第1章　授業の概要

1.1　講義で学ぶ事項

佐賀大学理工学部情報系 3 年生必修科目「プログラミング概論 III」では、2 年生科目「プログラミング概論/演習 I,II」の次のステップとして、最近では必須のプログラミング技術となっているオブジェクト指向プログラミングをダンジョンゲームを題材に丁寧に学びます。

みなさんが学んできた C++はオブジェクト指向プログラミングをサポートする言語のひとつですが、「プログラミング概論/演習 I,II」ではオブジェクト指向に踏み込まない基礎的な事項を学んできました。この講義ではオブジェクト指向を正面から取り上げます。

具体的には、

1. 初めに C++を用いてオブジェクト指向的ではないプログラムを作ります。
2. C++のプログラムを徐々にオブジェクト指向へ移行させていきます。
3. 講義の中盤で C++のオブジェクト指向プログラムを Java へ翻訳します。
4. Java のプログラムをより高度なオブジェクト指向へ移行させていきます。
5. さらに Java 特有のオブジェクト指向技術を適用します。

問題提起の意図もあって、この講義の途中では敢えて読みにくいエレガントでないプログラムを提示する箇所があります。しかし最終的には全て正しいプログラムの書き方へ修正していきます。

この授業を終えた後ではオブジェクト指向プログラミングの基本概念をほぼマスターできているはずです。ただしオブジェクト指向技術は学んですぐに使いこなせるようになるものではありません。少なくとも 2〜3 年程度以上のプログラミング経験があって初めてその重要性が実感できるようになります。その意味ではこの講義はあくまでもオブジェクト指向の機能や概念の習得が主な目的となります。

この講義は「オブジェクト指向プログラミング」について講義するものです。これと密接に関連する技術として「オブジェクト指向設計」があります。前者は、個々のプログラミング技術を、プログラムサイズで言えばせいぜい数十行程度のプログラムに限定し、種々ミクロに論じる内容で

す。建物に喩（たと）えるならば、玄関の扉の形、鍵の種類、階段の形状、鬼瓦の彫刻などに相当します。後者は、個々のプログラミング技術ではなく、プログラムサイズで言えば、数百行以上のプログラムについてマクロに論じる内容です。建物で言えば、家全体の外観、耐震強度、上下水道の配管などに相当するでしょう。もちろんいずれも大事な事項であって、両者は相互に関係するものです。この講義ではプログラム全体の設計についてはぼんやりと眺めつつも、主にプログラミングの細部の機能を丁寧に解説していきます。

1.2　講義の進め方、成績

まず、コロナ禍の状況によっては以下に述べる内容を変更することもありうるとあらかじめ了解してください。

講義は原則、対面授業で進めます。講義資料（pdf ファイル、解説動画）は Microsoft Teams にアップロードします。

成績は以下の 3 項目で評価し、0 点〜100 点の成績を付ける予定です。

小テスト ... 達成度確認のための小テストをオンライン試験で実施します。配点は毎週 2 点、計 30 点を予定しています。小テストは (1) 確認テスト、(2) その週の授業に関する疑問、質問、感想、の 2 問からなり、それぞれの配点は 1 点です。重要な質問、おもしろい質問には次回の授業で（または 動画で）回答します。

レポート点 ... ダンジョンゲームを自分なりに改造する課題を 2 回または 3 回予定しています。計 25 点の配点を予定しています。

期末試験点 ... 期末試験に 45 点の配点を予定します。

```
# # # # # # # # # # # # # #
# . . . . . . . . . . . . #
# . . . . . . . . . . . . #
# . . @ . . . . . . . . . #
# . . . . # # # . . . . . #
# . . . . # # # . . . . . #
# . . . . # # # . . . . . #
# . . . . . . . . . . . . #
# . . . . . . . . . . . . #
# . . . . . . . . . . . G #
# # # # # # # # # # # # # #
Player : (3,3) 100
```

図 1.1: プレイヤーのみが居るマップ

1.3　講義で作成するゲームプログラム

この節ではこの講義で作成するゲームプログラムを順に紹介します。

1.3.1　出発点のゲームプログラム

出発点となるゲームの実行画面は図 1.1 です。これはコンソール（C++の標準出力が表示されるウインドウ）を用いたゲームです。これを**キャラクタユーザインタフェース**（CUI）[1]によるゲームとも言います。

この段階では、岩（'#'）と床（'.'）で表されたマップにプレイヤー（'@'）が居て、右下のゴール（'G'）に向かって移動するだけです。岩はマップの周囲と中央（3×3 個の '#'）に配置しています。中央の岩は、これが無いとゲーム性が極端に下がるため、あえて置いています。マップの下にはプレイヤーの座標とその HP（health point）の値を表示しています。プレイヤーの移動はキーボードから入力します。これを C++でプログラミングします。

[1] あるいは、コンソールベースユーザインターフェース、コマンドラインユーザインターフェースとも言います。いずれも略語は CUI です。

```
# # # # # # # # # # # # #
# . . . . . . . . . . . #
# . @ . . . M . . . . . #
# . . . . . . . . . . . #
# . M . . # # # . . . . #
# . . . . # # # . . . . #
# . . . . # # # . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . G #
# # # # # # # # # # # # #
Player : (2,2) 100
Monster : (2,4) 150
Monster : (6,2) 150
```

図 1.2: プレイヤーと 2 体のモンスターが居るマップ

1.3.2　モンスターの追加

次に図 1.2 はモンスター（'M'）2 体を追加したゲームです。ゲームはターン制を取っており、プレイヤーとモンスターが交互を動いてゲームを進めて行きます。モンスターはプレイヤーの場所へ自動的に近づき、プレイヤーに隣接したならば攻撃します。

```
# # # # # # # # # # # # #
# . . . . . . . . . . . #
# @ . . . . . R . . . . #
# . . . . . . . . . . . #
# . . . . # # # . . . . #
# M . . . # # # . . . . #
# . . . . # # # . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . G #
# # # # # # # # # # # # #
Player : (1,2) 100
Monster : (1,5) 150
RedMonster : (7,2) 150
```

図 1.3: プレイヤーとモンスターとレッドモンスターが居るマップ

1.3.3　レッドモンスターの追加

図 1.3 は、モンスターのうちの 1 体をレッドモンスター（‘R’）に置き換えたものです。レッドモンスターは普通のモンスターよりも攻撃力が大きい設定になっています。オブジェクト指向の「継承」を用いてモンスターを拡張定義します。プログラムはここから Java へ移行します。

```
# # # # # # # # # # # # #
# @ . . . . . . R M . . #
# . . M . . . . . . . . #
# . . . . . . . . . . . #
# . M . . # # # . . . . #
# . . R . # # # . . . . #
# . R . . # # # . . R M #
# M . . . . . R . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . G #
# # # # # # # # # # # # #
Taro : (1,1) 100
Monster : (1,7) 150
Monster : (3,2) 150
Monster : (9,1) 150
RedMonster : (10,6) 150
RedMonster : (2,6) 150
Monster : (11,6) 150
RedMonster : (8,1) 150
RedMonster : (7,7) 150
RedMonster : (3,5) 150
Monster : (2,4) 150
```

図 1.4: プレイヤーと多数のモンスター、レッドモンスターが居るマップ

1.3.4　多数のモンスターの管理

図 1.4 は、多数のモンスター、レッドモンスターをマップ上にランダムに配置するように改造したゲームです。複数のモンスターを一括処理するため、オブジェクト指向の「多相性」を用います。

```
# # # # # # # # # # # # #
# . . . @ . . R o . o . #
# . . . . . R . . . o . #
# . . . . . . . . x . . #
# . o . . # # # . . . o #
# . . . . # # # . o . . #
# . . . . # # # M o . o #
# o o . o . . . M . . . #
# o . o . . . o . . x o #
# . . . o . . . o . o G #
# # # # # # # # # # # # #
Taro : (4,1) 200
RedMonster : (7,1) 190
RedMonster : (6,2) 150
Monster : (8,7) 150
Monster : (8,6) 240
```

図 1.5: 床に多数のポーションとポイズンがあるマップ

1.3.5　床の高機能化

図 1.5 は、床を高機能化（＝オブジェクト化）し、床にポーション（回復薬、‘o’） とポイズン（毒、‘x’） を配置できるように改造したゲームです。ポーションのある床に入ると HP が増えます。効果は最初にその床に入ったときの 1 回のみ有効です。ポイズンのある床に入ると HP が減ります。この床はトラップになっており、その床を踏まないとポイズンの有無が分からない仕様です。

図 1.6: グラフィカルなマップ

1.3.6　ゲームをグラフィカルに改造

図 1.6 は、ゲームを Java の GUI ツール AWT/Swing でグラフィカルに変更したものです。猫（＝プレイヤー）、緑のモンスター、赤いモンスターの画像を png ファイルで別途、用意しておき[2]、それをウインドウに描画します。茶色の正方形、水色の正方形、緑の正方形、黄色の正方形はそれぞれ岩、ポーション、ポイズン、ゴールのある床を表します。プレイヤーの移動はこの時点ではまだキーボードから入力します。

[2]モンスターのイラストは「かわいいフリー素材集：いらすとや」、`https://www.irasutoya.com` からダウンロードしました。モンスターについては色を変える加工を行いました。

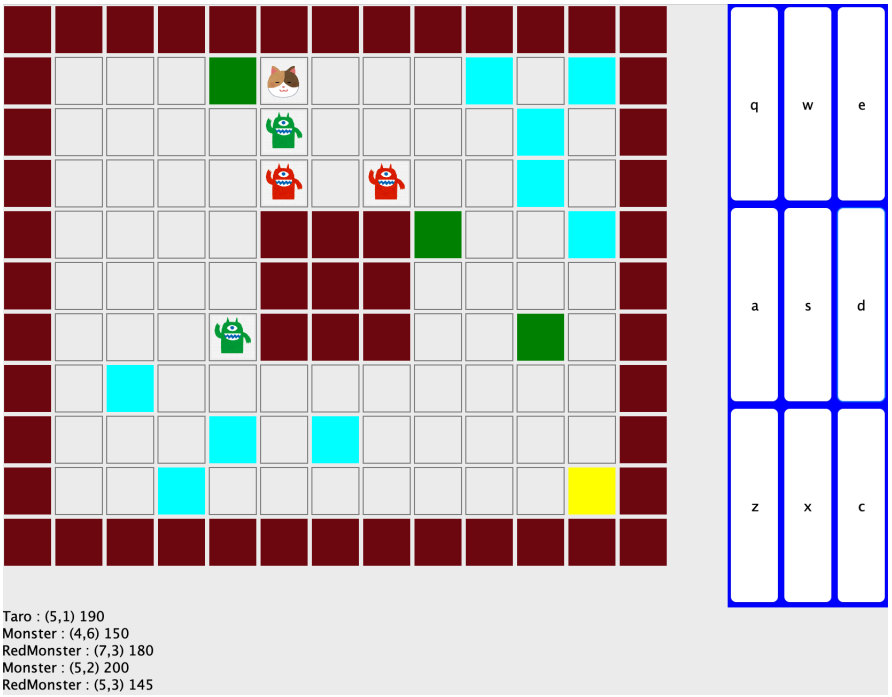


図 1.7: 動作ボタンが追加されたマップ

1.3.7　ボタンによる操作

図 1.7 は、ゲームにボタンを追加し、入出力全てをグラフィカルに改造したものです。右側の 9 個の縦長のボタンでプレイヤーの動作を入力することができます。このような見栄え、操作性のゲームは**グラフィカルユーザインタフェース**（GUI）[3]を用いるものと呼ばれます。

[3]コンピューターの画面上に表示されるウィンドウやアイコン、ボタン、プルダウンメニューなどを使い、マウスなどのポインティングデバイスで操作できるインターフェース（Wikipedia からの引用）

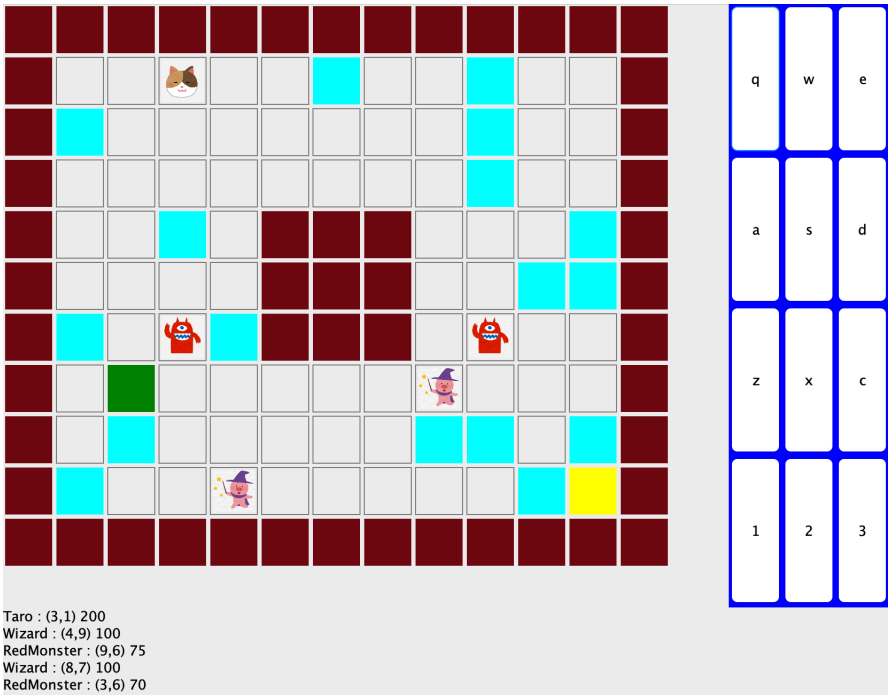


図 1.8: 遠隔攻撃ボタンと魔導師が追加されたマップ

1.3.8　遠隔攻撃の追加

図 1.8 の右側にはさらに 3 個のボタンを追加しました。これはプレイヤーの複数遠隔攻撃（マップ攻撃）を行うためのものです。また、モンスター側にも遠隔攻撃ができる魔導師（＝ブタ）が追加されます。遠隔攻撃を Java の「ラムダ式」で実装します。

また、ターン制を止め、プレイヤーのボタン操作を待たずにモンスターが勝手に動き出すリアルタイムシミュレーションゲームへの変更も行います。これには Java の並行プログラミング技法を用います。

この科目ではこのようなゲームの拡張を通してオブジェクト指向プログラミングの技術を順に学んでいきます。全体に内容が多く、やや難しい事項が出てきますが、できるだけ丁寧に解説していきます。

1.4　プログラムリストについて

この授業では様々なプログラムリストを示します。

テキストでは、混乱がないように、図表番号付きのダンジョンゲームのプログラムリストは以下の影付き枠で示します。

その他のプログラムリストは全て以下の丸枠で示します。

1

1

第2章　様々なプログラミング言語

オブジェクト指向プログラミングを学ぶに当たり、プログラミング言語の歴史を俯瞰しておくことは重要です。

2.1　プログラミング言語の役割

初めにプログラミング言語の必要性を確認しておきます。

C/C++プログラムの代入文:

```
x = a*b+c;
```

をアセンブリ・プログラム風に記述すると、以下のようになります。

```
load  r1 = [&a]        // a のアドレスからレジスタ r1 へ数値をロード
load  r2 = [&b]        // b のアドレスからレジスタ r2 へ数値をロード
mult  r3 = r1,r2       // r1 と r2 を乗算し、レジスタ r3 へ格納
load  r4 = [&c]        // c のアドレスからレジスタ r4 へ数値をロード
add   r5 = r3,r4       // r3 と r4 を加算し、レジスタ r5 へ格納
store [&x] = r5        // r5 の値を x のアドレスへストア
```

上の二つを比べれば、C プログラムを読む/書く労力がアセンブリ・プログラム片を読む/書く労力よりも小さいことは明らかです。C プログラムの改造がアセンブリ・プログラムの改造よりも容易であることも明らかです。プログラミング言語の重要性はアセンブリ・プログラミング（機械語プログラミング）を経験した者ならば誰でも身に染みて分かります。

では、どのようなプログラミング言語を用いれば、より効率的なプログラム開発が可能になるのでしょうか。その答えは一様ではありません。どのような処理を行うプログラムを作るか、プログラムの実行速度を重視するのか読み易さを重視するのか、どのようなコンピュータの上で実行するのか、ユーザーインターフェースはどうするのか、等々に依存しています。実際、用途に応じて様々なプログラミング言語がこれまでに設計・開発されてきました。

表 2.1: 低水準プログラミング言語の分類

	分類	言語名	備考
低水準	機械語 （ネイティブ・コード）	—	プロセッサ毎に異なる
	アセンブリ言語	—	プロセッサ毎に異なる
	中間言語	P コード	Pascal の中間言語
		バイトコード	狭義には Java の中間言語

プログラミング言語の違いは、絵画に喩（たと）えるならば絵筆の違いに相当します。油絵と水彩画、水墨画では使用する筆が違います。ひとつの筆で全ての絵画を描くことができないように、ひとつのプログラミング言語で全ての用途を満たそうと考えるのは現実的ではありません。

2.2　プログラミング言語の種類と歴史

この節では代表的なプログラミング言語について簡単に紹介していきます。

プログラミング言語は**高水準プログラミング言語**（high-level programming language[1]）と**低水準プログラミング言語**（low-level programming language）に大別されます。高水準言語はハードウェア・アーキテクチャを意識させない言語構造を持ち、人手によるプログラミングのしやすさ、プログラムの読みやすさを指向したプログラミング言語です。これに対して低水準言語はハードウェア・アーキテクチャを反映した言語構造を持ち、特定のハードウェアで高速に動作する利点を持ちます。

2.2.1　低水準プログラミング言語

機械語

機械語（machine language）は、プロセッサが直接解読可能な 2 進数データ列（ビット列）です。現在、機械語を読むことも機械語でプログラムを作ることはほとんどあり得ません。機械語プログラムのことを**ネイティブ・コード**（native code）と呼ぶことがありますが、その場合には、議論の文脈の中で、ある特定のプロセッサが暗黙に仮定していることが一般的です。

[1] しばしば「高級プログラミング言語」と訳されます。高級/低級という呼び方はあまり適切とは思えませんが、google 検索では「高級...」という訳語を用いているサイト数が「高水準...」の 2 倍近くあります。

アセンブリ言語

アセンブリ言語（assembly language）は、機械語に 1 対 1 対応するプログラミング言語です。2 進数データ列である機械語はそのままでは人間にとって非常に読みづらいため、キーワードなどを用いて少し読み易く表したものです。アセンブリ言語を機械語に変換する事をアセンブル（assemble）するとも言い、その変換を行うプログラムを**アセンブラ**（assembler）と呼びます[2]。

1952 年には Mark I コンピュータ用アセンブラが開発されており、これが世界初のアセンブラとも言われています。アセンブラで得られた知見は、後の高水準言語コンパイラの開発につながっていきます。

中間言語

中間言語（intermediate language）は、低水準言語と高水準言語の中間の性質を持つような言語の総称です。低水準言語は高速実行可能ですが、機械依存が強すぎて可搬性[3]に乏しく、高水準言語は可搬性に優れていますが、機械から離れすぎて実行効率が悪いという互いに相反する性質を持ちます。中間言語はこれら二つの性質の中間的な性質を持ちます。Pascal の P コード（P-code）や Java のバイトコード（Bytecode）が有名です。中間とは言うものの、どちらかと言えば低水準寄りのものが多いため、このテキストでは中間言語を低水準言語に含めました。

Java のバイトコードについては後章で解説します。

2.2.2　高水準プログラミング言語

高水準言語は**手続き型プログラミング言語**（procedural programming language、以下、手続き型言語）と**宣言型プログラミング言語**（declarative programming language、以下、宣言型言語）に大別されます（表 2.2 参照）。

手続き型言語とは手順の時系列によって処理内容を記述する種類の言語です。処理内容を順番に記述していくため、考え方が単純で平易なプログラミングが可能です。しかし、処理の順序を少し間違えただけで予想もしない結果になることが多く、複雑なプログラムの開発では注意力が要求されます。プログラムのデバッグでも手順の実行順序を丹念に追わねばならず、容易ではありません。

宣言型言語とは、処理内容をその数学的性質に着目して、あたかも数式を書くように記述する種類の言語です。処理内容を数学の公理のように表現可能であるときに、コンパクトで美しいプログ

[2] 三つの術語「アセンブリ（assembly）」、「アセンブル（assemble）」、「アセンブラ（assembler）」は似ています。品詞に注意して誤用のないように注意しましょう。

[3] 言語処理系が様々なコンピュータへ容易に移植できること、プログラムが様々な種類のコンピュータで実行できることを可搬性が良いと言います。

表 2.2: 様々な高水準プログラミング言語の分類

	分類 1	分類 2	言語名	誕生時期	利用状況（私見による）
高水準	手続き型 (命令型)	手続き型 (命令型)	FORTRAN FORTRAN77 Fortran90 ...	1950 年代	科学計算分野に利用されている
			COBOL	1950 年代	商業分野に利用されている
			ALGOL	1960 年代	歴史的使命を終えた
			Pascal	1970	ほとんど利用されていない
			C	1972	利用されている
			C++	1980 年代	使用されている
			Basic	1964 から	使用されている
			Ada	1980 年代	ほとんど利用されていない
			Java	1990 年代	最も利用されている
			Python	1990 年代	利用が拡大している
			Swift	2014	使用されている
	宣言型	関数型	LISP	1958	特定分野に使用されている
		論理型	Prolog	1972	特定分野に使用されている

ラムを書くことができます。反面、明確な時間順序を含むような処理内容の記述には不向きです。処理の順序を考慮する必要がない[4]ため、プログラムのデバッグは比較的容易と言われています。

代表的なプログラミング言語を表 2.2 にまとめました。以下ではこれらの言語の中からこの授業に特に関係する C/C++/Java について概説する。

C

AT&T のベル研究所（Bell laboratories）で開発された C（シー） は、UNIX を記述した言語として有名です。Pascal などと比較してエレガントな言語とは言いがたいのですが、OS をほとんどこの言語だけで記述できるなど応用範囲が広く、急速に普及しました。現在、ほとんどのプログラムの開発において C はアセンブリ言語に取って替わったと言ってよいでしょう。1972 年に最初の処理系が開発されました。C を普及させた要因のひとつはカーニハン（Brian Kernighan）とリッチー（Dennis Ritchie）が 1978 年に書いた C マニュアル[5]の貢献が大きいとも言われています。この当時の C を俗に「カーニハン&リッチーの C（K&R C）」と呼びます。C の仕様策定と標準化

[4] このような性質を一般に参照透明性（referential transparency）と呼びます。

[5] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall(1978). Second edition (1988).

```
int main(){
    int i;
    float sum = 0;
    for (i = 1; i <= 10; i++)
            sum += i;
    printf("%f\n",sum);
}
```

(a) C

```
int main(){
    float sum = 0;
    for (int i = 1; i<= 10; i++)
            sum += i;
    cout << sum << endl;
}
```

(b) C++

図 2.1: C/C++のプログラム例

は、1983 年に ANSI[6]に受け継がれ、これが今日 ANSI C と呼ばれているものです。

図 2.1(a) に、1 から 10 までの数の和を求めるプログラムを C で記述した例を示します。

混乱を避けるため、これ以降は C のことを「C 言語」と呼びます。

C++

C++（シープラスプラス） は、1980 年前半にベル研究所で開発された拡張 C 言語です。C 言語の構文規則を洗練し、機能を整理した言語でもあります。最も大きな特徴は C 言語にオブジェクト指向を導入した点にあります。C++の仕様は ANSI C にも影響を与えました。

図 2.1(b) に、1 から 10 までの数の和を求めるプログラムを C++で記述した例を示します。ただしこの例題では C 言語と C++の違いが際立ちません。

Java

Java（ジャバ） は、1990 年初めに家電組み込み用言語として開発が進められましたが、折しもインターネットの拡大期にあったため、その目的をネットワーク対応にシフトし、大きな注目を集めるようになりました。1995 年に最初の処理系が開発され、その後も改良が続けられています。Java の構文は C++に類似しており、C/C++プログラマには馴染みやすかった点も普及の一因と思われます。様々な興味深い特徴を持ちますが、その中でも、中間言語（バイトコード）を設定し、プラットフォーム非依存[7]である点は大きな特徴です。現在、最も普及している言語のひとつです。

図 2.2 に、1 から 10 までの数の和を求めるプログラムを Java で記述した例を示します。プログラムが C や C++に似ていることに気づくでしょう（図 2.1 参照）。

[6] the American National Standards Institute

[7] プラットフォームとは、コンピュータの機種（またはアーキテクチャ）および OS の種類のことです。機種や OS に依らず成り立つ性質をプラットフォーム非依存と言います。しばしば OS 非依存とも言います。

```
class Test {
    public static void main(String[] args){
        float sum = 0;
        for (int i = 1; i <= 10; i++)
            sum += i;
        System.out.println(sum);
    }
}
```

図 2.2: Java のプログラム例

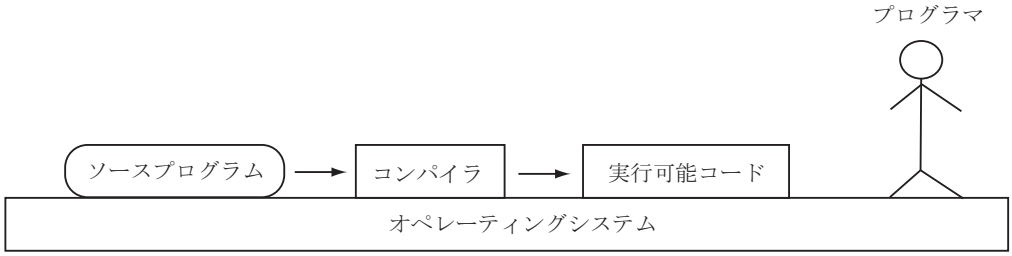


図 2.3: コンパイラを用いるプログラミング環境

2.3　プログラミング言語処理系

プログラミング言語を学ぶに当たり、その言語のプログラムを実際に実行する言語処理系について知識を整理しておきます。

プログラミング言語のプログラムを実際にコンピュータ上で実行する方法は、コンパイラを用いる方法とインタプリタを用いる方法の二種類に大別されます。

2.3.1　コンパイラ

コンパイラ（compiler）は、あるプログラミング言語 L_S で書かれたプログラムを別のプログラミング言語 L_T で書かれたプログラムへ自動変換するソフトウェアです。通常、変換元の言語（source language） L_S は高水準プログラミング言語であり、変換先の言語（target language） L_T は低水準言語（多くの場合は機械語）です。変換前の L_S のプログラムを**ソース・プログラム、原始プログラム**（source program）、**ソース・コード、原始コード**（source code）などと呼びます。変換後の L_T のプログラムを**オブジェクト・プログラム、目的プログラム**（object program）、**オブジェクト・コード、目的コード**（object code）などと呼びます。このテキストではソース・プログラムとオブジェクト・コードという呼び方を用います。図 2.3（20 ページ）はコンパイルによる実行環境を図示したものです。

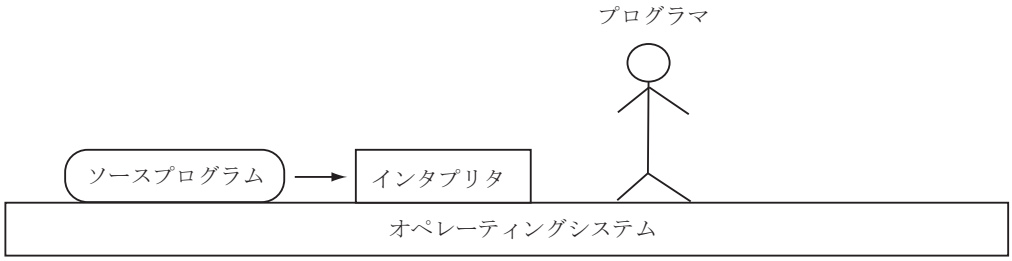


図 2.4: インタプリタを用いるプログラミング環境（バッチモード）

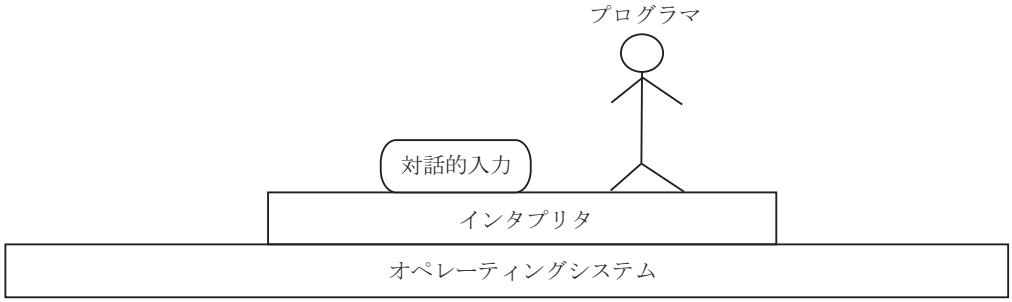


図 2.5: インタプリタを用いるプログラミング環境（対話モード）

なお、オブジェクト・コードはそのままでは実行できないことは注意しておきます。たとえばソース・プログラム中で三角関数 `sin()` を呼んでいる場合、オブジェクト・コードにもその関数呼び出しは存在しますが、 `sin()` の値を計算するプログラムはオブジェクト・コードには含まれません。そのため、**リンカ**（linker）を使ってシステムが用意する三角関数プログラムと結合し、**実行可能コード**（executable code）を作る必要があるのです。以下では、コンパイラにリンカも加えて、広い意味でコンパイラと呼ぶ場合があります。

コンパイラを用いたプログラム開発では、ソース・プログラムを作成した後にいちいちコンパイル（compile）せねばならないという手間が掛かることが欠点です。しかし、一旦実行可能コードが完成した後はプログラムが高速に実行できるという利点があります。高速性を重視する大規模科学技術計算などではコンパイラは必要不可欠なツールです。

2.3.2　インタプリタ

コンパイラがソース・プログラムを実行可能コードに変換するソフトウェアであるのに対して、**インタプリタ**（interpreter）はソース・プログラムを逐次解析（解釈）しつつ実行するソフトウェアです。図 2.4（21 ページ）はインタプリタによる実行環境を図示したものです。

インタプリタを用いる場合、ソース・プログラムをいちいち実行可能コードへ変換する必要はな

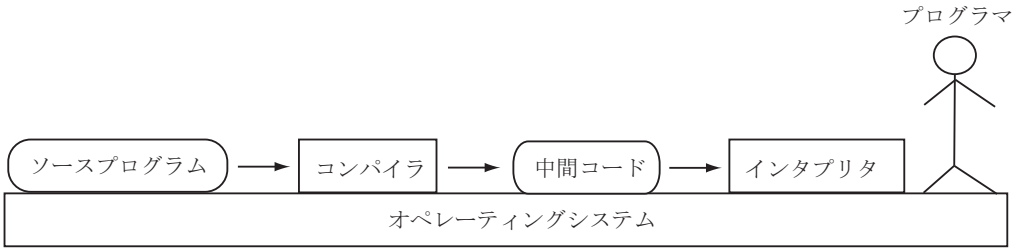


図 2.6: コンパイラ・インタプリタを用いるプログラミング環境（バッチモード）

く、プログラムを作ったその場でプログラムの実行ができます。あるいはプログラムを作りながら実行できます。そのため、プログラマはシステムと対話的にプログラムを作成することができ、応答性に優れた開発環境を得られるという利点があります。図 2.5（21 ページ）はインタプリタによる対話的な実行環境を図示したものです。しかし、インタプリタでは実行の度にプログラムを解析しなければならないため、実行速度が遅いという欠点もあります。

「プログラムを作りながら実行できる」という特性を活かすには、プログラムの意味内容が局所的に定まり、広範囲の解析を必要としない単純な構造の言語が適しています。あるいはプログラムの実行環境が動的に変化するため、コンパイルしても実行性能が期待できないような言語が適しています。宣言型言語の言語処理系はインタプリタで実装されます。前節で取り上げなかった Pearl や Ruby などのスクリプト言語もインタプリタまたは次に述べるコンパイラ・インタプリタ方式で実装されています。

2.3.3　コンパイラ・インタプリタ

中間言語を介した実行方法です。コンパイラがまずソース・プログラムを中間言語のプログラムに変換します。その中間言語プログラムをインタプリタが逐次実行します。この場合、コンパイラとインタプリタの両方を開発せねばなりません。しかし、中間言語プログラムが一旦生成されたならば、インタプリタが動く全ての環境でプログラムを実行可能であるため、可搬性に優れた方式です。実際、プログラミング言語 Pascal ではこの方式で様々なコンピュータに Pascal コンパイラが移植されました。Java では中間コードがネットワークを介して様々なコンピュータで実行できます。

可搬性以外のこの方式の特徴は、コンパイラによる実行方式とインタプリタによる実行方式の中間と考えてよいでしょう。

2.3.4　ジャストインタイム・コンパイラ

中間コードをインタプリタ内でネイティブ・コードにコンパイルして実行する方式、あるいはそのコンパイラを**ジャストインタイム・コンパイラ**（Just-In-time compiler）または **JIT コンパイラ**などと呼びます。Java のそれが有名です。Java の場合、新しい実行コードが呼び出される度に JIT コンパイラが起動します。コード中にループがあるような場合には中間コードをそのまま実行するよりも効果的です。

なお、実行速度を除けば、ユーザからは単に中間コードがインタプリタによって実行されているとしか見えないため、JIT コンパイラを用いても前節のコンパイラ・インタプリタの可搬性が損なわれることはありません。JIT コンパイラはインタプリタの実行速度を加速させるための仕組みのひとつと考えればよいでしょう。プログラム実行中にコンパイラを起動すると、プログラム本体の実行は一時中断されます。プログラム実行の不自然な挙動を避けるためにはコンパイラが長時間動作することは好ましくなく、コンパイルに長時間を費やすことはできません。そのため実行コードの質では妥協せざるを得ません。

2.4　ふたたび講義の概要：オブジェクト指向プログラミングの視点から

オブジェクト指向プログラミングには大きく二つの系統があります。ひとつは**クラスベース**（class based）のオブジェクト指向、もうひとつは**プロトタイプベース**（prototype based）のオブジェクト指向です。この講義テキストは前者に基づくものです。

前者のクラスベースのオブジェクト指向プログラミングには基本原理として (1) 継承、(2) 多相性、(3) 情報隠蔽などからなると一般に考えられています。もちろんこの講義でこれらを解説する訳ですが、以下にこれら原理と講義の流れについて概説します。

構造体の復習　クラスベースのオブジェクト指向の出発点は、データの集まりである構造体です。まず、これについて復習します。

構造体をクラスへ昇格　次に構造体に関数（あるいは手続き）を追加します。そのようにして出来上がったデータ型は、クラスを成します。そのクラスからオブジェクトを生成できるようになります。

クラス階層（継承）の導入　複数のクラス間に類似性を見つけ、それを整理することで、クラス間に親子関係を作ることができます。子供クラスは親クラスの性質を継承します。これによってプログラムの集中管理が可能になります。

多相性の利用 クラス階層上のオブジェクトの振る舞いは、オブジェクト実体に応じて適切な処理を動的に選択実行できます。これはオブジェクト指向の最も興味深い性質ですが、わかりにくい点でもあります。講義ではゲームを題材にしてできるだけ実感できる解説を行います。

アプリケーションフレームワークの利用 上記のオブジェクト指向プログラミングの性質を利用することで、ビジュアル的に優れたリアルタイムのゲームを簡単に作ることができます。

情報隠蔽 オブジェクトの中身を外部から不用意に変更できないように防御する仕組みはプログラムの信頼性を高める上で重要です。

なお、これらのいくつかはプロトタイプベースのオブジェクト指向にも共通していますが、そうでないものもあります。

1

1

第3章　このテキストでのプログラムの書き方

この講義は広い意味でプログラムの書き方を学ぶものですから、このテキストで用いるプログラムも世の中で広く知られた、標準的に認められたプログラミング作法に則った書き方を目指します。ただし、あまり拘（こだわ）りすぎると先へ進めなくなりますから、適度に妥協して先を急ぐところも多々あります。

3.1　標準的な指南書

現代的な標準的なプログラムの書き方を指南する資料は様々あります。

以下は Java のプログラムの書き方ですが、講義担当者に最も違和感がありません。

Code Conventions for the Java TM Programming Language

`https://www.oracle.com/java/technologies/javase/codeconventions-contents.html`

その和訳があるらしいのですが、残念ながら現在、リンク切れでアクセスできません[1]。

以下の Google の規約集も勉強になります。

Google Style Guides

`https://google.github.io/styleguide/`

ここから参照できる以下の資料

Google C++ Style Guide

`https://google.github.io/styleguide/cppguide.html`

が C++のプログラミングに特化した内容です。その和訳は、たとえば

Google C++ スタイルガイド 日本語全訳

`https://ttsuki.github.io/styleguide/cppguide.ja.html`

などがあります[2]。これを読むと、何故そのような書き方を用いるべきなのか、その理由も述べられており、非常に勉強になります。同様に、Java については

[1] 和訳を検索したい人は、たとえば、タイトル ＋ 和訳 などのキーワードで検索してみてください。
[2] 「Google C++ Style Guide 和訳」で検索すると、類似のサイトが見つかります。

Google Java Style Guide

https://google.github.io/styleguide/javaguide.html

があり、その和訳は、たとえば

Google Java Style Guide(非公式和訳)

https://kazurof.github.io/GoogleJavaStyle-ja/

などがあります。

これらは非常に参考になるプログラミング規範ですが、いくつかの理由でこの講義テキストでは上の指南に従わない書き方も採用する場合があります。最も大きな理由は、

- 講義テキストにプログラムを読み易い形で載せる

ためです。コンピュータディスプレイの場合には画面を縦横にスクロールしながら長大なプログラムを読むこともできます。しかしテキストの 1 ページは意外に狭く、長い変数名、関数名はスペースを圧迫し、改行を多用する書き方は行数が増大します。プログラム長が 1 ページを超えるとプログラムは途端に読みにくくなりますから、ここではスペースの圧縮を意識した書き方を用いることとします。

さて、上に挙げた資料にも書かれていますが、プログラムの書き方で最も大事なことは

- プログラムの書き方は統一すべきである

という点です。優れた書き方であっても、ひとつのルールで統一されていなければ、結局、それは良いプログラムの書き方とは言えません。

以下、この講義テキストでのプログラムの書き方を述べていきます。

3.2　インデント

インデントは必ず付けてください。インデントで注意すべき点は、仮に 4 文字幅を使うとして、その幅を

- タブ文字で開けるのか、それとも
- 半角空白文字 4 個で開けるのか

という選択です。一長一短あります。これについてはたとえば

インデントをしよう！（プログラミング）

https://blog.senseshare.jp/indent.html

などを参照してください。

3.3　中括弧

ブロックを表す中括弧 { } の使い方では、一般的には、たとえば if 構造の場合、

```
if (i == 0) {
    f(10);
} else {
    g(2);
}
```

などという書き方が多いのですが、最近は以下の書き方も広がりつつあります。

```
if (i == 0)
{
    f(10);
}
else
{
    g(2);
}
```

後者の方が括弧の対応付けが見やすい利点がありますが、行数を多く使用し、前者に慣れた目からすると空き過ぎているようにも思えます。講義担当者は後者が良いと思うのですが、すでに述べた理由からこの講義テキストでは少ない行数でコンパクトな記述ができる前者を採用します。なお、3.1 項の “Google C++ Style Guide” は前者を推奨しています。

次に、ブロック内の文の数がひとつの場合、以下の書き方の方が読みやすく思われます。

```
if (i == 0) { f(10); }
else { g(2); }
```

これは以下のように書いてもプログラムの実行では等価です。記述はよりコンパクトです。

```
if (i == 0) f(10);
else g(2);
```

しかし、後者はブロック構造が把握しにくくなりますから、ここでは前者を用います。

ブロック内が空の場合にも中括弧 { } は省略しないことにします。つまり

```
while (f(0) == 0) {}
```

を採用し、

```
while (f(0) == 0) ;
```

は採用しません。これは 3.1 項の “Google C++ Style Guide” の推奨する書き方でもあります。

3.4　名前

ユーザーが名前を付けることのできるデータ型名、変数名、関数名にも標準的な約束があります。名付けルールを名前の種類毎に分けるとプログラムが読みやすくなります。

3.4.1　データ型名、クラス名

このテキストではユーザー定義のデータ型、クラスの名前には以下のルールを用います。

- 英大文字で始まり、単語の区切り毎に英大文字を用いる。アンダースコアで区切らない。

たとえば後章では `RedMonster` というクラス名を用います。このような単語の区切り毎に英大文字を用いる方法を**キャメルケース**（camel case、ラクダのコブの凸凹を想像しましょう）と呼びます。大文字で始まる場合を特にアッパー（upper）キャメルケース、逆に小文字で始まる場合をローワー（lower）キャメルケースと呼びます。

アンダースコアで区切る方法とは、 `RedMonster` を `Red_monster` と命名する方法です。これを**スネークケース**（snake case、ヘビの歩いている姿を想像しましょう）と呼びます。これは 3.1 項の “Google C++ Style Guide” の推奨する書き方です。

3.4.2　関数名

このテキストでは関数名（メンバー関数名、メソッド名を含む）には以下のローワーキャメルケースのルールを用います。

- 英子文字で始まり、単語の区切り毎に英大文字を用いる。アンダースコアで区切らない。

たとえば後章では `mapCellPrint` という関数名を用います。なお、“Google C++ Style Guide” では、英大文字で始まるルールを推奨していますが、講義担当者はデータ型名との混乱を避けるため、大文字スタートは採用しません。

3.4.3　変数名

このテキストでは変数名（データメンバー名、インスタンス変数名を含む）には関数名と同様のルールを用います。たとえば後章では `playerHp` という変数名を用います。なお、“Google C++ Style Guide” ではスネークケースを推奨していますが、ここでは採用しません。

3.4.4　定数名

定数名には以下のルールを用います。

- 英大文字だけを用い、単語をアンダースコアで区切る。

たとえば後章では `MAP_XSIZE` という定数名を用います。上のルールは一般にマクロ名[3]のルールにして知られているもので、今や古いルールです。最近のプログラミング言語で「マクロ定義」を用いるものはありません[4]。その意味で古い書き方なのですが、名付けのルールとしては変数名などと一目で区別できるため分かりやすいと講義担当者は考えてます。なお、“Google C++ Style Guide” では定数名は変数名と同じルールを用い、ただし名前の初めに ‘`k`’ の付いたものを推奨しています。

3.5　空白文字の利用

プログラム片の随所に空白文字を入れてプログラムを読みやすくすることも重要です。

3.5.1　制御文キーワードの直後

C/C++/Java ではプログラムの制御を行うキーワード（ `if`、 `while`、 `for` など）の後には必ず左括弧 ‘(’ が現れますが、両者の間に空白文字をひとつ入れると、関数呼び出しと区別が付きやすくなります。たとえば、

[3] `#define` で定義される名前。

[4] “Google C++ Style Guide”（Google C++ スタイルガイド 日本語全訳）には「マクロを定義しようだなんて思ってないですよね？ ... マクロを定義してはいけません (特にヘッダでは)。 インライン関数や、列挙型、const 変数を使いましょう。 」と書かれています。

```
for(int x = 0; x < MAP_XSIZE; x++) {
```

ではなく、

```
for (int x = 0; x < MAP_XSIZE; x++) {
```

です。たったこれだけの工夫で可読性が上がります。

3.5.2　式の途中の空白

これもありがちですが、不要な空白を全て排除した

```
for(int x=0;x<MAP_XSIZE;x++){
```

はコンパイルエラーにこそなりませんが、非常に読みにくいものです。また、

```
x=a+b*(c+d);
```

も読みにくく感じます。かと言って、

```
x = a + b * ( c + d );
```

では間延びした印象になります。前掲の指南書では、

- 2 項演算子の前後には空白文字を置くべき

と書かれています。この辺の感覚は主観によるところも大きいのですが、空白を意識したプログラミングを心がけ、自分のスタイルを見つけてください。

この講義テキストでもこれを尊重しますが、スペースを減らしたい場合には圧縮した書き方を用います。

3.6　補足：単一代入

単一代入（single assignment）のルールは、「プログラムの書き方」からはやや外れる内容ですが、プログラミング作法として重要な事柄ですからここで紹介しておきます。これは、ひとつの変数には 1 回（single）しか値を代入（assign）しないプログラムの書き方です。

やや極端な例ですが、以下のプログラム片を考えましょう。ここでは花子と太郎の得点の平均を計算します。

```
int s = 0;
int p = 90;   // 花子の得点
s += p;
p = 80;       // 太郎の得点
s += p;
s = s/2;      // 二人の平均値
```

このプログラムの問題は、変数 s と p を使い回している点です。これを単一代入の原則で修正すると以下のようになります。

```
int s = 0;
int ph = 90;  // 花子の得点
s += ph;      // ここは単一代入に反する
int pt = 80;  // 太郎の得点
s += pt;      // ここは単一代入に反する
int a = s/2;  // 二人の平均値
```

使用する変数が増えている点に注意してください。単一代入では 1 回のみの代入であるため、変数の使い回しはできません。よって必然的に変数の数が増えるのですが、現在のコンピュータは潤沢なメモリを搭載していますから、それは問題になりません。変数 s のみ単一代入ではありませんが、これは問題の性質上、仕方ありません。

単一代入の長所は、プログラムが読みやすくなることです。単一代入の変数は特定の意味のみを持ちます。たとえば ph は花子の得点であってそれ以外ではありえません。これがプログラムの読解を容易にします。それに対して使い回す場合、変数の意味はプログラムの場所によって変化します。変数 p はある時点では花子の得点ですが、別の時点では太郎の得点です。これが可読性を損ね、バグの要因となります。

単一代入のもうひとつの長所は、実行文の記述位置に融通が効くことです。たとえば上の 2 番目のプログラム片は以下のように書くこともできます。

```
int s = 0;
int ph = 90;  // 花子の得点
int pt = 80;  // 太郎の得点
s += ph;      // ここは単一代入に反する
s += pt;      // ここは単一代入に反する
int a = s/2;  // 二人の平均値
```

このような変形は最初のプログラムでは出来ません。試しに最初のプログラムについて、あえて同じように位置を変えてみたのが以下のプログラムです。

```
int s = 0;
int p = 90;   // 花子の得点
p = 80;       // 太郎の得点
s += p;
s += p;
s = s/2;      // 二人の平均値
```

言うまでもなく、このプログラムは平均値を計算しません。単一代入の記述位置の融通性はプログラム読解に優れており、かつバグを減らします[5]。

上は当たり前のことを主張していると感じるかもしれません。しかし最近になってようやく単一代入が重要であることが認識されるようになってきました。最近のプログラミング言語では、単一代入される変数とそうでない変数の宣言方法を区別する言語仕様になっているものがあります。たとえば Javascript の `const` と `let`、Swift の `let` と `var` がそれです。これら仕様は上に述べた内容を背景したものです。もちろん単一代入変数が推奨されています。

プログラムの書き方にはこの他にも様々なルール、慣用がありますが、上記以外については適宜、紹介することとします。

[5] コンパイラによるプログラム最適化にも適しています。

1

1

第4章　とりあえずマップとプレイヤーをプログラム化する

この章では出発点のプログラムを示します。プログラミング概論/演習 I,II で学んだ内容の範囲で書かれた C++です。

4.1　プログラムの構造

図 4.1（38 ページ）がこの章のプログラムで定義、使用される関数と変数を図示したものです。このようなプログラム構造を表す図として UML（Unified Modeling Language）による図示もよく用いられますが、UML の図は慣れていないと理解が難しい（実は講義担当者も...）と考え、ここでは素朴な図を用いることとします。この章のプログラムでは、4 個のグローバル変数 `mapData`、`playerX`、 `playerY`、 `playerHp` を用います。`mapData` は 2 次元配列です。図では角丸長方形（角の丸い長方形）で描いています。関数は 14 個を定義します。図では長方形で描いています。図 4.1 の赤い矢印は関数の呼び出し関係（呼び出す関数 → 呼び出される関数）、青い矢印はデータのアクセス関係（関数 → 変数）を表します。変数名、関数名は次章以降の展開を事前に考慮したものになっています。

4.2　宣言部

図 4.2（39 ページ）はプログラムの宣言部です。いわゆる**ヘッダーファイル**（headr file）です。本来、ヘッダーファイルがあるならば、プログラムを複数のファイルに分割し、分割コンパイルするのが自然ですが、この講義テキストでは簡単のためにプログラムはひとつのファイルにまとめて記述されているものとします[1]。

図 4.2 はファイルの冒頭の部分に配置します。

4.2.1　宣言の順序

初めに宣言部に関する一般論です。

[1] 分割コンパイルの方法はこの講義では解説しません。

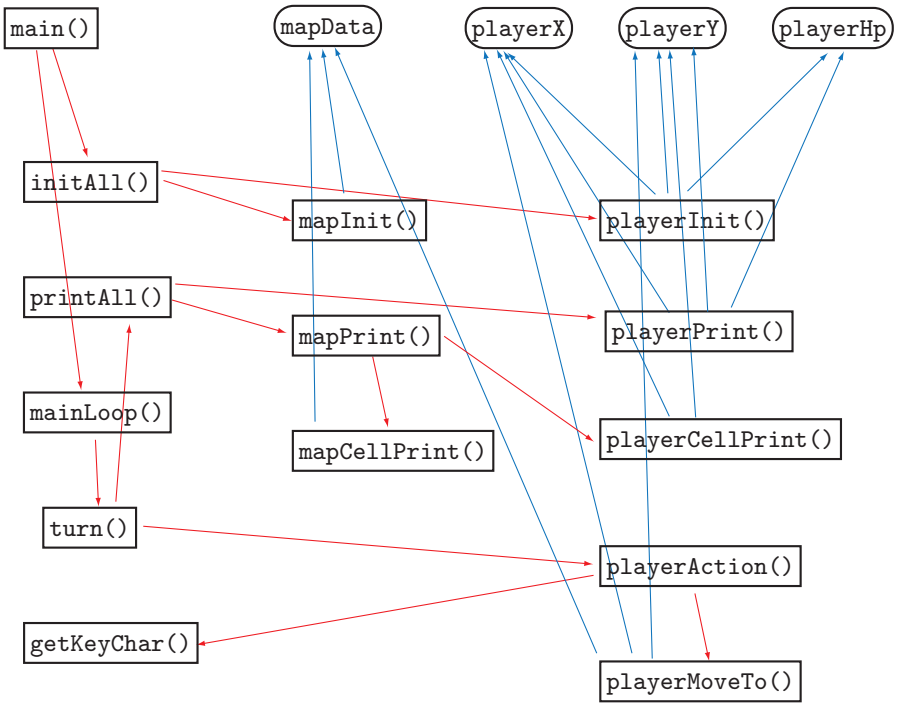


図 4.1: この章のプログラムの構造：長方形が関数、角丸長方形がデータ、赤い矢印が関数の呼び出し関係、青い矢印がデータのアクセスを表します。

ヘッダーファイルに書かれる宣言には種別毎に宣言の順序があり、C++では以下の順に宣言するのが一般的です。これは、種別間の参照関係を考慮して定まっており、順序を入れ替えるとコンパイルに支障が出る場合があります。

1. ファイルインクルード
2. 定数宣言
3. データ型宣言
4. プロトタイプ宣言
5. グローバル変数宣言

この順序になる理由は、データ型宣言はしばしば定数を参照しますから、定数宣言が先に来ます。プロトタイプ宣言（関数の外型の宣言）[2]で参照されるデータ型はしばしば直前のデータ型宣言のデータ型を参照します。グローバル変数は実装に寄った内容のため、最後に宣言されます。これらの参照関係から一般的に上の順序です。

[2]関数の名前、引数の数、引数のデータ型の組み合わせをシグネチャ（signature、シグニチャ）と言います。戻り値のデータ型を追加する場合もあります。名前を省いたものを特に型シグネチャ（type signature）とも言います。

```
#include <iostream>
using namespace std;

// 定数宣言
const int ROCK = 0;                 // 岩
const int EMPTY = 1;                // 床に何も居ないことを表す
const int GOAL = 2;                 // ゴール

const int MAP_XSIZE = 13;           // マップの横方向のセルの数
const int MAP_YSIZE = 11;           // マップの縦方向のセルの数

const int PLAYER_INIT_HP = 100;     // プレイヤーの初期 HP

// データ型宣言
// このプログラムにはありません

// プロトタイプ宣言
void initAll();                         // ゲーム全体の初期化
void printAll();                        // ゲーム状態の表示
void mainLoop();                        // ターンの繰り返し実行
void turn();                            // 1 回のターン実行
char getKeyChar();                      // キーボードからの 1 文字入力

void mapInit();                         // マップの初期化
void mapPrint();                        // マップの表示
void mapCellPrint(int x, int y);        // セルの表示

void playerInit();                      // プレイヤーの初期化
void playerPrint();                     // プレイヤーの状態の表示
bool playerCellPrint(int x, int y);     // プレイヤーのセル上への表示
bool playerMoveTo(int dx, int dy);      // プレイヤーの移動
void playerAction();                    // プレイヤーのアクション

// グローバル変数宣言
int mapData[MAP_XSIZE][MAP_YSIZE];      // マップ

int playerX, playerY;                   // プレイヤーの現在の位置
int playerHp;                           // プレイヤーの現在の HP
```

図 4.2: 出発点のダンジョンゲームのプログラム（各種宣言）

なお、出発点のこの章のプログラムにはデータ型宣言はありません。次章のプログラムではここに構造体を定義することになります。

4.2.2　定数宣言

図 4.2 の定数宣言部で使用されている修飾子 const はその変数の指し示す値が一旦初期化された後、プログラム実行中に変化しないことを表すものです。よって厳密に言えば、定数ではなく、値が変化しない変数ですが、実質的には定数として扱われます[3]。

考察　類似のキーワードとして、C++では constexpr が利用可能です。余裕のある人はその内容を調べなさい。

4.2.3　const、 #define、 enum の使い分け

上に述べたように、

```
const int ROCK = 0;                    // 岩
```

は、変数 ROCK の値を変更できないことを表すものです。これとよく似た宣言として、マクロ定義を用いる以下の方法があります。

```
#define ROCK 0
```

これはプログラム中に現れる字句（単語） ROCK を単に 0 に置き換えることを表します。上の二つの宣言方法の違いは、前者では ROCK には int 型という型情報が付いていますが、後者にはそれがありません。コンパイラは型情報を基にして、より詳細なプログラム解析を行うことができますから、その意味では前者の書き方が強く推奨されます。

さらに ROCK、 EMPTY、 GOAL は列挙型を用いて以下のように宣言することもできます。

```
enum CellType {
    ROCK = 0,
    EMPTY = 1,
    GOAL = 2,
};
```

あるいは

[3] const の話題の立ち入って議論するときに欠かせないのが const でポインタを修飾する場合です。たとえば const int * p; はポインタの指すデータ実体の内容が不変であることを表し、 int * const p; は、ポインタが指すアドレスが不変であることを表します。この講義ではこれらは用いません。

```
enum CellType {
    ROCK,
    EMPTY,
    GOAL,
};
```

この章のプログラムの場合、この列挙型を用いる方法が最も適切なのですが、不慣れな人が多いと考え、最初の const を用いる方法を採用しました。

4.2.4　グローバル変数の注意

グローバル変数は便利ですが、多用すべきではありません[4]。どこからでも値を変更できるため、潜在的なバグの危険性が高まります。その意味では図 4.2 は良いプログラムではありません。

4.3　トップレベルの実装

図 4.3 はゲームのトップレベルを統括するプログラム部分です。以下、各関数について解説します。

4.3.1　全体初期化関数 initAll()

マップ、プレイヤーの初期化関数を呼び出します。

4.3.2　全体状態表示関数 printAll()

マップ、プレイヤーの表示関数を呼び出し、最後に改行を 1 個追加表示します。全ての表示はコンソール（標準出力）です。

4.3.3　メインループ mainLoop()

ゲームの中核部分です。まずゲームの初期状態を表示し、その後、ターンの実行を繰り返します。

[4]たとえば、 https://kiyou-binbou.hatenablog.jp/entry/2013/03/19/グローバル変数を撲滅したい理由

```
int main(int argc, const char * argv[]) {
    initAll();
    mainLoop();
    return 0;
}

void initAll() {                    // ゲーム全体の初期化
    mapInit();                        // マップの初期化
    playerInit();                     // プレイヤーの初期化
}

void printAll() {                   // ゲーム状態の表示
    mapPrint();                       // マップの表示
    playerPrint();                    // プレイヤーの状態の表示
    cout << endl;                     // ターンの区切りのための改行
}

void mainLoop() {                   // ターンの繰り返し実行
    printAll();                       // ゲーム状態の表示
    while (true) {                    // 無限に繰り返す
        turn();                       // 1 回のターン実行
    }
}

void turn() {                       // 1 回のターン実行
    playerAction();                   // プレイヤーのアクション
    printAll();                       // ゲーム状態の表示
}

char getKeyChar() {                 // キーボードからの 1 文字入力
    char c;
    cout << "? " << flush;            // プロンプトの表示
    cin >> c;                         // 1 文字入力
    return c;
}
```

図 4.3: 出発点のダンジョンゲームのプログラム（トップレベル）

4.3.4　ひとつのターンの実行 turn()

このゲームはプレイヤーのアクションを交互に行うターン制です。ひとつのターンはプレイヤーのアクションとアクションの後のゲーム状態の表示です。プレイヤーのアクションは、キーボードからの 1 文字入力で行います。入力されるまでプログラムの実行はストップしますから、プレイヤーは長考が可能です。

4.3.5　キーボード入力 getKeyChar()

プレイヤーはキーボードから 1 文字を入力し、さらにリターンキーを打鍵せねばならない仕様です。リターンキーを打鍵する分、ゲームの操作性は悪いのですが、リターンキーを打鍵しないで済

```
void mapInit() {                          // マップの初期化
    for (int x = 0; x < MAP_XSIZE; x++) {
        for (int y = 0; y < MAP_YSIZE; y++) {
            if ((x == 0)||(x == MAP_XSIZE-1)|| // 左右端の岩の位置の判定
               (y == 0)||(y == MAP_YSIZE-1)||  // 上下端の岩の位置の判定
               ((abs(x-MAP_XSIZE/2) <= 1)&&    // x 値が中央値から 1 以内の距離
                (abs(y-MAP_YSIZE/2) <= 1))) {  // y 値が中央値から 1 以内の距離

                mapData[x][y] = ROCK;             // 岩
            }
            else { mapData[x][y] = EMPTY; }       // 何もない床
        }
    }
    mapData[MAP_XSIZE-2][MAP_YSIZE-2] = GOAL;     // ゴール
}

void mapPrint() {                         // マップの表示
    for (int y = 0; y < MAP_YSIZE; y++) {
        for (int x = 0; x < MAP_XSIZE; x++) {
            if (playerCellPrint(x,y)) { continue; } // プレイヤーを表示
            mapCellPrint(x,y);                      // セルの状態を表示
        }
        cout << endl;
    }
}

void mapCellPrint(int x, int y) {         // セルの表示
    int e = mapData[x][y];
    if (e == ROCK)       { cout << '#' << ' '; }
    else if (e == EMPTY) { cout << '.' << ' '; }
    else if (e == GOAL)  { cout << 'G' << ' '; }
    else                 { cout << 'E' << ' '; }
}
```

図 4.4: 出発点のダンジョンゲームのプログラム（マップ）

むプログラムは意外に複雑[5]な上に「オブジェクト指向」と直接、関係しない事項のため実装をサボります。

この関数は playerAction() から呼ばれます。

4.4　マップの実装

マップの実装は図 4.4（43 ページ）です。マップデータは 2 次元 int 型配列のグローバル変数 mapData （図 4.2（39 ページ）参照）に保持されます。以下、各関数の説明です。

[5]スピンウエイト（spin wait）を行いて実装可能です。興味のある人は「ゲーム & キーボード入力」などで検索すると種々ヒットします。

4.4.1　マップの初期化 mapInit()

配列への値の初期設定では、マップに岩（あるいは岩の壁）を設定する以下の箇所がやや煩雑です。

```
if ((x == 0)||(x == MAP_XSIZE-1)||  // 左右端の岩の位置の判定
   (y == 0)||(y == MAP_YSIZE-1)||  // 上下端の岩の位置の判定
   ((abs(x-MAP_XSIZE/2) <= 1)&&    // x 値が中央値から 1 以内の距離
    (abs(y-MAP_YSIZE/2) <= 1))) {  // y 値が中央値から 1 以内の距離

    mapData[x][y] = ROCK;
```

最初の 2 行がマップ全体を囲む岩の場所の判定です。次の 2 行がマップ中央の岩の塊の判定です[6]。

4.4.2　マップの表示 mapPrint()

ここでいうマップの表示とは、各セルの状態（岩なのか、プレイヤーが居るのか）の表示のことです。図 1.1（4 ページ）がその表示例です。x、 y による二重ループで処理を繰り返します。

ループ本体の

```
if (playerCellPrint(x,y)) { continue; } // プレイヤーを表示
mapCellPrint(x,y);                      // セルの状態を表示
```

の 1 行目は

- もし座標 (x,y) にプレイヤーが居るならば、プレイヤーを表す文字 ‘@’ を表示し、2 行目を実行しない

ことを表します。continue はループ本体のそれ以降の処理を行わないことを表す C/C++/Java のキーワードです。実際のプレイヤーの表示（‘@’ と空白文字の表示）は関数 playerCellPrint() 内で行います。次に 2 行目の mapCellPrint(x,y); は

- 座標 (x,y) のセルそのものの状態を対応する文字で表示する

ことを表します。実際のセルの表示（‘#’、‘.’ または ‘G’ の表示）は関数 cellPrint() 内で行います。

[6]マップの中央の岩は、それが無いとゲームとしての面白みがほとんど無くなるため、あえて置いています。

4.4.3　セルの状態の表示 mapCellPrint()

この関数では、もし座標 (x,y) のセルの値が ROCK ならば '#' と空白文字を表示し、EMPTY ならば '.' と空白文字を表示し、もし GOAL ならば 'G' と空白文字を表示します。さもなくばエラーを表す 'E' と空白文字を表示します。マップの表示に 'E' が現れたならば、プログラムに何らかのバグがあることを意味します。

4.5　プレイヤーの実装

プレイヤーの実装は図 4.5（46 ページ）です。プレイヤーの状態は当面、三つのグローバル変数 playerX、 playerY、 playerHp（図 4.2（39 ページ）参照）に保持します。以下、各関数の説明です。

4.5.1　プレイヤーの初期化 playerInit()

プレイヤーの初期位置は $(1, 1)$ です。座標原点 $(0, 0)$ ではない（そこにはマップの左上の岩がある）ことに注意してください。プレイヤーの初期 HP は PLAYER_INIT_HP です。

4.5.2　プレイヤーの状態表示 playerPrint()

プレイヤーの位置と HP を数値でコンソールに表示します。

4.5.3　プレイヤーのマップ表示 playerCellPrint()

引数で与えられた座標値とプレイヤーの座標値が等しいならば、'@' と空白文字を表示し、 true を戻します。true が戻されたならば、同じ座標値にセルの内容が表示されることはありません（4.4.2 項（44 ページ）参照）。座標値が等しくないならば、false を戻します。

4.5.4　プレイヤーの移動 playerMoveTo()

プレイヤーは、現在の位置 (playerX, playerY) に引数の方向ベクトル (dx, dy) を加えた位置 (targetX, targetY) へ移動することを試みます。移動できる条件は、目的の位置に岩がないことです。もし移動できたならば true を戻します。さもなくば false を戻します。

戻り値の真偽値はこの章のプログラムでは用いません。が、後章で攻撃を導入したときに必要になるため、先行してこの章で戻り値を付けておくことにします。

```
void playerInit() {                          // プレイヤーの初期化
    playerX = 1; playerY = 1;
    playerHp = PLAYER_INIT_HP;
}

void playerPrint() {                         // プレイヤーの状態の表示
    cout << "Player : (" << playerX << "," << playerY << ") "
    << playerHp << endl;
}

bool playerCellPrint(int x, int y) {         // プレイヤーのセル上への表示
    if ((x == playerX)&&(y == playerY)) {
        cout << '@' << ' ';                  // @と空白文字の表示
        return true;
    }
    return false;
}

bool playerMoveTo(int dx, int dy) {          // プレイヤーの移動
    int targetX = playerX+dx, targetY = playerY+dy;

    if (mapData[targetX][targetY] == ROCK) { return false; }

    playerX = targetX; playerY = targetY;    // 座標値を更新
    return true;
}

void playerAction() {                        // プレイヤーのアクション
    int dx, dy;
    switch (getKeyChar()) {
        case 'z': dx = -1; dy = +1; break;   // 移動ベクトルの設定
        case 'x': dx =  0; dy = +1; break;
        case 'c': dx = +1; dy = +1; break;
        case 'a': dx = -1; dy =  0; break;
        case 'd': dx = +1; dy =  0; break;
        case 'q': dx = -1; dy = -1; break;
        case 'w': dx =  0; dy = -1; break;
        case 'e': dx = +1; dy = -1; break;
        default:  dx =  0; dy =  0;
    }

    playerMoveTo(dx,dy);       // 移動処理、ここでは関数の戻り値は用いない
}
```

図 4.5: 出発点のダンジョンゲームのプログラム（プレイヤー）

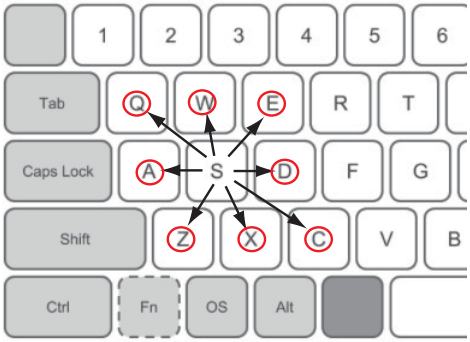


図 4.6: キーとプレイヤーの移動の関係：‘s’ の周囲の赤丸で囲った英子文字で 8 方向へ移動する。‘s’ キーは入力してもよいが、移動の効果を持たない。なお、入力確定にはリターンキーも押す必要がある。

4.5.5　プレイヤーのアクション playerAction()

getKeyChar() で求めた文字から相対位置 (dx, dy) を決定します。図 4.6（47 ページ）は、キーボード上のキーと相対位置の関係を図示したものです。そして、その相対位置へ移動できるならば移動します。できない場合には何も起きません。

4.6　プログラムの実行例

上に紹介したプログラム 図 4.2（39 ページ）、図 4.3（42 ページ）、図 4.4（43 ページ）、図 4.5（46 ページ）は、この順番にひとつのファイルにまとめてコンパイルすることができます。

プログラムの実行は、たとえば以下のように進みます。まだモンスターも勝ち負け判定もプログラム化していないため、現状では単にプレイヤーがマップ上を動くのみです。

```
# # # # # # # # # # # # #
# @ . . . . . . . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
# . . . . # # # . . . . #
# . . . . # # # . . . . #
# . . . . # # # . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . G #
# # # # # # # # # # # # #
Player : (1,1) 100

? d                          <---- テンキーから右方向（=d）を選択
# # # # # # # # # # # # #
# . @ . . . . . . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
```

```
# . . . . # # # . . . . #
# . . . . # # # . . . . #
# . . . . # # # . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . G #
# # # # # # # # # # # # # #
Player : (2,1) 100

? x                          <---- テンキーから下方向（=x）を選択
# # # # # # # # # # # # # #
# . . . . . . . . . . . #
# . @ . . . . . . . . . #
# . . . . . . . . . . . #
# . . . . # # # . . . . #
# . . . . # # # . . . . #
# . . . . # # # . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . G #
# # # # # # # # # # # # # #
Player : (2,2) 100

? z                          <---- テンキーから左下方向（=z）を選択
# # # # # # # # # # # # # #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
# @ . . . . . . . . . . #
# . . . . # # # . . . . #
# . . . . # # # . . . . #
# . . . . # # # . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . G #
# # # # # # # # # # # # # #
Player : (1,3) 100

? c                          <---- テンキーから右下方向（=c）を選択
# # # # # # # # # # # # # #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
# . @ . . # # # . . . . #
# . . . . # # # . . . . #
# . . . . # # # . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . G #
# # # # # # # # # # # # # #
Player : (2,4) 100

以下、省略
```

4.7　付録：この章の全プログラムリスト

プログラム読解の便利のために、以下にこの章で解説したプログラムのリストを載せます。このプログラムは一つのファイルとしてコンパイル、リンク、実行可能です。

```
//宣言部
#include <iostream>
using namespace std;

// 定数宣言
const int ROCK = 0;                 // 岩
const int EMPTY = 1;                // 床に何も居ないことを表す
const int GOAL = 2;                 // ゴール

const int MAP_XSIZE = 13;           // マップの横方向のセルの数
const int MAP_YSIZE = 11;           // マップの縦方向のセルの数

const int PLAYER_INIT_HP = 100;     // プレイヤーの初期 HP

//型宣言

//プロトタイプ宣言

void initAll();                         // ゲーム全体の初期化
void printAll();                        // ゲーム状態の表示
void mainLoop();                        // ターンの繰り返し実行
void turn();                            // 1 回のターン実行
char getKeyChar();                      // キーボードからの 1 文字入力

void mapInit();                         // マップの初期化
void mapPrint();                        // マップの表示
void mapCellPrint(int x, int y);        // セルの表示

void playerInit();                      // プレイヤーの初期化
void playerPrint();                     // プレイヤーの状態の表示
bool playerCellPrint(int x, int y);     // プレイヤーのセル上への表示
bool playerMoveTo(int dx, int dy);      // プレイヤーの移動
void playerAction();                    // プレイヤーのアクション

//グローバル変数宣言
int mapData[MAP_XSIZE][MAP_YSIZE];      // マップ

int playerX, playerY;                   // プレイヤーの現在の位置
int playerHp;                           // プレイヤーの現在の HP
```

```
//トップレベルの実装
int main(int argc, const char * argv[]) {
    initAll();
    mainLoop();
    return 0;
}

void initAll() {                    // ゲーム全体の初期化
    mapInit();                        // マップの初期化
    playerInit();                     // プレイヤーの初期化
}

void printAll() {                   // ゲーム状態の表示
    mapPrint();                       // マップの表示
    playerPrint();                    // プレイヤーの状態の表示
    cout << endl;                     // ターンの区切りのための改行
}

void mainLoop() {                   // ターンの繰り返し実行
    printAll();                       // ゲーム状態の表示
    while (true) {                    // 無限に繰り返す
        turn();                       // 1 回のターン実行
    }
}

void turn() {                       // 1 回のターン実行
    playerAction();                   // プレイヤーのアクション
    printAll();                       // ゲーム状態の表示
}

char getKeyChar() {                 // キーボードからの 1 文字入力
    char c;
    cout << "? " << flush;            // プロンプトの表示
    cin >> c;                         // 1 文字入力
    return c;
}
```

```
//マップの実装
void mapInit() {                          // マップの初期化
    for (int x = 0; x < MAP_XSIZE; x++) {
        for (int y = 0; y < MAP_YSIZE; y++) {
            if ((x == 0)||(x == MAP_XSIZE-1)|| // 左右端の岩の位置の判定
               (y == 0)||(y == MAP_YSIZE-1)||  // 上下端の岩の位置の判定
               ((abs(x-MAP_XSIZE/2) <= 1)&&    // x 値が中央値から 1 以内の距離
                (abs(y-MAP_YSIZE/2) <= 1))) {  // y 値が中央値から 1 以内の距離

                mapData[x][y] = ROCK;              // 岩
            }
            else { mapData[x][y] = EMPTY; }        // 何もない床
        }
    }
    mapData[MAP_XSIZE-2][MAP_YSIZE-2] = GOAL;      // ゴール
}

void mapPrint() {                         // マップの表示
    for (int y = 0; y < MAP_YSIZE; y++) {
        for (int x = 0; x < MAP_XSIZE; x++) {
            if (playerCellPrint(x,y)) { continue; } // プレイヤーを表示
            mapCellPrint(x,y);                        // セルの状態を表示
        }
        cout << endl;
    }
}

void mapCellPrint(int x, int y) {         // セルの表示
    int e = mapData[x][y];
    if (e == ROCK)       { cout << '#' << ' '; }
    else if (e == EMPTY) { cout << '.' << ' '; }
    else if (e == GOAL)  { cout << 'G' << ' '; }
    else                 { cout << 'E' << ' '; }
}
```

```
//プレイヤーの実装
void playerInit() {                          // プレイヤーの初期化
    playerX = 1; playerY = 1;
    playerHp = PLAYER_INIT_HP;
}

void playerPrint() {                         // プレイヤーの状態の表示
    cout << "Player : (" << playerX << "," << playerY << ") "
    << playerHp << endl;
}

bool playerCellPrint(int x, int y) {         // プレイヤーのセル上への表示
    if ((x == playerX)&&(y == playerY)) {
        cout << '@' << ' ';                  // @と空白文字の表示
        return true;
    }
    return false;
}

bool playerMoveTo(int dx, int dy) {          // プレイヤーの移動
    int targetX = playerX+dx, targetY = playerY+dy;

    if (mapData[targetX][targetY] == ROCK) { return false; }

    playerX = targetX; playerY = targetY;    // 座標値を更新
    return true;
}

void playerAction() {                        // プレイヤーのアクション
    int dx, dy;
    switch (getKeyChar()) {
        case 'z': dx = -1; dy = +1; break;   // 移動ベクトルの設定
        case 'x': dx =  0; dy = +1; break;
        case 'c': dx = +1; dy = +1; break;
        case 'a': dx = -1; dy =  0; break;
        case 'd': dx = +1; dy =  0; break;
        case 'q': dx = -1; dy = -1; break;
        case 'w': dx =  0; dy = -1; break;
        case 'e': dx = +1; dy = -1; break;
        default:  dx =  0; dy =  0;
    }

    playerMoveTo(dx,dy);       // 移動処理、ここでは関数の戻り値は用いない
}
```

1

1

第5章　プレイヤーのデータを集約する：構造体の導入

前章のプログラムではプレイヤーの状態を保持する三つの変数 playerX、playerY、PlayerHp はグローバル領域にバラバラに保持されており、まとまったデータ実体とはなっていません。そのせいでプレイヤーの全体像が把握しにくく、プレイヤーの保守、管理に不便です。これをこの章では**構造体**（structure）にまとめます。

構造体は、複数の関連するデータをひとつの高次のデータ型構造の中に集約し、プログラムの保守管理を高める仕組みです。データを集約する高次データ型構造としては配列もあります。しかし、配列は同じデータ型の多数のデータの並びを扱うものです。それに対して、構造体は様々なデータ型のデータを混在できる機能を持ち、より柔軟性の高い高次構造と言えます。

この章ではプレイヤーを座標値および HP 値を持つ構造体に改造します。次章ではさらにプレイヤーを操作する関数を集約する改造を行います。このようにプログラムの本質的な処理内容を変えず、プログラムの構文構造を変更する処理を**リファクタリング**（refactoring）と呼びます。最近ではリファクタリングをコマンドひとつで自動で行うツールもありますが、この講義テキストではその意義を確認しつつ、手動でのリファクタリングを行います。

5.1　簡単な例題

構造体を簡単に復習します。以下のようなプログラム片があるとき、

```
...
int a = 10;
float b = 20.0;
...
中略
...
cout << a+b << endl;
...
```

もし a、 b が互いに強く関連するデータであって、ひとつの高次構造としてまとめる方が自然と判断されるならば、このプログラムは次の構造体:

```
struct StructA {
    int   a;
    float b;
};
```

を用いて以下のように変形できます。なお、構造体の内部の変数 a、 b を一般に **メンバー変数** (member variable)、データメンバー、データフィールドなどと呼びます。

```
    ...
    StructA data;
    data.a = 10;
    data.b = 20.0;
    ...
    中略
    ...
    cout << data.a + data.b << endl;
    ...
```

この変更例では逆にプログラムが複雑になっただけと思うかもしれません。が、構造体と用いることで、二つのデータがまとまりであることをプログラマに強く意識させます。また、データをまとめて処理できる（たとえば、関数の引数として、あるいは関数の戻り値として一括して受け渡すことができる、一括代入ができるなど）というプログラミング上の利点があり、プログラムの可読性が高まります。

5.2　構造体導入の概要

ダンジョンゲームで用いる構造体について検討します。

前章の図 4.3（42 ページ）では、プレイヤーのデータを

```
int playerX, playerY;                    // プレイヤーの現在の位置
int playerHp;                            // プレイヤーの現在の HP
```

の三つのグローバル変数で表現しましたが、この章では

```
struct Player {                          // プレイヤーの構造体型
    int x, y;
    int hp;
};

Player player;                           // プレイヤーの構造体データ
```

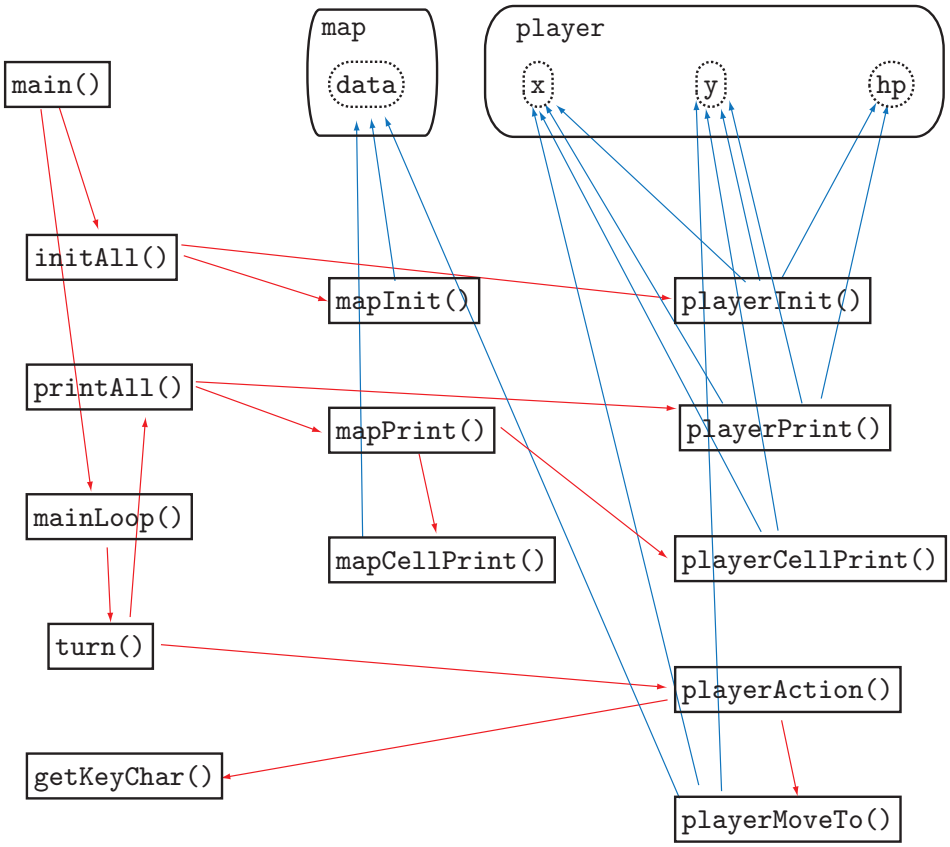


図 5.1: この章のプログラムの構造：長方形が関数、角丸長方形がデータ、赤い矢印が関数の呼び出し関係、青い矢印がデータのアクセスを表します。

と構造体データ型を定義し、そのデータ型のグローバル変数としてプレイヤーを表現します。前章の playerX は、 player.x（ player に所属するデータメンバー x）として参照することになります。構造体の導入によって三つのデータがひとつのまとまりとして扱われる訳です。とは言え、この章では Player 型のデータを 1 個しか使用していないため、この時点では構造体の有り難みがあまり感じられないかもしれません。

ついでにマップも構造体に変更します。この時点では構造体に格上げする必要性は低いのですが、次章を考慮し、この章であらかじめ構造体に変更します。

図 5.1（57 ページ）はこの章のプログラムの構造です。

5.3　宣言部の変更

図 5.2（58 ページ）は、図 4.2（39 ページ）を構造体を用いるように変更した宣言部です。

```
//定数宣言
** 前章から変更なし **

//型宣言
struct Map {                                // マップの構造体型
    int data[MAP_XSIZE][MAP_YSIZE];
};

struct Player {                             // プレイヤーの構造体型
    int x, y;
    int hp;
};

//プロトタイプ宣言
** 前章から変更なし **

//グローバル変数
Map map;                                    // マップの構造体データ
Player player;                              // プレイヤーの構造体データ
```

図 5.2: 構造体を導入したダンジョンゲームのプログラム（各種宣言）：図 4.2（39 ページ）参照

既に述べたように、プレイヤーのデータを構造体に変更しました。

マップは、図 4.2 では、単に

```
int mapData[MAP_XSIZE][MAP_YSIZE];
```

と宣言されていましたが、これを

```
struct Map {                                // マップの構造体型
    int data[MAP_XSIZE][MAP_YSIZE];
};

Map map;                                   // プレイヤーの構造体データ
```

へ変更します。

注意　構造体を筆記試験問題に出すと、宣言の末尾のセミコロンを忘れる学生が多いようです。実際、このセミコロンはほとんど意味をなさないため、実際のプログラミングの作業においてもしばしば忘れてコンパイルエラーを出してしまいます。このセミコロンが必要な理由は、実は C++の前身の C 言語では構造体の宣言と同時に変数宣言もできる仕様になっているためです。以下がその例です。

```
struct Player {                          // プレイヤーの構造体型
    int x, y;
    int hp;
} player;                                // Player 型変数の宣言
```

この場合、変数名 player の後にはセミコロンが付く方が自然です。これはちょうど

```
int i;
```

のセミコロンが必要であることと同じ理由です[1]。C++が C 言語の仕様を包含する限り、このセミコロンを省略できそうにありません。C 言語の仕様を引きずらない Java の場合には、こういう構文を認めないため、もはやセミコロンは不要です（Java のクラス宣言にセミコロンを付けてもコンパイルエラーにはなりませんが)。

5.4　トップレベルの実装の変更

プログラムの上位の処理を担う main 関数、 mainLoop()、 turn() などは全く変更ありません。というのも、この章の変更は実装の下層のデータ型の変更だからです。

5.5　マップの実装の変更

マップを操作する関数群は図 5.3（60 ページ）の通りです。ここでは、図 4.4（43 ページ）の mapData を map.data に置き換えるだけの変更です。

5.6　プレイヤーの実装の変更

プレイヤーを操作する関数群は図 5.4（61 ページ）の通りです。ここでは図 4.5（46 ページ）の

map、 playerX、 playerY、 playerHp

を、それぞれ

map.data、 player.x、 player.y、 player.hp

に置き換える変更のみです。

[1] プログラムを眺めてみると、このセミコロンがなくても i が変数であるとの認識はできるはずと気づくのですが、C 言語が開発された 1970 年代の非力なコンピュータによる構文解析処理ではセミコロンがあった方が解析の効率がよいため、当時はこれに疑問を挟む技術者はいませんでした。最近の新しいプログラミング言語ではこのセミコロンのような無駄な区切り記号は用いない仕様が増えています。

```
void mapInit() {
    for (int x = 0; x < MAP_XSIZE; x++) {
        for (int y = 0; y < MAP_YSIZE; y++) {
            if ((x == 0)||(x == MAP_XSIZE-1)||
               (y == 0)||(y == MAP_YSIZE-1)||
               ((abs(x-MAP_XSIZE/2) <= 1)&&
                (abs(y-MAP_YSIZE/2) <= 1))) {

                map.data[x][y] = ROCK;              // 変更
            }
            else { map.data[x][y] = EMPTY; }        // 変更
        }
    }
    map.data[MAP_XSIZE-2][MAP_YSIZE-2] = GOAL;      // 変更
}

void mapPrint() { ** 前章から変更なし ** }

void mapCellPrint(int x, int y) {
    int e = map.data[x][y];                         // 変更

    ** 以下、前章から変更なし **
}
```

図 5.3: 構造体を導入したダンジョンゲームのプログラム（マップ）：図 4.4（43 ページ）参照

分かるようにプログラムの変更は全く機械的です。もちろんゲームの実行は前章から変わりありません。

```
void playerInit() {
    player.x = 1; player.y = 1;                                  // 変更
    player.hp = PLAYER_INIT_HP;                                  // 変更
}

void playerPrint() {
    cout << "Player : (" << player.x << "," << player.y << ") " //変更
         << player.hp << endl;                                   //変更
}

bool playerCellPrint(int x, int y) {
    if ((x == player.x)&&(y == player.y)) {                      //変更
        cout << '@' << ' ';
        return true;
    }
    return false;
}

bool playerMoveTo(int dx, int dy) {
    int targetX = player.x+dx, targetY = player.y+dy;            //変更

    if (map.data[targetX][targetY] == ROCK) { return false; }   //変更

    player.x = targetX; player.y = targetY;                      //変更
    return true;
}

void playerAction() { ** 前章から変更なし ** }
```

図 5.4: 構造体を導入したダンジョンゲームのプログラム（プレイヤー）：図 4.5（46 ページ）参照

5.7　付録：この章の全プログラムリスト

```
//宣言部
#include <iostream>
using namespace std;

//定数宣言
const int ROCK = 0;
const int EMPTY = 1;
const int GOAL = 2;

const int MAP_XSIZE = 13;
const int MAP_YSIZE = 11;

const int PLAYER_INIT_HP = 100;

//型宣言
struct Map {                                // マップの構造体型
    int data[MAP_XSIZE][MAP_YSIZE];
};

struct Player {                             // プレイヤーの構造体型
    int x, y;
    int hp;
};

//プロトタイプ宣言
void initAll();
void printAll();
void turn();
void mainLoop();
char getKeyChar();

void mapInit();
void mapPrint();
void mapCellPrint(int x, int y);

void playerInit();
void playerPrint();
bool playerCellPrint(int x, int y);
bool playerMoveTo(int dx, int dy);
void playerAction();

//グローバル変数
Map map;                                    // マップの構造体データ
Player player;                              // プレイヤーの構造体データ
```

```
//トップレベルの実装
int main(int argc, const char * argv[]) {
    initAll();
    mainLoop();
    return 0;
}

void initAll() {
    mapInit();
    playerInit();
}

void printAll() {
    mapPrint();
    playerPrint();
    cout << endl;
}

void mainLoop() {
    printAll();
    while (true) {
        turn();
    }
}

void turn() {
    playerAction();
    printAll();
}

char getKeyChar() {
    char c;
    cout << "? " << flush;
    cin >> c;
    return c;
}
```

```
//マップの実装
void mapInit() {
    for (int x = 0; x < MAP_XSIZE; x++) {
        for (int y = 0; y < MAP_YSIZE; y++) {
            if ((x == 0)||(x == MAP_XSIZE-1)||
               (y == 0)||(y == MAP_YSIZE-1)||
               ((abs(x-MAP_XSIZE/2) <= 1)&&
                (abs(y-MAP_YSIZE/2) <= 1))) {

                map.data[x][y] = ROCK;             // 変更
            }
            else { map.data[x][y] = EMPTY; }       // 変更
        }
    }
    map.data[MAP_XSIZE-2][MAP_YSIZE-2] = GOAL;     // 変更
}

void mapPrint() {
    for (int y = 0; y < MAP_YSIZE; y++) {
        for (int x = 0; x < MAP_XSIZE; x++) {
            if (playerCellPrint(x,y)) { continue; }
            mapCellPrint(x,y);
        }
        cout << endl;
    }
}

void mapCellPrint(int x, int y) {
    int e = map.data[x][y];                        // 変更
    if (e == ROCK) { cout << '#' << ' '; }
    else if (e == EMPTY) { cout << '.' << ' '; }
    else if (e == GOAL) { cout << 'G' << ' '; }
    else { cout << 'E' << ' '; }
}
```

```
//プレイヤーの実装
void playerInit() {
    player.x = 1; player.y = 1;                                  // 変更
    player.hp = PLAYER_INIT_HP;                                  // 変更
}

void playerPrint() {
    cout << "Player : (" << player.x << "," << player.y << ") " //変更
         << player.hp << endl;                                   //変更
}

bool playerCellPrint(int x, int y) {
    if ((x == player.x)&&(y == player.y)) {                      //変更
        cout << '@' << ' ';
        return true;
    }
    return false;
}

bool playerMoveTo(int dx, int dy) {
    int targetX = player.x+dx, targetY = player.y+dy;           //変更

    if (map.data[targetX][targetY] == ROCK) { return false; }   //変更

    player.x = targetX; player.y = targetY;                     //変更
    return true;
}

void playerAction() {
    int dx, dy;
    switch (getKeyChar()) {
        case 'z': dx = -1; dy = +1; break;
        case 'x': dx =  0; dy = +1; break;
        case 'c': dx = +1; dy = +1; break;
        case 'a': dx = -1; dy =  0; break;
        case 'd': dx = +1; dy =  0; break;
        case 'q': dx = -1; dy = -1; break;
        case 'w': dx =  0; dy = -1; break;
        case 'e': dx = +1; dy = -1; break;
        default:  dx =  0; dy =  0;
    }

    playerMoveTo(dx,dy);
}
```

1

1

第6章　プレイヤーの処理をまとめる：メンバー関数の導入、クラスへの格上げ

前章までのプログラムには 14 個の様々な関数が含まれますが、それら関数の中には**マップの操作に強く関連**しているもの、**プレイヤーの操作に強く関連**しているものがそれぞれあります。この章の目的は、そのような関数を構造体の中に移動させることです。これによって、単なる構造体が

$$\text{構造体} + \text{その操作関数} = \text{クラス}$$

へ改良されます。

6.1　簡単な例題

たとえば以下は構造体を用いるプログラムの例です。

```
struct StructA {
    int   a;
};
    ...
    StructA data;
    data.a = 10;
    data.a++;
    ...
```

このとき、もしメンバー変数 a に 10 を代入すること、そして a の値を 1 増やすことが構造体 StructA にとって基本演算であるならば、その処理を構造体の中に定義してしまうべきですし、それが可能です。以下がその改造例です。

```
struct StructA {
    int   a;
    void init() { a = 10; }
    void inc() { a++; }
};
    ...
    StructA data;
    data.init();
    data.inc();
    ...
```

上の構造体内の関数 `init()`、`inc()` を**メンバー関数**（member function）と呼びます。メンバー関数は構造体に関する操作を主な役割とします。メンバー変数のアクセスを

構造体データ. メンバー変数名

で行うように、メンバー関数の呼び出しは

構造体データ. メンバー関数名（実引数）

で行います。メンバー関数の中から様々な変数を参照するルールは 6.5.3 項（77 ページ）において詳細に述べます。

　さて、上の変更によって構造体の内部構造を外に見せることなくその操作が可能になります。これを**データ抽象**（data abstraction）と呼びます（データ抽象の詳しい議論は再度、14 章で行います）。データ抽象化された構造体は、もはや単なる構造体ではありません。オブジェクト指向ではこのような、メンバー変数とメンバー関数の両方を持つデータ型を**クラス**（class）と呼びます。そしてこの場合、上の宣言は以下のように書き換え可能です。

```
class StructA {            // struct を class へ置換
  public:                  // アクセス制御子
    int   a;
    void init() { a = 10; }
    void inc() { a++; }
};
```

なお、`public:`キーワードについては後ほど簡単に紹介し、詳細は 32 章で解説します。

　C++ではクラスの宣言から関数の実装を切り離すこともできます。

```
class StructA {
  public:
    int   a;
    void init();
    void inc();
};

void StructA::init() { a = 10; }
void StructA::inc() { a++; }
```

ここに、`::` はスコープ解決演算子と呼ばれますが、これについては 6.5.1 項（76 ページ）で詳しく述べます。

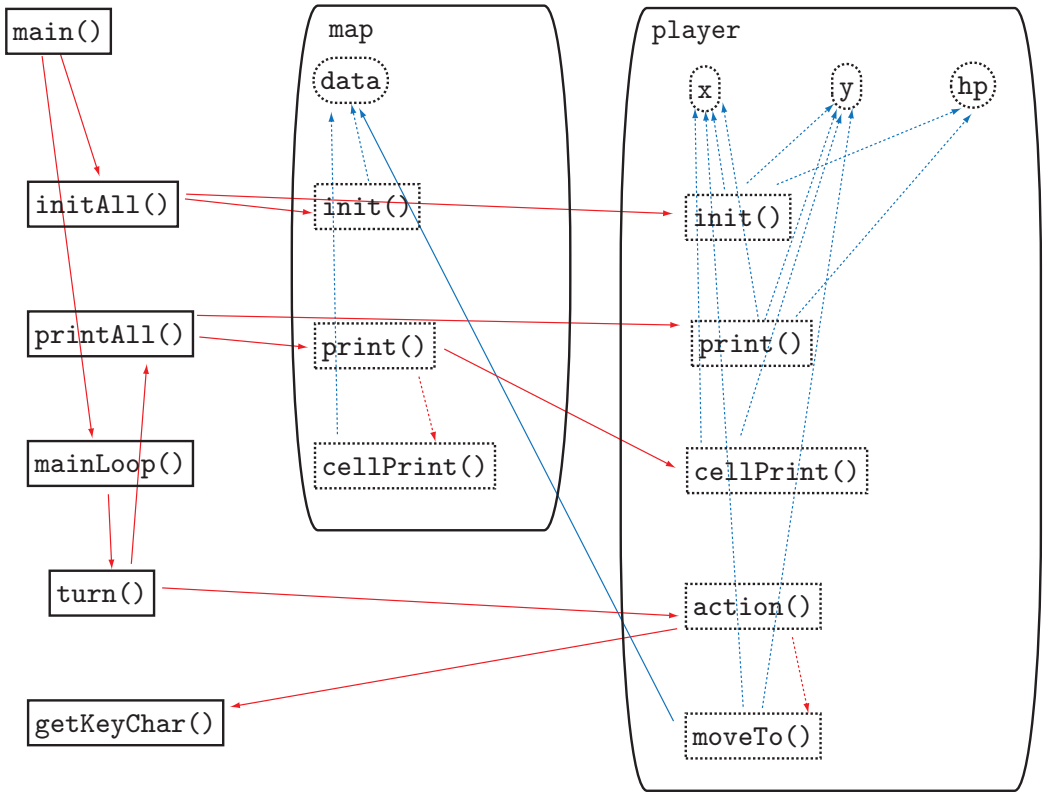


図 6.1: この章のプログラムの構造：長方形が関数、角丸長方形がデータ、赤い矢印が関数の呼び出し関係、青い矢印がデータのアクセスを表します。点線矢印は、クラス（構造体）内の呼び出し関係、アクセス関係を表します。青い実線が 1 本しかないことに注意。

6.2　クラスへの格上げの概要

図 6.1（70 ページ）にこの章のゲームプログラムの構造を図示しました。

図の中央のクラス（構造体） map の中に init() があります。これは前章の図 5.1（57 ページ）で mapInit() だった関数です。mapInit() はマップの初期化を行う関数でしたから、 map の中に移動されることは自然です。その他の関数も同様です。

次に関数からデータへのアクセスを表す青い矢印を見てください。点線の矢印はクラス内のアクセスを表します。この図では、13 本の青い矢印の中、12 本が点線の矢印です。ほとんどのデータアクセスはクラス内での局所的なアクセスに置き換えることができました。**局所性はプログラム理解を容易にします**。逆に言えば、局所性の低いプログラムではプログラマは様々なデータアクセスに目を配る必要があり、プログラム理解を妨げる大きな要因になります。

```
class Map {
  public:
    static const int ROCK = 0;            // Map 内へ移動
    static const int EMPTY = 1;           // 同上
    static const int GOAL = 2;            // 同上

    static const int XSIZE = 13;          // 同上
    static const int YSIZE = 11;          // 同上

    int data[XSIZE][YSIZE];

    void init();                          // Map 内へ移動
    void print();                         // 同上
    void cellPrint(int x, int y);         // 同上
};

class Player {
  public:
    static const int INIT_HP = 100;       // Map 内へ移動

    int x, y;
    int hp;

    void init();
    void print();                        // Map 内へ移動
    bool cellPrint(int x, int y);        // 同上
    bool moveTo(int dx, int dy);         // 同上
    void action();                       // 同上
};

//プロトタイプ宣言
void initAll();
void printAll();
void turn();
void mainLoop();
char getKeyChar();

//グローバル宣言
Map map;
Player player;
```

図 6.2: メンバー関数を導入したダンジョンゲームのプログラム（各種宣言）：図 5.2（58 ページ）参照

6.3　宣言部の変更

図 6.2（71 ページ）は宣言部です。前章までと異なり、クラス内の記述が増えています。

6.3.1　class キーワードの導入

C++では、構造体の宣言：

```
struct Player {
```

は、以下のクラス宣言

```
class Player {
  public:
```

の省略表記と見なされます。つまり構造体はクラスの特殊形式です。struct は C 言語のキーワードです。しかし、C++ではオブジェクト指向の観点からこれが class の特殊形として再解釈された訳です。

前章のまま struct を用い続けてもよいのですが、Java では struct は使えませんから、いずれ struct とは決別せねばなりません。そこでこのテキストでは構造体の使用は前章で終え、この章からクラスへ移行します。「構造体のデータ実体」などという言い方も止め、これ以降はクラスの定義に基づいて生成されるデータを**オブジェクト**（object）と呼ぶこととします。また、生成された個々のオブジェクトを特に**インスタンス**（instance、英和辞書的意味は「例、実例」です）と呼び、インスタンスを作ることを**インスタンス化**（instantiation）すると言います。

「クラス」と「オブジェクト」はしばしばあいまいに用いられますが、それが定義（クラス）なのか、個々の実体（オブジェクト）なのか、という観点で両者を区別してください。同様に、「オブジェクト」と「インスタンス」もあいまいに用いられますが、前者が総称であるのに対して後者は個別の実体を意味します。

6.3.2　public 修飾子

クラス宣言の 2 行目の public: はクラス内へのアクセスをプログラマが制御するためのアクセス修飾子（access modifier/specifier）であり、

- その行以降からクラス宣言の末尾まで（あるいは他のアクセス修飾子が現れるまで）に宣言された変数、関数は全てオブジェクトの外部から自由にアクセスできる

ことを表します。アクセス制御はオブジェクト指向技術の重要事項ですが、高度に洗練された技術でもあるため、このテキストでは他の重要事項の解説を終えるまで全ての変数、関数はアクセスフリーな状態（つまり、 public な状態）で進めていきます。

6.3.3　static 修飾子（その 1）

図 6.2 に static キーワードを用いた以下のような宣言部分があります。

```
class Map {
  public:
    static const int ROCK = 0;
```

static が付いた変数は、クラスのオブジェクトに所属するのではなくクラスそれ自体に所属します。C++では、このような変数を**静的メンバー変数**（static member variable）と呼びます。

さて、Map クラスそれ自身はプログラム空間[1]に唯一存在します。よって静的メンバー変数と宣言された ROCK もプログラム空間に唯ひとつだけ存在します。ひとつだけという点では静的メンバー変数はグローバル変数と同じです。両者の違いは、静的メンバー変数はクラスに所属し、グローバル変数は如何なるクラスにも属さない点です。後章に述べるように、クラスに所属する場合、アクセス制御が可能であり、野放図なアクセスを規制できます。その他の定数 EMPTY、...、 YSIZE も同様です。

なお、通常のメンバー変数は各オブジェクトの中に存在します。よって、オブジェクトが複数個あれば通常のメンバー変数はその個数分だけ存在し、それぞれ異なる値を持つことができます。

static については 9.2.4 項で追加解説します。

補足　静的メンバー関数の位置づけについて図 6.3（74 ページ）を用いて補足します。構造体の拡張として導入されたクラスは、そのクラスのオブジェクトの挙動を記載した設計図としての役割があります。しかし単なる設計図ではありません。プログラム空間に実体を持つ**メタクラス**（metaclass）のオブジェクトでもあり、静的メンバー変数はそのオブジェクトの中に保持されています。Map クラスも Player クラスもメタクラスからインスタンス化されたオブジェクトです。

[1] ここで言うプログラム空間は、プログラム実行時に利用されるメモリ領域を抽象化して名付けたものです。プログラム空間には様々な変数、関数、オブジェクトなどが存在します。メモリ領域は 1 次元のメモリに過ぎませんが、プログラム空間ではメモリの 1 次元性などを考慮する必要はありません。

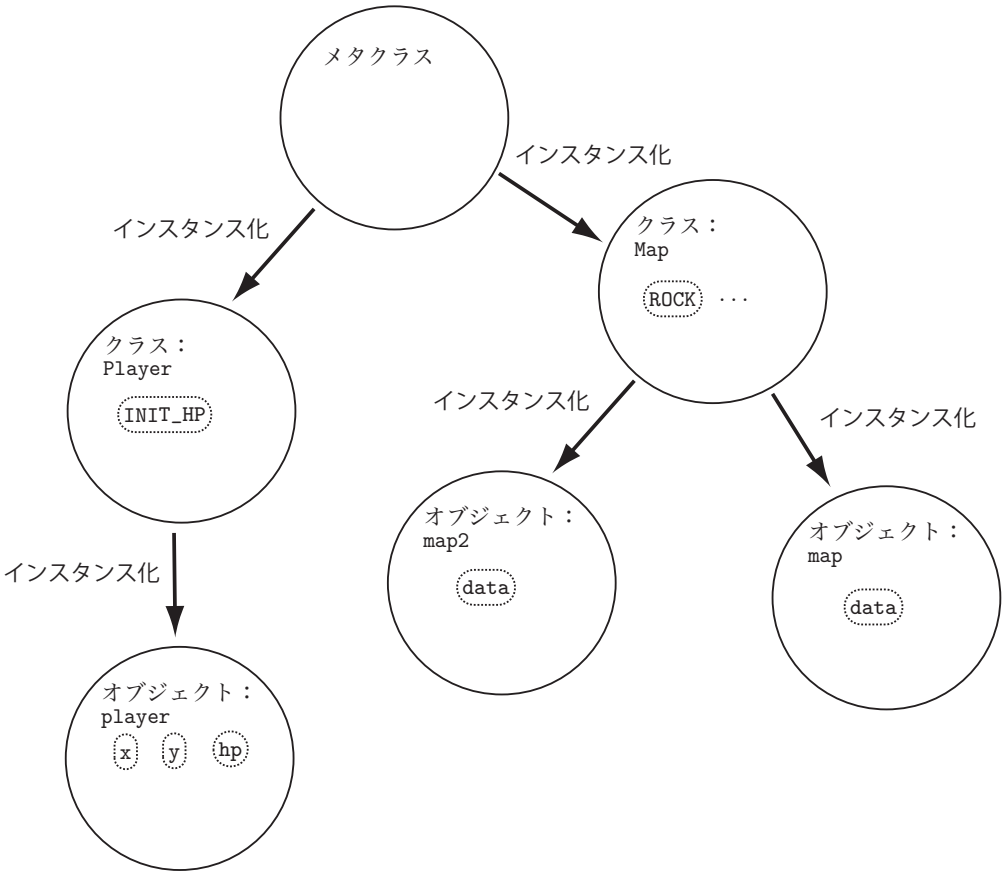


図 6.3: メタクラス、クラス、オブジェクトの生成関係：クラスはメタクラスからインスタンス化され、オブジェクトはクラスからインスタンス化されます。静的メンバー変数（たとえば Map クラスの ROCK）はクラス自体に所属する変数です。通常のメンバー変数（たとえば Map クラスの data）はオブジェクトに所属する変数です。

実はメタクラスもオブジェクトであり、メタクラスはメタクラスのオブジェクトという奇妙な位置づけになっています。奇妙ですが、この循環する関係によってクラス → オブジェクトの関係は閉じた構造を形作っています。

Map を例にします。インスタンス (オブジェクト実体） map は Map クラスからインスタンス化されたオブジェクトであり、通常のメンバー変数 data を持ちます。もし仮に Map クラスから map2 をインスタンス化したならば、map2 もやはり通常のメンバー変数 data を持ちます。メンバー変数はそれぞれのオブジェクトに属するものですから、もしプログラム空間にオブジェクトが複数個存在するならば、メンバー変数もその数だけ存在します。対して、クラス Map は静的メンバー変数 ROCK を持ちます。Map クラスはプログラム空間に唯一存在しますから、ROCK も唯一存在することになります。図 6.3 はそれを図示したものです。

```
int main(int argc, const char * argv[]) { ** 前章から変更なし ** }

void initAll() {
    map.init();                         // mapInit() から変更
    player.init();                      // playerInit() から変更
}

void printAll() {
    map.print();                        // mapPrint() から変更
    player.print();                     // playerPrint() から変更
    cout << endl;
}

void mainLoop() { ** 前章から変更なし ** }

void turn() {
    player.action();                    // playerAction() から変更
    printAll();
}

char getKeyChar() { ** 前章から変更なし ** }
```

図 6.4: メンバー関数を導入したダンジョンゲームのプログラム（トップレベル）：図 4.3（42 ページ）参照

ところで、通常、プログラマがメタクラスにアクセスすることはできません[2]が、概念上はメタクラスがオブジェクト指向の世界の総元締めとして鎮座しています。

6.4　トップレベルの実装の変更

図 6.4（75 ページ）がトップレベルの変更です。前章まで用いていた関数のいくつかがメンバー関数に置き換わっています。

6.5　マップクラスの実装

マップを操作する関数群は図 6.5（76 ページ）の通りです。関数本体では前章の図 5.3（60 ページ）の `map.data` を `data` に置き換え、関数呼び出す関数名をメンバー関数名に変更しています。

ここで、C++のメンバー関数の実装方法（インライン/非インライン実装）、メンバー変数のスコープについて以下、少し詳細に解説します。

[2] この辺に興味の湧いた人は「メタクラス」、あるいは「プログラミング & リフレクション」などで検索してみてください。プログラミングの新たな一面、可能性を知ることができます。

```
void Map::init() {                                  // 変更
    for (int x = 0; x < XSIZE; x++) {
        for (int y = 0; y < YSIZE; y++) {
            if ((x == 0)||(x == XSIZE-1)||
               (y == 0)||(y == YSIZE-1)||
               ((abs(x-XSIZE/2) <= 1)&&
                (abs(y-YSIZE/2) <= 1))) {

                data[x][y] = ROCK;                  // 変更
            }
            else { data[x][y] = EMPTY; }            // 変更
        }
    }
    data[XSIZE-2][YSIZE-2] = GOAL;                  // 変更
}

void Map::print() {                                 // 変更
    for (int y = 0; y < YSIZE; y++) {
        for (int x = 0; x < XSIZE; x++) {
            if (player.cellPrint(x,y)) { continue; }// 変更
            cellPrint(x,y);                          // 変更
        }
        cout << endl;
    }
}

void Map::cellPrint(int x, int y) {                 // 変更
    int e = data[x][y];                             // 変更
    if (e == ROCK)       { cout << '#' << ' '; }
    else if (e == EMPTY) { cout << '.' << ' '; }
    else if (e == GOAL)  { cout << 'G' << ' '; }
    else                 { cout << 'E' << ' '; }
}
```

図 6.5: メンバー関数を導入したダンジョンゲームのプログラム（マップ）：図 5.3（60 ページ）参照

6.5.1　メンバー関数の実装方法、スコープ解決演算子

図 6.5 の各関数本体の実装の冒頭部分：

```
void Map::init() {
```

に注意しましょう。`::` は**スコープ解決演算子**（scope resolution operator）と呼ばれ、 `::` の直後に現れる変数や関数が、`::` の直前に現れるクラスに所属することを明示するために用います。よって `Map::init()` は `Map` クラスのメンバー関数 `init()` のことを意味します。

図 6.2（71 ページ）の宣言部では、 `Map` クラスにメンバー関数 `init()` があることを宣言していますが、実装は行っていません。図 6.5 の `void Map::init() {...}` がその実装です。

6.5.2　インラインか非インラインか

C++ではクラスの宣言と実装を分けて記述するのが一般的です。よって図 6.2 と図 6.5 のように宣言と実装を分ける訳です。すでに 6.1 節（68 ページ）の簡単な例題で見たように、宣言部に実装を埋め込んでしまう書き方も可能です。これを**インライン記述**（inline description）と呼びます。たとえば、 `Map` クラスはインライン記述では以下の通りです。

```
class Map {
    public:
    ...
    void init() {  // クラス定義の中にそのまま関数の実装を記述している
        for (int x = 0; x < XSIZE; x++) {
            ...
        }
        data[XSIZE-2][YSIZE-2] = GOAL;
    }
    ...
};
```

これに対して、宣言と実装を分ける記述方を、**非インライン記述**（non-inline description）と呼びます。

一般的に C++では宣言と実装は分けて記述する非インライン記述が推奨されています。大規模プログラムを開発する場合、複数人で分担してプログラム開発プを行いますが、他のプログラマには自分の作ったプログラムの宣言部だけを見せ、実装を見せないためです[3]。

6.5.3　メンバー関数内のスコープ

スコープ（scope）は、変数や関数が参照可能になる範囲（プログラムの文面上の範囲）を意味します。日本語では多くの場合、有効範囲と訳しています。スコープのルールはプログラミング言語毎に微妙に異なることがあり、新しい言語を学ぶときには特に注意が必要です。

C 言語のスコープは比較的単純です。C++のクラスにおいても C 言語のスコープの考え方をほぼ素直に拡張すればよいだけですが、以下、図を用いて解説します。

図 6.6（78 ページ）は前章のスコープの一例です。通常の関数 `mapInit()` から構造体 `map` のメンバー `data` にアクセスする場合のスコープを図示したものです。`mapInit()` から `data` は直接参照できないため、 `map.data` と記述し、データを手繰り（たぐり）寄せる必要があります。

[3]納期に追われる切羽詰まったプログラマは、実装を見せられると、ついその実装を自分に都合のいいように改変したくなります。しかしそれをやってしまうと、後々、泥沼に嵌（はま）ってしまいます。

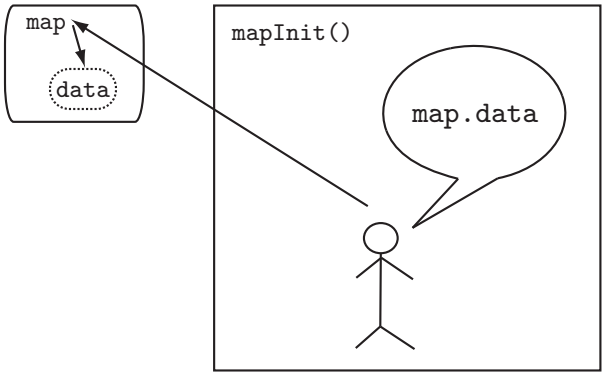


図 6.6: 外部から構造体のメンバー変数へのアクセスの例：外部関数から構造体のメンバー変数にアクセスするには 構造体名. メンバー名 で指定します。

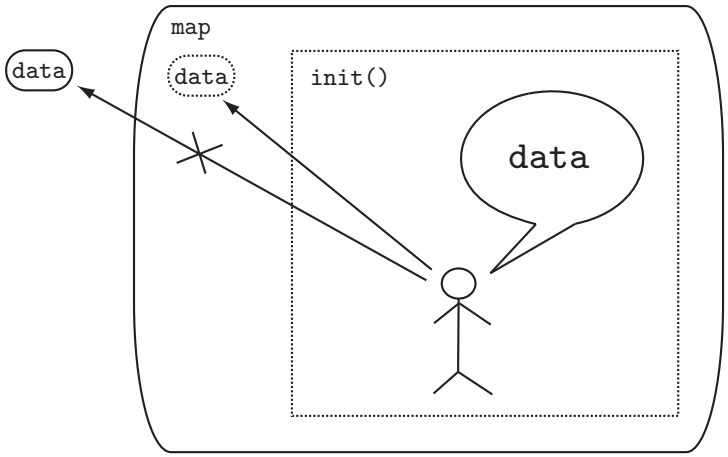


図 6.7: オブジェクト内からのメンバー変数へのアクセスの例：メンバー関数から同じオブジェクト内のメンバー変数にアクセスするには、メンバー変数名 のみでアクセスできます。もし同じ変数名のグローバル変数がある場合、そのままではグローバル変数にはアクセスできません。

図 6.7（78 ページ）はこの章のスコープの一例です。この例ではプログラマの視点はオブジェクトの中のメンバー関数 `init()` の中にあります。この視点からは同じオブジェクト内のメンバー `data` へは単にそのメンバー名を指定することでアクセスできます。ここで仮に同じ名前 `data` のグローバル変数が宣言されていると仮定しましょう。そのグローバル変数には `data` という名前ではアクセスできません。何故ならばメンバー関数 `init()` の中から見た最も近くの `data` はメンバー変数の `data` だからです。同じ名前のグローバル変数にアクセスするには、スコープ解決演算子を用いて、 `::data` と記述します。直前にクラス名を付けない `::` はグローバルスコープ解決演算子と呼ばれます。しかし本来、スコープに重複する名前が存在するようなプログラムは書くべきではありません。

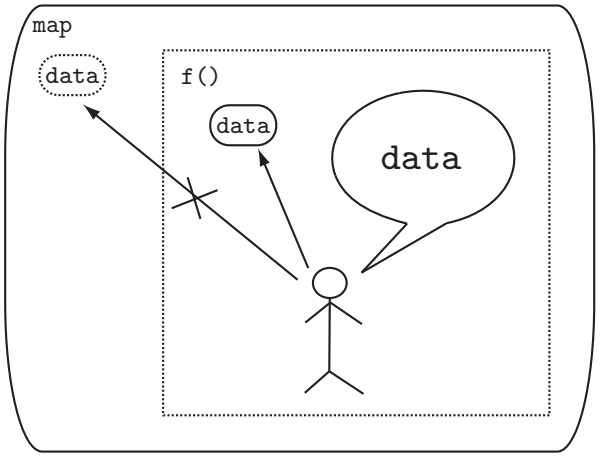


図 6.8: オブジェクト内からのローカル変数へのアクセス：メンバー関数からその関数内へローカル変数へのアクセスするには、ローカル変数名 のみでアクセスできます。もし同じ変数名のメンバー変数がある場合、そのままではメンバー変数にはアクセスできません。

図 6.8（79 ページ）は別の例です。Map クラスにメンバー関数 f() があると仮定します。しかも f() の中ではローカル変数 data が宣言されていると仮定します。このとき、 f() から見た最も近くの data はローカル変数の data です。メンバー変数の data には名前 data ではアクセスできません。メンバー変数（静的メンバー変数を含みます）にアクセスするにはスコープ解決演算子を用いて、 Map::data と記述します。あるいは、静的ではないメンバー変数に限って言えば、9.3.1 項で解説する this を用いて this->data と記述します。重ねての注意ですが、スコープに重複する名前が存在するようなプログラムは書くべきではありません。

6.6　プレイヤークラスの実装

プレイヤーを操作する関数群は図 6.9（80 ページ）の通りです。

関数名の変更などを除き、主な変更は以下の 2 点です。まず、図 5.4 の

player.x、 player.y、 player.hp

を、それぞれ

x、 y、 hp

に置き換えています。次に、 Map クラスの定数 ROCK をスコープ解決演算子を用いて Map::ROCK に変更しています。

もうひとつ、小さな変更ですが、前章（図 5.4（61 ページ））の

```
bool playerCellPrint(int x, int y) {
```

```
void Player::init() {                                   // 変更
    x = 1; y = 1;                                       // 変更
    hp = INIT_HP;                                       // 変更
}

void Player::print() {                                  // 変更
    cout << "Player : (" << x << "," << y << ") "       // 変更
         << hp << endl;                                 // 変更
}

bool Player::cellPrint(int x_, int y_) {                // 変更
    if ((x == x_)&&(y == y_)) {                         // 変更
        cout << '@' << ' ';
        return true;
    }
    return false;
}

bool Player::moveTo(int dx, int dy) {                   // 変更
    int targetX = x+dx, targetY = y+dy;                 // 変更

    if (map.data[targetX][targetY] == Map::ROCK) { return false; }

    x = targetX; y = targetY;
    return true;
}

void Player::action() {                                 // 変更
    ** 以下の末尾の 1 行を除き、変更なし **

    moveTo(dx,dy);                              // playerMoveTo() から変更
}
```

図 6.9: メンバー関数を導入したダンジョンゲームのプログラム（プレイヤー）：図 5.4（61 ページ）参照

では、仮引数名が x、y でしたが、本章では

```
bool Player::cellPrint(int x_, int y_) {                // 変更
```

と、末尾にアンダースコアを付けた仮引数名 x_、y_ へ変更しました。これは仮引数名がメンバー変数名と重複するのを避けるためです。似た名前の混乱回避、衝突回避の命名法については他にも様々なルールが用いられています。

この章の変更も全く機械的です。ゲームの実行も前々章、前章から変わりありません。

6.7　付録：この章の全プログラムリスト

```
//宣言部
#include <iostream>
using namespace std;

class Map {
  public:
    static const int ROCK = 0;            // Map 内へ移動
    static const int EMPTY = 1;           // 同上
    static const int GOAL = 2;            // 同上

    static const int XSIZE = 13;          // 同上
    static const int YSIZE = 11;          // 同上

    int data[XSIZE][YSIZE];

    void init();                          // Map 内へ移動
    void print();                         // 同上
    void cellPrint(int x, int y);         // 同上
};

class Player {
  public:
    static const int INIT_HP = 100;       // Map 内へ移動

    int x, y;
    int hp;

    void init();
    void print();                        // Map 内へ移動
    bool cellPrint(int x, int y);        // 同上
    bool moveTo(int dx, int dy);         // 同上
    void action();                       // 同上
};

//プロトタイプ宣言
void initAll();
void printAll();
void turn();
void mainLoop();
char getKeyChar();

//グローバル宣言
Map map;
Player player;
```

```
//トップレベルの実装
int main(int argc, const char * argv[]) {
    initAll();
    mainLoop();
    return 0;
}

void initAll() {
    map.init();                         // mapInit() から変更
    player.init();                      // playerInit() から変更
}

void printAll() {
    map.print();                        // mapPrint() から変更
    player.print();                     // playerPrint() から変更
    cout << endl;
}

void mainLoop() {
    printAll();
    while (true) {
        turn();
    }
}

void turn() {
    player.action();                    // playerAction() から変更
    printAll();
}

char getKeyChar() {
    char c;
    cout << "? " << flush;
    cin >> c;
    return c;
}
```

```
//マップクラスの実装
void Map::init() {                                  // 変更
    for (int x = 0; x < XSIZE; x++) {
        for (int y = 0; y < YSIZE; y++) {
            if ((x == 0)||(x == XSIZE-1)||
               (y == 0)||(y == YSIZE-1)||
               ((abs(x-XSIZE/2) <= 1)&&
                (abs(y-YSIZE/2) <= 1))) {

                data[x][y] = ROCK;                  // 変更
            }
            else { data[x][y] = EMPTY; }            // 変更
        }
    }
    data[XSIZE-2][YSIZE-2] = GOAL;                  // 変更
}

void Map::print() {                                 // 変更
    for (int y = 0; y < YSIZE; y++) {
        for (int x = 0; x < XSIZE; x++) {
            if (player.cellPrint(x,y)) { continue; }// 変更
            cellPrint(x,y);                          // 変更
        }
        cout << endl;
    }
}

void Map::cellPrint(int x, int y) {                 // 変更
    int e = data[x][y];                             // 変更
    if (e == ROCK)       { cout << '#' << ' '; }
    else if (e == EMPTY) { cout << '.' << ' '; }
    else if (e == GOAL)  { cout << 'G' << ' '; }
    else                 { cout << 'E' << ' '; }
}
```

```
//プレイヤークラスの実装
void Player::init() {                                   // 変更
    x = 1; y = 1;                                       // 変更
    hp = INIT_HP;                                       // 変更
}

void Player::print() {                                  // 変更
    cout << "Player : (" << x << "," << y << ") "       // 変更
         << hp << endl;                                 // 変更
}

bool Player::cellPrint(int x_, int y_) {                // 変更
    if ((x == x_)&&(y == y_)) {                         // 変更
        cout << '@' << ' ';
        return true;
    }
    return false;
}

bool Player::moveTo(int dx, int dy) {                   // 変更
    int targetX = x+dx, targetY = y+dy;                 // 変更

    if (map.data[targetX][targetY] == Map::ROCK) { return false; }

    x = targetX; y = targetY;
    return true;
}

void Player::action() {                                 // 変更
    int dx, dy;
    switch (getKeyChar()) {
        case 'z': dx = -1; dy = +1; break;
        case 'x': dx =  0; dy = +1; break;
        case 'c': dx = +1; dy = +1; break;
        case 'a': dx = -1; dy =  0; break;
        case 'd': dx = +1; dy =  0; break;
        case 'q': dx = -1; dy = -1; break;
        case 'w': dx =  0; dy = -1; break;
        case 'e': dx = +1; dy = -1; break;
        default:  dx =  0; dy =  0;
    }

    moveTo(dx,dy);                              // playerMoveTo() から変更
}
```

1

1

第7章　モンスターを追加する

この章ではプレイヤーをまねて 2 体のモンスターを導入します。そのために、以下の処理を新たにプログラム化します。

1. プレイヤーはモンスターを、モンスターはプレイヤーを攻撃する。
2. モンスターは自動的に移動、攻撃を行う。

なお、この章にはプログラミング技術に関する新規事項はありません。

7.1　プログラムの構造

図 7.1（88 ページ）が、この章のプログラムの構造です。関数呼び出し関係、データへのアクセスの赤い矢印、青い矢印はこの図では省略しました。新たに `Monster` クラスのインスタンス（オブジェクト）がグローバル変数 `monster1`、`monster2` として追加されている点に注意してください。

7.2　宣言部の変更

図 7.2（89 ページ）は変更された宣言部です。`Monster` クラスが追加されています。`Player` クラスに攻撃処理のメンバー関数 `atackTo()` が新たに追加されています。`Monster` クラスは `Player` クラスと同じ構造ですが、メンバー関数の中身が異なります。

7.3　トップレベルの実装の変更

図 7.3（90 ページ）はトップレベルのプログラムです。プログラムの上位の処理を担う関数群にはモンスター 2 体の処理を追加しています。

まず、モンスターの初期化：

```
monster1.init(1, Map::YSIZE-2);        // モンスターの初期化の追加
monster2.init(Map::XSIZE-2, 1);        // 2 体で初期値が異なる
```

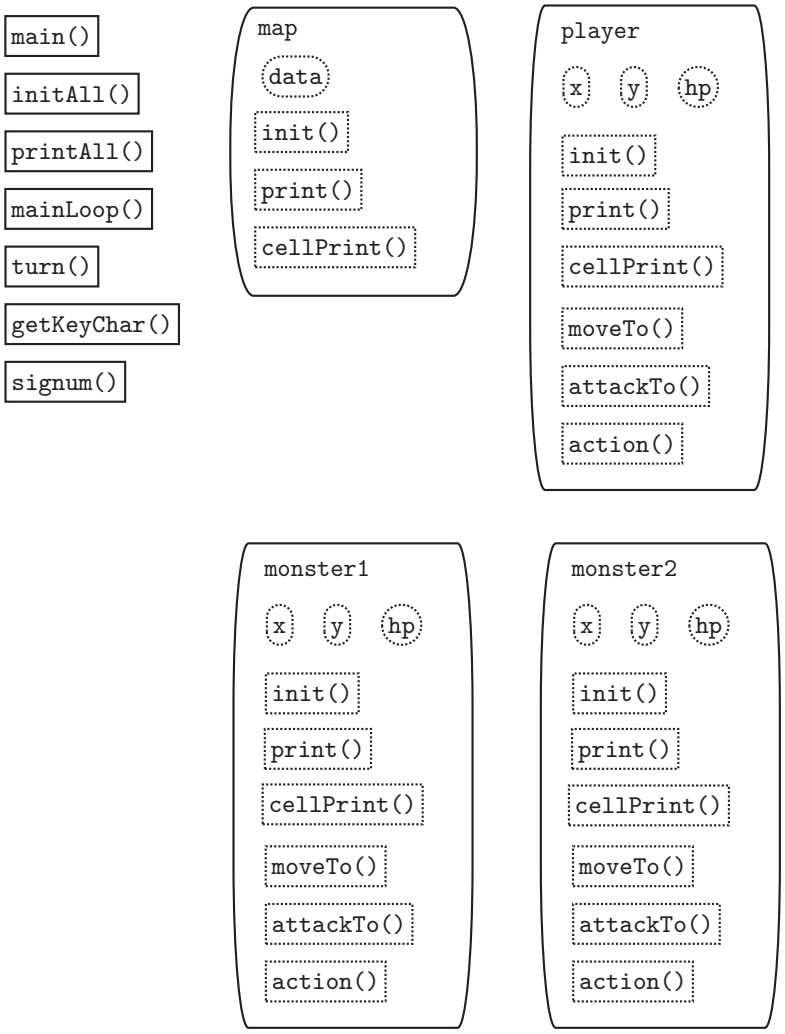


図 7.1: この章のプログラムの構造：グローバル関数群と map、 player、 monster1、 monster2 から成る。

では、1 体のモンスターを左下、もう 1 体のモンスターを右上に初期配置します。初期位置は引数で与えるようにしました。

関数 printAll()、turn() にもモンスターの処理を追加しています。

関数 signum() はいわゆる符号関数（sign function、signum function）で、モンスターの移動を計算するために用います。C/C++にはこの関数が標準ライブラリに組み込まれていないため、自作しました。

```
class Map { ** 前章から変更なし ** };

class Player {
  ** 以下を追加する以外、前章から変更なし **
    void attackTo(int dx, int dy);          // プレイヤーの攻撃
};

class Monster {                             // モンスターのクラスの追加
  public:
    static const int INIT_HP = 150;         // HP の初期値

    int x, y;
    int hp;

    void init(int x, int y);                // 初期化
    bool cellPrint(int x, int y);           // モンスターのセル上への表示
    void print();                           // モンスターの状態の表示
    bool moveTo(int dx, int dy);            // モンスターの移動
    void attackTo(int dx, int dy);          // モンスターの攻撃
    void action();                          // モンスターのアクション
};

//プロトタイプ宣言
** 以下の関数を追加すること以外、前章から変更なし **
int signum(int a);                      // 符号関数、モンスターの action() で使用

//グローバル宣言
Map map;
Player player;
Monster monster1;                           // モンスターのグローバル変数の追加
Monster monster2;                           // モンスターのグローバル変数の追加
```

図 7.2: モンスターを追加したダンジョンゲームのプログラム（宣言部）：図 6.2（71 ページ）参照

```
int main(int argc, const char * argv[]) { ** 前章から変更なし ** }

void initAll() {
    map.init();
    player.init();
    monster1.init(1, Map::YSIZE-2);       // モンスターの初期化の追加
    monster2.init(Map::XSIZE-2, 1);       // 2 体で初期値が異なる
}

void printAll() {
    map.print();
    player.print();
    monster1.print();                     // モンスターの表示の追加
    monster2.print();
    cout << endl;
}

void mainLoop() { ** 前章から変更なし ** }

void turn() {
    player.action();
    monster1.action();                    // モンスターのアクションの追加
    monster2.action();
    printAll();
}

char getKeyChar() { ** 前章から変更なし ** }

int signum(int a) {                      // モンスターの action() で利用
    if (a < 0) return -1;                 // a が負数ならば -1 を戻り値とする
    else if (a > 0) return 1;             // a が正数ならば  1 を戻り値とする
    else return 0;                        // a が 0  ならば  0 を戻り値とする
}
```

図 7.3: モンスターを追加したダンジョンゲームのプログラム（トップレベル）

```
void Map::init() { ** 前章から変更なし ** }

void Map::print() {
    for (int y = 0; y < YSIZE; y++) {
        for (int x = 0; x < XSIZE; x++) {
            if (player.cellPrint(x,y)) { continue; }
                                        // player が (x,y) の位置ならば表示
            if (monster1.cellPrint(x,y)) { continue; }
                                        // monster1 が (x,y) の位置ならば表示
            if (monster2.cellPrint(x,y)) { continue; }
                                        // monster2 が (x,y) の位置ならば表示
            cellPrint(x,y);
        }
        cout << endl;
    }
}

void Map::cellPrint(int x, int y) { ** 前章から変更なし ** }
```

図 7.4: モンスターを追加したダンジョンゲームのプログラム（マップ）：図 5.3（60 ページ）参照

7.4　マップクラスの実装の変更

図 7.4 は、Map クラスのプログラムです。

マップのプリント（ print()）時にモンスターを表示する処理（以下の 2 行目、3 行目）：

```
            if (monster1.cellPrint(x,y)) { continue; }
                                        // monster1 が (x,y) の位置ならば表示
            if (monster2.cellPrint(x,y)) { continue; }
                                        // monster2 が (x,y) の位置ならば表示
```

を追加しました。monster1.cellPrint(x,y) では、座標位置 (x,y) にモンスター monster1 が居る場合には、そのモンスターを表示し、 continue します（ループ本体の後続の処理を行わず、次の繰り返しへ進みます）。さもなくば（(x,y) に monster1 が居ないならば）、次に monster2 について同じ処理を繰り返します。

7.5　プレイヤークラスの実装の変更

図 7.5（92 ページ）はプレイヤーのクラスの改造プログラムです。

moveTo() の中の

```
void Player::init() { ** 前章から変更なし ** }

bool Player::cellPrint(int x_, int y_) { ** 前章から変更なし ** }

void Player::print() { ** 前章から変更なし ** }

bool Player::moveTo(int dx, int dy) {
    int targetX = x+dx, targetY = y+dy;

    if (map.data[targetX][targetY] == Map::ROCK) { return false; }

    if ((targetX == monster1.x)&&(targetY == monster1.y)) { return false; }
                            // 移動先に monster1 が居るならば、移動できない

    if ((targetX == monster2.x)&&(targetY == monster2.y)) { return false; }
                            // 移動先に monster2 が居るならば、移動できない

    x = targetX; y = targetY;
    return true;
}

void Player::attackTo(int dx, int dy) {
    int targetX = x+dx, targetY = y+dy;

    if ((targetX == monster1.x)&&(targetY == monster1.y)) {
        monster1.hp -= 20;   // 攻撃先に monster1 が居るならば hp を 20 減らす
    }

    if ((targetX == monster2.x)&&(targetY == monster2.y)) {
        monster2.hp -= 20;   // 攻撃先に monster2 が居るならば hp を 20 減らす
    }
}

void Player::action() {
    ** 以下の末尾の 1 行を除き、変更なし **

    if (!moveTo(dx,dy)) { attackTo(dx,dy); }  // 移動できないならば、攻撃する
}
```

図 7.5: モンスターを追加したダンジョンゲームのプログラム（プレイヤー）：図 6.9（80 ページ）参照

```
    if ((targetX == monster1.x)&&(targetY == monster1.y)) { return false; }
                                // 移動先に monster1 が居るならば、移動できない
```

は、もし (x2, y2) にモンスター monster1 が居るならば、移動できないため、 false を戻り値とすることを表しています。monster2 についても同様です。

attackTo() の中の

```
    if ((targetX == monster1.x)&&(targetY == monster1.y)) {
        monster1.hp -= 20;    // 攻撃先に monster1 が居るならば hp を 20 減らす
    }
```

の処理は、(x2, y2) に monster1 が居る場合にそのモンスターの hp を 20 減らす攻撃処理です。monster2 についても同様です。

このプログラム片では、ダメージ量の生の数値 20 を条件分岐の中にそのまま書き込んでおり、あまり良いプログラミング作法ではありません。これは後章でオブジェクト指向の技法を用いてスッキリと書き直しますが、当面このままにしておきます。

7.6　モンスタークラスの実装

図 7.6（94 ページ）は、モンスターの実装です。

ほぼプレイヤーと同様の実装ですが、異なるのは以下の 2 点です。

まず、初期化関数 init() に新たに引数を導入しました。複数のモンスターを同じ位置に配置できませんから、初期位置は引数で異なる値を与えるようにします。

次に、action() 内で移動位置を求める方法がプレイヤーと異なります。モンスターは自分の位置とプレイヤーの位置の相対位置を求め、プレイヤーの方向へ x 座標値、y 座標値それぞれで最大 ± 1 の移動を試みます。その計算に signum() を用います。

この移動距離の計算方法はシンプルすぎて、ひとつ問題があります。たとえばゲームが図 7.7（95 ページ）の状況にあると仮定します。プレイヤーは $(6, 3)$、左下のモンスターは $(4, 6)$ の位置にいます。座標原点が左上であることに注意してください。

プレイヤーと左下のモンスターの相対位置は $(2, -3)$ ですから、これに符号関数を適用すると、モンスターの移動方向は $(1, -1)$ と計算できます。モンスターは $(1, -1)$ の方向（右上の方向）へ移動したいのですが、そこには岩（'#' の文字）があり、移動できないと判定されます。マップを大局的に見れば、少しだけ迂回すれば岩を回避してプレイヤーに近づく経路が見つかります。が、こ

```
void Monster::init(int x_, int y_) {            // モンスターの初期化
    x = x_; y = y_;
    hp = INIT_HP;
}

void Monster::print() {                         // モンスターの状態の表示
    cout << "Monster: (" << x << "," << y << ") "
         << hp << endl;
}

bool Monster::cellPrint(int x_, int y_) {       // モンスターのセル上への表示
    if ((x == x_)&&(y == y_)) {
        cout << 'M' << ' ';
        return true;
    }
    return false;
}

bool Monster::moveTo(int dx, int dy) {          // モンスターの移動
    int targetX = x+dx, targetY = y+dy;

    if (map.data[targetX][targetY] == Map::ROCK) { return false; }

    if ((targetX == player.x)&&(targetY == player.y)) { return false; }
                              // 移動先に player が居るならば、移動できない

    if ((targetX == monster1.x)&&(targetY == monster1.y)) { return false; }
                              // 移動先に monster1 が居るならば、移動できない

    if ((targetX == monster2.x)&&(targetY == monster2.y)) { return false; }
                              // 移動先に monster2 が居るならば、移動できない

    x = targetX; y = targetY;
    return true;
}

void Monster::attackTo(int dx, int dy) {        // モンスターの攻撃
    int targetX= x+dx, targetY = y+dy;

    if ((targetX == player.x)&&(targetY == player.y)) { player.hp -= 30; }
               // 攻撃先にプレイヤーが居るならばプレイヤーの hp を 30 減らす
}

void Monster::action() {
    int dx = signum(player.x - x);
    int dy = signum(player.y - y);

    if (!moveTo(dx,dy)) { attackTo(dx,dy); } // 移動できないならば、攻撃する
}
```

図 7.6: モンスターを追加したダンジョンゲームのプログラム（モンスター）

```
# # # # # # # # # # # # #
# . . . . . . . . . . . #
# . . . . . . M . . . . #
# . . . . . @ . . . . . #  <-- プレイヤー
# . . . . # # # . . . . #
# . . . . # # # . . . . #
# . . . M # # # . . . . #  <--
# . . . . . . . . . . . #
# . . . . . . . . . . G #
# # # # # # # # # # # # #
Player : (6,3) 40
Monster: (4,6) 150
Monster: (7,2) 150
```

図 7.7: モンスターを追加したダンジョンゲームのプログラム（モンスター）：左下のモンスターは岩が邪魔してプレイヤーに近づくことができない

れにはかなり高度な判断が必要となるため、講義テキストのプログラムにはそれを実装していません。結果、このような状況では左下のモンスターは動けなくなります。なお、右上のモンスターはプレイヤーに隣接しており、左下に向かってプレイヤーを攻撃します。

7.7　プログラムの実行例

この章のプログラムの実行は、たとえば以下のように進みます。

```
# # # # # # # # # # # # #
# @ . . . . . . . . . M #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
# . . . . # # # . . . . #
# . . . . # # # . . . . #
# . . . . # # # . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
# M . . . . . . . . . G #
# # # # # # # # # # # # #
Player : (1,1) 100
Monster: (1,9) 150
Monster: (11,1) 150

? c                          <---- 右下方向をキー入力
# # # # # # # # # # # # #
# . . . . . . . . . . . #
# . @ . . . . . . . M . #
# . . . . . . . . . . . #
# . . . . # # # . . . . #
# . . . . # # # . . . . #
# . . . . # # # . . . . #
# . . . . . . . . . . . #
# . M . . . . . . . . . #
# . . . . . . . . . . G #
# # # # # # # # # # # # #
```

```
Player : (2,2) 100
Monster: (2,8) 150
Monster: (10,2) 150

? c                          <---- 右下方向をキー入力
# # # # # # # # # # # # #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
# . . @ . . . . . M . . #
# . . . . # # # . . . . #
# . . . . # # # . . . . #
# . . . . # # # . . . . #
# . . M . . . . . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . G #
# # # # # # # # # # # # #
Player : (3,3) 100
Monster: (3,7) 150
Monster: (9,3) 150

? x                          <---- 下方向をキー入力
# # # # # # # # # # # # #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
# . . @ . # # # M . . . #
# . . . . # # # . . . . #
# . . M . # # # . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . G #
# # # # # # # # # # # # #
Player : (3,4) 100
Monster: (3,6) 150
Monster: (8,4) 150

? s                          <---- 無動作をキー入力（モンスターを待つ）
# # # # # # # # # # # # #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
# . . @ . # # # M . . . #
# . . M . # # # . . . . #
# . . . . # # # . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . G #
# # # # # # # # # # # # #
Player : (3,4) 100
Monster: (3,5) 150
Monster: (8,4) 150

? x                          <---- 右下方向をキー入力（モンスターを攻撃）
# # # # # # # # # # # # #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
# . . @ . # # # M . . . #
```

```
# . . M . # # # . . . . #
# . . . . # # # . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . G #
# # # # # # # # # # # # #
Player : (3,4) 70            <---- 30 のダメージ
Monster: (3,5) 130           <---- 20 のダメージ
Monster: (8,4) 150

? c                          <---- 右下方向をキー入力（逃げる）
# # # # # # # # # # # # #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
# . . . . # # # M . . . #
# . . M @ # # # . . . . #
# . . . . # # # . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . G #
# # # # # # # # # # # # #
Player : (4,5) 40
Monster: (3,5) 130
Monster: (8,4) 150

? x                          <---- 下方向をキー入力（逃げる）
# # # # # # # # # # # # #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
# . . . . # # # M . . . #
# . . M . # # # . . . . #
# . . . @ # # # . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . G #
# # # # # # # # # # # # #
Player : (4,6) 10
Monster: (3,5) 130
Monster: (8,4) 150

? c                          <---- 右下方向をキー入力（逃げる）
# # # # # # # # # # # # #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
# . . . . # # # M . . . #
# . . . . # # # . . . . #
# . . . M # # # . . . . #
# . . . . @ . . . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . G #
# # # # # # # # # # # # #
Player : (5,7) 10
Monster: (4,6) 130
Monster: (8,4) 150
```

以下、同様

7.8　補足：メンバー関数の実体

前章から導入されたメンバー関数は、図 7.1（88 ページ）ではオブジェクトの中に存在するように描かれています。概念上、そう考えて問題ありませんし、そう考えないと前章の図 6.7 のスコープの説明などでは理解できません。

しかし、C++の言語処理系がアプリケーションプログラムを実行するときには実際にはメンバー関数（のオブジェクトコード）は、Map、Player、Monster のオブジェクトの中には含まれていません。

仮に Monster クラスのメンバー関数 moveTo() のオブジェクトコードのサイズが 1K バイトであったと仮定します。もしこのオブジェクトコードが Monster オブジェクトの中に含まれているならば、この章の 2 体の Monster オブジェクトそれぞれに 1K バイトのメモリを確保する必要がありますが、これは全く無駄です。何故ならば、コードの内容はオブジェクトに依らず、全く同じものだからです。

実際には全てのメンバー関数のオブジェクトコードはプログラム空間のコード領域（データではなく、オブジェクトコードを保存しておくメモリ領域）に通常のグローバル関数と同様に保存されています。言語処理系内部ではメンバー関数の実行は通常の関数の実行に解釈されて行われています。ですから、オブジェクトのメモリサイズはメンバー変数だけの合計サイズ — つまり構造体と見なした場合のサイズ — と同じです。試しに main 関数内で

```
int main(int argc, const char * argv[]) {
    initAll();
    cout << sizeof(monster1) << endl; // Monster オブジェクトのサイズを表示
    return 0;
}
```

を実行し[1]、 Monster オブジェクトのサイズを求めると、12 バイトという結果が出ます。これはモンスターのメンバー変数 x、y、hp の 3 個のメモリサイズと一致します。

ただし、この議論は後章の多相性の原理を解説する章以降では成り立たなくなります。メンバー関数の情報がオブジェクト内に保存され、同じモンスターであっても異なる関数を呼び出す場合があるからです。

[1] sizeof() は引数のメモリサイズを返答する C/C++の組み込み関数です。

7.9　付録：この章の全プログラムリスト

```
//宣言部
#include <iostream>
using namespace std;

class Map {
  public:
    static const int ROCK = 0;
    static const int EMPTY = 1;
    static const int GOAL = 2;

    static const int XSIZE = 13;
    static const int YSIZE = 11;

    int data[XSIZE][YSIZE];

    void init();
    void print();
    void cellPrint(int x, int y);
};

class Player {
  public:
    static const int INIT_HP = 100;

    int x, y;
    int hp;

    void init();
    void print();
    bool cellPrint(int x, int y);
    bool moveTo(int dx, int dy);
    void attackTo(int dx, int dy);          // プレイヤーの攻撃
    void action();
};

class Monster {                             // モンスターのクラスの追加
  public:
    static const int INIT_HP = 150;         // HP の初期値

    int x, y;
    int hp;

    void init(int x, int y);                // 初期化
    bool cellPrint(int x, int y);           // モンスターのセル上への表示
    void print();                           // モンスターの状態の表示
    bool moveTo(int dx, int dy);            // モンスターの移動
    void attackTo(int dx, int dy);          // モンスターの攻撃
    void action();                          // モンスターのアクション
};

//プロトタイプ宣言
void initAll();
void printAll();
void mainLoop();
void turn();
char getKeyChar();
```

```
int signum(int a);                    // 符号関数、モンスターの action() で使用

//グローバル宣言
Map map;
Player player;
Monster monster1;                     // モンスターのグローバル変数の追加
Monster monster2;                     // モンスターのグローバル変数の追加
```

```
//トップレベルの実装
int main(int argc, const char * argv[]) {
    initAll();
    mainLoop();
    return 0;
}

void initAll() {
    map.init();
    player.init();
    monster1.init(1, Map::YSIZE-2);       // モンスターの初期化の追加
    monster2.init(Map::XSIZE-2, 1);       // 2 体で初期値が異なる
}

void printAll() {
    map.print();
    player.print();
    monster1.print();                     // モンスターの表示の追加
    monster2.print();
    cout << endl;
}

void mainLoop() {
    printAll();
    while (true) {
        turn();
    }
}

void turn() {
    player.action();
    monster1.action();                    // モンスターのアクションの追加
    monster2.action();
    printAll();
}

char getKeyChar() {
    char c;
    cout << "? " << flush;
    cin >> c;
    return c;
}

int signum(int a) {                      // モンスターの action() で利用
    if (a < 0) return -1;                 // a が負数ならば -1 を戻り値とする
    else if (a > 0) return 1;             // a が正数ならば  1 を戻り値とする
    else return 0;                        // a が 0  ならば  0 を戻り値とする
}
```

```
//マップクラスの実装
void Map::init() {
    for (int x = 0; x < XSIZE; x++) {
        for (int y = 0; y < YSIZE; y++) {
            if ((x == 0)||(x == XSIZE-1)||
               (y == 0)||(y == YSIZE-1)||
               ((abs(x-XSIZE/2) <= 1)&&
                (abs(y-YSIZE/2) <= 1))) {

                data[x][y] = ROCK;
            }
            else { data[x][y] = EMPTY; }
        }
    }
    data[XSIZE-2][YSIZE-2] = GOAL;
}

void Map::print() {
    for (int y = 0; y < YSIZE; y++) {
        for (int x = 0; x < XSIZE; x++) {
            if (player.cellPrint(x,y)) { continue; }
                                        // player が (x,y) の位置ならば表示
            if (monster1.cellPrint(x,y)) { continue; }
                                        // monster1 が (x,y) の位置ならば表示
            if (monster2.cellPrint(x,y)) { continue; }
                                        // monster2 が (x,y) の位置ならば表示
            cellPrint(x,y);
        }
        cout << endl;
    }
}

void Map::cellPrint(int x, int y) {
    int e = data[x][y];
    if (e == ROCK)       { cout << '#' << ' '; }
    else if (e == EMPTY) { cout << '.' << ' '; }
    else if (e == GOAL)  { cout << 'G' << ' '; }
    else                 { cout << 'E' << ' '; }
}
```

```
//プレイヤークラスの実装
void Player::init() {
    x = 1; y = 1;
    hp = INIT_HP;
}

void Player::print() {
    cout << "Player : (" << x << "," << y << ") "
         << hp << endl;
}

bool Player::cellPrint(int x_, int y_) {
    if ((x == x_)&&(y == y_)) {
        cout << '@' << ' ';
        return true;
    }
    return false;
}

bool Player::moveTo(int dx, int dy) {
    int targetX = x+dx, targetY = y+dy;

    if (map.data[targetX][targetY] == Map::ROCK) { return false; }

    if ((targetX == monster1.x)&&(targetY == monster1.y)) { return false; }
                            // 移動先に monster1 が居るならば、移動できない

    if ((targetX == monster2.x)&&(targetY == monster2.y)) { return false; }
                            // 移動先に monster2 が居るならば、移動できない

    x = targetX; y = targetY;
    return true;
}

void Player::attackTo(int dx, int dy) {
    int targetX = x+dx, targetY = y+dy;

    if ((targetX == monster1.x)&&(targetY == monster1.y)) {
        monster1.hp -= 20;   // 攻撃先に monster1 が居るならば hp を 20 減らす
    }

    if ((targetX == monster2.x)&&(targetY == monster2.y)) {
        monster2.hp -= 20;   // 攻撃先に monster2 が居るならば hp を 20 減らす
    }
}

void Player::action() {
    int dx, dy;
    switch (getKeyChar()) {
        case 'z': dx = -1; dy = +1; break;
        case 'x': dx =  0; dy = +1; break;
        case 'c': dx = +1; dy = +1; break;
        case 'a': dx = -1; dy =  0; break;
        case 'd': dx = +1; dy =  0; break;
        case 'q': dx = -1; dy = -1; break;
        case 'w': dx =  0; dy = -1; break;
        case 'e': dx = +1; dy = -1; break;
        default:  dx =  0; dy =  0;
```

```
    }

    if (!moveTo(dx,dy)) { attackTo(dx,dy); }  // 移動できないならば、攻撃する
}
```

```
//モンスタークラスの実装
void Monster::init(int x_, int y_) {            // モンスターの初期化
    x = x_; y = y_;
    hp = INIT_HP;
}

void Monster::print() {                         // モンスターの状態の表示
    cout << "Monster: (" << x << "," << y << ") "
         << hp << endl;
}

bool Monster::cellPrint(int x_, int y_) {     // モンスターのセル上への表示
    if ((x == x_)&&(y == y_)) {
        cout << 'M' << ' ';
        return true;
    }
    return false;
}

bool Monster::moveTo(int dx, int dy) {        // モンスターの移動
    int targetX = x+dx, targetY = y+dy;

    if (map.data[targetX][targetY] == Map::ROCK) { return false; }

    if ((targetX == player.x)&&(targetY == player.y)) { return false; }
                            // 移動先に player が居るならば、移動できない

    if ((targetX == monster1.x)&&(targetY == monster1.y)) { return false; }
                            // 移動先に monster1 が居るならば、移動できない

    if ((targetX == monster2.x)&&(targetY == monster2.y)) { return false; }
                            // 移動先に monster2 が居るならば、移動できない

    x = targetX; y = targetY;
    return true;
}

void Monster::attackTo(int dx, int dy) {      // モンスターの攻撃
    int targetX= x+dx, targetY = y+dy;

    if ((targetX == player.x)&&(targetY == player.y)) { player.hp -= 30; }
               // 攻撃先にプレイヤーが居るならばプレイヤーの hp を 30 減らす
}

void Monster::action() {
    int dx = signum(player.x - x);
    int dy = signum(player.y - y);

    if (!moveTo(dx,dy)) { attackTo(dx,dy); } // 移動できないならば、攻撃する
}
```

1

1

第8章　マップ、プレイヤー、モンスターを動的オブジェクトに変更する

後章ではC++のプログラムをJavaへ移植します。C++とJavaには様々な違いがありますが、移植を行う上での大きな違いは

- Javaのオブジェクトは全て動的オブジェクトに制限されている

点です。この制限のせいでJavaはC++に比べてやや不自由です。しかし、逆に不自由な分、オブジェクトの操作がよりシンプルに、より安全になる[1]などの効果があります。また、後章で述べる多相性は動的オブジェクトの上でのみ機能しますから、C++であっても動的オブジェクトを避けては通れません。

この章では前章のオブジェクト `map`、`player`、`monster1`、`monster2` を全て動的オブジェクトへ変更します。

8.1　オブジェクトと記憶領域の関係

復習になりますが、C/C++ではデータやオブジェクトを割り当てるメモリ領域を以下の3種類に大別できます。

静的メモリ領域（static memory area）： コンパイル時にメモリ使用量が確定できるグローバル変数や `static` キーワードで修飾されたローカル変数のデータは、プログラムの実行開始時点にこの領域に割り当てられ、プログラム実行終了まで生存し続けます。この領域に割り当てられる変数を**静的変数**（static variable）と呼びます。

自動メモリ領域（automatic memory area）： 関数あるいは関数内のブロック { } に宣言されたローカル変数のデータは、その変数が宣言された時点でこの領域に割り当てられ、制御がブロックから抜けるときにそのデータは消滅します。このメモリ領域を**局所メモリ領域**（local memory area）、**スタック領域**（stack area、あるいは単にスタック）などとも呼びます。この領域に割り当てられる変数を**自動変数**（automatic variable）とも呼びます。

[1]「プログラミング概論/演習 I,II」でポインタに苦労した人にとっては「操作がシンプル、安全になる」とは全く思えないかもしれませんが、それはC++が多機能すぎることに起因します。

動的メモリ領域（dynamic memory area）： プログラム内で new 演算子[2]によって生成されるデータは全てこの領域に割り当てられ、delete 演算子[3]によってこの領域から削除されます。このメモリ領域を**ヒープ領域**（heap area）とも呼びます。この領域に生成されるデータ、オブジェクトを動的データ、動的オブジェクトと呼びます。なお、動的データ、動的オブジェクトはメモリアドレスで管理されるだけであり、直接的には変数と関連付けられない[4]ため、動的変数と呼ばれることはありません[5]。

前章のプログラムのオブジェクト map、 player、 monster1 、 monster2 はいずれも静的メモリ領域に作られていました。この章ではこれらを動的メモリ領域へ割り付けます。

一般に動的オブジェクトの利用には熟練したプログラミング技術が必要です。特に不要になった動的オブジェクトを適切に delete する処理には高度な技術を要します。たとえば delete が実行されないとヒープ領域に利用されないオブジェクトが無駄に居残ることになり、delete が重複して実行されるとプログラムが暴走するなど、様々な危険が隣り合わせになっています。そのため、Java を含め、最近のプログラミング言語では delete はシステムに任せる方式が主流です。これについては後章で解説します。

この講義テキストの C++のプログラムでは new は行いますが、delete は行いません。その意味では少々手抜きしたプログラムになっています。

8.2　簡単な例題

以下は 6.1 節（68 ページ）の例題プログラムです。

```
struct StructA {
    int   a;
};
    ...
    StructA data;
    data.a = 10;
    data.a++;
    ...
```

このプログラムの StructA data; は、自動変数としての構造体の宣言です。これを StructA *data = new StructA; に変更した場合、すなわち動的オブジェクトをポインタが指している状況に変更した場合、データのアクセスは以下のように変更されます。

[2]C 言語では malloc()
[3]C 言語では free()
[4]ポインタ型変数にそのアドレス値が保持されることはありますが、動的データに特定の変数名が割り当てられることはありません。
[5]稀に自動変数のことを動的変数と呼ぶことがあるようです。

```
struct StructA {
    int   a;
};
    ...
    StructA *data = new StructA;
    data->a = 10;
    data->a++;
    ...
```

ここまでは「プログラミング概論/演習 I,II」の復習です。

次に、以下も 6.1 節（68 ページ）の例題プログラムです。

```
struct StructA {
    int   a;
    void init() { a = 10; }
    void inc() { a++; }
};
    ...
    StructA data;
    data.init();
    data.inc();
    ...
```

このプログラムにおいても `StructA data;` を `StructA *data = new StructA;` に変更した場合、メンバー関数の呼び出し方は以下のように変更されます。メンバー関数へのアクセスの変更はメンバー変数へのアクセスの変更と同様であって、ドット演算子 `.` をアロー演算子 `->` に変更するだけです。

```
struct StructA {
    int   a;
    void init() { a = 10; }
    void inc() { a++; }
};
    ...
    StructA *data = new StructA;
    data->init();
    data->inc();
    ...
```

8.3　宣言部の変更

図 8.1（111 ページ）はこの章の宣言部です。たとえばマップオブジェクトの場合、前章において

```
//型宣言
  ** 前章から変更なし **

//プロトタイプ宣言
  ** 前章から変更なし **

//グローバル宣言
Map *map = new Map;                    // グローバル変数をポインタに変更
Player *player = new Player;
Monster *monster1 = new Monster;
Monster *monster2 = new Monster;
```

図 8.1: メンバー関数を導入したダンジョンゲームのプログラム（）：図 7.2（89 ページ）参照

```
Map map;
```

と宣言されていたものを

```
Map *map = new Map;
```

と変更しています。つまり、グローバル変数としてポインタを用意し、プログラム実行開始時に Map クラスの動的オブジェクトを生成し、ポインタはそのオブジェクトを指すように初期位置しています。プレイヤー、モンスターを同様です。

8.4 トップレベル/マップクラス/プレイヤークラス/モンスタークラスの実装の変更

変数 map、 player、 monster1、 monster2 をオブジェクト実体からポインタへ変更したため、実装では、プログラム中の

- “map.” を “map->” へ変更、
- “player.” を “player->” へ変更、
- “monster1.” を “monster1->” へ変更、
- “monster2.” を “monster2->” へ変更

すればよいだけです。個々の変更箇所は示しません。

この章の解説は以上です。

8.5　付録：この章の全プログラムリスト

```
//宣言部
#include <iostream>
using namespace std;

class Map {
  public:
    static const int ROCK = 0;
    static const int EMPTY = 1;
    static const int GOAL = 2;

    static const int XSIZE = 13;
    static const int YSIZE = 11;

    int data[XSIZE][YSIZE];

    void init();
    void print();
    void cellPrint(int x, int y);
};

class Player {
  public:
    static const int INIT_HP = 100;

    int x, y;
    int hp;

    void init();
    void print();
    bool cellPrint(int x, int y);
    bool moveTo(int dx, int dy);
    void attackTo(int dx, int dy);
    void action();
};

class Monster {
  public:
    static const int INIT_HP = 150;

    int x, y;
    int hp;

    void init(int x, int y);
    void print();
    bool cellPrint(int x, int y);
    bool moveTo(int dx, int dy);
    void attackTo(int dx, int dy);
    void action();
};

//プロトタイプ宣言
void initAll();
void printAll();
void turn();
void mainLoop();
char getKeyChar();
```

```
int signum(int a);

//グローバル宣言
Map *map = new Map;                    // グローバル変数をポインタに変更
Player *player = new Player;
Monster *monster1 = new Monster;
Monster *monster2 = new Monster;
```

```
//トップレベルの実装
int main(int argc, const char * argv[]) {
    initAll();
    mainLoop();
    return 0;
}

void initAll() {
    map->init();
    player->init();
    monster1->init(1, Map::YSIZE-2);
    monster2->init(Map::XSIZE-2, 1);
}

void printAll() {
    map->print();
    player->print();
    monster1->print();
    monster2->print();
    cout << endl;
}

void mainLoop() {
    printAll();
    while (true) {
        turn();
    }
}

void turn() {
    player->action();
    monster1->action();
    monster2->action();
    printAll();
}

char getKeyChar() {
    char c;
    cout << "? " << flush;
    cin >> c;
    return c;
}

int signum(int a) {
    if (a < 0) return -1;
    else if (a > 0) return 1;
    else return 0;
}
```

```
//マップクラスの実装
void Map::init() {
    for (int x = 0; x < XSIZE; x++) {
        for (int y = 0; y < YSIZE; y++) {
            if ((x == 0)||(x == XSIZE-1)||
               (y == 0)||(y == YSIZE-1)||
               ((abs(x-XSIZE/2) <= 1)&&
                (abs(y-YSIZE/2) <= 1))) {

                data[x][y] = ROCK;
            }
            else { data[x][y] = EMPTY; }
        }
    }
    data[XSIZE-2][YSIZE-2] = GOAL;
}

void Map::print() {
    for (int y = 0; y < YSIZE; y++) {
        for (int x = 0; x < XSIZE; x++) {
            if (player->cellPrint(x,y)) { continue; }
            if (monster1->cellPrint(x,y)) { continue; }
            if (monster2->cellPrint(x,y)) { continue; }
            cellPrint(x,y);
        }
        cout << endl;
    }
}

void Map::cellPrint(int x, int y) {
    int e = data[x][y];
    if (e == ROCK)       { cout << '#' << ' '; }
    else if (e == EMPTY) { cout << '.' << ' '; }
    else if (e == GOAL)  { cout << 'G' << ' '; }
    else                 { cout << 'E' << ' '; }
}
```

```
//プレイヤークラスの実装
void Player::init() {
    x = 1; y = 1;
    hp = INIT_HP;
}

void Player::print() {
    cout << "Player : (" << x << "," << y << ") "
         << hp << endl;
}

bool Player::cellPrint(int x_, int y_) {
    if ((x == x_)&&(y == y_)) {
        cout << '@' << ' ';
        return true;
    }
    return false;
}

bool Player::moveTo(int dx, int dy) {
    int targetX = x+dx, targetY = y+dy;

    if (map->data[targetX][targetY] == Map::ROCK) { return false; }
    if ((targetX == monster1->x)&&(targetY == monster1->y)) { return false; } //!
    if ((targetX == monster2->x)&&(targetY == monster2->y)) { return false; } //!

    x = targetX; y = targetY;
    return true;
}

void Player::attackTo(int dx, int dy) {
    int targetX = x+dx, targetY = y+dy;
    if ((targetX == monster1->x)&&(targetY == monster1->y)) { monster1->hp -= 20; }
    if ((targetX == monster2->x)&&(targetY == monster2->y)) { monster2->hp -= 20; }
}

void Player::action() {
    int dx, dy;
    switch (getKeyChar()) {
        case 'z': dx = -1; dy = +1; break;
        case 'x': dx =  0; dy = +1; break;
        case 'c': dx = +1; dy = +1; break;
        case 'a': dx = -1; dy =  0; break;
        case 'd': dx = +1; dy =  0; break;
        case 'q': dx = -1; dy = -1; break;
        case 'w': dx =  0; dy = -1; break;
        case 'e': dx = +1; dy = -1; break;
        default:  dx =  0; dy =  0;
    }

    if (!moveTo(dx,dy)) { attackTo(dx,dy); }
}
```

```
//モンスタークラスの実装
void Monster::init(int x_, int y_) {
    x = x_; y = y_;
    hp = INIT_HP;
}

void Monster::print() {
    cout << "Monster: (" << x << "," << y << ") "
         << hp << endl;
}

bool Monster::cellPrint(int x_, int y_) {
    if ((x == x_)&&(y == y_)) {
        cout << 'M' << ' ';
        return true;
    }
    return false;
}

bool Monster::moveTo(int dx, int dy) {
    int targetX = x+dx, targetY = y+dy;

    if (map->data[targetX][targetY] == Map::ROCK) { return false; }
    if ((targetX == player->x)&&(targetY == player->y)) { return false; }
    if ((targetX == monster1->x)&&(targetY == monster1->y)) { return false; }
    if ((targetX == monster2->x)&&(targetY == monster2->y)) { return false; }
    x = targetX; y = targetY;
    return true;
}

void Monster::attackTo(int dx, int dy) {
    int targetX = x+dx, targetY = y+dy;
    if ((targetX == player->x)&&(targetY == player->y)) { player->hp -= 30; }
}

void Monster::action() {
    int dx = signum(player->x - x);
    int dy = signum(player->y - y);

    if (!moveTo(dx,dy)) { attackTo(dx,dy); }
}
```

1

1

第9章　ゲーム全体をオブジェクト化する：グローバルの削除

変数 map、 player、 monster1、 monster2 は、それがオブジェクト実体であれポインタであれ、前章までは一貫してグローバル変数として参照されてきました。しかし、後章で利用する **Java ではグローバル変数を用いることはできません**。如何なる変数もクラスの中に宣言せねばなりません。4.2.4 項（41 ページ）に書いたように、グローバル変数は無秩序なアクセスを許す危険な存在であり、その意味でグローバルを許さない Java の考え方は（やや窮屈に感じる場合もありますが）理に適ったものと言えます。また、**Java ではグローバル関数も用いることができません**。グローバル関数とは、非オブジェクト指向では通常の、グローバル変数と同じスコープを持ち、どこからでも呼び出すことができる、前章までの `initAll()`、 `printAll()` などです。

後章で C++のプログラムを Java のプログラムへ変換することを考慮し、この章では前章のプログラム中のグローバル変数、グローバル関数を全て、新たに導入する `Game` オブジェクトの中へ移動します。これによってプログラムはより一層、オブジェクト指向風に変更されます。

この章にはオブジェクト指向とは直接関係しない新規事項がいくつかあります。プログラムの解説の中で紹介していきます。

9.1　プログラムの構造

図 9.1（121 ページ）は、この章のプログラムの構造を図示したものです。新たに `Game` クラスを定義し、`map`、 `player`、 `monster1`、 `monster2` は `Game` オブジェクトのメンバー変数へ移動します。`Game` オブジェクトは `main` 関数が管理します（`main` 関数がオブジェクトを生成し、ポインタ `game` を保持します）。また、`map`、 `player`、 `monster1`、 `monster2` は相互に参照する必要があるため、それぞれのクラスに `Game` オブジェクトへのポインタ `game` を追加します。オブジェクト間の相互参照は `game` を介して行います。

具体的な動作は後節で解説します。

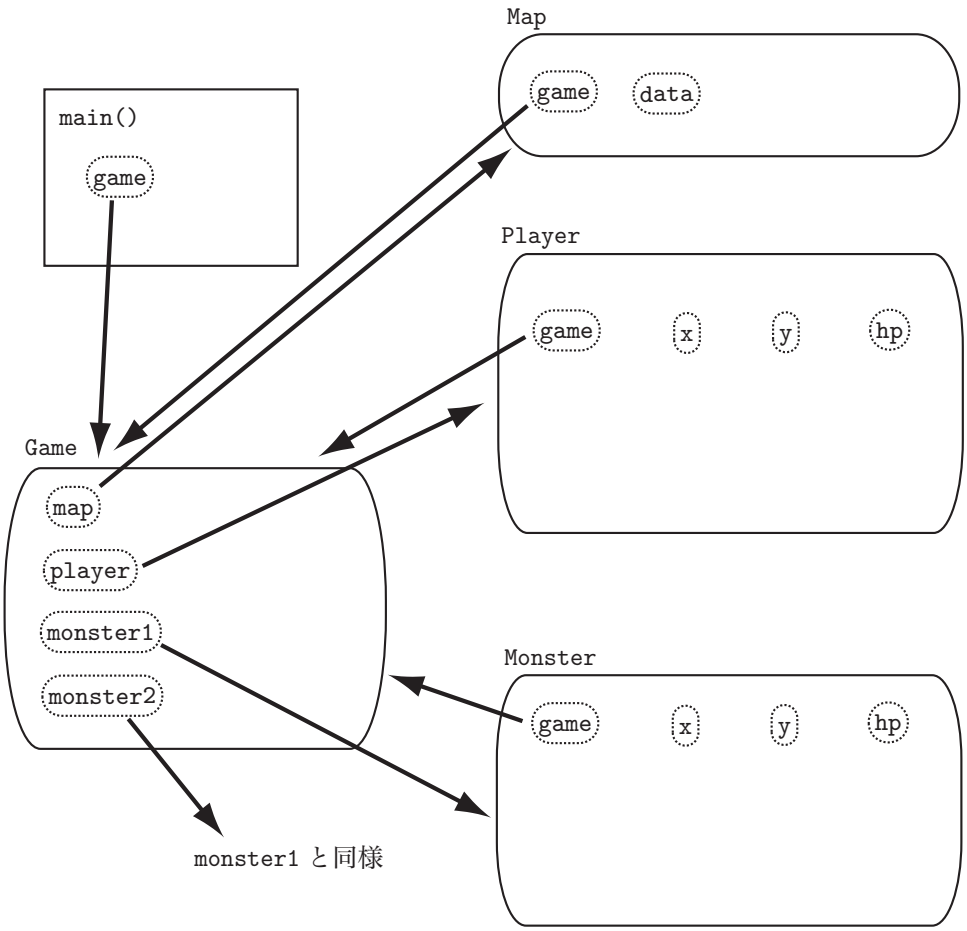


図 9.1: この章のプログラムの構造：ポインタの参照関係に注意してください。メンバー関数は省略しています。

9.2　宣言部の変更

少し長くなっていますが、図 9.2（122 ページ）、図 9.3（123 ページ）がこの章のプログラムの宣言部です。最も大きな変更点は Game クラスの導入です。グローバル変数およびグローバル関数を Game クラスに移動しました。

前章のグローバル変数 map、 player、 monster1、 monster2 はグローバルであったため、これらはどこからでも自由に参照できました。しかしこの章でそれらを Game オブジェクト内に閉じ込めました。しかし、それらを相互に参照できるようにするために Map、 Player、 Monster のそれぞれのオブジェクトに Game オブジェクトを指すポインタ game を追加しました。これらの変更に関連していくつかの新規事項、関連事項を解説します。

```
class Map;                          // クラス名の前方宣言
class Player;                       // 同上
class Monster;                      // 同上

class Game {                        // ゲーム全体を統括するクラス
  public:
    Map *map;                       // ゲームの中のマップ
    Player *player;                 // ゲームの中のプレイヤー
    Monster *monster1;               // ゲームの中のモンスター
    Monster *monster2;               // ゲームの中のモンスター

    void init();                    // ゲームの初期化
    void print();                   // ゲームの表示
    void mainLoop();                // ループ
    void turn();                    // ひとつのターン
    static char getKeyChar();       // キー入力の static 関数
    static int signum(int a);       // 浮動関数の static 関数
};

class Map {
  public:
    static const int ROCK = 0;
    static const int EMPTY = 1;
    static const int GOAL = 2;

    static const int XSIZE = 13;
    static const int YSIZE = 11;

    Game *game;                     // ゲームへのポインタ
    int data[XSIZE][YSIZE];

    void init(Game *game_);  // Game オブジェクトのアドレスを受け取って初期化
    void cellPrint(int x, int y);
    void print();
};

class Player {
  public:
    static const int INIT_HP = 100;

    Game *game;                    // ゲームへのポインタ
    int x, y;
    int hp;

    void init(Game *game_);  // Game オブジェクトのアドレスを受け取って初期化
    bool cellPrint(int x, int y);
    void print();
    bool moveTo(int dx, int dy);
    void attackTo(int dx, int dy);
    void action();
};
```

図 9.2: グローバルを全て削除したダンジョンゲームのプログラム（その 1、宣言部）

```
class Monster {
  public:
    static const int INIT_HP = 150;

    Game *game;                        // ゲームへのポインタ
    int x, y;
    int hp;

    void init(Game *game_, int x, int y);
                              // Game オブジェクトのアドレスを受け取って初期化
    bool cellPrint(int x, int y);
    void print();
    bool moveTo(int dx, int dy);
    void attackTo(int dx, int dy);
    void action();
};

//プロトタイプ宣言
** 全て削除 **

//グローバル宣言
** 全て削除 **
```

図 9.3: グローバルを全て削除したダンジョンゲームのプログラム（その 2、宣言部）

9.2.1　前方宣言

この項の内容は最近のプログラミング言語の潮流から見ればもはや古い内容なのですが、そうは言っても、これを分かっていないと C++でオブジェクト指向プログラムを書くことができません（なお、Java はその限りではありません）。

図 9.2 の 3 行目：

```
class Map;                             // クラス名の前方宣言
```

は**前方宣言**（forward declaration）と呼ばれるクラス宣言です。この宣言は、 `Map` がクラス名であることを宣言しているだけです。この前方宣言が必要な理由は、図 9.2 の `Game` クラスの定義中に以下のように `Map` クラスのポインタを用いているためです。

```
class Game {                           // ゲーム全体を統括するクラス
  public:
    Map *map;                          // ゲームの中のマップ
```

つまり、Map がクラス（あるいはユーザー定義型）であることが Game クラスの定義の前に分かっている必要があり、そのために前方宣言をおこなっています。そして実際の Map クラスの具体的な定義は Game クラスの定義の後に書かれています。

前方宣言を行わないために Map クラスの定義を Game クラスの定義の前に置いたらどうでしょうか。残念ながら、やはり前方宣言は必要です。Map クラスが Game 型ポインタをメンバー変数として持つため, 今度は Game の前方宣言が必要になっるからです。つまり、複数のクラスがポインタを介して相互に参照する形になっている場合には前方宣言は避けられません。

さて、前方宣言が必要である根本的な理由は、**C++コンパイラは 1 パスコンパイルを行う**からです。1 パスコンパイルとは、ソースプログラムを 1 回だけ解析してコンパイル処理を終える方式のことを言います。もし 2 パスコンパイル — ソースプログラムを 2 回解析してコンパイル処理を終える方式 — であるならば、コンパイラは

- 1 回目の解析で Map がクラス名であることを解析しておいて、
- 2 回目の解析で Game クラスの内部で参照されている Map の解析に 1 回目の解析結果（ Map がクラス名であること）を用いる

ことができます。しかし、1 回の解析だけでコンパイルを終えるには、 Game クラスの解析の前に Map がクラス名（データ型名）であることを知っておく必要があります。前方宣言はそのための宣言です。なお、メンバー変数 map はポインタであるため、 Map がクラス名であることさえ分かれば十分であり、 Map クラスの詳細な定義は不要です。何故ならば、ポインタに必要なメモリサイズは、64bit CPU の場合、どんなクラスであっても常に 8 バイト（= 64bit）であり、それさえ分かれば Game オブジェクトのメモリサイズが計算できるからです。詳細を書かないことから前方宣言は不完全なクラス宣言とも呼ばれます。

現在はほとんど意識されることがありませんが、コンピュータの実行速度 – それは CPU の速度、メモリの容量、メモリ、HD のアクセス速度に依存します – が十分でなかった 1990 年代以前にはコンパイルは非常に時間の掛かる処理でした。1000 行のプログラムのコンパイルに数分掛かるのは通常のことでした[1]よって当時は 2 パスコンパイルを行うなどは論外であって、1 パスコンパイルができることがプログラミング言語設計の必須条件でした。しかし最近のコンピュータは十分高速であることから、言語の設計は 2 パスコンパイル（あるいはマルチパスコンパイル）を前提としているものに変わってきています。後章で学ぶ Java では前方宣言は不要です。

結論を言えば、古い言語仕様の C++では 1 パスコンパイルが前提です。必要に応じて前方宣言を付けてください。

[1]人は 10 秒待たされるとイライラし始めるとも言われています（参考：シチズン、ビジネスパーソンの待ち時間意識調査、2003 年）

9.2.2　前方参照

前方宣言に似た用語に**前方参照**（forward reference）があります。これは「前方から後方の定義を参照する」ことを意味します。前項の前方宣言は「前方参照を回避するために、わざわざ前方に宣言を置く」ことですので、その違いに注意してください。

もし図 9.2 のプログラムに前方宣言を置かないならば、前方の Game クラスの定義は、後方の Map クラスの定義を前方参照していることになります。

インライン記述を用いた以下の小さなプログラムも前方参照の例です。

```
class ClassA {
  public:
    void init() { a = 10; }   // ２行後の a の宣言を参照している

    int a;
};
```

メンバー関数 init() の本体内から後方に宣言されているメンバー変数 a を参照しています。

ややこしいのですが、C++ は 1 パスコンパイルを行いますが、例外的にひとつのクラス内については 2 パスコンパイルを行うため、上の例のような前方参照は許しています。

9.2.3　オブジェクトの相互参照

Game オブジェクトが全体を統括するオブジェクトです。このオブジェクトはゲームの構成要素であるマップ、プレイヤー、モンスター 2 体のオブジェクトをポインタで参照しています。プレイヤーは Game オブジェクトを参照しているため、マップデータを参照したい場合には、 game → map → data とリンクを辿っていきます（図 9.4（126 ページ）参照）。

このようなオブジェクト間相互の参照はオブジェクト指向設計上はあまり良くない構造と考えられています。とは言え、完全に相互参照は完全には排除できない場合もあり、悩ましいところです。このテキストではこの相互参照については妥協することとします。

9.2.4　static 修飾子（その 2）

この解説は 6.3.3 項（73 ページ）の続きです。

図 9.2 の Game クラスの定義に以下があります。

```
    static char getKeyChar();         // キー入力の static 関数
    static int signum(int a);         // 浮動関数の static 関数
```

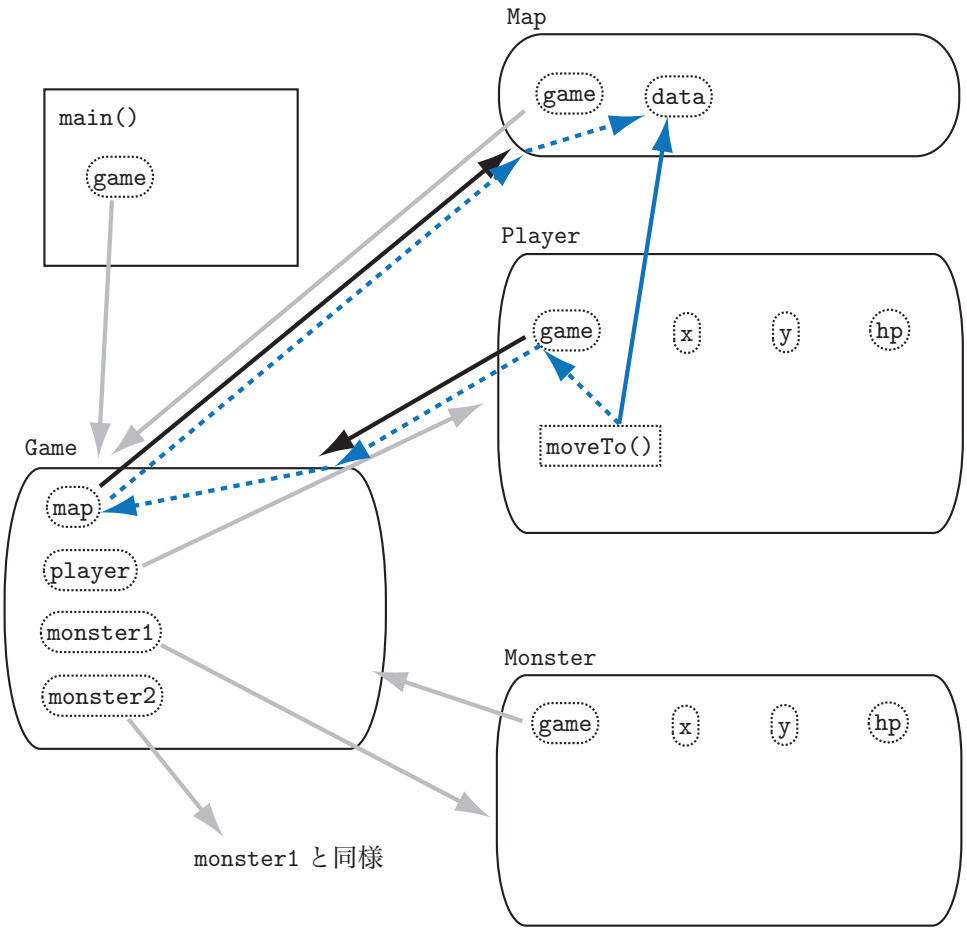


図 9.4: Player オブジェクトの moveTo() から Map オブジェクトの data へアクセスする方法：点線矢印を辿ってアクセスします。

メンバー関数に static が付いたものです。関数と変数の違いを除けば、 static 修飾されたメンバー変数（6.3.3 項参照）と類似した扱いになります。つまり、プログラム空間に唯一存在する、Game クラスそれ自身のオブジェクトに所属する関数であるという点です。このような関数を C++ では静的メンバー関数（static member function）と呼びます。

9.3　main 関数およびゲームクラスの実装

main() 関数および Game クラスの実装は図 9.5（127 ページ）です。

main() 関数の

```
int main(int argc, const char * argv[]) {
    Game *game = new Game;                    // ゲームの生成
    game->init();                             // ゲームの初期化
    game->mainLoop();                         // ゲーム内のループ
    return 0;
}

void Game::init() {                           // ゲームの初期化
    map = new Map;                            // マップの生成
    map->init(this);                          // マップの初期化
    player = new Player;                      // プレイヤーの生成
    player->init(this);                       // プレイヤーの初期化
    monster1 = new Monster;                   // モンスターの生成
    monster1->init(this, 1, Map::YSIZE-2);    // モンスターの初期化
    monster2 = new Monster;                   // モンスターの生成
    monster2->init(this, Map::XSIZE-2, 1);    // モンスターの初期化
}

void Game::print() {
    ** 関数本体は前章の printAll() の本体と同じ **
}

void Game::mainLoop() {
    ** 前章と同じ、ただし printAll() を print() に置換 **
}
void Game::turn() {
    ** 前章と同じ、ただし printAll() を print() に置換 **
}
char Game::getKeyChar() {
    ** 関数本体は前章の getKeyChar() の本体と同じ **s
}

int Game::signum(int a) {
    ** 関数本体は前章の signum() の本体と同じ **
}
```

図 9.5: グローバルを全て削除したダンジョンゲームのプログラム（ main() および Game クラス）

```
    Game *game = new Game;     // ゲームの生成
```

で、 Game オブジェクトを動的オブジェクトとして生成し、そのポインタを main() 関数内に保持します。次に

```
    game->init();              // ゲームの初期化
```

の中で、マップ、プレイヤー、モンスターの各オブジェクトを生成し、初期化します。その具体的

な内容は Game::init() の通りです。

9.3.1　自分自身を指し示すポインタ： this

図 9.5 の Game::init() の中の以下に注意してください。

```
map->init(this);                          // マップの初期化
...
player->init(this);                       // プレイヤーの初期化
...
monster1->init(this, 1, Map::YSIZE-2);    // モンスターの初期化
...
monster2->init(this, Map::XSIZE-2,  1);    // モンスターの初期化
```

引数で this を渡しています。この this は、当該オブジェクト自身のメモリアドレスを表す C++ のキーワードです。メンバー関数の中だけで利用可能です。

上に述べたように、マップ、プレイヤー、モンスターは Game オブジェクトを参照できる必要があります。そこで、 Game オブジェクトのメモリアドレス this を引数として渡しているのです。

this は Java でも同様に利用可能です。this は、オブジェクトが「自分自身」に言及するときに必須の、オブジェクト指向プログラミングの重要な変数です。後章でも随所で利用します。

9.4　マップクラス/プレイヤークラス/モンスタークラスの実装の変更

プログラムの変更は以下の 3 点です。

1. Map クラス、Player クラス、Monster クラスの初期化関数の引数には Game オブジェクトのアドレスを受け取る引数が追加されています。

   Map クラスならば

   ```
   void Map::init(Game *game_) {
       game = game_;
       ...
   ```

   Player クラスならば

```
void Player::init(Game *game_) {
    game = game_;
    ...
```

Monster クラスならば

```
void Monster::init(Game *game_, int x_, int y_) {
    game = game_;
    ...
```

と、メンバー変数 game の初期化処理を追加します。

2. 変数 map、 player、 monster1、 monster2 がグローバル変数から Game クラスのメンバー変数に変更されたことにより、

   - “map->” を “game->map->” へ変更、
   - “player->” を “game->player->” へ変更、
   - “monster1->” を “game->monster1->” へ変更、
   - “monster2->” を “game->monster2->” へ変更

   に変更します。

3. グローバル関数 getKeyChar()、 signum() が Game クラスの静的メンバー関数に変更されたことにより、これらの関数の呼び出しの直前にスコープ修飾子 Game:: を追加します（6.5.1 項（76 ページ）参照)。つまり、getKeyChar() の場合は、前章の

```
    switch (getKeyChar()) {
```

   をこの章では

```
    switch (Game::getKeyChar()) {    // Game::を追加する
```

   に変更します。

煩雑ですから個々の変更箇所は示しません。

9.5　付録：この章の全プログラムリスト

```
//宣言部
#include <iostream>
using namespace std;

//クラス名の宣言
class Map;                          // クラス名の前方宣言
class Player;                       // 同上
class Monster;                      // 同上

class Game {                        // ゲーム全体を統括するクラス
  public:
    Map *map;                       // ゲームの中のマップ
    Player *player;                 // ゲームの中のプレイヤー
    Monster *monster1;               // ゲームの中のモンスター
    Monster *monster2;               // ゲームの中のモンスター

    void init();                    // ゲームの初期化
    void print();                   // ゲームの表示
    void mainLoop();                // ループ
    void turn();                    // ひとつのターン
    static char getKeyChar();       // キー入力の static 関数
    static int signum(int a);       // 浮動関数の static 関数
};

class Map {
  public:
    static const int ROCK = 0;
    static const int EMPTY = 1;
    static const int GOAL = 2;

    static const int XSIZE = 13;
    static const int YSIZE = 11;

    Game *game;                     // ゲームへのポインタ
    int data[XSIZE][YSIZE];

    void init(Game *game_);  // Game オブジェクトのアドレスを受け取って初期化
    void cellPrint(int x, int y);
    void print();
};

class Player {
  public:
    static const int INIT_HP = 100;

    Game *game;                    // ゲームへのポインタ
    int x, y;
    int hp;

    void init(Game *game_);  // Game オブジェクトのアドレスを受け取って初期化
    bool cellPrint(int x, int y);
    void print();
    bool moveTo(int dx, int dy);
    void attackTo(int dx, int dy);
    void action();
};
```

```
1
2  class Monster {
3    public:
4      static const int INIT_HP = 150;
5
6      Game *game;                        // ゲームへのポインタ
7      int x, y;
8      int hp;
9
10     void init(Game *game_, int x, int y);
11                               // Game オブジェクトのアドレスを受け取って初期化
12     bool cellPrint(int x, int y);
13     void print();
14     bool moveTo(int dx, int dy);
15     void attackTo(int dx, int dy);
16     void action();
17 };
18
19 //プロトタイプ宣言
20
21 //グローバル宣言
22
```

```
//main 関数およびゲームクラスの実装
int main(int argc, const char * argv[]) {
    Game *game = new Game;                    // ゲームの生成
    game->init();                             // ゲームの初期化
    game->mainLoop();                         // ゲーム内のループ
    return 0;
}

void Game::init() {                           // ゲームの初期化
    map = new Map;                            // マップの生成
    map->init(this);                          // マップの初期化
    player = new Player;                      // プレイヤーの生成
    player->init(this);                       // プレイヤーの初期化
    monster1 = new Monster;                   // モンスターの生成
    monster1->init(this, 1, Map::YSIZE-2);    // モンスターの初期化
    monster2 = new Monster;                   // モンスターの生成
    monster2->init(this, Map::XSIZE-2, 1);    // モンスターの初期化
}

void Game::print() {
    map->print();
    player->print();
    monster1->print();
    monster2->print();
    cout << endl;
}

void Game::turn() {
    player->action();
    monster1->action();
    monster2->action();
    print();
}

void Game::mainLoop() {
    print();
    while (true) {
        turn();
    }
}

char Game::getKeyChar() {
    char c;
    cout << "? " << flush;
    cin >> c;
    return c;
}

int Game::signum(int a) {
    if (a < 0) return -1;
    else if (a > 0) return 1;
    else return 0;
}
```

```
//マップクラスの実装
void Map::init(Game *game_) {
    game = game_;
    for (int x = 0; x < XSIZE; x++) {
        for (int y = 0; y < YSIZE; y++) {
            if ((x == 0)||(x == XSIZE-1)||
               (y == 0)||(y == YSIZE-1)||
               ((abs(x-XSIZE/2) <= 1)&&
                (abs(y-YSIZE/2) <= 1))) {

                data[x][y] = ROCK;
            }
            else { data[x][y] = EMPTY; }
        }
    }
    data[XSIZE-2][YSIZE-2] = GOAL;
}

void Map::print() {
    for (int y = 0; y < YSIZE; y++) {
        for (int x = 0; x < XSIZE; x++) {
            if (game->player->cellPrint(x,y)) { continue; }
            if (game->monster1->cellPrint(x,y)) { continue; }
            if (game->monster2->cellPrint(x,y)) { continue; }
            cellPrint(x,y);
        }
        cout << endl;
    }
}

void Map::cellPrint(int x, int y) {
    int e = data[x][y];
    if (e == ROCK)       { cout << '#' << ' '; }
    else if (e == EMPTY) { cout << '.' << ' '; }
    else if (e == GOAL)  { cout << 'G' << ' '; }
    else                 { cout << 'E' << ' '; }
}
```

```
//プレイヤークラスの実装
void Player::init(Game *game_) {
    game = game_;
    x = 1; y = 1;
    hp = INIT_HP;
}

void Player::print() {
    cout << "Player : (" << x << "," << y << ") "
         << hp << endl;
}

bool Player::cellPrint(int x_, int y_) {
    if ((x == x_)&&(y == y_)) {
        cout << '@' << ' ';
        return true;
    }
    return false;
}

bool Player::moveTo(int dx, int dy) {
    int targetX = x+dx, targetY = y+dy;

    if (game->map->data[targetX][targetY] == Map::ROCK) { return false; }
    if ((targetX == game->monster1->x)&&(targetY == game->monster1->y)) {
        return false;
    }
    if ((targetX == game->monster2->x)&&(targetY == game->monster2->y)) {
        return false;
    }

    x = targetX; y = targetY;
    return true;
}

void Player::attackTo(int dx, int dy) {
    int targetX = x+dx, targetY = y+dy;
    if ((targetX == game->monster1->x)&&(targetY == game->monster1->y)) {
        game->monster1->hp -= 20;
    }
    if ((targetX == game->monster2->x)&&(targetY == game->monster2->y)) {
        game->monster2->hp -= 20;
    }
}

void Player::action() {
    int dx, dy;
    switch (Game::getKeyChar()) {
        case 'z': dx = -1; dy = +1; break;
        case 'x': dx =  0; dy = +1; break;
        case 'c': dx = +1; dy = +1; break;
        case 'a': dx = -1; dy =  0; break;
        case 'd': dx = +1; dy =  0; break;
        case 'q': dx = -1; dy = -1; break;
        case 'w': dx =  0; dy = -1; break;
        case 'e': dx = +1; dy = -1; break;
        default:  dx =  0; dy =  0;
    }
```

```
1
2     if (!moveTo(dx,dy)) { attackTo(dx,dy); }
3 }
```

```
//モンスタークラスの実装
void Monster::init(Game *game_, int x_, int y_) {
    game = game_;
    x = x_; y = y_;
    hp = INIT_HP;
}

void Monster::print() {
    cout << "Monster: (" << x << "," << y << ") "
         << hp << endl;
}

bool Monster::cellPrint(int x_, int y_) {
    if ((x == x_)&&(y == y_)) {
        cout << 'M' << ' ';
        return true;
    }
    return false;
}

bool Monster::moveTo(int dx, int dy) {
    int targetX = x+dx, targetY = y+dy;

    if (game->map->data[targetX][targetY] == Map::ROCK) { return false; }
    if ((targetX == game->player->x)&&(targetY == game->player->y)) {
        return false;
    }
    if ((targetX == game->monster1->x)&&(targetY == game->monster1->y)) {
        return false;
    }
    if ((targetX == game->monster2->x)&&(targetY == game->monster2->y)) {
        return false;
    }

    x = targetX; y = targetY;
    return true;
}

void Monster::attackTo(int dx, int dy) {
    int targetX = x+dx, targetY = y+dy;
    if ((targetX == game->player->x)&&(targetY == game->player->y)) {
        game->player->hp -= 30;
    }
}

void Monster::action() {
    int dx = Game::signum(game->player->x - x);
    int dy = Game::signum(game->player->y - y);

    if (!moveTo(dx,dy)) { attackTo(dx,dy); }
}
```

1

1

第10章　マップ、プレイヤー、モンスターの初期化をオブジェクト指向風にする：コンストラクタへの置き換え

ここまでオブジェクトの初期化にメンバー関数 `init()` を用いてきました。しかし初期化はオブジェクト指向では非常に重要な処理であって、全てのオブジェクト指向プログラミング言語には初期化処理には専用の仕組みが用意されています。それが **コンストラクタ**（constructor[1]）です。この章ではメンバー関数 `init()` をコンストラクタに置き換えます。

初期化は何故、重要なのでしょうか。

たとえば `Player` オブジェクトを考えましょう。`Player` オブジェクトに必要なメモリは 20 バイト（ `game` ポインタに 8 バイト、 `x`、 `y`、 `hp` にそれぞれ 4 バイト）です。しかし、20 バイトのメモリが確保できたとしてもそれだけでは `Player` オブジェクトを作ったことになりません。メモリの内容がプレイヤーに相応しい値に設定されている必要があります。プレイヤーが岩の上に配置されたり、HP の初期値が 0 であっては不都合です。正しい初期化はオブジェクトを正常に動作させるための第一歩であり、極めて重要なのです。

10.1　簡単な例題

以下は、メンバー関数 `init()` でオブジェクトを初期化するクラス定義の例です。

```
class ClassA {
  public:
    int   a;
    void init() { a = 10; }
    void inc() { a++; }
};
```

実際に動的オブジェクトを作るときには

[1]構築子などと訳されます。また、英文では constructor のことを ctor などと略す場合もあります。

```
    ...
    ClassA  *objectA = new ClassA;
    objectA->init();
    ....
```

などと書くのが前章の初期化の方法でした。

これに対して、コンストラクタを用いるクラス定義は以下の通りです。

```
class ClassA {
  public:
    int   a;
    ClassA() { a = 10; }      // コンストラクタ
    void inc() { a++; }
};
```

C++のコンストラクタ（Java も同様）は、クラス名と同じ名前を持ちます。コンストラクタの戻り値はそのオブジェクトそれ自身ですから、戻り値のデータ型を記述する必要はありません。コンストラクタ本体に `return` 文を書く必要もありません（書いても構いませんが)。

オブジェクトを作るときには

```
    ...
    ClassA  *objectA = new ClassA;
    ....
```

の 1 行だけで済みます。あるいは以下もこれと等価です。

```
    ...
    ClassA  *objectA = new ClassA();
    ....
```

なお厳密に言えば、最初の `init()` を用いるプログラムと 2 番目のコンストラクタを用いるプログラムとではコンピュータの動作はわずかに異なります。というのは、 `init()` を用いるプログラムは C++処理系内部では以下のクラス定義は解釈されます。

```
class ClassA {
  public:
    int   a;
    ClassA() { a = 0; }        // デフォルトコンストラクタ
    void init() { a = 10; }
    void inc() { a++; }
};
```

つまりコンストラクタをユーザ定義しない場合、オブジェクト生成時に暗黙の初期化を行うコストラクタが呼び出される仕様となっています。このような暗黙のコンストラクタを **デフォルトコンストラクタ**（default constructor）と呼びます。暗黙には文字型、数値型のメンバー変数は 0 で初期化され、ポインタ型のメンバー変数はヌルアドレス `NULL`[2] で初期化されます。前章までのプログラムでは気づかないままにデフォルトコンストラクタが実行されており、その分、少しだけ無駄は動作をしていました。

引数の有る無しに関わらず、コンストラクタがユーザ定義された場合には、仮引数と実引数が整合するならば、そのコンストラクタの内容で初期化を行います。実引数とマッチするコンストラクタが定義されていない場合にはコンパイルエラーとなります。

仮引数と実引数の違いについての注意　用語に関する小さな解説です。関数の引数（argument, parameter）には仮引数（formal argument, formal parameter）と実引数（actual argument, actual parameter）の 2 種類があります。前者は、関数定義プログラムに現れる引数のことを指します。後者は、関数呼び出し時の引数値のことを指します。なお、引数は湯桶（ゆとう）読みで「ひきすう」と読むのがほぼ正解です[3]。

10.2　宣言部の改造

図 10.1（142 ページ）が宣言部のプログラムです。全てのクラスで `init()` 関数をコンストラクタ に置き換えます。

10.3　`main` およびゲームクラス、マップクラス、プレイヤークラス、モンスタークラスの実装の変更

図 10.2（143 ページ）、図 10.3（143 ページ）、図 10.4（144 ページ）、図 10.5（144 ページ）は、`main` 関数および各クラスの実装の変更点を示すものです。変更は機械的です。`init()` をコンストラクタに置き換えるだけです。詳細は省きます。

[2] `NULL` の値は 64bit 型整数値の 0 です。
[3]「いんすう」でも間違いではないとの見解も一部にあるようですが、避けた方が無難です。プレゼンテーション時に「いんすう」と読むとプログラミング独学者、初心者と思われます。

```
class Map;
class Player;
class Monster;

class Game {
    ** void init() を以下に置換、その他の変更なし **
    Game();
};

class Map {
    ** void init(Game *game_) を以下に置換、その他の変更なし **
    Map(Game *game_);
};

class Player {
    ** void init(Game *game_) を以下に置換、その他の変更なし **
    Player(Game *game_);
};

class Monster {
    ** void init(Game *game_, int x, int y) を以下に置換、その他の変更なし **
    Monster(Game *game_, int x, int y);
};
```

図 10.1: コストラクタを導入したダンジョンゲームのプログラム（宣言部）: 図 9.2（122 ページ）、図 9.3（123 ページ）参照

```
int main(int argc, const char * argv[]) {
    Game *game = new Game;              // Game::Game() の呼び出し
    game->mainLoop();
    return 0;
}

Game::Game() {
    map = new Map(this);               // Map::Map(Game*) の呼び出し
    player = new Player(this);         // Player::Player(Game*) の呼び出し
    monster1 = new Monster(this, 1, Map::YSIZE-2);
                            // Monster::Monster(Game*, int, int) の呼び出し
    monster2 = new Monster(this, Map::XSIZE-2, 1);
                            // Monster::Monster(Game*, int, int) の呼び出し
}

void Game::print() { ** 前章から変更なし** }

void Game::turn() { ** 前章から変更なし** }

void Game::mainLoop() { ** 前章から変更なし** }

char Game::getKeyChar() { ** 前章から変更なし** }

int Game::signum(int a) { ** 前章から変更なし** }
```

図 10.2: コストラクタを導入したダンジョンゲームのプログラム（ main() および Game クラス）：図 9.5（127 ページ）参照

```
Map::Map(Game *game_) {
    ** 関数本体は前章の Map::init(Game*) の本体と同じ **
}

void Map::cellPrint(int x, int y) { ** 前章から変更なし** }

void Map::print() { ** 前章から変更なし** }
```

図 10.3: コストラクタを導入したダンジョンゲームのプログラム（マップ）

```
Player::Player(Game *game_) {
    ** 関数本体は前章の Player::init(Game*) の本体と同じ **
}

bool Player::cellPrint(int x2, int y2) { ** 前章から変更なし** }

void Player::print() { ** 前章から変更なし** }

bool Player::moveTo(int dx, int dy) { ** 前章から変更なし** }

void Player::attackTo(int dx, int dy) { ** 前章から変更なし** }

void Player::action() { ** 前章から変更なし** }
```

図 10.4: コストラクタを導入したダンジョンゲームのプログラム（プレイヤー）

```
Monster::Monster(Game *game_, int x_, int y_) {
    ** 関数本体は前章の Monster::init(Game*, int, int) の本体と同じ **
}

bool Monster::cellPrint(int x2, int y2) { ** 前章から変更なし** }

void Monster::print() { ** 前章から変更なし** }

bool Monster::moveTo(int dx, int dy) { ** 前章から変更なし** }

void Monster::attackTo(int dx, int dy) { ** 前章から変更なし** }

void Monster::action() { ** 前章から変更なし** }
```

図 10.5: コストラクタを導入したダンジョンゲームのプログラム（モンスター）

10.4　補足：デストラクタ

コンストラクタはオブジェクトを作るときに呼ばれる関数ですが、C++ではオブジェクトを消滅させるときに呼ばれる関数 **デストラクタ**（destructor[4]）もユーザ定義可能です。これによってオブジェクトが消滅するときに特別な処理を行うことが可能です。デストラクタは、クラス名の前にチルダ '~' を付けた名前を持ち、かつ引数のない関数です。引数がない理由は、一般にオブジェクトが消滅する時点を特定することは難しい[5]ため、デストラクタは引数を持つことができないものと規定されています。

この章の冒頭の簡単な例題にデストラクタを追加すると以下の通りです。オブジェクトが消滅するときに "deleted" とメッセージが出るようにプログラム化しました。

```
class ClassA {
  public:
    int   a;
    ClassA() { a = 10; }                      // コンストラクタ
    ~ClassA() { cout << "deleted" << endl; }  // デストラクタ
    void inc() { a++; }
};
```

しかし、8.1 節（108 ページ）にも述べたように、オブジェクトを破壊、消滅させる処理は高度な技術を要します。そこを詳細に議論することは講義テキストの範囲を超えますから、これ以上の議論は省略します。多くの場合、デストラクタについて知らなくても大きな問題にはなりません。実際、Java を含め、最近のプログラミング言語ではオブジェクトの破壊はシステムが自動的に行うものがほとんどです（これについては後に述べます）。

[4]破壊子、消滅子などと訳されます。また、英文では destructor のことを dtor などと略す場合もあります。
[5]オブジェクトが常に `delete` 演算で消滅するならば、消滅する時点を特定できます。しかし、オブジェクトが自動変数として生成された場合には制御がスコープを抜けた瞬間に自動的に消滅するため、消滅時点をプログラマが特定できません。

10.5　付録：この章の全プログラムリスト

```
//宣言部
#include <iostream>
using namespace std;

//クラス名の宣言
class Map;
class Player;
class Monster;

class Game {
  public:
    Map *map;
    Player *player;
    Monster *monster1;
    Monster *monster2;

    Game();
    void print();
    void turn();
    void mainLoop();
    static char getKeyChar();
    static int signum(int a);
};


class Map {
  public:
    static const int ROCK = 0;
    static const int EMPTY = 1;
    static const int GOAL = 2;

    static const int XSIZE = 13;
    static const int YSIZE = 11;

    Game *game;
    int data[XSIZE][YSIZE];

    Map(Game *game_);
    void print();
    void cellPrint(int x, int y);
};

class Player {
  public:
    static const int INIT_HP = 100;

    Game *game;
    int x, y;
    int hp;

    Player(Game *game_);
    void print();
    bool cellPrint(int x, int y);
    bool moveTo(int dx, int dy);
    void attackTo(int dx, int dy);
    void action();
```

```
};

class Monster {
  public:
    static const int INIT_HP = 150;

    Game *game;
    int x, y;
    int hp;

    Monster(Game *game_, int x, int y);
    void print();
    bool cellPrint(int x, int y);
    bool moveTo(int dx, int dy);
    void attackTo(int dx, int dy);
    void action();
};
```

```
//main 関数およびゲームクラスの実装
int main(int argc, const char * argv[]) {
    Game *game = new Game;              // Game::Game() の呼び出し
    game->mainLoop();
    return 0;
}

Game::Game() {
    map = new Map(this);               // Map::Map(Game*) の呼び出し
    player = new Player(this);         // Player::Player(Game*) の呼び出し
    monster1 = new Monster(this, 1, Map::YSIZE-2);
                            // Monster::Monster(Game*, int, int) の呼び出し
    monster2 = new Monster(this, Map::XSIZE-2, 1);
                            // Monster::Monster(Game*, int, int) の呼び出し
}

void Game::print() {
    map->print();
    player->print();
    monster1->print();
    monster2->print();
    cout << endl;
}

void Game::mainLoop() {
    print();
    while (true) { turn(); }
}

void Game::turn() {
    player->action();
    monster1->action();
    monster2->action();
    print();
}

char Game::getKeyChar() {
    char c;
    cout << "? " << flush;
    cin >> c;
    return c;
}

int Game::signum(int a) {
    if (a < 0) return -1;
    else if (a > 0) return 1;
    else return 0;
}
```

```
//マップクラスの実装
Map::Map(Game *game_) {
    game = game_;
    for (int x = 0; x < XSIZE; x++) {
        for (int y = 0; y < YSIZE; y++) {
            if ((x == 0)||(x == XSIZE-1)||
               (y == 0)||(y == YSIZE-1)||
               ((abs(x-XSIZE/2) <= 1)&&
                (abs(y-YSIZE/2) <= 1))) {
                data[x][y] = ROCK;
            }
            else { data[x][y] = EMPTY; }
        }
    }
    data[XSIZE-2][YSIZE-2] = GOAL;
}

void Map::print() {
    for (int y = 0; y < YSIZE; y++) {
        for (int x = 0; x < XSIZE; x++) {
            if (game->player->cellPrint(x,y)) { continue; }
            if (game->monster1->cellPrint(x,y)) { continue; }
            if (game->monster2->cellPrint(x,y)) { continue; }
            cellPrint(x,y);
        }
        cout << endl;
    }
}

void Map::cellPrint(int x, int y) {
    int e = data[x][y];
    if (e == ROCK) { cout << '#' << ' '; }
    else if (e == EMPTY) { cout << '.' << ' '; }
    else if (e == GOAL) { cout << 'G' << ' '; }
    else { cout << 'E' << ' '; }
}
```

```
//プレイヤークラスの実装
Player::Player(Game *game_) {
    game = game_;
    x = 1; y = 1;
    hp = INIT_HP;
}

void Player::print() {
    cout << "Player : (" << x << "," << y << ") "
         << hp << endl;
}

bool Player::cellPrint(int x_, int y_) {
    if ((x == x_)&&(y == y_)) {
        cout << '@' << ' ';
        return true;
    }
    return false;
}

bool Player::moveTo(int dx, int dy) {
    int targetX = x+dx, targetY = y+dy;

    if (game->map->data[targetX][targetY] == Map::ROCK) { return false; }
    if ((targetX == game->monster1->x)&&(targetY == game->monster1->y)) {
        return false;
    }
    if ((targetX == game->monster2->x)&&(targetY == game->monster2->y)) {
        return false;
    }

    x = targetX; y = targetY;
    return true;
}

void Player::attackTo(int dx, int dy) {
    int targetX = x+dx, targetY = y+dy;
    if ((targetX == game->monster1->x)&&(targetY == game->monster1->y)) {
        game->monster1->hp -= 20;
    }
    if ((targetX == game->monster2->x)&&(targetY == game->monster2->y)) {
        game->monster2->hp -= 20;
    }
}

void Player::action() {
    int dx, dy;
    switch (Game::getKeyChar()) {
        case 'z': dx = -1; dy = +1; break;
        case 'x': dx =  0; dy = +1; break;
        case 'c': dx = +1; dy = +1; break;
        case 'a': dx = -1; dy =  0; break;
        case 'd': dx = +1; dy =  0; break;
        case 'q': dx = -1; dy = -1; break;
        case 'w': dx =  0; dy = -1; break;
        case 'e': dx = +1; dy = -1; break;
        default:  dx =  0; dy =  0;
    }
```

```

    if (!moveTo(dx,dy)) { attackTo(dx,dy); }
}
```

```
//モンスタークラスの実装
Monster::Monster(Game *game_, int x_, int y_) {
    game = game_;
    x = x_; y = y_;
    hp = INIT_HP;
}

void Monster::print() {
    cout << "Monster : (" << x << "," << y << ") "
         << hp << endl;
}

bool Monster::cellPrint(int x_, int y_) {
    if ((x == x_)&&(y == y_)) {
        cout << 'M' << ' ';
        return true;
    }
    return false;
}

bool Monster::moveTo(int dx, int dy) {
    int targetX = x+dx, targetY = y+dy;

    if (game->map->data[targetX][targetY] == Map::ROCK) { return false; }
    if ((targetX == game->player->x)&&(targetY == game->player->y)) { return false; }
    if ((targetX == game->monster1->x)&&(targetY == game->monster1->y)) {
        return false;
    }
    if ((targetX == game->monster2->x)&&(targetY == game->monster2->y)) {
        return false;
    }

    x = targetX; y = targetY;
    return true;
}

void Monster::attackTo(int dx, int dy) {
    int targetX = x+dx, targetY = y+dy;
    if ((targetX == game->player->x)&&(targetY == game->player->y)) {
        game->player->hp -= 30;
    }
}

void Monster::action() {
    int dx = Game::signum(game->player->x - x);
    int dy = Game::signum(game->player->y - y);

    if (!moveTo(dx,dy)) { attackTo(dx,dy); }
}
```

1

1

第11章　Javaについて

前章まで C++で作ってきたダンジョンゲームを次章で Java に移植します。この章ではその準備として Java について概説します。この講義テキストの後章でも随所で C++と Java の違いについて解説しますので、ここではポイントを絞って述べることとします。

11.1　ハローワールド

まず、Java を全く知らない人のためにハローワールドのプログラムを示します。

```
class Test {
    public static void main(String[] args){
         System.out.println("Hello World!");
   }
}
```

Java も C/C++ と同様に `main` 関数からプログラムの実行が始まります。ただし、Java では全ての関数はクラス内に記述する必要がありますから、上のプログラムでは `Test` クラスに `main` 関数を置きました。キーワード `public` の意味はすでに 6.3.2 項（72 ページ）に述べた内容とおおむね同じですが、詳細は後章において述べます。キーワード `static` についても 9.2.4 節（125 ページ） に述べた内容と同じです。`main` 関数は特定のオブジェクトの動作を記述するものではないため、 `static` を付けます。

`main` 関数の仮引数（`String[] args`）は、C++の `main` 関数の仮引数（`int argc, const char * argv[]`）と同じ役割です。引数の個数が異なりますが、`argc` に相当する数は、Java では `args.length` で取得できます。詳細は省略します。

関数呼び出し `System.out.println()` は、Java での標準出力への出力を行うものです。上のプログラムでは `" "` で囲まれた文字列を出力しています。

`System.out.println()` の引数として、文字列を連接する式を与えることもできます。以下はその例です。ここに + は文字列どうしを連接する演算子として働きます。文字列でないデータ – この場合は整数値 — は文字列に自動変換されます。数式 `1+2` はそれが評価された後に文字列へ変換されます。

```
class Test {
    public static void main(String[] args){
         System.out.println(1 + " plus " + 2 + " is " + (1+2));
   }
}
```

System.out.println() の ln は改行（line）を表しますから、改行したくない場合には ln の付かない System.out.print() を用います。たとえば、以下のプログラムは上のプログラムと同じ出力結果を与えます。

```
class Test {
    public static void main(String[] args){
         System.out.print(1);
         System.out.print(" plus ");
         System.out.print(2);
         System.out.print(" is " + (1+2));
         System.out.println();          // 改行のみを行う
   }
}
```

なお、System.out.println() の System は、プログラム実行時のオペレーティングシステムとのやり取りを行うクラス[1]のことです。System.out は System に所属している標準出力へのストリーム処理を担当するオブジェクトを表します。println() はそのオブジェクトの出力処理関数です。

11.2　C++と Java の言語仕様の違い

C++の言語仕様は非常に複雑です。と言うのも、前身である C 言語との整合性を考慮し、C 言語の言語仕様をほぼ全て包含し、拡張 C 言語としても通用するような仕様になっているためです。C 言語の古い仕様を捨てなかった上、オブジェクト指向の機能をしっかりと導入したため、言語仕様が複雑で大きなものになっています。それに対して Java では C++ の仕様のうち、複雑すぎる仕様を切り捨てて比較的単純な仕様になっています。上の関係を講義担当者の主観で図式化したものが図 11.1 です。

C++と Java の言語仕様はコアの部分では非常に似ており、C++を学んだ人はわずかな学習で Java の基本構文を理解できます。

[1]参考：マンガで分かる Java 入門講座、第 11 章 5、https://manga.crocro.com/?cat=java （各項目が分かりやすくかかれています）

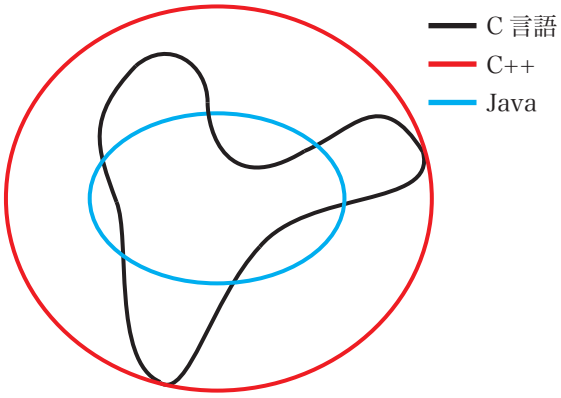


図 11.1: C 言語、C++、Java の言語仕様の印象（講義担当者の主観）をあえて図式化してみた：C 言語は古い言語であり、言語仕様が元々やや歪（いびつ）な形になっています。C++は拡張 C 言語としても機能するように C 言語の古い仕様を捨てなかった上、オブジェクト指向の機能をしっかりと導入したため、言語仕様が異常に大きくなっています。Java は C++ の仕様のうち、複雑すぎる仕様を切り捨てて程よい複雑さに抑え、比較的単純な仕様にまとめています。

11.2.1　全体構造

Java のプログラムで使用される変数、関数（メソッドと呼びます）は全ていずれかのクラスに所属している必要があります。クラスに所属しない C++のグローバル変数、グローバル関数のようなものは許されません。よって Java のプログラムは必ず

```
class ... {
    ....
}

class ... {
    ....
}

...
```

というクラスの記述の並びになります。クラスを記述する順番は任意で構いません。C++で説明した前方参照（9.2.2 項（125 ページ））を気にする必要はありません[2]。6.5.2 項（77 ページ）で述べたように、Java のプログラムはインライン記述のみ可能です。

11.2.2　関数の種類

Java では関数のことを**メソッド**（method）と呼びます。

[2]前方参照を気にしないといけないのは C/C++などの前世代のプログラミング言語です。

関数と呼ばない理由は、1980 年代に流行ったオブジェクト指向プログラミング言語 Smalltalk に由来すると思われます。Smalltalk ではオブジェクト間の「メッセージ」のやり取りで処理が進みます。メッセージを受け取ったオブジェクトはそれに関連する「メソッド」を実行します。メソッドという用語はこの辺りから来ています。わざわざ「メッセージ」と「メソッド」を分けるのは、後章で述べる多相性を意識しているためです。

Java のメソッドには 2 種類があり、呼び方こそ違うものの C++のメンバー関数と同等のものです。

インスタンスメソッド（instance method）　個々のオブジェクト（インスタンス）の操作に関する関数のことです。C++のメンバー関数に相当します。`static` で修飾されない関数がインスタンスメソッドです。コンストラクタはインスタンスメソッドの特殊なものであり、その点も C++と同等です。

クラスメソッド（class method）　個々のオブジェクトではなく、クラスに付随する関数のことです。C++の静的メンバー関数（9.2.4 項（125 ページ）参照）に相当します。`static` で修飾された関数がクラスメソッドです。呼び出す際には、C++ではスコープ解決演算子 `::` を用いて“クラス名`::`関数名”によって静的メンバー関数を特定しましたが、Java ではドット演算子 `.` を用いて“クラス名`.`メソッド名”によって特定します。

11.2.3　変数の種類

Java の変数は以下のように分類されています。関数同様、呼び方こそ異なる[3]ものの、C++のメンバー変数と同等です。

ローカル変数（local variable）　メソッド内で一時的に用いる変数です。C/C++と同様です。

インスタンス変数（instance variable）　C++のメンバー変数に相当します。C++のメンバー変数と Java のインスタンス変数では、正確に言えば初期化の方法がわずかに異なりますが、違いを気にするほどではなく、それ以外については同様に取り扱いができます。クラス内宣言され、`static` で修飾されない関数がインスタンス変数です。

クラス変数（class variable）　C++の静的メンバー変数に相当します。クラス内宣言され、`static` で修飾された変数がクラス変数です。

[3]一部の書籍、ホームページでは、Java であってもメンバー変数、静的メンバー変数という呼び方を用いているものがあります。

11.3　オブジェクトの扱いの違い

この節は重要です。C++と Java のオブジェクトの扱いには大きな違いがあります。

11.3.1　Java のデータ型

Java のデータ型は、**プリミティブ型**（primitive type）と**参照型**（reference type）[4]の 2 種類に大別されます。

プリミティブ型は、`byte`、`short`、`int`、`long`、`float`、`double`、`boolean`、`char` の 8 個のデータ型です。その使用方法は C/C++とほぼ同じですので解説を省略します。なお、Java には符号無し（`unsigend`）の数値はありません。符号の有無の使い分けがプログラマを混乱させ、バグを生み易いためです。

参照型は、プリミティブ型以外の、クラス記述によって定義されるデータ型や配列型、`String` 型（Java の文字列）などです。参照型のデータは、全て動的メモリ領域に作られるオブジェクトです。そのため、参照型をオブジェクト型と呼ぶこともあります。

参照型変数は、C/C++のポインタ型変数と似ています。ただし安全性のため（バグの回避のため）、C/C++のポインタに比べて機能が強く制限されており、オブジェクトを参照する（ポイントする）以外の機能を持ちません。

注意　Java の「参照」は勘違いしやすい概念として、しばしば物議を醸します。特に混乱するのは、C++にも「参照」という概念があることです。Java と C++の「参照」は似ていますが別のものと考えてください。この辺に関する解説記事はネットに溢れています。多くの人がもやもやするためでしょう[5]。

11.3.2　オブジェクトの生成

Java の場合

Java のプログラムでクラス `classA` がすでに定義されていると仮定しましょう。`ClassA` のオブジェクトの生成は、`new` 演算子とコンストラクタを用いて、

```
new ClassA()
```

[4]参照型をしばしばオブジェクト型、クラス型とも言います。
[5]参考：「Java の参照渡しと値渡しについてこの世で 1 番分かり易く解説してみた」、https://freelance-jak.com/technology/java/998/

の形式でのみ可能です。この式が実行されるとオブジェクトが動的メモリ領域に作成されます。

上の式の値はオブジェクトへの**参照**（reference）です。よって、以下のように参照型の変数を用意し、初期化を行うと、変数がそのオブジェクトを参照するように設定されます。

```
Java:    ClassA objectJ = new ClassA();
```

こう述べると、参照はポインタのような... と気づく人もいるでしょう。あながち間違いではありません。以下、その解説です。

C++の場合

上の Java でのオブジェクト生成は以下の C++の動的オブジェクトの生成のほとんど同じです。

```
C++:     ClassA *objectC = new ClassA();
```

C++の場合、`objectC` はポインタです。先ほどの Java の場合、変数に* は付いていませんが、実は `objectJ` もその実体はほぼポインタです。

ところで、C++の場合、オブジェクトの生成は以下でも可能です。

```
C++:     ClassA objectC();
```

この場合の `objectC` は自動変数またはグローバル変数です。自動変数ならば、このスコープから抜け出した瞬間にオブジェクトは自動的に消滅します。グローバル変数ならば、プログラムの実行開始直後にオブジェクトは自動的に作成され、プログラム終了時に自動的に消滅します。しかし、Java では自動変数、グローバル変数としてオブジェクトを生成することはできない仕様になってます。つまり、C++よりも言語仕様を制限している訳ですが、この制限によって Java は参照に関する実行時エラーが出にくい安全な言語になっています。次にそれを考察します。

11.3.3　ポインタ vs 参照

危険な C++のポインタ

C++のポインタは柔軟な処理に便利である一方、危険と隣り合わせです。いくつか事例を紹介します。

今、クラス ClassA はすでに定義されているものと仮定します。以下は、**ダングリングポインタ**(dangling pointer、ぶら下がりポインタ)[6]に delete を行なう C++のプログラム片です。

```
    ClassA *object = new ClassA();
    ...
    delete object;    // この時点で object はダングリングポインタになっている
    ...
    delete object;    // ダングリングポインタの delete はプログラムの暴走を招く
```

少し複雑なプログラムではやってしまいがちなミスです。

以下の例も危険です。

```
ClassA *f() {
    ClassA object1(); // ClassA 型のオブジェクト object1 を生成
    return &object1;  // object1 のメモリアドレスを戻り値にする
}

    ...
    ClassA *object2 = f(); // f() で生成したアドレスを代入
```

呼び出し側のポインタ object2 には object1 のアドレスが代入されます。しかし自動変数オブジェクト object1 は制御が関数 f() から戻った後、自動消滅しますから、 object2 の保持するアドレスにはもはやオブジェクトは存在せず、ダングリングポインタになっています。

C++では次のようなことも可能です。

```
    ClassA *object = new ClassA();
    ...
    object++;
```

ポインタのインクリメント演算は、そのポインタが配列を指している場合には有効ですが、上の場合はそうではありません。**ワイルドポインタ**（wild pointer、野良ポインタ)[7]になってしまいます。

Java の参照では上のような問題は生じません。何故ならば、Java には delete 演算子がありません。自動変数オブジェクトは生成できません。参照型変数の算術演算はできません。つまり、上に例示した C++の危険な状況を作り出せない仕様になっています[8]。なお、Java の「参照」の仕様は公開されていません。メモリアドレスに酷似していると考えて問題ありませんが、メモリアドレスそのものと同じか否かは明らかにされていません。

[6]既にオブジェクトが消滅した後のメモリ領域を指しているポインタ
[7]オブジェクトが存在しないアドレスを指しているポインタ
[8]危険な状況を容易に招く C++が悪い言語かと言えば、それは的外れです。C++は、プロフェッショナルのプログラマが自分の責任で巧妙なプログラムを作ることを許す言語という位置づけです。

Java の参照の問題点

Java の参照について唯一安全ではないとされているのはヌル値（null）の取り扱いです。

Java の参照値は、オブジェクトの参照値（ポインタでいうメモリアドレス）または null[9]です。null は、オブジェクト実体を参照していない状態ですから、ある参照型変数 object の値が null であるときに、object のメソッドを呼び出すと実行時エラーが発生し、プログラムの実行は強制中断されます。以下がそのプログラム片の例です。

```
class ClassA {
    void print() { System.out.println("ClassA!"); }
}
    ...
    ClassA object;
    if (0 < 1) { object = null; }
    else       { object = new ClassA(); }
    object.print();
    ....
```

上のプログラムにコンパイルエラーはありません。if 文の条件は必ず成り立ちますから、object の値は null です。null に対してメソッド print() を呼び出すと、以下のような実行時エラーが出ます。

```
Exception in thread "..." java.lang.NullPointerException:
Cannot invoke "ClassA.print()" because "<local1>" is null
```

この問題を解決すべく、Swift などの最近のプログラミング言語では、参照型変数の値が null を取りうるか否かを言語処理系が厳格に管理する機能が用意されています。

11.4　ガベージコレクション

既に述べたように、Java では不要になった動的オブジェクトをシステムが自動的に delete します。この処理を一般には**ガベージコレクション**（garbage collection、ゴミ集め）と呼びます。ゴミ（不要になった動的オブジェクト）を集めて消去する訳です。ガベージコレクションを行うソフトウェアを**ガベージコレクター**（garbage collector、ゴミ集め器）と呼びます。

たとえば次の C++のプログラム：

[9]参考までに、C++のヌル値は大文字の NULL です。

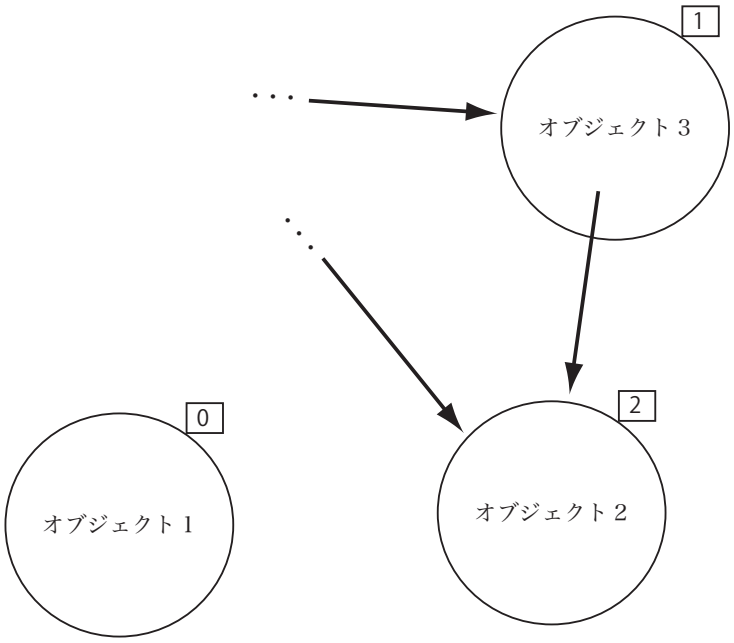


図 11.2: 参照カウントが付けられたオブジェクト：オブジェクトの右上にそのオブジェクトの参照数を表示しています。オブジェクト 1 は参照数が 0 であるから、ゴミであり、`delete` の候補です。

```
for (int i = 0; i < 1000000; i++) {
    ClassA *object = new ClassA();
    ...                             // object を用いた処理など
}
```

では、ループ本体でオブジェクトをひとつ生成し、ある処理を行います。オブジェクトの `delete` は行わないものと仮定します。そうすると、C++ではあっという間に動的メモリ領域を使い切ってしまいます。しかし、同様のプログラムを Java で実行した場合、Java ではループ実行の途中でガベージコレクターが自動的にゴミ集めを行うため、メモリを使い切ることはありません。

11.4.1　ガベージコレクションの原理

ガベージコレクターはどのような仕組みでゴミ集めを行なうのでしょうか。

最もシンプルな方法は、各オブジェクトにそれが参照されている数を保持させる方法です。図 11.2（163 ページ）はその例です。この例ではオブジェクト 1 は 0 回、オブジェクト 2 は 2 回、オブジェクト 3 は 1 回参照されています。この場合、オブジェクト 1 はどこからも参照されておらず、これを `delete` してもプログラムの実行に何の影響も及ぼしません。この参照回数は、プログラム中での参照の変更（参照型変数への代入）など起きる度にカウントアップ、カウントダウンす

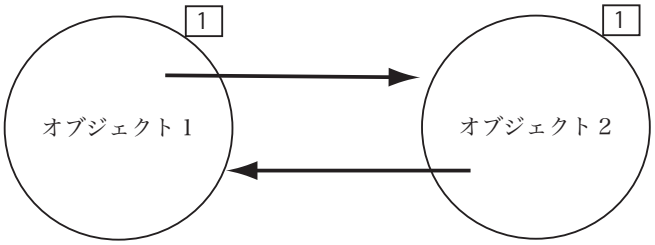


図 11.3: 相互参照のあるオブジェクト：相互に参照しあうとカウント数が 0 になりません。しかし、オブジェクト 1、オブジェクト 2 はそれ以外の参照がないため、ゴミです。

れば正しく追従できます。

この方法はシンプルです。しかし、図 11.3（164 ページ）のような、オブジェクト間の相互参照がある場合は大きな問題です。一旦、複数オブジェクト間にこの種の相互参照が成立すると、以降、これらのオブジェクトの参照数が 0 になることはなく、実際にはゴミであるにも関わらず、オブジェクトは動的メモリ領域に不当に居残ってしまいます。これを**メモリリーク**（memory leak）と呼びます。これを解決するにはアルゴリズムの改良が必要ですが、この講義テキストではそれを解説している余裕がありません。ガベージコレクションの議論は奥が深く、これだけで一冊の書籍になるほどの内容があります。興味のある人は以下などを参考にしてください。

リチャード・ジョーンズ、アントニー・ホスキング：“ガベージコレクション”、翔泳社（2016）.

11.4.2　ガベージコレクションの問題点

ガベージコレクションの欠点は、ガベージコレクターが動いているときには本来のプログラム実行が一時中断される点です。そのためプログラム実行が多少ギクシャクするかもしれません。

しかし、プログラマが危険な `delete` 処理を行なう必要がないことのメリットの方ははるかに大きいと考えられており、最近のほぼ全てのプログラミング言語がガベージコレクションを用いています。

11.5　Java の言語処理系

Java の言語処理系は 2.3.3 項（22 ページ）で紹介したコンパイラ・インタプリタ方式です。

図 11.4（165 ページ）は、図 2.6（22 ページ）を Java に当てはめた図です。Java のソースプログラムは Java コンパイラ `javac` でバイトコード（Java の中間言語）へ変換され、バイトコードは Java インタプリタ `java` によって実行されます。

次章では実際に Java のプログラムを動かすための環境設定について解説します。

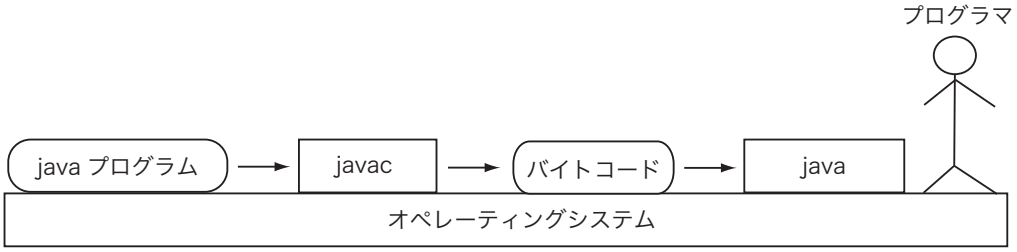


図 11.4: コンパイラ・インタプリタを用いるプログラミング環境（バッチモード）

1

1

第12章　Javaプログラミング環境のインストールと実行例

Java は様々な統合開発環境（IDE、Integrated Development Environment）— Eclipse がよく利用されています — によって実行可能です。たとえば「java プログラミング インストール」でネット検索してください。様々な実行方法があることが分かります。この授業では特定の環境でしか実行できないようなプログラムは扱いませんので、自由に環境構築してください。特定のプログラミング環境を指定しません。

しかしそうは言っても、全く自由だと途方に暮れる人も出てくるでしょうから、この章では前章の図 11.4 が実感できる素朴な方法を紹介します。以下の内容がこの講義テキストの想定する環境に最も近いものになります。参考にしてください。

`https://sukkiri.jp/technologies/devtools/vscode_java_win.html`

この章で紹介する事項は上の記事をほぼ言い換えたようなものになっています。作業は、以下の 3 ステップからなっています。

1. Java のコンパイラ、インタプリタ（Java Development Kit）のインストール
2. エディタ（VisualStudio Code）のインストール
3. 上の二つを用いたプログラム開発

以下、順に解説していきます。

12.1　Javaのコンパイラ、インタプリタのインストール

Java のコンパイラ、インタプリタは、オラクル社が提供するものが一般的[1]です。ただし、これは個人用には無償提供されていますが、商用その他には有償ですから、注意してください。

ネットで「jdk インストール windows」で検索してください（Mac の場合は「Windows」を「Mac」に置き換えます）。そうすると、以下のようなサイトがヒットするはずです。

[1]その他に、全くフリーな、Open Java Development Kit があります。

【初心者でもすぐわかる】JDK のインストール方法 Windows 編

https://eng-entrance.com/java-install-jdk-windows

Java JDK のインストール方法（Windows & Mac 対応）

https://codeforfun.jp/how-to-install-java-jdk-on-windows-and-mac/

上のようなサイトを参考に、JDK（Java Development Kit）をダウンロードし、インストールします。このとき、必ずパスの設定を行ってください。パスを設定しないと、コマンド名だけではコンパイル、実行ができません。

12.2　エディタのインストール

この講義テキストでは Java のプログラムを作るためのエディタも指定しません。手近にある使い慣れたもので問題ありません。

この授業では当初、Windows の「メモ帳」をエディタとして想定していましたが、「メモ帳」はタブが 8 文字固定でこの授業の想定する 4 文字と合わないため、やや不都合です。そこでもしエディタとして思いつくものがないならば、以下の Visual Studio Code（以下、VS Code と略します）を使ってみてください。

VS Code は、マイクロソフトが開発したプログラミング専用エディタです。統合開発環境の「Visual Studio」とは名前が似ていますが、別のソフトウェアです。中核部は軽量のエディタですが、拡張機能を追加することで、プログラマの目的にカスタマイズした環境を構築可能です。

Visual Studio Code のインストールは以下に載っています。

Visual Studio Code インストール手順＜ Windows 向け＞

https://sukkiri.jp/technologies/devtools/vscode_win.html

メニューなどの日本語化は以下などに解説されています。

Visual Studio Code を日本語化する

https://www.javadrive.jp/vscode/install/index4.html

12.3　プログラム開発の手順

12.3.1　Java ソースプログラムの作成

エディタで Java のソースプログラムを作成します。このとき、**ファイル名のサフィックスは必ず、“.java” に設定**してください。サフィックスが異なる場合、コンパイラがファイルを受け付け

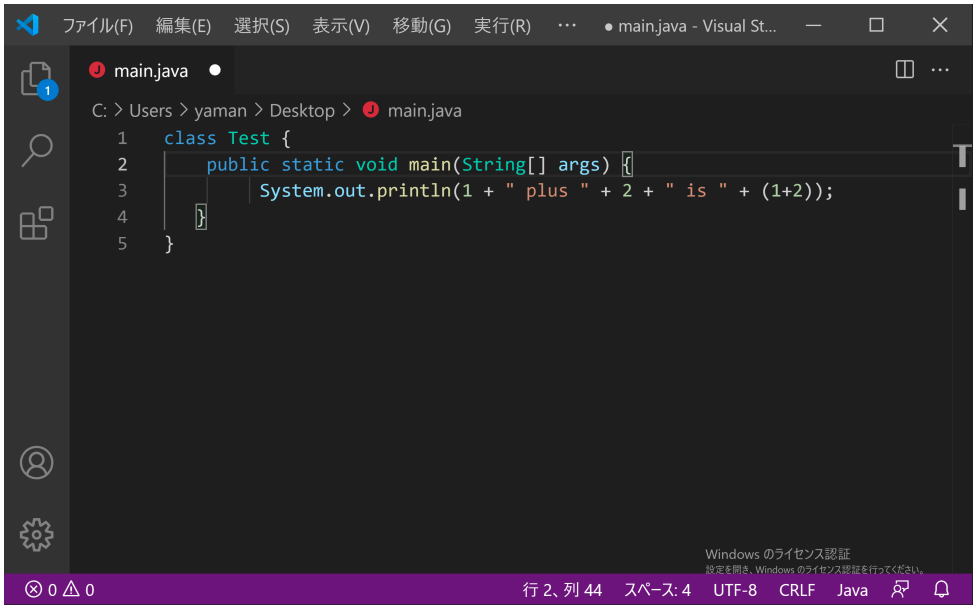


図 12.1: VS Code で Java のプログラムを編集中の画面

ません。図 12.1（170 ページ）は、VS Code で前章のプログラムを入力したところです。

12.3.2　コマンドプロンプト

Java のプログラムの実行は VS Code の上でも可能です[2]が、最も素朴な方法としてコンパイルと実行を Windows のシステムツールである「コマンドプロンプト」（command prompt）の上で行う方法を紹介します。コマンドプロンプトは Windows を CUI で操作するツールです。これは、Windows の左下のスタートボタンから図 12.2（171 ページ）のようにして起動できます。

図 12.3（172 ページ）は、起動直後のコマンドプロンプトです。右下に VS Code のウインドウが見えます。

次に、コマンドプロンプトを利用する上で最低限知っておくべきコマンドを以下に紹介します。なお、以下に出てくる**ディレクトリ**（directory）とは、GUI で言うフォルダーのことです。その時点で作業中のフォルダー（作業位置）を、以下では現在のディレクトリ（current directory）、当該ディレクトリ、カレントディレクトリなどと呼びます。作業位置を現在のフォルダーに含まれるフォルダーへ移動することを子ディレクトリへ移ると言います。逆に現在の作業フォルダー（作業位置）を外側のフォルダーへ移動することを親ディレクトリへ移ると言います。

`cd` ディレクトリ名 … 現在のディレクトリを、パラメータで指定したディレクトリへ移動します。

コマンド名 cd は “change directory” の略です。たとえば、

[2]その方法は自ら調べてください。

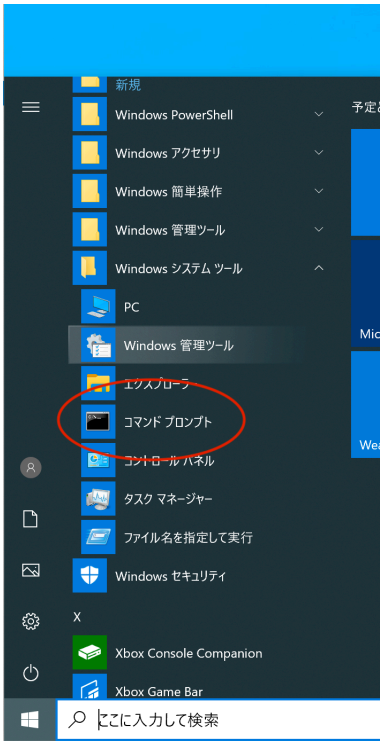


図 12.2: スタートボタンからのコマンドプロンプトの選択

```
> cd tmp
```

と入力すると、現在のディレクトリに含まれる子ディレクトリ tmp へ移動します。tmp が現在のディレクトリに含まれない場合にはエラーとなり、ディレクトリの移動は起きません。特殊なパラメータとして “..” は親ディレクトリを表します。よって、たとえば

```
> cd tmp
> cd ..
```

をコマンド入力すると、子ディレクトリ tmp へ移動し、次に親ディレクトリへ移動するため、結局、元に戻ります。

dir … 現在のディレクトリ内のファイルのリストを表示します。コマンドのパラメータは不要です。コマンド名は “directory” の略です。

del ファイル名 … パラメータで指定されたファイルを消去します。

図 12.4（173 ページ）に、コマンド cd と dir の使用例を示します。

その他のコマンドなどは以下などを参考にしてください。

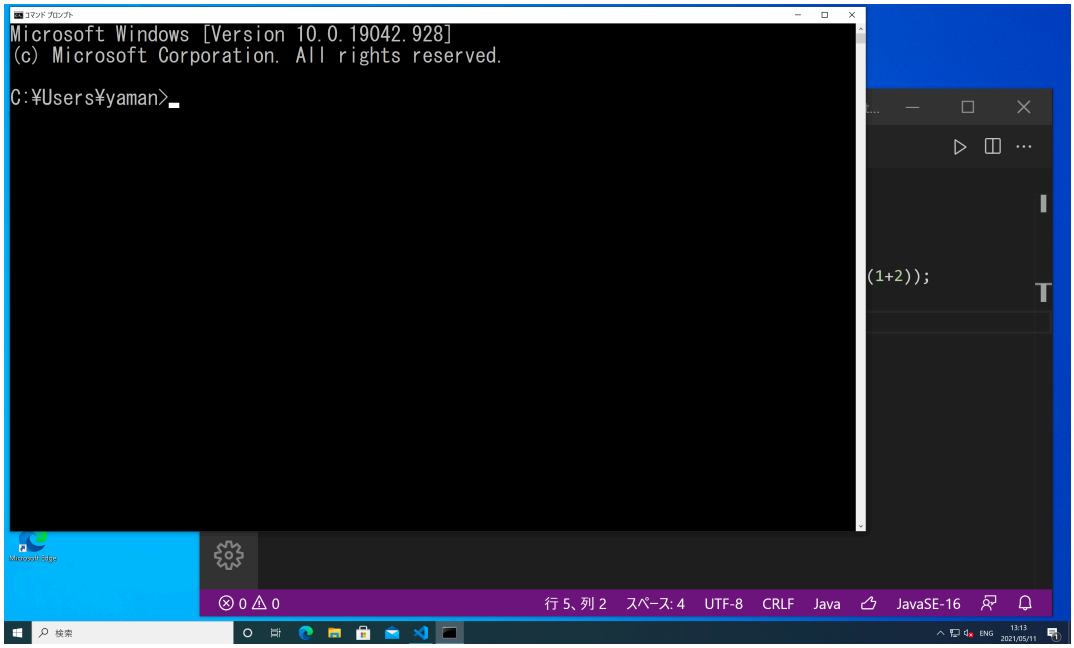


図 12.3: コマンドプロンプトのウインドウ：右下は VS Code のウインドウ

コマンドプロンプトの使い方とコマンド一覧

https://www.modis.jp/staffing/insight/column_28/

コンパイル

11.1 節（155 ページ）のプログラムをファイル “main.java” に格納していると仮定します。ディレクトリを Java ソースプログラムがある位置へ移動しておきます。そして以下を実行します。javac が Java コンパイラのコマンド名です。

```
> javac main.java
```

直後に、dir コマンドで現在のディレクトリのファイルリストを表示させてみます。もしファイル “Test.class” が生成されていればコンパイルは成功です。図 12.5（174 ページ）に、コンパイルし、“Test.class” の確認をする操作の様子を示します。

生成されたファイル “Test.class” の Test はクラス名を意味します。これは、図 11.1 のプログラムに定義されていたクラスをコンパイルした結果のバイトコードのファイルです。このファイルを**クラスファイル**（class file）と呼びます。ソースプログラムに複数のクラス定義が含まれる場合にはクラス毎に “クラス名.class” の名前のクラスファイルが生成されます。

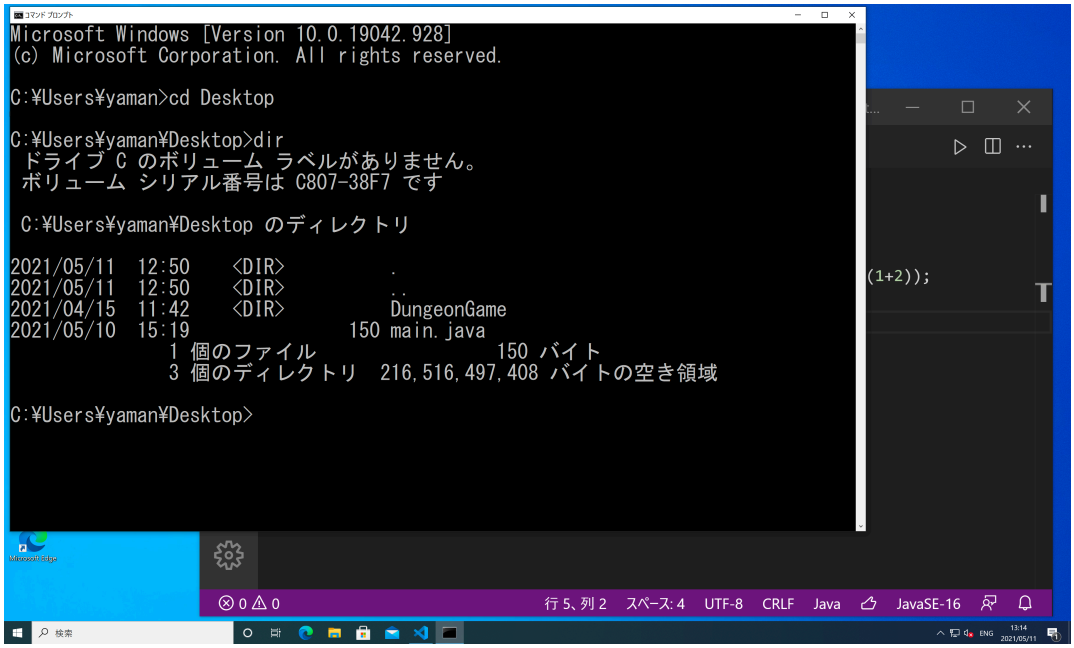


図 12.4: ディレクトリの移動とファイルリストの表示：(1) 現在のディレクトリから直下の Desktop へ移動、(2) 移動後のディレクトリの内容を表示

実行

インタプリタを起動するコマンドは以下です。コマンドのパラメータは、main 関数が定義されているクラス名であって、現在のディレクトリにそのクラスファイルが含まれるものです。その際、クラスファイルのサフィックス .class を付けてはいけません。コマンドパラメータはクラス名のみです。

```
> java Test
```

実行の様子を図 12.5（174 ページ）に示します。正しく “ 1 plus 2 is 3” が出力されています。

12.3.3　補足：単一のファイルのコンパイル、実行

前節の方法で、コンパイルには javac コマンドを用い、実行には java コマンドを用います。しかし、Java ソースファイルが単一のファイルだけからなる場合（この講義テキストの方法はそれに該当します）、コンパイルと実行を java コマンドだけで行うことが可能です。つまり、以下の 1 行だけで済みます。

```
> java main.java
```

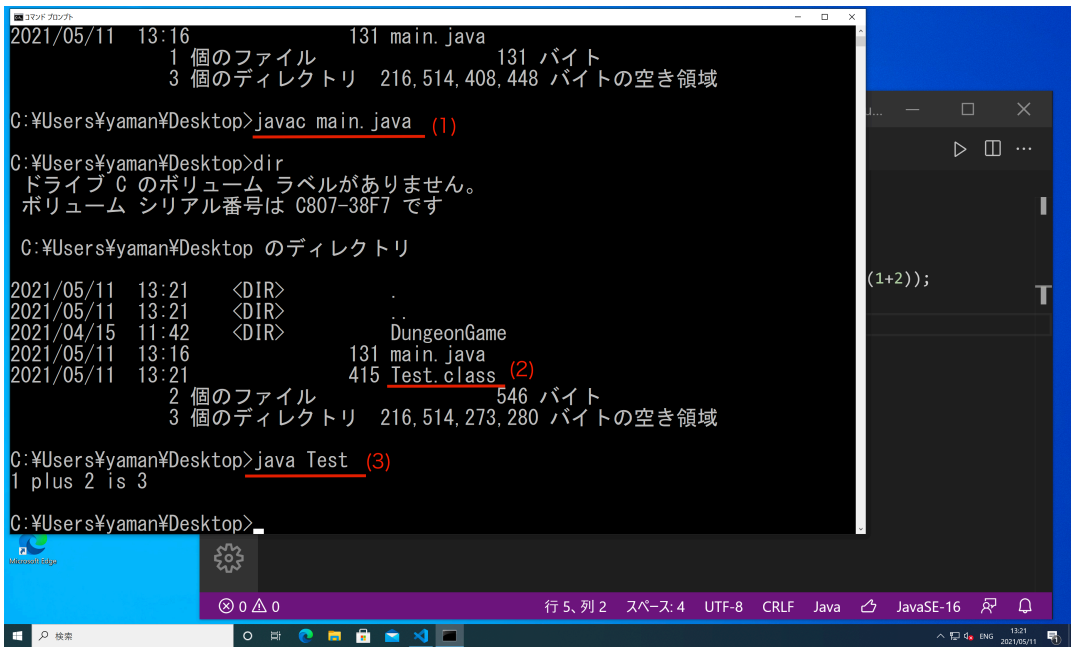


図 12.5: コンパイルと実行：(1) ソースプログラムのコンパイル、(2) ディレクトリにクラスファイル（バイトコード）`Test.class` が作成されていることを確認、(3) クラスファイルの実行

```
0000000    ca  fe  ba  be  00  00  00  3b  00  1d  0a  00  02  00  03  07
0000020    00  04  0c  00  05  00  06  01  00  10  6a  61  76  61  2f  6c
0000040    61  6e  67  2f  4f  62  6a  65  63  74  01  00  06  3c  69  6e
0000060    69  74  3e  01  00  03  28  29  56  09  00  08  00  09  07  00
0000100    0a  0c  00  0b  00  0c  01  00  10  6a  61  76  61  2f  6c  61
0000120    6e  67  2f  53  79  73  74  65  6d  01  00  03  6f  75  74  01
0000140    00  15  4c  6a  61  76  61  2f  69  6f  2f  50  72  69  6e  74
0000160    53  74  72  65  61  6d  3b  08  00  0e  01  00  0d  31  20  70
0000200    6c  75  73  20  32  20  69  73  20  33  0a  00  10  00  11  07
0000220    00  12  0c  00  13  00  14  01  00  13  6a  61  76  61  2f  69
0000240    6f  2f  50  72  69  6e  74  53  74  72  65  61  6d  01  00  07
0000260    70  72  69  6e  74  6c  6e  01  00  15  28  4c  6a  61  76  61
0000300    2f  6c  61  6e  67  2f  53  74  72  69  6e  67  3b  29  56  07
0000320    00  16  01  00  04  54  65  73  74  01  00  04  43  6f  64  65
0000340    01  00  0f  4c  69  6e  65  4e  75  6d  62  65  72  54  61  62
0000360    6c  65  01  00  04  6d  61  69  6e  01  00  16  28  5b  4c  6a
0000400    61  76  61  2f  6c  61  6e  67  2f  53  74  72  69  6e  67  3b
0000420    29  56  01  00  0a  53  6f  75  72  63  65  46  69  6c  65  01
0000440    00  09  6d  61  69  6e  2e  6a  61  76  61  00  20  00  15  00
0000460    02  00  00  00  00  00  02  00  00  00  05  00  06  00  01  00
0000500    17  00  00  00  1d  00  01  00  01  00  00  00  05  2a  b7  00
0000520    01  b1  00  00  00  01  00  18  00  00  00  06  00  01  00  00
0000540    00  01  00  09  00  19  00  1a  00  01  00  17  00  00  00  25
0000560    00  02  00  01  00  00  00  09  b2  00  07  12  0d  b6  00  0f
0000600    b1  00  00  00  01  00  18  00  00  00  0a  00  02  00  00  00
0000620    03  00  08  00  04  00  01  00  1b  00  00  00  02  00  1c
0000637
```

図 12.6: `Test.class` の 16 進ダンプリスト

12.4　補足：クラスファイルの中を覗（のぞ）いてみる

クラスファイルはテキストファイルではありませんから、その内容を直接、プリントアウトして確かめることはできません。が、いくつかの手段で内容を知ることができます。ここでは覗く程度のことを試してみます。

12.4.1　16 進ダンプ出力

クラスファイルの内容を確認するひとつの方法は、それを 16 進数でダンプ出力してみることです。UNIX 環境ならば、`od` コマンドで可能です。図 12.6（175 ページ）は前節の `Test.class` の内容を 16 進ダンプ出力したものです。取り立てて見るべきところがないように思えますが、先頭の 4 バイトに注目しましょう。

```
0000000    ca  fe  ba  be
```

16 進数ですが、これは “cafe babe” と読めます。

```
Classfile /Users/yaman/Test.class
  Last modified 2021/04/20; size 415 bytes
  SHA-256 checksum 467e1871a088c827b7a2362b73331bf3aef08a884f0f77bf0a84546e8f7...
  Compiled from "main.java"
class Test
  minor version: 0
  major version: 59
  flags: (0x0020) ACC_SUPER
  this_class: #21                         // Test
  super_class: #2                         // java/lang/Object
  interfaces: 0, fields: 0, methods: 2, attributes: 1
Constant pool:
   #1 = Methodref          #2.#3          // java/lang/Object."<init>":()V
   #2 = Class              #4             // java/lang/Object
   #3 = NameAndType        #5:#6          // "<init>":()V
   #4 = Utf8               java/lang/Object
   #5 = Utf8               <init>
   #6 = Utf8               ()V
   #7 = Fieldref           #8.#9          // java/lang/System.out:Ljava/io/Pr...
   #8 = Class              #10            // java/lang/System
   #9 = NameAndType        #11:#12        // out:Ljava/io/PrintStream;
  #10 = Utf8               java/lang/System
  #11 = Utf8               out
  #12 = Utf8               Ljava/io/PrintStream;
  #13 = String             #14            // 1 plus 2 is 3
  #14 = Utf8               1 plus 2 is 3
  #15 = Methodref          #16.#17        // java/io/PrintStream.println:(Lja...
  #16 = Class              #18            // java/io/PrintStream
  #17 = NameAndType        #19:#20        // println:(Ljava/lang/String;)V
  #18 = Utf8               java/io/PrintStream
  #19 = Utf8               println
  #20 = Utf8               (Ljava/lang/String;)V
  #21 = Class              #22            // Test
  #22 = Utf8               Test
```

図 12.7: `Test.class` のアセンブリコード（その 1、次ページへ続く）

実は Java の全てのクラスファイルはこの 4 バイトから始まることが規定されています。このような数字を一般に**マジックナンバー**と呼びます。クラスファイルの先頭 4 バイトが “cafe babe” でなければ、そのファイルはクラスファイルではないと即座に判別できる簡単な仕組みです。もちろん偶然 “cafe babe” になることもありますが、その確率は約 42 億分の 1（$= 1/2^{32}$）ですから、識別子として十分安全に機能します。

ところで “cafe babe” の由来についてはたとえば以下のページに載っています。

`http://www.it-a.jp/2009/05/cafe-babe-へようこそ/`

```
  #23 = Utf8               Code
  #24 = Utf8               LineNumberTable
  #25 = Utf8               main
  #26 = Utf8               ([Ljava/lang/String;)V
  #27 = Utf8               SourceFile
  #28 = Utf8               main.java
{
  Test();
    descriptor: ()V
    flags: (0x0000)
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method java/lang/Object."<in...
         4: return
      LineNumberTable:
        line 1: 0

  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
    flags: (0x0009) ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=1, args_size=1
         0: getstatic     #7                  // Field java/lang/System.out:L...
         3: ldc           #13                 // String 1 plus 2 is 3
         5: invokevirtual #15                 // Method java/io/PrintStream.p...
         8: return
      LineNumberTable:
        line 3: 0
        line 4: 8
}
SourceFile: "main.java"
```

図 12.8: Test.class のアセンブリコード（その 2）

12.4.2　逆アセンブル

クラスファイルの内容をテキスト形式で表示するツール javap が Java SDK に含まれています。このツールはバイトコードをアセンブリコード風に変換して表示する逆アセンブラーです。

実行方法は以下の通りです[3]。クラスファイルを指定する前に、必ずコマンドオプション -v を付けてください。

```
> javap -v Test.class
```

図 12.7（176 ページ）、図 12.8（177 ページ）はその表示例です。この表示の詳細は述べませんが、バイトコード内部でどのような定数が使用され、どのようなアセンブリ命令が実行されているかを

[3] コマンドプロンプトから「javap を知らない」とエラーメッセージが出ることがあるようです。上で述べたパスの設定が javap について適切に設定されていないためです。

1 垣間見ることができます。

1

1

第13章　C++のプログラムをJavaへ翻訳する

この章では10章（139ページ）で作成したC++のプログラムをJavaのプログラムへ翻訳します。C++とJavaの基本構文、オブジェクト指向プログラミングのアイディアはほとんど一致しますから、1行毎の翻訳が可能です。ところで、すでに述べていますが、Javaではインライン記述のみが可能です。その点に注意しながら比較解説していきます。

13.1　ゲームクラスの翻訳

まず、 `Game` クラスを翻訳します。

図13.1（182ページ）は、 `Game` クラス `Game` の宣言について、C++のプログラムとそれをJavaへ翻訳したプログラムを並べて比較したものです。Javaについては実装部分（関数の本体部分）をとりあえず中点 { ... } で省略しています。

以下、両者の違いについて解説します。

1. C++では前方宣言（9.2.1項（123ページ）参照）：

```
class Map;
class Player;
class Monster;
```

   が必要ですが、Javaでは必要ありません。

2. C++では、クラス内のメンバー変数、メンバー関数へのアクセスはデフォルトでは非公開です。それを解除し、外部に全て公開するため、

```
class Game {
  public:
```

```
class Map;
class Player;
class Monster;

class Game {
  public:

    Map *map;
    Player *player;
    Monster *monster1;
    Monster *monster2;

    Game();
    void print();
    void mainLoop();
    void turn();
    static char getKeyChar();
    static int signum(int a);
};
```

```
class Game {

    public static void main(String args[]) {...}

    Map map;
    Player player;
    Monster monster1;
    Monster monster2;

    Game() { ... }
    void print() { ... }
    void mainLoop() { ... }
    void turn() { ... }
    static char getKeyChar() { ... }

}
```

図 13.1: C++と Java のプログラムの比較（その 1、 `Game` クラスの外形）

と `public:` を付けました（6.3.2 項（72 ページ）参照）。しかし、Java では暗黙の設定では同じソースプログラムファイル内では全ての変数、メソッドは互いに自由にアクセスできます。そこで、特別の事情がない限り、Java のプログラムには `public` を付けません。

C++のアクセス制御子（`public:`など）はそれが付けられた行以降の全ての変数、関数を修飾します。それに対して Java のアクセス制御子はそれが付けられた変数、メソッドのみを修飾します。

3. Java の `main` 関数は適当なクラスのクラスメソッドとして定義する必要があります。しかし、`main` 関数のためにわざわざクラスをひとつ新たに定義するのは返って煩雑なので、通常は中核クラスの中に定義します。ダンジョンゲームの場合、 `Game` クラスが中核クラスになります。

4. C++でのオブジェクトを指すポインタの宣言：

```
Map *map;
Player *player;
Monster *monster1;
Monster *monster2;
```

では、**間接演算子**（indirection operator）‘*’ が必要です。それに対して Java ではプリミティブ型以外の変数は全て参照型であり、宣言に ‘*’ は不要です。よって Java の宣言は以下のように変更されます。

```
int main(int argc, const char * argv[]) {
    Game *game = new Game;
    game->mainLoop();
    return 0;
}
```

```
class Game {
...
    public static void main(String args[]) {
        Game game = new Game();
        game.mainLoop();

    }
...
}
```

図 13.2: C++と Java のプログラムの比較（その 2、 main() 関数）

```
    Map map;
    Player player;
    Monster monster1;
    Monster monster2;
```

重要なので繰り返します。ここを勘違いすると、この講義の以降の内容が全て理解できなくなります。上の Java のインスタンス変数の宣言は、C/C++で言うポインタを宣言していることに注意してください。

5. Java には、符号関数 signum() が標準装備されていますから、ユーザ定義する必要はありません。

以上の違いを踏まえて、図 13.1 の二つのプログラムが 1 行毎の翻訳になっていることを確認してください。

次に、図 13.2（183 ページ）は、 main 関数の翻訳です。違いは、

6. C++でデフォルトコンストラクタを呼び出すときには、“= new Game;” のようにクラス名の後ろの括弧 () は不要です（有っても構いません）。しかし Java では必ず括弧を付ける必要があります。Java の場合、オブジェクトの生成を明確に意識させる意図があると思われます。

7. C++で動的オブジェクトの実体を求めるには、game->のようにアロー演算子 ->を用いますが、Java では game. のようにピリオド演算子 . を用います。

図 13.3（184 ページ）は、 Game クラスのメンバー関数をメソッドへ翻訳する場合です。違いは以下の通りです。

9. 標準出力への文字列の表示は、C++では cout を用いますが、Java では System.out.println() を用います。

```
Game::Game() {
    map = new Map(this);
    player = new Player(this);
    monster1 = new Monster(this,
                           1,
                           Map::YSIZE-2);
    monster2 = new Monster(this,
                           Map::XSIZE-2,
                           1);
}

void Game::print() {
    map->print();
    player->print();
    monster1->print();
    monster2->print();
    cout << endl;
}

void Game::mainLoop() {
    print();
    while (true) { turn(); }
}

void Game::turn() {
    player->action();
    monster1->action();
    monster2->action();
    print();
}
```

```
class Game {
    ...
    Game() {
        map = new Map(this);
        player = new Player(this);
        monster1 = new Monster(this,
                               1,
                               Map.YSIZE-2);
        monster2 = new Monster(this,
                               Map.XSIZE-2,
                               1);
    }

    void print() {
        map.print();
        player.print();
        monster1.print();
        monster2.print();
        System.out.println();
    }

    void mainLoop() {
        print();
        while (true) { turn(); }
    }

    void turn() {
        player.action();
        monster1.action();
        monster2.action();
        print();
    }
    ...
}
```

図 13.3: C++と Java のプログラムの比較（その 3、 Game のメンバー関数、メソッド）

次に、図 13.4（185 ページ）は、標準入力から 1 文字を読み込む処理です。ここは言語依存の度合いが強く、C++から Java への 1 行毎の翻訳は不可能です。そこで Java の独自処理を組み込みました。Java の入出力の仕組みを完全に理解するには深い知識を必要とするため、ここでは説明を省略します。余裕のある人は「Java 入出力」などでネット検索するなどして調べてください。

13.2　マップクラス/プレイヤークラス/モンスタークラスの翻訳

図 13.5（185 ページ）は Map クラスの宣言部についてです。違いは以下の通りです。

10. C++の定数化キーワード const に対応する Java のキーワードは final です。両者の機能上の微妙な違いは、ダンジョンゲームのプログラムの場合には影響しません。

マップクラスのプログラムに関する違いはありません。図 13.6（187 ページ）を確認してください。1 行毎の翻訳が可能になっています。

プレイヤー、モンスターの宣言部（図 13.6（187 ページ）、図 13.9（190 ページ））に関する上記以外の違いはありません。

プレイヤークラス、モンスタークラスのプログラムの C++と Java の違いは以下の通りです。

```
char Game::getKeyChar() {
    char c;
    cout << "? " << flush;
    cin >> c;
    return c;
}
```

```
class Game {
    ...
    static char getKeyChar() {
        char c = 0;
        System.out.print("? ");
        try {
            BufferedReader in =
                new BufferedReader(
                new InputStreamReader(
                  System.in));
            c = in.readLine().charAt(0);
        } catch(IOException e){
            System.out.println(
                "ERROR:: getKeyChar()");
        }
        return c;
    }
    ...
}
```

図 13.4: C++と Java のプログラムの比較（その 4、 `Game` の `getKeyChar()`）

```
class Map {
  public:
    static const int ROCK = 0;
    static const int EMPTY = 1;
    static const int GOAL = 2;

    static const int XSIZE = 13;
    static const int YSIZE = 11;

    Game *game;
    int** data;

    Map(Game *game_);
    void cellPrint(int x, int y);
    void print();
};
```

```
class Map {

    static final int ROCK = 0;
    static final int EMPTY = 1;
    static final int GOAL = 2;

    static final int XSIZE = 13;
    static final int YSIZE = 11;

    Game game;
    int  data[][];

    Map(Game game) { ... }
    void cellPrint(int x, int y) { ... }
    void print() { ... }
}
```

図 13.5: C++と Java のプログラムの比較（その 1、 `Map` クラスの外形）

11. Java のメソッド定義ではしばしば、インスタンス変数（仮に `inst` とします）と同じ名前の仮引数 `inst` を用います。その場合、単に名前 “`inst`” で参照すると引数の方が参照されます（C++に関する図 6.8（79 ページ）と同じ議論が成り立ちます）。インスタンス変数を参照したい場合には、オブジェクト自身を表すキーワード `this`（C++の 9.3.1 項（128 ページ）を思い出そう）を付けて “`this.inst`”で参照可能です。Java のプログラムでしばしば見かけるのはコンストラクタの中での以下の代入文のパターンです。

```
this.inst = inst;
```

C++の場合には、上のパターンを用いることは少なく[1]、この講義テキストでは

[1]同じ代入文は C++では “`this->inst = inst;` となり、あまり美しくないのが理由と思われます。

```
    inst = inst_;
```

となるパターンを用いています。ここに、インスタンス変数名の後にアンダースコアを付けたものを引数名としています。これは指南書（3.1 項（27 ページ）参照）に書かれている作法のひとつです。

図 13.6（187 ページ）、図 13.7（188 ページ）、図 13.8（189 ページ）、図 13.9（190 ページ）、図 13.10（191 ページ）を確認してください。

```
Map::Map(Game *game_) {
    game = game_;
    data = new int*[XSIZE];
    for (int x = 0; x < XSIZE; x++) {
        data[x] = new int[YSIZE];
        for (int y = 0; y < YSIZE; y++) {
            if ((x == 0)||(x == XSIZE-1)||
               (y == 0)||(y == YSIZE-1)||
               ((abs(x-XSIZE/2) <= 1)&&
                (abs(y-YSIZE/2) <= 1))) {
                data[x][y] = ROCK;
            }
            else { data[x][y] = EMPTY; }
        }
    }
    data[XSIZE-2][YSIZE-2] = GOAL;
}

void Map::cellPrint(int x, int y) {
    int e = data[x][y];
    if (e == ROCK) {
        cout << '#' << ' ';
    }
    else if (e == EMPTY) {
        cout << '.' << ' ';
    }
    else if (e == GOAL) {
        cout << 'G' << ' ';
    }
    else { cout << 'E' << ' '; }
}

void Map::print() {
    for (int y = 0; y < YSIZE; y++) {
        for (int x = 0; x < XSIZE; x++) {
            if (game->player->cellPrint(x,y))
                { continue; };
            if (game->monster1->cellPrint(x,y))
                { continue; };
            if (game->monster2->cellPrint(x,y))
                { continue; };
            cellPrint(x,y);
        }
        cout << endl;
    }
}
```

```
class Map {
    ...
    Map(Game game) {
        this.game = game;
        data = new int[XSIZE][];
        for (int x = 0; x < XSIZE; x++) {
            data[x] = new int[YSIZE];
            for (int y = 0; y < YSIZE; y++) {
                if ((x == 0)||(x == XSIZE-1)||
                   (y == 0)||(y == YSIZE-1)||
                   ((Math.abs(x-XSIZE/2) <= 1)&&
                    (Math.abs(y-YSIZE/2) <= 1))) {
                    data[x][y] = ROCK;
                }
                else { data[x][y] = EMPTY; }
            }
        }
        data[XSIZE-2][YSIZE-2] = GOAL;
    }

    void cellPrint(int x, int y) {
        int e = data[x][y];
        if (e == ROCK) {
            System.out.print('#'+" ");
        }
        else if (e == EMPTY) {
            System.out.print('.'+" ");
        }
        else if (e == GOAL) {
            System.out.print('G'+" ");
        }
        else { System.out.print('E'+' '); }
    }

    void print() {
        for (int y = 0; y < YSIZE; y++) {
            for (int x = 0; x < XSIZE; x++) {
                if (game.player.cellPrint(x,y))
                    { continue; };
                if (game.monster1.cellPrint(x,y))
                    { continue; };
                if (game.monster2.cellPrint(x,y))
                    { continue; };
                cellPrint(x,y);
            }
            System.out.println();
        }
    }
    ...
}
```

図 13.6: C++と Java のプログラムの比較（その 2、 Map クラスの関数）

```
class Player {
  public:
    static const int INIT_HP = 100;

    Game *game;
    int x, y;
    int hp;

    Player(Game *game_);
    bool cellPrint(int x, int y);
    void print();
    bool moveTo(int dx, int dy);
    void attackTo(int dx, int dy);
    void action();
};
```

```
class Player {

    static final int INIT_HP = 100;

    Game game;
    int x, y;
    int hp;

    Player(Game game) { ... }
    boolean cellPrint(int x, int y) { ... }
    void print() { ... }
    boolean moveTo(int dx, int dy) { ... }
    void attackTo(int dx, int dy) { ... }
    void action() { ... }
}
```

図 13.7: C++と Java のプログラムの比較（その 1、 `Player` クラスの外形）

```
Player::Player(Game *game_) {
    game = game_;
    x = 1; y = 1;
    hp = INIT_HP;
}

bool Player::cellPrint(int x_, int y_) {
    if ((x == x_)&&(y == y_)) {
        cout << '@' << ' ';
        return true;
    }
    return false;
}

void Player::print() {
    cout << "Player : ("
         << x << "," << y << ") "
         << hp << endl;
}

bool Player::moveTo(int dx, int dy) {
    int x2 = x+dx, y2 = y+dy;

    if (game->map->data[x2][y2] == Map::ROCK) {
        return false;
    }
    if ((x2 == game->monster1->x)&&
        (y2 == game->monster1->y)) {
        return false;
    }
    if ((x2 == game->monster2->x)&&
        (y2 == game->monster2->y)) {
        return false;
    }

    x = x2; y = y2;
    return true;
}

void Player::attackTo(int dx, int dy) {
    int x2 = x+dx, y2 = y+dy;
    if ((x2 == game->monster1->x)&&
        (y2 == game->monster1->y)) {
        game->monster1->hp -= 20;
    }
    if ((x2 == game->monster2->x)&&
        (y2 == game->monster2->y)) {
        game->monster2->hp -= 20;
    }
}

void Player::action() {
    int dx, dy;
    switch (Game::getKeyChar()) {
        case 'z': dx = -1; dy = +1; break;
        case 'x': dx =  0; dy = +1; break;
        case 'c': dx = +1; dy = +1; break;
        case 'a': dx = -1; dy =  0; break;
        case 'd': dx = +1; dy =  0; break;
        case 'q': dx = -1; dy = -1; break;
        case 'w': dx =  0; dy = -1; break;
        case 'e': dx = +1; dy = -1; break;
        default:  dx =  0; dy =  0;
    }

    if (!moveTo(dx,dy)) { attackTo(dx,dy); }
}
```

```
class Player {
    ...
    Player(Game game) {
        this.game = game;
        x = 1; y = 1;
        hp = INIT_HP;
    }

    boolean cellPrint(int x, int y) {
        if ((this.x == x)&&(this.y == y)) {
            System.out.print('@'+" ");
            return true;
        }
        return false;
    }

    void print() {
        System.out.println(
            "Player : (" + x +","+ y + ") "
            + hp);
    }

    boolean moveTo(int dx, int dy) {
        int x2 = x+dx, y2 = y+dy;

        if (game.map.data[x2][y2] == Map.ROCK) {
            return false;
        }
        if ((x2 == game.monster1.x)&&
            (y2 == game.monster1.y)) {
            return false;
        }
        if ((x2 == game.monster2.x)&&
            (y2 == game.monster2.y)) {
            return false;
        }

        x = x2; y = y2;
        return true;
    }

    void attackTo(int dx, int dy) {
        int x2 = x+dx, y2 = y+dy;
        if ((x2 == game.monster1.x)&&
            (y2 == game.monster1.y)) {
            game.monster1.hp -= 20;
        }
        if ((x2 == game.monster2.x)&&
            (y2 == game.monster2.y)) {
            game.monster2.hp -= 20;
        }
    }

    void action() {
        int dx, dy;
        switch (Game.getKeyChar()) {
            case 'z': dx = -1; dy = +1; break;
            case 'x': dx =  0; dy = +1; break;
            case 'c': dx = +1; dy = +1; break;
            case 'a': dx = -1; dy =  0; break;
            case 'd': dx = +1; dy =  0; break;
            case 'q': dx = -1; dy = -1; break;
            case 'w': dx =  0; dy = -1; break;
            case 'e': dx = +1; dy = -1; break;
            default:  dx =  0; dy =  0;
        }

        if (!moveTo(dx,dy)) { attackTo(dx,dy); }
    }
    ...
}
```

図 13.8: C++と Java のプログラムの比較（その 2、 Player クラスの関数）

```
class Monster {
  public:
    static const int INIT_HP = 150;

    Game *game;
    int x, y;
    int hp;

    Monster(Game *game_, int x, int y);
    bool cellPrint(int x, int y);
    void print();
    bool moveTo(int dx, int dy);
    void attackTo(int dx, int dy);
    void action();
};
```

```
class Monster {

    static final int INIT_HP = 150;

    Game game;
    int x, y;
    int hp;

    Monster(Game game, int x, int y) { ... }
    boolean cellPrint(int x, int y) { ... }
    void print() { ... }
    boolean moveTo(int dx, int dy) { ... }
    void attackTo(int dx, int dy) { ... }
    void action() { ... }
}
```

図 13.9: C++と Java のプログラムの比較（その 1、 `Monster` クラスの外形）

```
Monster::Monster(Game *game_,
                 int x_, int y_) {
    game = game_;
    x = x_; y = y_;
    hp = INIT_HP;
}

bool Monster::cellPrint(int x_, int y_) {
    if ((x == x_)&&(y == y_)) {
        cout << 'M' << ' ';
        return true;
    }
    return false;
}

void Monster::print() {
    cout << "Monster : ("
         << x << "," << y << ") "
         << hp << endl;
}

bool Monster::moveTo(int dx, int dy) {
    int x2 = x+dx, y2 = y+dy;

    if (game->map->data[x2][y2] == Map::ROCK) {
        return false;
    }
    if ((x2 == game->player->x)&&
        (y2 == game->player->y)) {
        return false;
    }
    if ((x2 == game->monster1->x)&&
        (y2 == game->monster1->y)) {
        return false;
    }
    if ((x2 == game->monster2->x)&&
        (y2 == game->monster2->y)) {
        return false;
    }

    x = x2; y = y2;
    return true;
}

void Monster::attackTo(int dx, int dy) {
    int x2 = x+dx, y2 = y+dy;
    if ((x2 == game->player->x)&&
        (y2 == game->player->y)) {
        game->player->hp -= 30;
    }
}

void Monster::action() {
    int dx = Game::
             signum(game->player->x - x);
    int dy = Game::
             signum(game->player->y - y);

    if (!moveTo(dx,dy)) { attackTo(dx,dy); }
}
```

```
class Monster {
    ...
    Monster(Game game, int x, int y) {

        this.game = game;
        this.x = x; this.y = y;
        hp = INIT_HP;
    }

    boolean cellPrint(int x, int y) {
        if ((this.x == x)&&(this.y == y)) {
            System.out.print('M'+" ");
            return true;
        }
        return false;
    }

    void print() {
        System.out.println("Monster : ("
                           + x + "," + y + ") "
                           + hp);
    }

    boolean moveTo(int dx, int dy) {
        int x2 = x+dx, y2 = y+dy;

        if (game.map.data[x2][y2] == Map.ROCK) {
            return false;
        }
        if ((x2 == game.player.x)&&
            (y2 == game.player.y)) {
            return false;
        }
        if ((x2 == game.monster1.x)&&
            (y2 == game.monster1.y)) {
            return false;
        }
        if ((x2 == game.monster2.x)&&
            (y2 == game.monster2.y)) {
            return false;
        }

        x = x2; y = y2;
        return true;
    }

    void attackTo(int dx, int dy) {
        int x2 = x+dx, y2 = y+dy;
        if ((x2 == game.player.x)&&
            (y2 == game.player.y)) {
            game.player.hp -= 30;
        }
    }

    void action() {
        int dx = Integer.
                 signum(game.player.x - x);
        int dy = Integer.
                 signum(game.player.y - y);

        if (!moveTo(dx,dy)) { attackTo(dx,dy); }
    }
    ...
}
```

図 13.10: C++と Java のプログラムの比較（その 2、 Monster クラスの関数）

13.3　付録：この章の全プログラムリスト

```
//ゲームクラスのプログラム
import java.io.*;

class Game {
    public static void main(String args[]) {
        Game game = new Game();
        game.mainLoop();
    }

    Map map;
    Player player;
    Monster monster1;
    Monster monster2;

    Game() {
        map = new Map(this);
        player = new Player(this);
        monster1 = new Monster(this, 1, Map.YSIZE-2);
        monster2 = new Monster(this, Map.XSIZE-2, 1);
    }

    void print() {
        map.print();
        player.print();
        monster1.print();
        monster2.print();
        System.out.println();
    }

    void mainLoop() {
        print();
        while (true) { turn(); }
    }

    void turn() {
        player.action();
        monster1.action();
        monster2.action();
        print();
    }

    static char getKeyChar() {
        char c = 0;
        System.out.print("? ");
        try {
            BufferedReader in = new BufferedReader(
                                new InputStreamReader(System.in));
            c = in.readLine().charAt(0);
        } catch(IOException e){
            System.out.println("ERROR:: getKeyChar()");
        }
        return c;
    }
}
```

```
//マップクラスのプログラム
class Map {
    static final int ROCK = 0;
    static final int EMPTY = 1;
    static final int GOAL = 2;

    static final int XSIZE = 13;
    static final int YSIZE = 11;

    Game game;
    int data[][]= new int[XSIZE][YSIZE];

    Map(Game game) {
        this.game = game;
        for (int x = 0; x < XSIZE; x++) {
            for (int y = 0; y < YSIZE; y++) {
                if ((x == 0)||(x == XSIZE-1)||
                   (y == 0)||(y == YSIZE-1)||
                   ((Math.abs(x-XSIZE/2) <= 1)&&
                    (Math.abs(y-YSIZE/2) <= 1))) {
                    data[x][y] = ROCK;
                }
                else { data[x][y] = EMPTY; }
            }
        }
        data[XSIZE-2][YSIZE-2] = GOAL;
    }

    void print() {
        for (int y = 0; y < YSIZE; y++) {
            for (int x = 0; x < XSIZE; x++) {
                if (game.player.cellPrint(x,y)) { continue; }
                if (game.monster1.cellPrint(x,y)) { continue; }
                if (game.monster2.cellPrint(x,y)) { continue; }
                cellPrint(x,y);
            }
            System.out.println();
        }
    }

   void cellPrint(int x, int y) {
        int e = data[x][y];
        if (e == ROCK) { System.out.print('#'+" "); }
        else if (e == EMPTY) { System.out.print('.'+" "); }
        else if (e == GOAL) { System.out.print('G'+" "); }
        else { System.out.print('E'+' '); }
    }
}
```

```
//プレイヤークラスのプログラム
class Player {
    static final int INIT_HP = 100;

    Game game;
    int x, y;
    int hp;

    Player(Game game) {
        this.game = game;
        x = 1; y = 1;
        hp = INIT_HP;
    }

    void print() {
        System.out.println("Player : (" + x +","+ y + ") " + hp);
    }

    boolean cellPrint(int x, int y) {
        if ((this.x == x)&&(this.y == y)) {
            System.out.print('@'+" ");
            return true;
        }
        return false;
    }

    boolean moveTo(int dx, int dy) {
        int targetX = x+dx, targetY = y+dy;

        if (game.map.data[targetX][targetY] == Map.ROCK) { return false; }
        if ((targetX == game.monster1.x)&&(targetY == game.monster1.y)) { return false; }
        if ((targetX == game.monster2.x)&&(targetY == game.monster2.y)) { return false; }

        x = targetX; y = targetY;
        return true;
    }

    void attackTo(int dx, int dy) {
        int targetX = x+dx, targetY = y+dy;
        if ((targetX == game.monster1.x)&&(targetY == game.monster1.y)) {
            game.monster1.hp -= 20;
        }
        if ((targetX == game.monster2.x)&&(targetY == game.monster2.y)) {
            game.monster2.hp -= 20;
        }
    }

    void action() {
        int dx, dy;
        switch (Game.getKeyChar()) {
            case 'z': dx = -1; dy = +1; break;
            case 'x': dx =  0; dy = +1; break;
            case 'c': dx = +1; dy = +1; break;
            case 'a': dx = -1; dy =  0; break;
            case 'd': dx = +1; dy =  0; break;
            case 'q': dx = -1; dy = -1; break;
            case 'w': dx =  0; dy = -1; break;
            case 'e': dx = +1; dy = -1; break;
```

```
            default:  dx =  0; dy =  0;
        }

        if (!moveTo(dx,dy)) { attackTo(dx,dy); }
    }
}
```

```
//モンスタークラスのプログラム
class Monster {
    static final int INIT_HP = 150;

    Game game;
    int x, y;
    int hp;

    Monster(Game game, int x, int y) {
        this.game = game;
        this.x = x; this.y = y;
        hp = INIT_HP;
    }

    void print() {
        System.out.println("Monster : (" + x + "," + y + ") " + hp);
    }

    boolean cellPrint(int x, int y) {
        if ((this.x == x)&&(this.y == y)) {
            System.out.print('M'+" ");
            return true;
        }
        return false;
    }

    boolean moveTo(int dx, int dy) {
        int targetX = x+dx, targetY = y+dy;

        if (game.map.data[targetX][targetY] == Map.ROCK) { return false; }
        if ((targetX == game.player.x)&&(targetY == game.player.y)) { return false; }
        if ((targetX == game.monster1.x)&&(targetY == game.monster1.y)) { return false; }
        if ((targetX == game.monster2.x)&&(targetY == game.monster2.y)) { return false; }

        x = targetX; y = targetY;
        return true;
    }

    void attackTo(int dx, int dy) {
        int targetX = x+dx, targetY = y+dy;
        if ((targetX == game.player.x)&&(targetY == game.player.y)) {
            game.player.hp -= 30;
        }
    }

    void action() {
        int dx = Integer.signum(game.player.x - x);
        int dy = Integer.signum(game.player.y - y);

        if (!moveTo(dx,dy)) { attackTo(dx,dy); }
    }
}
```

1

1

第14章　視界の悪いダンジョンを作ってみる：抽象データ型の導入

前章まで２次元座標は x 座標値、y 座標値の各要素毎の扱いをしてきました。たとえばプログラム中に以下のような文がありました。

```
        if ((this.x == x)&&(this.y == y)) {
...
        int targetX = x+dx, targetY = y+dy;
        if ((targetX == game.player.x)&&(targetY == game.player.y)) {
...
        int dx = Integer.signum(game.player.x - x);
        int dy = Integer.signum(game.player.y - y);
```

これをよりスマートな表現に変えることがこの章の目的です。上のプログラム辺について言えば、この章以降、以下のように変更されます。x 座標値、y 座標値は陽には用いず、抽象化されたベクトルクラスのオブジェクトを用います。

```
        if (this.position.equal(position)) {
...
        Int2 target = position.add(direction);
        if (target.equal(game.player.position)) {
...
        Int2 direction = game.player.position.sub(position).signum();
```

このベクトルのようなデータ型を**抽象データ型**（abstract data type）と呼びます。ここでいう「抽象」とは、「内部構造は外部に見せず、必要な演算はメソッドとして実現する」という意味です。これによってプログラムの可読性が向上します。

14.1　整数型２次元ベクトルオブジェクト： Int2

図 14.1（200 ページ）は、要素が整数型の２次元ベクトルを保持するオブジェクトのクラス Int2 の定義です。なお、座標原点を始点とするベクトルは座標位置を表しますから、Int2 はベクトルだけではなく、２次元座標位置を表すオブジェクトとしても利用できます。

```
class Int2 {
    int x,y;

    Int2(int x, int y) { this.x = x; this.y = y; }       // コンストラクタ

    Int2 add(Int2 a) { return new Int2(x+a.x, y+a.y); }  // 加算

    Int2 sub(Int2 a) { return new Int2(x-a.x, y-a.y); }  // 減算

    Int2 signum() {                                      // 要素毎の符号関数
        return new Int2(Integer.signum(x), Integer.signum(y));
    }

    boolean equal(Int2 a) { return (x == a.x)&&(y == a.y); } // 比較

    double length() { return Math.sqrt((x*x+y*y)); }     // 長さ

    public String toString(){ return "("+x+","+y+")"; }  // 文字列への変換
}
```

図 14.1: 整数型 2 次元ベクトルクラス

さて、図 14.1 のオブジェクトは、x 座標値、y 座標値に相当するインスタンス変数 x、 y を持っています。コンストラクタは、x 座標値、y 座標値を受け取り、それをそのままインスタンス変数 x、 y へコピーし、オブジェクトを生成、初期化します。ベクトルどうしの加算、減算はメソッド add()、 sub で可能です。ベクトルの要素毎に符号関数を適用する関数はメソッド signum() です。当該オブジェクトと引数のオブジェクトが等しいか否かを判定する関数はメソッド equal() です。length() はベクトルの長さを求めます。ここに Math.sqrt() は Java の Math モジュールの中の平方根を計算する関数の呼び出しです。いずれのメソッドでもオブジェクト自身の内容を変更することはなく、新しいオブジェクトを生成し、それを戻り値とします。

少し謎めいているのが、図 14.1 の最後の行のメソッド toString() かもしれません。これについて以下に解説します。

14.1.1　オブジェクトから文字列への自動変換： toString()

Java では、クラスにメソッド public String toString() が定義されているとき、そのクラスのオブジェクトを文字列（ String）に変換すべき局面で自動的にこの関数 toString() が呼び出されると約束されています。何故そのような便利な仕組みが自動的に機能するのか、その正体は後章で多相性（ポリモルフィズム）を学んだところで判明します。

たとえば式：

```
        "Player : " + position + " " + hp
```

のように、文字列定数 "Player : " の後に position が連接されている局面では、 position は文字列であることが期待されます（文字列と連接できるのは文字列のみです）。このとき、もし position が文字列ではなくオブジェクトであるならば、この式は自動的に以下の式に変換されて実行されます。

```
        "Player : " + position.toString() + " " + hp
```

なお、オブジェクトではない、プリミティブ型データである整数 hp は、Java 言語仕様により文字列に自動変換されると約束されています。

Java では全てのオブジェクトがこの性質を持つため、表示関連のメソッドとして toString() は重宝します。この章では他にも Player、 Monster にも新たに toString() を定義します。

以下、 Int2 の導入で変更されるダンジョンプログラムの部分をひとつずつ見ていきましょう。

14.2　ゲームクラスの変更

Game クラスの記述は図 14.2（202 ページ）の通りです。先ほど述べたように Player クラスには新たに toString() が定義されますから（図 14.4（204 ページ）参照）、前章の関数呼び出し：

```
        player.print();
```

は、以下の関数呼び出し：

```
        System.out.println(player);   // toString() で自動変換されて表示される
```

に変更します。引数の Player オブジェクトは文字列に自動変換されて表示されます。Monster オブジェクトも同様です。

前章の Player クラス、 Monster クラスのメソッド print() は不要ですから、削除します。

```
class Game {
    public static void main(String args[]) { ** 前章から変更なし** }

    ** インスタンス変数の宣言に変更なし **

    Game() { ** 前章から変更なし** }

    void print() {
        map.print();
        System.out.println(player);   // toString() で自動変換されて表示される
        System.out.println(monster1); // toString() で自動変換されて表示される
        System.out.println(monster2); // toString() で自動変換されて表示される
        System.out.println();
    }

    void mainLoop() { ** 前章から変更なし** }

    void turn() { ** 前章から変更なし** }

    static char getKeyChar() { ** 前章から変更なし** }
}
```

図 14.2: Int2 を導入したダンジョンゲームのプログラム（ゲーム）

14.3　マップ・クラスの変更

図 14.3（203 ページ）がこの章のマップ定義です。

Int2 の導入に伴い、新たに以下の便利なメソッドを定義します。

```
    int getAt(Int2 position) { return data[position.x][position.y]; }
                                        // Int2 が示すセル位置のデータを取り出す
```

これは Int2 で与えられた座標位置のセルデータを取り出すメソッドです。これはオブジェクト指向で言う **ゲッター**（getter）—データをゲットしてくれるメソッド — に相当するのです。ゲッターの反対語は**セッター**（setter）— データをセットしてくれるメソッド — です。ゲッター、セッターの詳細は後章で解説します。このメソッドは、以下のように、 cellPrint() の中で使います。

```
    void cellPrint(Int2 position) {                    // 引数を Int2 型へ変更
        int e = getAt(position);                       // セルデータを取り出す
```

print() の中も、 Int2 で以下のようにすっきりしました。

```
class Map {
    ** 変数部は前章から変更なし **

    Map(Game game) { ** 前章から変更なし ** }

    int getAt(Int2 position) { return data[position.x][position.y]; }
                                    // Int2 が示すセル位置のデータを取り出す

    void print() {
        for (int y = 0; y < YSIZE; y++) {
            for (int x = 0; x < XSIZE; x++) {
                Int2 position = new Int2(x,y);     // Int2 オブジェクトを生成
                if (game.player.cellPrint(position))   { continue; };
                if (game.monster1.cellPrint(position)) { continue; };
                if (game.monster2.cellPrint(position)) { continue; };
                cellPrint(position);
            }
            System.out.println();
        }
    }

    void cellPrint(Int2 position) {                   // 引数を Int2 型へ変更
        int e = getAt(position);                      // セルデータを取り出す
        if (e == ROCK)       { System.out.print('#'+" "); }
        else if (e == EMPTY) { System.out.print('.'+" "); }
        else if (e == GOAL)  { System.out.print('G'+" "); }
        else                 { System.out.print('E'+' '); }
    }
}
```

図 14.3: Int2 を導入したダンジョンゲームのプログラム（マップ）

```
                Int2 position = new Int2(x,y);     // Int2 オブジェクトを生成
                if (game.player.cellPrint(position))   { continue; };
                if (game.monster1.cellPrint(position)) { continue; };
                if (game.monster2.cellPrint(position)) { continue; };
                cellPrint(position);
```

14.4　プレイヤー・クラスの変更

図 14.4（204 ページ）、図 14.5（205 ページ）は Player の定義です。

まず、プレイヤーの位置を保持するインスタンス変数を Int2 型に変更しました。

```
    Int2 position;                          // 位置を Int2 オブジェクトに変更
```

```
class Player {
    static final int INIT_HP = 100;

    Game game;
    Int2 position;                          // 位置を Int2 オブジェクトに変更
    int hp;

    Player(Game game) {
        this.game = game;
        position = new Int2(1,1);         // 位置を Int2 オブジェクトで初期設定
        hp = INIT_HP;
    }

    public String toString() {                // print() を toString() に置換
        return "Player : "+position+" "+hp;     // position を文字列へ変換
    }

    boolean cellPrint(Int2 position) {       // 引数を Int2 型へ変更
        if (this.position.equal(position)) { // 位置の比較を Int2 どうしで行う
            System.out.print('@'+" ");
            return true;
        }
        return false;
    }

    boolean moveTo(Int2 direction) {                // 引数を Int2 型へ変更
        Int2 target = position.add(direction);         // Int2 どうしの加算

        if (game.map.getAt(target) == Map.ROCK) { return false; }
                                                        // セルデータは岩か？
        if (target.equal(game.monster1.position)) { return false; }
                                                              // 位置の比較
        if (target.equal(game.monster2.position)) { return false; } // 同上

        position = target;                                    // 位置の更新
        return true;
    }

    void attackTo(Int2 direction) {                   // 引数を Int2 型へ変更
        Int2 target = position.add(direction);          // Int2 どうしの加算
        if (target.equal(game.monster1.position)) { // 位置の比較
            game.monster1.hp -= 20;
        }
        if (target.equal(game.monster2.position)) { // 同上
            game.monster2.hp -= 20;
        }
    }
```

図 14.4: Int2 を導入したダンジョンゲームのプログラム（その 1、プレイヤー）

```
    void action() {
        Int2 direction;                                  // Int2 型変数を用意する
        switch (Game.getKeyChar()) {
            case 'z': direction = new Int2(-1,+1); break; // Int2 オブジェク
            case 'x': direction = new Int2( 0,+1); break; // トを条件に応じ
            case 'c': direction = new Int2(+1,+1); break; // て生成し、その
            case 'a': direction = new Int2(-1, 0); break; // 参照を
            case 'd': direction = new Int2(+1, 0); break; // direction に
            case 'q': direction = new Int2(-1,-1); break; // 代入
            case 'w': direction = new Int2( 0,-1); break;
            case 'e': direction = new Int2(+1,-1); break;
            default:  direction = new Int2( 0, 0);
        }

        if (!moveTo(direction)) { attackTo(direction); }
                                        // Int2 を引数としてメソッド呼び出しを行う
    }
}
```

図 14.5: Int2 を導入したダンジョンゲームのプログラム（その 2、プレイヤー）

これに伴って、メソッドの関連箇所を全て変更します。既に 14.1.1 項で述べたように、 toString() 内の

```
        return "Player : "+position+" "+hp;// position を文字列へ変換
```

では、 position をそのまま文字列の間に記述していますが、この position は図 14.1（200 ページ）の toString() によって文字列に自動変換されて出力されます。

moveTo() の冒頭の

```
        Int2 target = position.add(direction);    // Int2 どうしの加算
```

では、 Int2 の加算を行なっています。同じく、

```
        if (target.equal(game.monster.position)) { return false; }
```

では、プレイヤーが移動したい位置 target とモンスターの位置が一致するかどうかの判定を equal() メソッドで行なっています。

座標位置に関する記述がすっきりと簡略化されました。

図 14.5 の action() の中も Int2 に置き換えています。

14.5　モンスター・クラスの変更

図 14.6（207 ページ）のモンスターの記述もプレイヤーと同様です。詳細は省略します。

14.6　視界の悪いダンジョン

ダンジョン内部が煙や霞で視界が悪く、せいぜいセル 3 個分先までしか見えないと仮定しましょう。そうすると、マップの表示が以下のようになります。プレイヤーの北、南東に岩が見えています。南にはモンスターが 1 体見えていますが、それよりも遠い場所の様子は見えません。

```

        #
    . . . . .
    . . . . .
  . . . @ . . .
    . . . # #
    . . . # #
        M



Player : (4,3) 100
Monster : (4,6) 150
Monster : (8,3) 150
```

このようにゲームを改造するのは簡単です。図 14.7（208 ページ）がその改造部分です。`Map` クラスの `print()` において、表示位置 `position` がプレイヤーの位置 `game.player.position` から 3 セル分離れている場合には空白文字 2 文字“　”を表示するように改造します。その距離の計算には `Int2` クラスの `length()` メソッドを用います。

```
class Monster {
    static final int INIT_HP = 150;

    Game game;
    Int2 position;                          // 位置を Int2 オブジェクトに変更
    int hp;

    Monster(Game game, Int2 position) {
        this.game = game;
        this.position = position;                     // Int2 型引数で初期化
        hp = INIT_HP;
    }

    public String toString() {                // print() を toString() に置換
        return "Monster : "+position+" "+hp;   // position を文字列へ変換
    }

    boolean cellPrint(Int2 position) {                // 引数を Int2 型へ変更
        if (this.position.equal(position)) { // 位置の比較を Int2 どうしで行う
            System.out.print('M'+" ");
            return true;
        }
        return false;
    }

    boolean moveTo(Int2 direction) {                  // 引数を Int2 型へ変更
        Int2 target = this.position.add(direction);     // Int2 どうしの加算
        if (game.map.getAt(target) == Map.ROCK) { return false; }
                                                        // セルデータは岩か？
        if (target.equal(game.player.position)) { return false; }
                                                                // 位置の比較
        if (target.equal(game.monster1.position)) { return false; } // 同上
        if (target.equal(game.monster2.position)) { return false; } // 同上
        position = target;                            // Int2 での位置の更新
        return true;
    }

    void attackTo(Int2 direction) {                   // 引数を Int2 型へ変更
        Int2 target = this.position.add(direction);     // Int2 どうしの加算
        if (target.equal(game.player.position)) {
                                              // 位置の比較を Int2 どうしで行う
            game.player.hp -= 30;
        }
    }

    void action() {
        Int2 direction = game.player.position.sub(position).signum();
                // Int2 上で、プレイヤーとモンスターの位置の差に符号関数を適用

        if (!moveTo(direction)) { attackTo(direction); }
                                        // Int2 を引数としてメソッド呼び出しを行う
    }
}
```

図 14.6: Int2 を導入したダンジョンゲームのプログラム（モンスター）

```
class Map {
    ...
    void print() {
        for (int y = 0; y < YSIZE; y++) {
            for (int x = 0; x < XSIZE; x++) {
                Int2 position = new Int2(x,y);
                if (position.sub(game.player.position).length() > 3) {
                          // position がプレイヤーから 3 セル分よりも遠いならば
                    System.out.print("  ");  // 空白を表示
                    continue;
                }
                if (game.player.cellPrint(position)) { continue; }
                if (game.monster1.cellPrint(position)) { continue; }
                if (game.monster2.cellPrint(position)) { continue; }
                cellPrint(position);
            }
            System.out.println();
        }
     }
    ...
}
```

図 14.7: Int2 を導入したダンジョンゲームのプログラム（視界の悪い場合を実装するために、図 14.3 を改造）

14.7　抽象データ型の実装技術としてのオブジェクト指向

繰り返しになりますが、オブジェクト指向のクラス定義は、データ構造と演算をクラス内に抽象化する技法として役に立ちます。

この章では 2 次元ベクトル、2 次元座標位置のデータ構造と演算を Int2 クラスに実装しました。これによって、前章まで

```
int targetX = x+dx, targetY = y+dy;
```

と座標毎に計算していたプログラム部分が

```
Int2 target = position.add(direction);
```

とコンパクトに記述できるようになりました。ここで行いたいことは 2 次元座標上の加算です。実際の演算内容はもちろん座標軸毎の 2 回の加算ですが、それをわざわざダンジョンゲームの表層の Player オブジェクトの動作レベルに見せる必要はありません。オブジェクトとして抽象化された記述方法は、実行したい内容の本質を素直に表現できており、座標軸毎の具体的な演算が見える記述よりも可読性に優れた記述法と考えられます。

ところで Game クラス、Map クラス、Player クラスを抽象データ型と見るにはやや無理があります。というのも、ゲーム、マップ、プレイヤーは相互に関連して処理を進めていくものであり、Int2 クラスのようにひとつのクラス内で実装が閉じていません。どんなものでも抽象データ型として記述できる訳ではありません。元の対象が数学的に抽象化できる性質のものであることが大前提になります。その意味では抽象データ型は、他には複素数データ型などを実装する場合に適しています。

14.8　補足：2 次元ベクトルクラスの C++での実装

抽象データ型は C++でも記述可能です。しかも C++では、 +、 -、 ==などの演算子をユーザー定義することも可能です。ユーザー定義される演算子、あるいはそのプログラムを**演算子関数**（operator function）と呼びます。

図 14.8（210 ページ）は C++で Int2 を定義したものです。これを用いれば、Java の

```
// これは C++のプログラム

class Int2 {
  public:
    int x,y;

    Int2(int x_, int y_) { x = x_; y = y_; }

    Int2 operator + (Int2 a) { return Int2(x+a.x, y+a.y); }

    Int2 operator - (Int2 a) { return Int2(x-a.x, y-a.y); }

    boolean operator == (Int2 a) { return (x == a.x)&&(y == a.y); }
}
```

図 14.8:　C++による整数型 2 次元座標値クラスの定義：演算子メンバー関数を含む。signum()、length() は省略しました。

```
        Int2 target = position.add(direction);    // Int2 どうしの加算

        if (target.equal(game.monster.position)) { return false; }
```

は、以下のように、さらに簡潔に記述することもできます。

```
        Int2 target = position + direction;    // Int2 どうしの加算

        if (target == *(game->monster->position)) { return false; }
```

しかし簡潔すぎて逆にバグを作り込みやすいとも考えられるため、演算子関数には賛否両論があり、Java では導入は見送られています。プログラミング言語のデザインはなかなか難しいものです。

14.9　付録：この章の全プログラムリスト

以下は視界の悪いダンジョンのプログラムです。次章は、視界良好、かつ Int2 を用いる場合を前提として進めていきます。

```
//ゲームクラスのプログラム
import java.io.*;

class Game {
    public static void main(String args[]) {
        Game game = new Game();
        game.mainLoop();
    }

    Map map;
    Player player;
    Monster monster1;
    Monster monster2;

    Game() {
        map = new Map(this);
        player = new Player(this);
        monster1 = new Monster(this, new Int2(1, Map.YSIZE-2));
        monster2 = new Monster(this, new Int2(Map.XSIZE-2, 1));
    }

    void print() {
        map.print();
        System.out.println(player);   // toString() で自動変換されて表示される
        System.out.println(monster1); // toString() で自動変換されて表示される
        System.out.println(monster2); // toString() で自動変換されて表示される
        System.out.println();
    }

    void mainLoop() {
        print();
        while (true) { turn(); }
    }

    void turn() {
        player.action();
        monster1.action();
        monster2.action();
        print();
    }

    static char getKeyChar() {
        char c = 0;
        System.out.print("? ");
        try {
            BufferedReader in = new BufferedReader(
                                new InputStreamReader(System.in));
            c = in.readLine().charAt(0);
        } catch(IOException e){
            System.out.println("ERROR:: getKeyChar()");
        }
        return c;
```

```
        }
}
```

```
//マップクラスのプログラム
class Map {
    static final int ROCK = 0;
    static final int EMPTY = 1;
    static final int GOAL = 2;

    static final int XSIZE = 13;
    static final int YSIZE = 11;

    Game game;
    int data[][]= new int[XSIZE][YSIZE];

    Map(Game game) {
        this.game = game;
        for (int x = 0; x < XSIZE; x++) {
            for (int y = 0; y < YSIZE; y++) {
                if ((x == 0)||(x == XSIZE-1)||
                   (y == 0)||(y == YSIZE-1)||
                   ((Math.abs(x-XSIZE/2) <= 1)&&
                    (Math.abs(y-YSIZE/2) <= 1))) {
                    data[x][y] = ROCK;
                }
                else { data[x][y] = EMPTY; }
            }
        }
        data[XSIZE-2][YSIZE-2] = GOAL;
    }

    int getAt(Int2 position) { return data[position.x][position.y]; }
                                    // Int2 が示すセル位置のデータを取り出す

    void print() {
        for (int y = 0; y < YSIZE; y++) {
            for (int x = 0; x < XSIZE; x++) {
               Int2 position = new Int2(x,y);     // Int2 オブジェクトを生成
                if (position.sub(game.player.position).length() > 3) {
                    System.out.print("  ");
                    continue;
                }
                if (game.player.cellPrint(position)) { continue; }
                if (game.monster1.cellPrint(position)) { continue; }
                if (game.monster2.cellPrint(position)) { continue; }
                cellPrint(position);
            }
            System.out.println();
        }
    }

    void cellPrint(Int2 position) {                  // 引数を Int2 型へ変更
        int e = getAt(position);                     // セルデータを取り出す
        if (e == ROCK) { System.out.print('#'+" "); }
        else if (e == EMPTY) { System.out.print('.'+" "); }
        else if (e == GOAL) { System.out.print('G'+" "); }
        else { System.out.print('E'+' '); }
    }
}
```

```
1  //プレイヤークラスのプログラム
2  class Player {
3      static final int INIT_HP = 100;
4
5      Game game;
6      Int2 position;                          // 位置を Int2 オブジェクトに変更
7      int hp;
8
9      Player(Game game) {
10         this.game = game;
11         position = new Int2(1,1);         // 位置を Int2 オブジェクトで初期設定
12         hp = INIT_HP;
13     }
14
15     public String toString() {                // print() を toString() に置換
16         return "Player : "+position+" "+hp;     // position を文字列へ変換
17     }
18
19     boolean cellPrint(Int2 position) {       // 引数を Int2 型へ変更
20         if (this.position.equal(position)) { // 位置の比較を Int2 どうしで行う
21             System.out.print('@'+" ");
22             return true;
23         }
24         return false;
25     }
26
27     boolean moveTo(Int2 direction) {                 // 引数を Int2 型へ変更
28         Int2 target = position.add(direction);         // Int2 どうしの加算
29
30         if (game.map.getAt(target) == Map.ROCK) { return false; }
31                                                          // セルデータは岩か？
32         if (target.equal(game.monster1.position)) { return false; }
33                                                                  // 位置の比較
34         if (target.equal(game.monster2.position)) { return false; } // 同上
35
36         position = target;                                       // 位置の更新
37         return true;
38     }
39
40     void attackTo(Int2 direction) {                   // 引数を Int2 型へ変更
41         Int2 target = position.add(direction);          // Int2 どうしの加算
42         if (target.equal(game.monster1.position)) { // 位置の比較
43             game.monster1.hp -= 20;
44         }
45         if (target.equal(game.monster2.position)) { // 同上
46             game.monster2.hp -= 20;
47         }
48     }
49
50     void action() {
51         Int2 direction;                              // Int2 型変数を用意する
52         switch (Game.getKeyChar()) {
53             case 'z': direction = new Int2(-1,+1); break; // Int2 オブジェク
54             case 'x': direction = new Int2( 0,+1); break; // トを条件に応じ
55             case 'c': direction = new Int2(+1,+1); break; // て生成し、その
56             case 'a': direction = new Int2(-1, 0); break; // 参照を
57             case 'd': direction = new Int2(+1, 0); break; // direction に
58             case 'q': direction = new Int2(-1,-1); break; // 代入
```

```
            case 'w': direction = new Int2( 0,-1); break;
            case 'e': direction = new Int2(+1,-1); break;
            default:  direction = new Int2( 0, 0);
        }

        if (!moveTo(direction)) { attackTo(direction); }
                                        // Int2 を引数としてメソッド呼び出しを行う
    }
}
```

```
//モンスタークラスのプログラム
class Monster {
    static final int INIT_HP = 150;

    Game game;
    Int2 position;                          // 位置を Int2 オブジェクトに変更
    int hp;

    Monster(Game game, Int2 position) {
        this.game = game;
        this.position = position;                     // Int2 型引数で初期化
        hp = INIT_HP;
    }

    public String toString() {                // print() を toString() に置換
        return "Monster : "+position+" "+hp;   // position を文字列へ変換
    }

    boolean cellPrint(Int2 position) {                // 引数を Int2 型へ変更
        if (this.position.equal(position)) { // 位置の比較を Int2 どうしで行う
            System.out.print('M'+" ");
            return true;
        }
        return false;
    }

    boolean moveTo(Int2 direction) {                  // 引数を Int2 型へ変更
        Int2 target = this.position.add(direction);     // Int2 どうしの加算
        if (game.map.getAt(target) == Map.ROCK) { return false; }
                                                         // セルデータは岩か？
        if (target.equal(game.player.position)) { return false; }
                                                                  // 位置の比較
        if (target.equal(game.monster1.position)) { return false; } // 同上
        if (target.equal(game.monster2.position)) { return false; } // 同上
        position = target;                            // Int2 での位置の更新
        return true;
    }

    void attackTo(Int2 direction) {                   // 引数を Int2 型へ変更
        Int2 target = this.position.add(direction);     // Int2 どうしの加算
        if (target.equal(game.player.position)) {
                                                // 位置の比較を Int2 どうしで行う
            game.player.hp -= 30;
        }
    }

    void action() {
        Int2 direction = game.player.position.sub(position).signum();
                 // Int2 上で、プレイヤーとモンスターの位置の差に符号関数を適用

        if (!moveTo(direction)) { attackTo(direction); }
                                      // Int2 を引数としてメソッド呼び出しを行う
    }
}
```

```
//2 次元ベクトルクラスのプログラム
class Int2 {
    int x,y;

    Int2(int x, int y) { this.x = x; this.y = y; }       // コンストラクタ

    Int2 add(Int2 a) { return new Int2(x+a.x, y+a.y); }  // 加算

    Int2 sub(Int2 a) { return new Int2(x-a.x, y-a.y); }  // 減算

    Int2 signum() {                                      // 要素毎の符号関数
        return new Int2(Integer.signum(x), Integer.signum(y));
    }

    boolean equal(Int2 a) { return (x == a.x)&&(y == a.y); } // 比較

    double length() { return Math.sqrt((x*x+y*y)); }     // 長さ

    public String toString(){ return "("+x+","+y+")"; }  // 文字列への変換
}
```

1

1

第15章　プレイヤーとモンスターの共通データをアクターにまとめる：クラス階層の導入（その1）

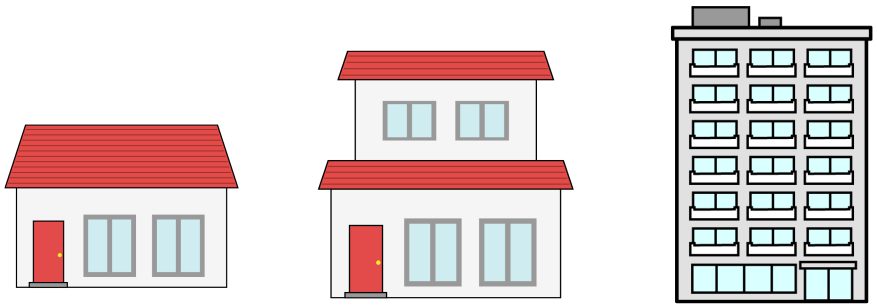
この章から、オブジェクト指向の基本原理のひとつ、**継承**（inheritance、インヘリタンス）の解説に入ります。継承とは、親の性質を子供が受け継ぐことです。これをクラス階層に適用します。これによってプログラムの集中管理と再利用が可能になります。

プログラムを家に例えるならば、クラス階層を用いないプログラムは平屋建ての家に相当するイメージです（図 15.1（221 ページ）(a) 参照）。延べ床面積（建物の各階の床面積の合計）を広げようとすれば、広い敷地に何棟もの平家を建てる必要があります。敷地内を移動する距離も増えます。ただし家全体の構造は比較的単純です。クラス階層を用いるプログラムは、2 階建て、3 階建ての家に相当するイメージです（図 15.1(b) 参照）。コンパクトにまとまっており、狭い空間に機能を充実させることができます。ただし構造が複雑になりがちです。そしてこのテキストの終盤では Java から提供される高機能クラスをクラス階層を介して利用します。それはインテリジェントビルの一角に住むイメージです（図 15.1(c) 参照）。提供される高機能インフラを活用して快適なプログラミングを実現します。

この講義テキストでは、継承をこの章と次章に分けて解説します。この章ではインスタンス変数を継承する場合のみを扱います。次章ではさらにメソッドを継承する場合を扱います。

15.1　簡単な例題

まず簡単な例題です。以下は、「`ClassB` が `ClassA` を拡張（extends）する」場合のプログラムです。



(a) クラス階層がない場合　(b) クラス階層がある場合　(c) クラス階層を高度に利用する場合

図 15.1: 講義担当者のクラス階層のイメージ図

```
class ClassA {
    int a = 111;
}

class ClassB extends ClassA {          // ClassB は ClassA を拡張したクラス
    int b = 222;
    void g() {
        System.out.println(a+" / "+b);  // a は ClassB には定義されていない
    }
}
```

ClassA を ClassB の**親クラス**（parent class）、**スーパークラス**（super class）、**ベースクラス**（base class）などと呼びます。逆に ClassB は ClassA の**子クラス**（child class）、**サブクラス**（subclass）などと呼びます。

さて、オブジェクト指向では子クラスは親クラスの性質を受け継ぐ（inherit）ことができます。この機能を継承と呼びます。

上の例では ClassA にインスタンス変数 a が定義されています。よって ClassA の性質を受け継ぐ ClassB は自動的にインスタンス変数 a を持つことになり、変数 a と b を出力するメソッド g() は正常に実行されて、以下の出力を得ます。

```
111 / 222
```

ClassA は複数の子クラスを持つことができます。以下は、ClassA を親クラスとする ClassC の定義です。

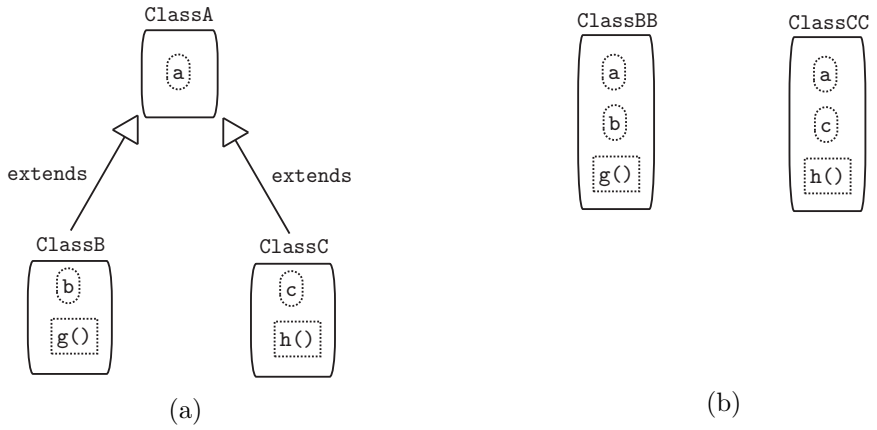


図 15.2: 簡単な例題の五つのクラスの継承関係の図式化

```
class ClassC extends ClassA {          // ClassC は ClassA を拡張したクラス
    int c = 333;
    void h() {
        System.out.println(a+" : "+c);  // a は ClassC には定義されていない
    }
}
```

ClassB とは異なる定義になっていることに注意してください。しかし ClassA からインスタンス変数 a を継承する点では共通しています。この講義テキストではこれら三つのクラス間の継承関係を図 15.2 (a) のように図式化します。

ところで ClassB、ClassC は以下のように定義した ClassBB、ClassCC と等価です。これを図示したものが図 15.2 (b) です。

```
class ClassBB {
    int a = 111;
    int b = 222;
    void g() {
        System.out.println(a+" / "+b);
    }
}

class ClassCC {
    int a = 111;
    int c = 333;
    void h() {
        System.out.println(a+" : "+b);
    }
}
```

上のプログラムでは ClassBB、ClassCC に重複する記述 `int a = 111;` が見られます。これがプログラム記述の集中管理の点から問題になります。

たとえば変数 a の初期値を変更する必要が生じたと仮定しましょう。そのとき、元の ClassA、ClassB、ClassC のプログラムでは、ClassA 内を変更するだけで済みます。しかし、ClassBB、ClassCC のプログラムでは、それぞれの a の初期化部分を変更する必要があり、非効率です。単に手間が 2 倍に増えただけではありません。同一の記述がプログラム中にばら撒かれているとき、変更すべきか箇所を全て見つけ出さねばなりません。一部を変更し、一部を変更しないなどという不都合は事態をより混乱させてしまうからです。継承は、クラス間の自然な関係性の中でプログラムの集中管理をすっきりと実現する優れた手法と考えられています。

ところで、以下のプログラムは正常にコンパイルできますが、これは良くないプログラムです。というのも、親クラスのインスタンス変数名と子クラスのインスタンス変数名に重複があります。

```
class ClassA {
    int a = 111;
}

class ClassB extends ClassA {
    int a = 333;             // インスタンス変数名が親クラスと重複
    int b = 222;
    void g() {
        System.out.println(a+" "+b);
    }
}
```

ちなみに、これを実行した結果は以下のとおりです。

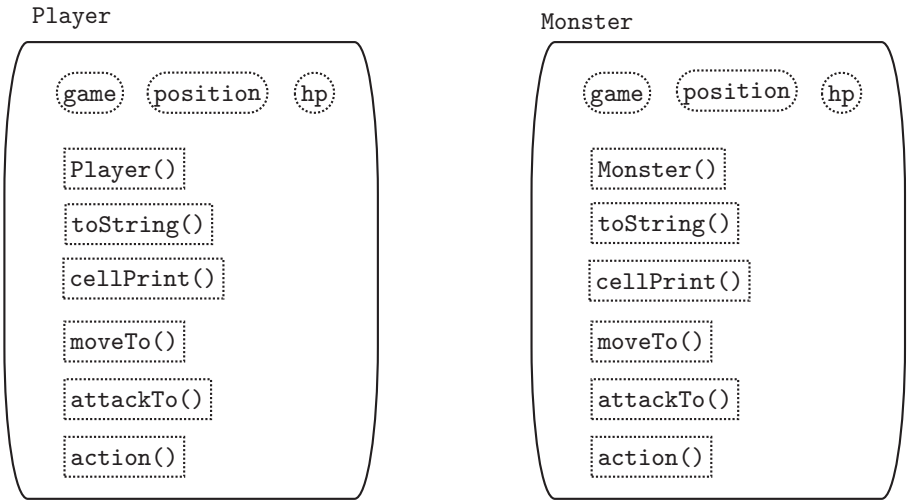
```
333 222
```

重複した場合、親クラスの変数は見えません。一般にクラスの継承関係の上下で同じ名前の変数名を使うことは避けるべきです。

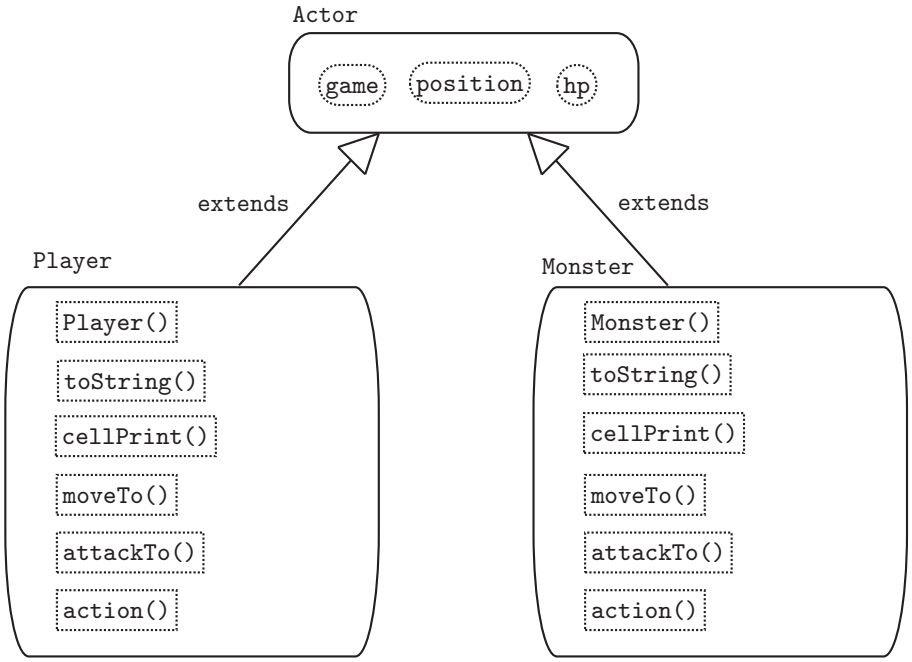
この節の簡単な例題は継承の機能を説明するために紹介しました。次節からはダンジョンゲームを題材に継承の効果的な使い方について解説します。

15.2　アクターの定義

図 15.3（224 ページ）は前章のプログラムのプレイヤーとモンスターの内部構造を図示したものです。同じインスタンス変数を持ち、同じ名前のインスタンスメソッドを持ちます。というのも、



(a) 前章の構造



(b) この章の構造

図 15.3: 前章とこの章のプレイヤーとモンスターの構造

```
class Actor {
    Game game;
    Int2 position;
    int hp;
}
```

図 15.4: Actor を導入したダンジョンゲームのプログラム（アクター）

```
class Player extends Actor {           // Actor が親クラス
    static final int INIT_HP = 100;

    Player(Game game) { ** 前章から変更なし ** }

    public String toString() { ** 前章から変更なし ** }

    boolean cellPrint(Int2 position) { ** 前章から変更なし ** }

    boolean moveTo(Int2 direction)  { ** 前章から変更なし ** }

    void attackTo(Int2 direction) { ** 前章から変更なし ** }

    void action() { ** 前章から変更なし ** }
}
```

図 15.5: Actor を導入したダンジョンゲームのプログラム（プレイヤー）

プレイヤーとモンスターはダンジョンゲームのキャラクターとして本質的に共通する性質を持つからです。そこで、その共通する役割を抽出し、共通化し、プログラム記述を集中管理することがこの章と次章の目的です。特にこの章ではインスタンス変数を共通化します。

図 15.3 が示すように、プレイヤーとモンスターには共通するインスタンス変数 game、position、hp が定義されています。それぞれのデータ型も同じです。そこでこれら三つのインスタンス変数を括（くく）り出し、それらを持つ親クラス Actor（アクター）を図 15.4（225 ページ）のように定義します。

15.3　サブクラスの定義

次に、プレイヤーとモンスターをアクターを拡張（extend）する形式に変更します。三つのインスタンス変数はアクターへ移動したため、プレイヤーとモンスターからはそれらを削除します。図 15.5（225 ページ）、図 15.6（226 ページ） がそれぞれサブクラス化されたプレイヤー、モンスターのプログラムです。この章の変更では、単にインスタンス変数を親クラスに移動しただけですから、メソッド記述に変更はありません。

```
class Monster extends Actor {           // Actor が親クラス
    static final int INIT_HP = 150;

    Player(Game game) { ** 前章から変更なし ** }

    public String toString() { ** 前章から変更なし ** }

    boolean cellPrint(Int2 position) { ** 前章から変更なし ** }

    boolean moveTo(Int2 direction)  { ** 前章から変更なし ** }

    void attackTo(Int2 direction) { ** 前章から変更なし ** }

    void action() { ** 前章から変更なし ** }
}
```

図 15.6: Actor を導入したダンジョンゲームのプログラム（モンスター）

図 15.7（227 ページ）は、このように定義された Player オブジェクトの内部構造です。クラスの継承関係（図 15.3 (b)）ではないことに注意してください。オブジェクトの内部には、中核部にアクター固有のオブジェクト部分があり、その外側にプレイヤー固有のオブジェクト部分が拡張されています。プレイヤーのメソッドからはアクターオブジェクトのインスタンス変数に自由にアクセスできます[1]。オブジェクトの外からもアクターオブジェクトのインスタンス変数にアクセスできます。結果として、 Player オブジェクトは前章までのオブジェクトと同等の機能を持ちます。Monster オブジェクトの内部構造も同様です。

15.4　補足：クラス階層の C++での記述

C++でも Java と同様にクラス階層を構築することができます。以下は、この章の最初の簡単な例題を C++に翻訳したものです。

[1] アクター部分へのアクセスに制限を加え、堅牢なオブジェクトを作ることもできますが、アクセス制限については最終章でまとめて述べます。

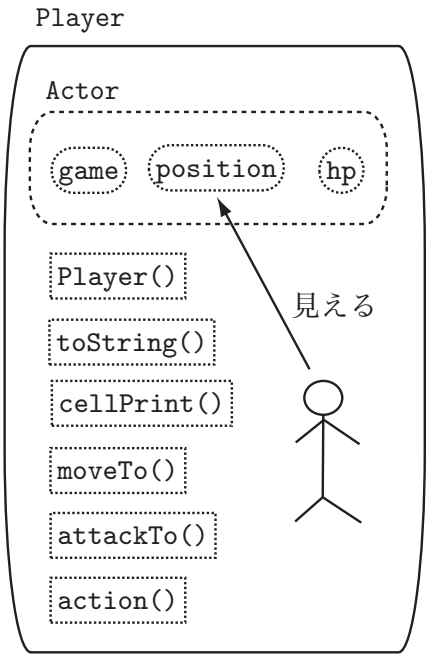


図 15.7: Player オブジェクトの内部構造：内部にアクターオブジェクトが含まれ、そこはプレイヤーのメソッドからアクセスできる。

```
class ClassA {
  public:
    int a = 111;
};

class ClassB : public ClassA {
  public:
    int b = 222;
    void g() {
        cout << a << " / " << b << endl;
    }
};
```

C++では、以下で親クラスを記述します。

```
class 子クラス : public 親クラス {
```

ここに、public は親クラスの定義を子クラスに公開することを表します。親クラスの定義を子クラスに公開＊しない＊ことは通常あり得ないため、この public はほとんどの場合に付けます。

クラス階層に関する基本概念は C++と Java で共通していますから、この章の Java に関する説

明は C++についても成り立ちます。

注意　上の C++プログラムのメンバー変数のクラス内初期化

```
    int a = 111;
    int b = 222;
```

は、C++11（C++の 2011 年国際標準規格）以前のコンパイラではエラーになる可能性があります。以前の C++では初期化はコンストラクタの冒頭で行なう仕様になっています。それが以下の `: a(111)`、`: b(222)` の部分です。詳細は省略します。

```
class ClassA {
  public:
    int a;
    ClassA() : a(111) {} // メンバー変数の初期化
};

class ClassB : public ClassA {
  public:
    int b;
    ClassB() : b(222) {} // メンバー変数の初期化
    void g() {
        cout << a << " / " << b << endl;
    }
};
```

15.5　付録：この章の全プログラムリスト

```
//ゲームクラスのプログラム
import java.io.*;

class Game {
    public static void main(String args[]) {
        Game game = new Game();
        game.mainLoop();
    }

    Map map;
    Player player;
    Monster monster1;
    Monster monster2;

    Game() {
        map = new Map(this);
        player = new Player(this);
        monster1 = new Monster(this, new Int2(1, Map.YSIZE-2));
        monster2 = new Monster(this, new Int2(Map.XSIZE-2, 1));
    }

    void print() {
        map.print();
        System.out.println(player);
        System.out.println(monster1);
        System.out.println(monster2);
        System.out.println();
    }

    void mainLoop() {
        print();
        while (true) { turn(); }
    }

    void turn() {
        player.action();
        monster1.action();
        monster2.action();
        print();
    }

    static char getKeyChar() {
        char c = 0;
        System.out.print("? ");
        try {
            BufferedReader in = new BufferedReader(
                                new InputStreamReader(System.in));
            c = in.readLine().charAt(0);
        } catch(IOException e){
            System.out.println("ERROR:: getKeyChar()");
        }
        return c;
    }
}
```

```
//マップクラスのプログラム
class Map {
    static final int ROCK = 0;
    static final int EMPTY = 1;
    static final int GOAL = 2;

    static final int XSIZE = 13;
    static final int YSIZE = 11;

    Game game;
    int data[][]= new int[XSIZE][YSIZE];

    Map(Game game) {
        this.game = game;
        for (int x = 0; x < XSIZE; x++) {
            for (int y = 0; y < YSIZE; y++) {
                if ((x == 0)||(x == XSIZE-1)||
                   (y == 0)||(y == YSIZE-1)||
                   ((Math.abs(x-XSIZE/2) <= 1)&&
                    (Math.abs(y-YSIZE/2) <= 1))) {
                    data[x][y] = ROCK;
                }
                else { data[x][y] = EMPTY; }
            }
        }
        data[XSIZE-2][YSIZE-2] = GOAL;
    }

    int getAt(Int2 position) { return data[position.x][position.y]; }

    void print() {
        for (int y = 0; y < YSIZE; y++) {
            for (int x = 0; x < XSIZE; x++) {
                Int2 position = new Int2(x,y);
                if (game.player.cellPrint(position)) { continue; }
                if (game.monster1.cellPrint(position)) { continue; }
                if (game.monster2.cellPrint(position)) { continue; }
                cellPrint(position);
            }
            System.out.println();
        }
    }

    void cellPrint(Int2 position) {
        int e = getAt(position);
        if (e == ROCK)       { System.out.print('#'+" "); }
        else if (e == EMPTY) { System.out.print('.'+" "); }
        else if (e == GOAL)  { System.out.print('G'+" "); }
        else                 { System.out.print('E'+' '); }
    }
}
```

```
//アクタークラスのプログラム
class Actor {
    Game game;
    Int2 position;
    int hp;
}
```

```
//プレイヤークラスのプログラム
class Player extends Actor {           // Actor が親クラス
    static final int INIT_HP = 100;

    Player(Game game) {
        this.game = game;
        position = new Int2(1,1);
        hp = INIT_HP;
    }

    public String toString() {
        return "Player : "+position+" "+hp;
    }

    boolean cellPrint(Int2 position) {
        if (this.position.equal(position)) {
            System.out.print('@'+" ");
            return true;
        }
        return false;
    }

    boolean moveTo(Int2 direction) {
        Int2 target = position.add(direction);

        if (game.map.getAt(target) == Map.ROCK) { return false; }
        if (target.equal(game.monster1.position)) { return false; }
        if (target.equal(game.monster2.position)) { return false; }

        position = target;
        return true;
    }

    void attackTo(Int2 direction) {
        Int2 target = position.add(direction);
        if (target.equal(game.monster1.position)) {
            game.monster1.hp -= 20;
        }
        if (target.equal(game.monster2.position)) {
            game.monster2.hp -= 20;
        }
    }

    void action() {
        Int2 direction;
        switch (Game.getKeyChar()) {
            case 'z': direction = new Int2(-1,+1); break;
            case 'x': direction = new Int2( 0,+1); break;
            case 'c': direction = new Int2(+1,+1); break;
            case 'a': direction = new Int2(-1, 0); break;
            case 'd': direction = new Int2(+1, 0); break;
            case 'q': direction = new Int2(-1,-1); break;
            case 'w': direction = new Int2( 0,-1); break;
            case 'e': direction = new Int2(+1,-1); break;
            default:  direction = new Int2( 0, 0);
        }

        if (!moveTo(direction)) { attackTo(direction); }
```

```
    }
}
```

```
//モンスタークラスのプログラム
class Monster extends Actor {           // Actor が親クラス
    static final int INIT_HP = 150;

    Monster(Game game, Int2 position) {
        this.game = game;
        this.position = position;
        hp = INIT_HP;
    }

    public String toString() {
        return "Monster : "+position+" "+hp;
    }

    boolean cellPrint(Int2 position) {
        if (this.position.equal(position)) {
            System.out.print('M'+" ");
            return true;
        }
        return false;
    }

    boolean moveTo(Int2 direction) {
        Int2 target = position.add(direction);

        if (game.map.getAt(target) == Map.ROCK) { return false; }
        if (target.equal(game.player.position)) { return false; }
        if (target.equal(game.monster1.position)) { return false; }
        if (target.equal(game.monster2.position)) { return false; }

        position = target;
        return true;
    }

    void attackTo(Int2 direction) {
        Int2 target = this.position.add(direction);
        if (target.equal(game.player.position)) {
            game.player.hp -= 30;
        }
    }

    void action() {
        Int2 direction = game.player.position.sub(position).signum();

        if (!moveTo(direction)) { attackTo(direction); }
    }
}
```

```
//2 次元ベクトルクラスのプログラム
class Int2 {
    int x,y;

    Int2(int x, int y) { this.x = x; this.y = y; }

    Int2 add(Int2 a) { return new Int2(x+a.x,y+a.y); }
    Int2 sub(Int2 a) { return new Int2(x-a.x,y-a.y); }
    Int2 signum() { return new Int2(Integer.signum(x),Integer.signum(y)); }
    boolean equal(Int2 a) { return (x == a.x)&&(y == a.y); }

    public String toString(){ return "("+x+","+y+")"; }
}
```

第 15 章　プレイヤーとモンスターの共通データをアクターにまとめる：クラス階層の導入（その 1）

1

第 15 章　プレイヤーとモンスターの共通データをアクターにまとめる：クラス階層の導入（その 1）

1

第16章　プレイヤーとモンスターの共通メソッドをアクターにまとめる：クラス階層の導入（その2）

この章では前章から引き続いて継承について解説します。この章では親クラスにメソッドがある場合を扱います。

16.1　簡単な例題

以下はこの章の簡単な例です。前章同様に ClassB は ClassA を拡張（extends）したクラスです。前章と異なるのは ClassA がメソッド f() を持つことです。

```
class ClassA {
    int a = 111;
    void f() {
        System.out.println(a); // b にはアクセスできない。
    }
}

class ClassB extends ClassA {
    int b = 222;
    void g() {
        f();
        System.out.println(a+" / "+b);
    }
}
```

親クラスのメソッドも子クラスへ継承されます。よって ClassB から f() を呼び出すことができます。しかし、 f() の本体から ClassB のインスタンス変数 b、インスタンスメソッド g() を呼び出すことはできません。というのも、ClassA は ClassB とは全く独立に定義されているクラスであり、ClassB の性質を継承することはないからです（子は親を継承しますが、親は子を継承しません）。

さて、以下は ClassA の内容をそのまま ClassB へコピーし、継承をしない例です。メソッド f()、 g() の実行結果は元のプログラムと同じです。

```
class ClassBB {
    int a = 111;
    int b = 222;
    void f() {
        System.out.println(a);
    }
    void g() {
        f();
        System.out.println(a+" / "+b);
    }
}
```

しかし前章でも述べたように、仮に実行結果が同じであっても、もし ClassA の子クラスが複数あるならば、クラス階層を用いない後者のプログラムは良いプログラムではありません。

前章ではインスタンス変数名がクラス階層の上下で重複する場合について、好ましくないと述べました。同様にして、メソッド名がクラス階層の上下で重複する場合を考えることができます。実はこの重複は好ましくないどころか、積極的に利用すべき事項になります。その理由、使い方を含め、次章から最終章まで様々に論じていくことになります。

16.2　アクタークラスの変更

図 16.1（240 ページ） (a) が前章のクラス構造でした。この章では、図 16.1 (b) のように、プレイヤー、モンスターのメソッド cellPrint() と moveTo() をアクターに移動し、共通化します。

図 16.2（241 ページ）は Actor クラスの定義です。以下、具体的にプログラムを見ていきましょう。

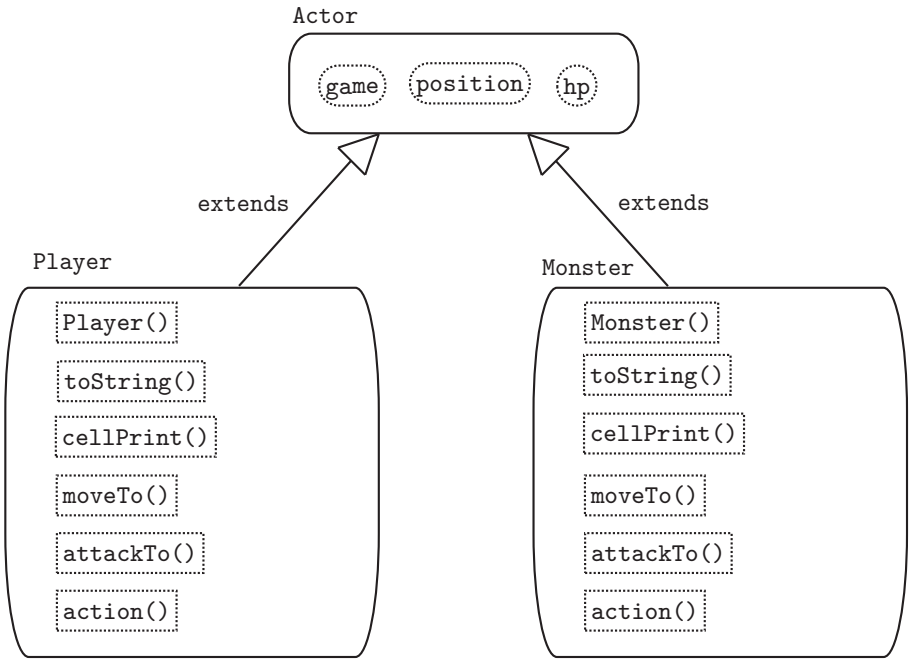
16.2.1　インスタンス変数 cellChar の追加

プレイヤーの cellPrint() とモンスターのメソッド cellPrint() では処理内容が異なります。プレイヤーの場合には '@' を表示し、モンスターの場合には'M' を表示します。この違いをアクタークラスの cellPrint() で吸収解消するために、新たに char 型のインスタンス変数 cellChar を導入します。

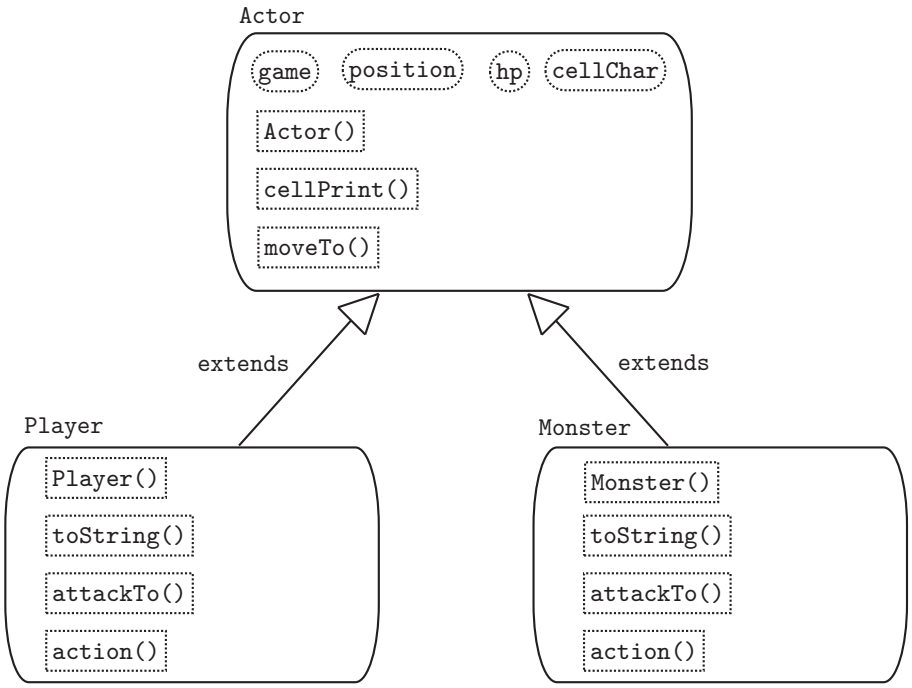
16.2.2　アクターのコンストラクタ

アクタークラスは通常のクラス定義に従いますから、コンストラクタを持つことができます。

図 16.2 のアクターのコンストラクタは四つの引数を四つのインスタンス変数にそれぞれ代入する単純なものとして定義しています。



(a) 前章の構造



(b) この章の構造

図 16.1: 前章とこの章のプレイヤーとモンスターの構造

```
class Actor {
    Game game;
    Int2 position;
    int hp;
    char cellChar;                              // cellPrint() での表示文字

    Actor(Game game, Int2 position, int hp, char cellChar) {
        this.game = game;
        this.position = position;
        this.hp = hp;
        this.cellChar = cellChar;        // サブクラスから受け取った文字を保存
    }

    boolean cellPrint(Int2 position) {
        if (this.position.equal(position)) {
            System.out.print(cellChar+" ");      //cellChar を出力
            return true;
        }
        return false;
    }

    boolean moveTo(Int2 direction) {
        Int2 target = position.add(direction);

        if (game.map.getAt(target) == Map.ROCK) { return false; }

        if (target.equal(game.player.position)) { return false; }
        if (target.equal(game.monster1.position)) { return false; }
        if (target.equal(game.monster2.position)) { return false; }

        position = target;
        return true;
    }
}
```

図 16.2: Actor に二つのメソッドを移動したダンジョンゲームのプログラム（アクター）

16.2.3　アクターのマップ表示 cellPrint()

メソッド cellPrint() は、もし引数の position がアクターの位置 this.position に等しいならば、インスタンス変数 cellChar を表示し、 true を戻します。さもなくば、 false を戻します。

cellChar の値はプレイヤーならば ‘@’、モンスターならば ‘M’ ですが、その値はコンストラクタで事前に設定するものとします。

```
class Player extends Actor {
    static final int INIT_HP = 100;

    Player(Game game) {
        super(game, new Int2(1,1), INIT_HP, '@');
                                        //親クラスのコンストラクタの呼び出し
    }

    public String toString() { ** 前章から変更ない ** }

    void attackTo(Int2 direction)  { ** 前章から変更ない ** }

    void action()  { ** 前章から変更ない ** }
}
```

図 16.3: Actor に二つのメソッドを移動したダンジョンゲームのプログラム（プレイヤー）

16.2.4　アクターの移動 MoveTo()

プレイヤーの MoveTo() も、モンスターの MoveTo() も、引数で指定された移動したい位置にプレイヤーもモンスターもいなければ、そこへ移動します。その意味でアクターへの共通化は容易です。

図 16.2 の MoveTo() は、前章のプレイヤーの MoveTo() に次の 1 行：

```
        if (target.equal(game.player.position)) { return false; }
```

を追加したものです。アクターがプレイヤーの場合、この 1 行の実行は無駄なのですが、共通化のためのコストとして容認することとします。また、図 16.2 の MoveTo() は前章のモンスターの MoveTo() と全く同じものです。

16.3　プレイヤークラスの変更

図 16.3（242 ページ）は Player クラスの定義です。メソッド cellPrint()、 moveTo() が消えたことに注意してください。

16.3.1　親クラスを表す super

キーワード this はオブジェクトが自分自身を指すときに用いる変数でした（9.3.1 項（128 ページ）参照）。類似の変数として、オブジェクトが自分の中の直上の親クラスのオブジェクト部分を

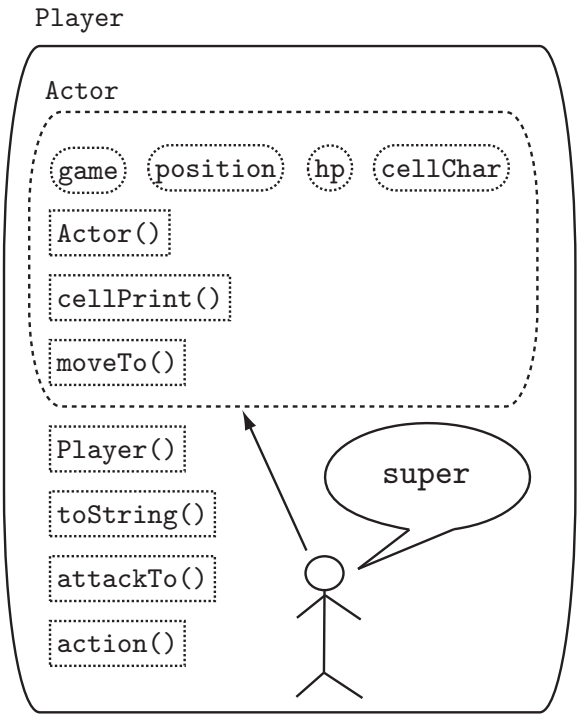


図 16.4: Player オブジェクトから見た super：自分の中のアクターオブジェクトを表す

表す変数 super があります[1]。文章では分かりにくいため、図 16.4（243 ページ）に図示しました。

Player オブジェクトのメソッドは、 super を使って親クラス部分のメソッドを呼び出すことができます。

以下は、図 16.3 のコンストラクタの本体です。ここに super が使われています。

```
        super(game, new Int2(1,1), INIT_HP, '@');
```

これは、 Player オブジェクトのコンストラクタの中で親クラスのコンストラクタを呼び出すことを表しています。

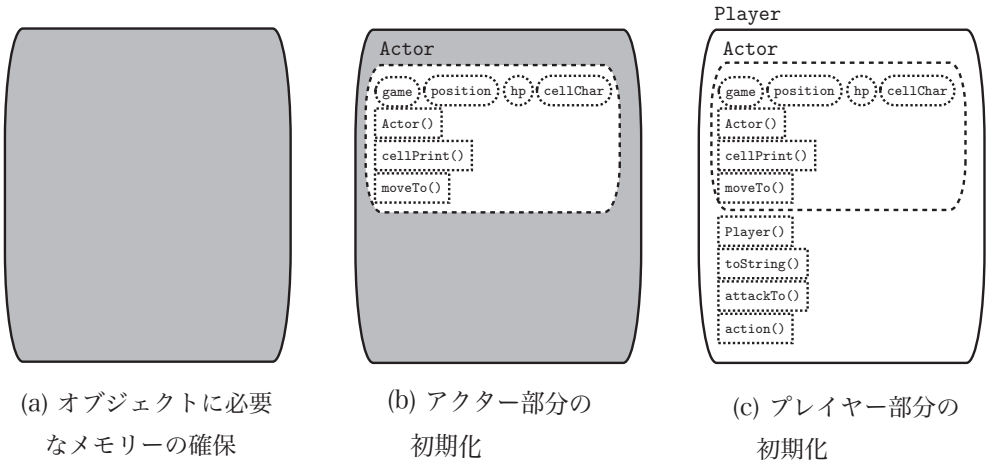
では、プレイヤーのコンストラクタとアクターのコンストラクタがどのように働いて Player オブジェクトが作られるのでしょうか。次項で解説します。

16.3.2　クラス階層におけるオブジェクトの構築処理

図 16.5（244 ページ）は、 Player オブジェクトが構築される手順を図式したものです。

一般にクラス階層内のオブジェクトは以下の 3 ステップで構築されます。

[1] super は Java のキーワードです。C++にはありません。



(a) オブジェクトに必要なメモリーの確保　(b) アクター部分の初期化　(c) プレイヤー部分の初期化

図 16.5: Player オブジェクトが構築される過程

1. そのオブジェクトに必要なメモリが確保されます。
2. 確保されたメモリ内で、親クラスに相当するメモリ領域[2]を親クラスのコンストラクタで初期化します。
3. 子クラスに相当するメモリ領域を子クラスのコンストラクタで初期化します。

これをプレイヤーのコンストラクタに当てはめると

1. プレイヤーのコンストラクタが Player(game) が起動されると、 Player オブジェクトに必要なメモリ領域が動的メモリ領域（図 8.1（108 ページ）参照）から確保されます。図 16.5 では、灰色の部分が確保されたメモリ領域です。領域内はまだ初期化されていません。
2. 次に、プレイヤーのコンストラクタの本体の

```
super(game, new Int2(1,1), INIT_HP, '@');
```

が実行されます。ここで言う super は Actor ですから、結局、コンストラクタ：

```
Actor(game, new Int2(1,1), INIT_HP, '@');
```

が呼び出され、図 16.5 のアクター部分を初期化します。プレイヤー部分はまだ灰色のままです。

[2]通常、メモリの先頭部分

3. 処理は、再度、プレイヤーのコンストラクタに戻り、プレイヤー部分の初期化を行います。この例題のプレイヤー固有の明示的な初期化処理はありませんが、実際には明示されていない処理が多々あり（後章で解説します）、それを実行します。結果として、図 16.5 ではプレイヤー固有の部分が初期化され、全体の処理化が完了します。

この手順から類推されるように、

```
        super(game, new Int2(1,1), INIT_HP, '@');
```

は、プレイヤーのコンストラクタの本体の冒頭に記述されねばなりません。

いろいろ複雑ですが、しっかり理解しましょう。

16.3.3　クラス階層におけるスコープ

これについては前章でも触れましたが、ここでは親クラスにメソッドを許す場合を含めて解説します。

図 16.6（246 ページ）は、クラス階層のスコープを図示したものです。前章同様、子クラスのメソッド内からは、親クラスのインスタンス変数、インスタンスメソッドにアクセスできます。プレイヤーは自分の中核に**必ず**アクターオブジェクト部分があることを知っているので、そこへアクセスできます。しかし、親クラスのメソッド内からは、子クラスのインスタンス変数、インスタンスメソッドにアクセスできません。そのオブジェクトがアクターそのものだった場合、アクターの外側に子クラス部分はないかも知れず、あるいは外側にプレイヤー部分があるかも知れず、モンスターがあるかも知れないため、一般にはアクセスが許されないのです[3]。

図 16.7（247 ページ）は、図 16.6 と同じ状況を、クラス階層図の上に図示したものです。簡単に言えば、上は見えるが、下は見えないというスコープルールです。

16.4　モンスタークラスの変更

図 16.8（248 ページ）は `Monster` クラスの定義です。変更内容はプレイヤーと同じです。

16.5　ゲームクラス/マップクラス/2 次元座標値クラスの変更

タイトルの 3 クラスについてはプログラムの変更はありません。

[3]例外として、プレイヤー責任で、アクターオブジェクトを `Player` オブジェクトに強引にデータ型変換をする（ダウンキャスト）も可能です。しかし、潜在的なバグの原因となるため、一般には推奨されません。

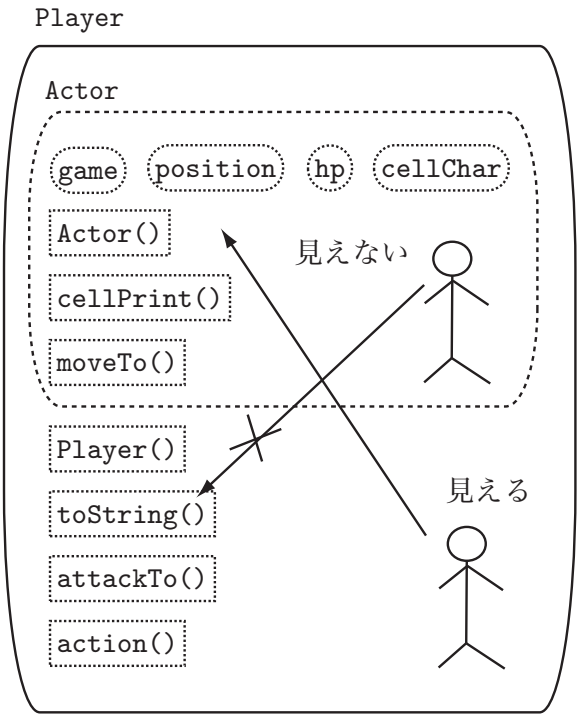


図 16.6: Player オブジェクトの内部構造：内部にアクターオブジェクトが含まれ、そこはプレイヤーのメソッドからアクセスできます。アクターのメソッドからはプレイヤーの部分は原理的に見えません。

16.6　補足：クラス階層の C++での記述

前章同様、この章の内容は C++でも成り立ちます。以下は、この章の簡単例題を C++でプログラミングしたものです。

```
class ClassA {
  public:
    int a = 111;
    void f() {
        cout << a << endl;
    }
};

class ClassB : public ClassA {
  public:
    int b = 222;
    void g() {
        f();
        cout << a << " / " << b << endl;
    }
};
```

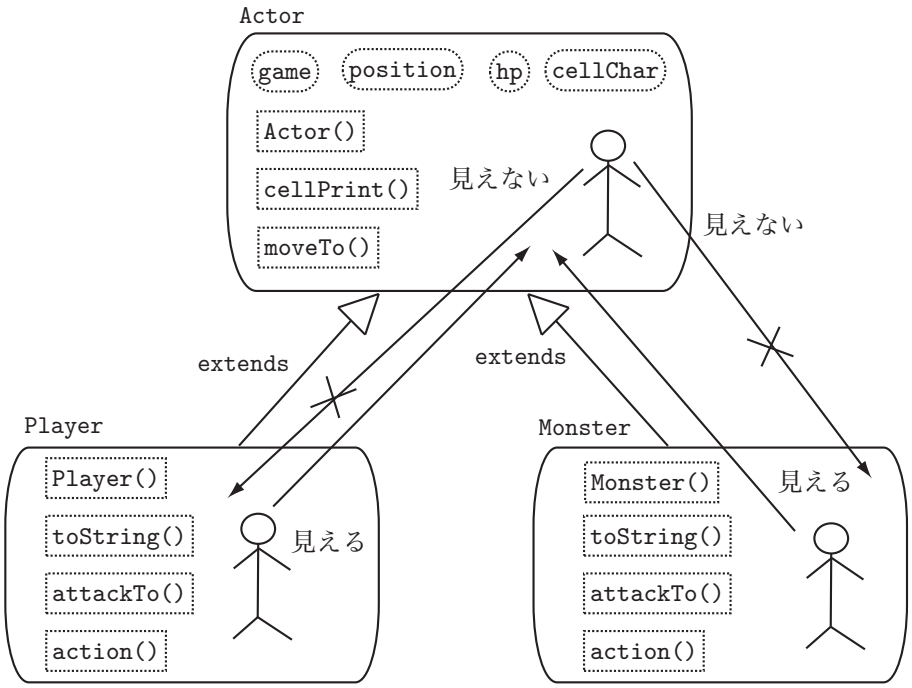


図 16.7: クラス階層でのスコープ：子クラスは親クラスを継承するから、プレイヤー、モンスターのメソッドからアクター部分は見えます。逆に、親クラスは子クラスを関知できませんから、アクターのメソッドからプレイヤー、モンスターの部分は見えません。

```
class Monster extends Actor {
    static final int INIT_HP = 150;

    Monster(Game game, Int2 position) {
        super(game, position, INIT_HP, 'M');
    }

    public String toString() { ** 前章から変更ない ** }

    void attackTo(Int2 direction) { ** 前章から変更ない ** }

    void action() { ** 前章から変更ない ** }
}
```

図 16.8: Actor に二つのメソッドを移動したダンジョンゲームのプログラム（モンスター）

16.7　付録：この章の全プログラムリスト

```
//ゲームクラスのプログラム
import java.io.*;

class Game {
    public static void main(String args[]) {
        Game game = new Game();
        game.mainLoop();
    }

    Map map;
    Player player;
    Monster monster1;
    Monster monster2;

    Game() {
        map = new Map(this);
        player = new Player(this);
        monster1 = new Monster(this, new Int2(1, Map.YSIZE-2));
        monster2 = new Monster(this, new Int2(Map.XSIZE-2, 1));
    }

    void print() {
        map.print();
        System.out.println(player);
        System.out.println(monster1);
        System.out.println(monster2);
        System.out.println();
    }

    void mainLoop() {
        print();
        while (true) { turn(); }
    }

    void turn() {
        player.action();
        monster1.action();
        monster2.action();
        print();
    }

    static char getKeyChar() {
        char c = 0;
        System.out.print("? ");
        try {
            BufferedReader in = new BufferedReader(
                                new InputStreamReader(System.in));
            c = in.readLine().charAt(0);
        } catch(IOException e){
            System.out.println("ERROR:: getKeyChar()");
        }
        return c;
    }
}
```

```
//マップクラスのプログラム
class Map {
    static final int ROCK = 0;
    static final int EMPTY = 1;
    static final int GOAL = 2;

    static final int XSIZE = 13;
    static final int YSIZE = 11;

    Game game;
    int data[][]= new int[XSIZE][YSIZE];

    Map(Game game) {
        this.game = game;
        for (int x = 0; x < XSIZE; x++) {
            for (int y = 0; y < YSIZE; y++) {
                if ((x == 0)||(x == XSIZE-1)||
                   (y == 0)||(y == YSIZE-1)||
                   ((Math.abs(x-XSIZE/2) <= 1)&&
                    (Math.abs(y-YSIZE/2) <= 1))) {
                    data[x][y] = ROCK;
                }
                else { data[x][y] = EMPTY; }
            }
        }
        data[XSIZE-2][YSIZE-2] = GOAL;
    }

    int getAt(Int2 position) { return data[position.x][position.y]; }

    void print() {
        for (int y = 0; y < YSIZE; y++) {
            for (int x = 0; x < XSIZE; x++) {
                Int2 position = new Int2(x,y);
                if (game.player.cellPrint(position)) { continue; }
                if (game.monster1.cellPrint(position)) { continue; }
                if (game.monster2.cellPrint(position)) { continue; }
                cellPrint(position);
            }
            System.out.println();
        }
    }

    void cellPrint(Int2 position) {
        int e = getAt(position);
        if (e == ROCK) { System.out.print('#'+" "); }
        else if (e == EMPTY) { System.out.print('.'+" "); }
        else if (e == GOAL) { System.out.print('G'+" "); }
        else { System.out.print('E'+' '); }
    }
}
```

```
//アクタークラスのプログラム
class Actor {
    Game game;
    Int2 position;
    int hp;
    char cellChar;                              // cellPrint() での表示文字

    Actor(Game game, Int2 position, int hp, char cellChar) {
        this.game = game;
        this.position = position;
        this.hp = hp;
        this.cellChar = cellChar;        // サブクラスから受け取った文字を保存
    }

    boolean cellPrint(Int2 position) {
        if (this.position.equal(position)) {
            System.out.print(cellChar+" ");      //cellChar を出力
            return true;
        }
        return false;
    }

    boolean moveTo(Int2 direction) {
        Int2 target = position.add(direction);

        if (game.map.getAt(target) == Map.ROCK) { return false; }
        if (target.equal(game.player.position)) { return false; }
        if (target.equal(game.monster1.position)) { return false; }
        if (target.equal(game.monster2.position)) { return false; }

        position = target;
        return true;
    }
}
```

```
//プレイヤークラスのプログラム
class Player extends Actor {
    static final int INIT_HP = 100;

    Player(Game game) {
        super(game, new Int2(1,1), INIT_HP, '@');
                                        //親クラスのコンストラクタの呼び出し
    }

    public String toString() {
        return "Player : "+position+" "+hp;
    }

    void attackTo(Int2 direction) {
        Int2 target = position.add(direction);
        if (target.equal(game.monster1.position)) {
            game.monster1.hp -= 20;
        }
        if (target.equal(game.monster2.position)) {
            game.monster2.hp -= 20;
        }
    }

    void action() {
        Int2 direction;
        switch (Game.getKeyChar()) {
            case 'z': direction = new Int2(-1,+1); break;
            case 'x': direction = new Int2( 0,+1); break;
            case 'c': direction = new Int2(+1,+1); break;
            case 'a': direction = new Int2(-1, 0); break;
            case 'd': direction = new Int2(+1, 0); break;
            case 'q': direction = new Int2(-1,-1); break;
            case 'w': direction = new Int2( 0,-1); break;
            case 'e': direction = new Int2(+1,-1); break;
            default:  direction = new Int2( 0, 0);
        }

        if (!moveTo(direction)) { attackTo(direction); }
    }
}
```

```
//モンスタークラスのプログラム
class Monster extends Actor {
    static final int INIT_HP = 150;

    Monster(Game game, Int2 position) {
        super(game, position, INIT_HP, 'M');
    }

    public String toString() {
        return "Monster : "+position+" "+hp;
    }

    void attackTo(Int2 direction) {
        Int2 target = this.position.add(direction);
        if (target.equal(game.player.position)) {
            game.player.hp -= 30;
        }
    }

    void action() {
        Int2 direction = game.player.position.sub(position).signum();

        if (!moveTo(direction)) { attackTo(direction); }
    }
}
```

```
//2次元ベクトルクラスのプログラム
class Int2 {
    int x,y;

    Int2(int x, int y) { this.x = x; this.y = y; }

    Int2 add(Int2 a) { return new Int2(x+a.x,y+a.y); }
    Int2 sub(Int2 a) { return new Int2(x-a.x,y-a.y); }
    Int2 signum() { return new Int2(Integer.signum(x),Integer.signum(y)); }
    boolean equal(Int2 a) { return (x == a.x)&&(y == a.y); }

    public String toString(){ return "("+x+","+y+")"; }
}
```

第16章　プレイヤーとモンスターの共通メソッドをアクターにまとめる：クラス階層の導入（その2）

1

1

第17章　レッドモンスターはモンスターの拡張である：メソッドの上書き

この章では、前章で予告した同じメソッド名がクラス階層の上下で重複する場合について検討を始めます。「始めます」というのは、この重複はオブジェクト指向の多相性の原理に関わる重要事項のため、最終章まで様々な形でこの重複について検討していくことになるからです。

17.1　簡単な例題

以下は ClassB は ClassA を拡張（extends）したクラスです。メソッド f() が ClassA にも ClassB にも定義されていることに注意してください。

```
class ClassA {
    int a = 111;
    void f() {
        System.out.println(a);
    }
}

class ClassB extends ClassA {
    int b = 222;
    void f() {
        System.out.println(a+" / "+b);
    }
}
```

このとき、以下のプログラム片：

```
    ClassB objectB = new ClassB();

    objectB.f();
```

が実行された場合、 ClassB の f() が実行されます。よって出力は以下のとおりです。

```
111 / 222
```

ClassB は ClassA の f() を継承しますが、 ClassB に同じ名前のメソッドが定義されている場合には、親クラスから継承されるメソッドは無視されます。これを当該クラスのメソッドで親クラスのメソッドが**上書き**（override、オーバライド）されていると言います。

もちろん以下のプログラム：

```
    ClassA objectA = new ClassA();

    objectA.f();
```

が実行された場合には ClassA の f() が実行されます。親クラスの挙動が子クラスの定義に影響されることはありませんから、この場合の出力は以下のとおりです。

```
111
```

上書きを使えば、親クラスから継承させる性質の中で子クラスで採用したくない性質を無視することができます。無視する場合には上書きし、子クラス固有の処理を記述すればよいのです。

有名な例を紹介します。鳥は羽を持ちます。飛ぶことができます。ペンギンは鳥の一種ですから、鳥の性質を継承するのは自然です。よってペンギンは飛ぶことができません。この関係をプログラム化したものが以下です。

```
class Bird {                            // 鳥
    boolean hasWings() { return true; }
    boolean canFly() { return true; }   // 飛ぶことができる
}

class Penguin extends Bird {            // ペンギン
    boolean canFly() { return false; }  // 飛ぶことはできない
}
```

この章のダンジョンゲームでは、モンスターの攻撃力を増したレッドモンスター[1]をモンスターのサブクラスとして導入し、攻撃部分を上書きします。

17.2　レッドモンスタークラスの定義

まず、図 17.1（259 ページ）はこの章のクラス階層を図示したものです。RedMonster クラスは Monster クラスのサブクラスになっています。このクラスにはインスタンス変数はなく、コンスト

[1]釣りの世界ではアオリイカのことをレッドモンスターとも呼ぶのだそうです。このテキストのレッドモンスターはそれとは関係ありません。強そうなモンスターのための安易な命名に過ぎません。

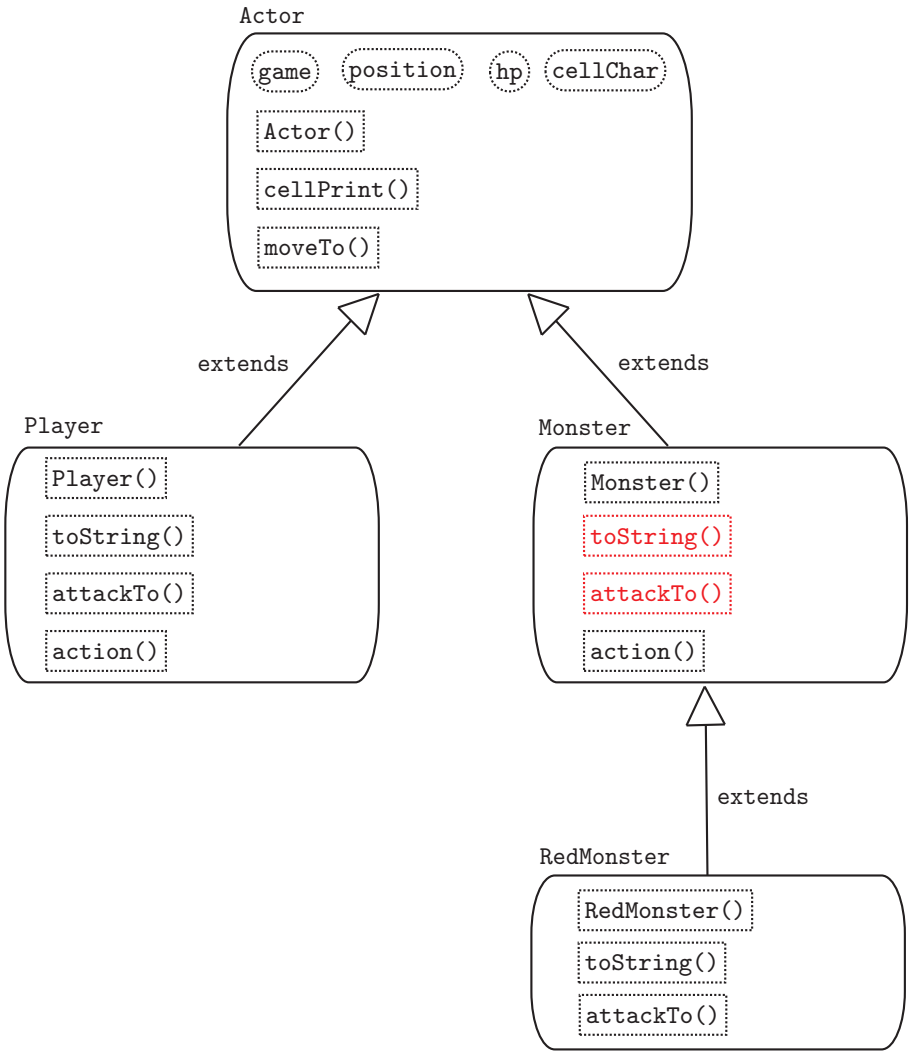


図 17.1: この章のアクター、プレイヤー、モンスター、レッドモンスターのクラス階層：赤字のメソッドが上書きされます

ラクタとメソッド attackTo() のみを持っています。attackTo() は Monster クラスにも定義されていますが、これが RedMonster クラスで上書きされます。

具体的にプログラムを見ていきましょう。図 17.2（260 ページ）が RedMonster クラスの定義です。Monster クラスを継承しています。

17.2.1　レッドモンスターのコンストラクタ

RedMonster クラスのコンストラクタの本体 1 行目：

```
class RedMonster extends Monster {
    RedMonster(Game game, Int2 position) {
        super(game, position);           // 親クラスのコンストラクタの呼び出し
        cellChar = 'R';            // 親の親のクラスのインスタンス変数への代入
    }

    @Override                                        // 上書きのアノテーション
    public String toString() {                  // オブジェクトを文字列に変換
        return "Red"+super.toString(); // "Red"の後に"Monster : ..."が続く
    }

    @Override                                        // 上書きのアノテーション
    void attackTo(Int2 direction) {
        Int2 target = this.position.add(direction);
        if (target.equal(game.player.position)) {
            game.player.hp -= 60;             // プレイヤーのダメージが大きい
        }
    }
}
```

図 17.2: レッドモンスターを導入したダンジョンゲームのプログラム（レッドモンスター）

```
        super(game, position);           // 親クラスのコンストラクタの呼び出し
```

は、親クラスである Monster クラスのコンストラクタ：

```
    Monster(Game game, Int2 position) {
        super(game, position, INIT_HP, 'M');
    }
```

を呼び出します。このコンストラクタがモンスターの親クラスであるアクターのコンストラクタ：

```
    Actor(Game game, Int2 position, int hp, char cellChar) {
        this.game = game;
        this.position = position;
        this.hp = hp;
        this.cellChar = cellChar;
    }
```

を呼び出すことは前章で見た通りです。

　さて、この時点ではアクターのインスタンス変数 cellChar には文字 'M' が格納されていることに注意してください。cellChar はマップ上の表示文字でした（図 16.2.1（239 ページ）、図 16.2.3

（241 ページ）参照)。このままではレッドモンスターはマップ上に ‘M’ と表示され、モンスターとレッドモンスターの区別が付きません。これを回避するために、RedMonster クラスのコンストラクタの本体 2 行目：

```
        cellChar = 'R';              // 親の親のクラスのインスタンス変数への代入
```

で、レッドモンスターのコンストラクタから、アクタークラスのインスタンス変数の値を ‘R’ に書き換えています。つまり、cellChar に ‘M’ を設定した直後に ‘R’ を再設定しています。このプログラムでは 2 回の設定を行なっており、あまりエレガントな設定方法とは言えません。この解決法は次章で示します。

17.2.2　アノテーション @Override

図 17.2 の toString() と attackTo() のメソッド記述の直前の

```
    @Override                                          // 上書きのアノテーション
```

は、その直後のメソッドが親クラスのメソッドを上書きしていることを表す**アノテーション**（annotation、注釈）です。一般にアノテーションは、プログラムの実行に全く影響を与えないものの、プログラマからコンパイラへ有益な情報を与える記述です。Override アノテーション（これを「上書き注釈」とは訳さないようです）は、そのメソッドが親クラスにも定義されていることを確認するためのものです。プログラマがメソッド名を書き間違えるなどして意図した通りの上書きになっていない場合にコンパイラからプログラマへ警告が出ますから、バグの発見に効果的です。

この講義では、メソッドの上書き部分には全てこのアノテーションを付けることにします。

17.2.3　レッドモンスターの表示 toString()

レッドモンスターの toString() は、モンスターの toString() を上書きしています。モンスターの toString() では、たとえば

```
Monster : (4,5) 150
```

のようにモンスターの状態が文字列に変換されます。同様に、レッドモンスターの toString() では

```
RedMonster : (4,5) 150
```

のように変換されるべきと考えるならば、両者の違いは先頭に "Red" があるか否かだけです。そこで、レッドモンスターの toString() の本体を以下のようにプログラム化します。

```
return "Red"+super.toString(); // "Red"の後に"Monster : ..."が続く
```

つまり、 文字列 "Red" の後に文字列 super.toString() を連接します。super は RedMonster オブジェクトのスーパークラス部分、すなわち Monster オブジェクトを意味します。よって、super.toString() では Monster クラスの toString() が呼び出されます。結果、"RedMonster : ..." という文字列が得られます。

17.2.4　レッドモンスターの攻撃 attackTo()

レッドモンスターは、モンスターの attackTo() も上書きしています。両者のメソッドの違いは、モンスターのメソッド（たとえば図 14.6（207 ページ）参照）では

```
game.player.hp -= 30;
```

となっているプログラム部分が、レッドモンスターのメソッドでは

```
game.player.hp -= 60;            // プレイヤーのダメージが大きい
```

です。よってレッドモンスターの攻撃力はモンスターから倍増しています。

なお、後章ではこの攻撃部分のプログラムをよりエレガントに記述する方法を紹介します。

17.3　ゲームクラスの変更

図 17.3（263 ページ）を見てください。ここまでモンスターは 2 体でしたが、そのうちの 1 体をレッドモンスターに変更します。そのために、変数 monster2 のデータ型を RedMonster に変更し、オブジェクトの生成を new RedMonster(...) に変更します。

17.4　マップクラス/アクタークラス/モンスタークラスの変更

Map クラス、Actor クラスに変更はありません。Monster クラスの変更もありません。

```
class Game {
    public static void main(String args[]) { ** 前章から変更なし** }

    Map map;
    Player player;
    Monster monster1;
    RedMonster monster2;          // 2番目のモンスターをレッドモンスターへ変更

    Game() {
        map = new Map(this);
        player = new Player(this);
        monster1 = new Monster(this, new Int2(1, Map.YSIZE-2));
        monster2 = new RedMonster(this, new Int2(Map.XSIZE-2, 1));  // 変更
    }

    void print() { ** 前章から変更なし** }

    void mainLoop() { ** 前章から変更なし** }

    void turn() { ** 前章から変更なし** }

    static char getKeyChar() { ** 前章から変更なし** }
}
```

図 17.3: レッドモンスターを導入したダンジョンゲームのプログラム（ゲーム）

17.5　プレイヤークラスの変更

Player クラスの記述（図 17.4（264 ページ）参照）には 1 箇所だけ変更があります。レッドモンスターはモンスターよりも防御力が大きいと仮定し、プレイヤーが monster2 を攻撃するときには、hp の減少を普通のモンスターの半分としました。

```
class Player extends Actor {
    static final int INIT_HP = 100;

    Player(Game game) { ** 前章と同じ ** }

    @Override
    public String toString()  { ** 前章と同じ ** }

    void attackTo(Int2 direction) {
        Int2 target = position.add(direction);
        if (target.equal(game.monster1.position)) {
            game.monster1.hp -= 20;
        }
        if (target.equal(game.monster2.position)) {
            game.monster2.hp -= 10;       // レッドモンスターは防御力が大きい
        }
    }

    void action()  { ** 前章と同じ ** }
}
```

図 17.4: RedMonster を導入したダンジョンゲームのプログラム（プレイヤー）

17.6　プログラムの実行

この章のプログラムの実行は、たとえば以下のように進みます。想定通り、プレイヤーはレッドモンスターからの 1 撃で、 hp が 60 減少します。

```
# # # # # # # # # # # # #
# @ . . . . . . . . . R #     <---- R はレッドモンスター
# . . . . . . . . . . . #
# . . . . . . . . . . . #
# . . . . # # # . . . . #
# . . . . # # # . . . . #
# . . . . # # # . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
# M . . . . . . . . . G #
# # # # # # # # # # # # #
Player : (1,1) 100
Monster : (1,9) 150
RedMonster : (11,1) 150

? d                         <---- 右方向をキー入力
# # # # # # # # # # # # #
# . @ . . . . . . . R . #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
# . . . . # # # . . . . #
# . . . . # # # . . . . #
# . . . . # # # . . . . #
# . . . . . . . . . . . #
# . M . . . . . . . . . #
# . . . . . . . . . . G #
# # # # # # # # # # # # #
Player : (2,1) 100
Monster : (2,8) 150
RedMonster : (10,1) 150

? d                         <---- 右方向をキー入力
# # # # # # # # # # # # #
# . . @ . . . . . R . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
# . . . . # # # . . . . #
# . . . . # # # . . . . #
# . . . . # # # . . . . #
# . . M . . . . . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . G #
# # # # # # # # # # # # #
Player : (3,1) 100
Monster : (3,7) 150
RedMonster : (9,1) 150

? d                         <---- 右方向をキー入力
# # # # # # # # # # # # #
# . . . @ . . . R . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
```

```
# . . . . # # # . . . . #
# . . . . # # # . . . . #
# . . . M # # # . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . G #
# # # # # # # # # # # # #
Player : (4,1) 100
Monster : (4,6) 150
RedMonster : (8,1) 150

? d                          <---- 右方向をキー入力
# # # # # # # # # # # # #
# . . . . @ . R . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
# . . . . # # # . . . . #
# . . . . # # # . . . . #
# . . . M # # # . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . G #
# # # # # # # # # # # # #
Player : (5,1) 100
Monster : (4,6) 150
RedMonster : (7,1) 150

? s                          <---- 何もしないをキー入力
# # # # # # # # # # # # #
# . . . . @ R . . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
# . . . . # # # . . . . #
# . . . . # # # . . . . #
# . . . M # # # . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . G #
# # # # # # # # # # # # #
Player : (5,1) 100
Monster : (4,6) 150
RedMonster : (6,1) 150

? d                          <---- 右方向をキー入力（攻撃）
# # # # # # # # # # # # #
# . . . . @ R . . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
# . . . . # # # . . . . #
# . . . . # # # . . . . #
# . . . M # # # . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . . G #
# # # # # # # # # # # # #
Player : (5,1) 40            <---- -60 のダメージ
Monster : (4,6) 150
RedMonster : (6,1) 140       <---- -10 のダメージ
```

以下同様

17.7　差分プログラミング

この章ではレッドモンスターを導入しましたが、新規に開発したプログラム部分はモンスターとは異なるレッドモンスター固有の処理の追加のみで済みました。

このように、多くの処理を親クラスから継承し、親クラスと異なる部分のみを追加してプログラムを開発していく手法を**差分プログラミング**（difference programming, difference coding）と呼びます。オブジェクト指向の継承の機能を有効活用するプログラミング手法として広く知られています。

ただし、有効だからと言って全てを差分プログラミングで実装しようなどという極端なことはやめるべきです。非常に不自然で分かりにくいプログラムを作ってしまう恐れがあります。これに関連する話題を 23.1 節で取り上げます。

17.8　補足：クラス階層の C++での記述

この章の内容も C++で同様に成り立ちます。ただし、`@override` アノテーションは、C++11 では以下のように記述できます。

```
class ClassA {
  public:
    int a = 111;
    void f() {
        cout << a << endl;
    }
};

class ClassB : public ClassA {
  public:
    int b = 222;
    void f() override {              // オーバーライド・アノテーション
        cout << a << " " << b << endl;
    }
};
```

C++11 以前の規格のコンパイラでは上のオーバーライド・アノテーションはエラーになる可能性があります。

17.9　付録：この章の全プログラムリスト

```
//ゲームクラスのプログラム
import java.io.*;

class Game {
    public static void main(String args[]) {
        Game game = new Game();
        game.mainLoop();
    }

    Map map;
    Player player;
    Monster monster1;
    RedMonster monster2;          // ２番目のモンスターをレッドモンスターへ変更

    Game() {
        map = new Map(this);
        player = new Player(this);
        monster1 = new Monster(this, new Int2(1, Map.YSIZE-2));
        monster2 = new RedMonster(this, new Int2(Map.XSIZE-2, 1));  // 変更
    }

    void print() {
        map.print();
        System.out.println(player);
        System.out.println(monster1);
        System.out.println(monster2);
        System.out.println();
    }

    void mainLoop() {
        print();
        while (true) { turn(); }
    }

    void turn() {
        player.action();
        monster1.action();
        monster2.action();
        print();
    }

    static char getKeyChar() {
        char c = 0;
        System.out.print("? ");
        try {
            BufferedReader in = new BufferedReader(
                                new InputStreamReader(System.in));
            c = in.readLine().charAt(0);
        } catch(IOException e){
            System.out.println("ERROR:: getKeyChar()");
        }
        return c;
    }
}
```

```
//マップクラスのプログラム
class Map {
    static final int ROCK = 0;
    static final int EMPTY = 1;
    static final int GOAL = 2;

    static final int XSIZE = 13;
    static final int YSIZE = 11;

    Game game;
    int data[][]= new int[XSIZE][YSIZE];

    Map(Game game) {
        this.game = game;
        for (int x = 0; x < XSIZE; x++) {
            for (int y = 0; y < YSIZE; y++) {
                if ((x == 0)||(x == XSIZE-1)||
                   (y == 0)||(y == YSIZE-1)||
                   ((Math.abs(x-XSIZE/2) <= 1)&&
                    (Math.abs(y-YSIZE/2) <= 1))) {
                    data[x][y] = ROCK;
                }
                else { data[x][y] = EMPTY; }
            }
        }
        data[XSIZE-2][YSIZE-2] = GOAL;
    }

    int getAt(Int2 position) { return data[position.x][position.y]; }

    void print() {
        for (int y = 0; y < YSIZE; y++) {
            for (int x = 0; x < XSIZE; x++) {
                Int2 position = new Int2(x,y);
                if (game.player.cellPrint(position)) { continue; }
                if (game.monster1.cellPrint(position)) { continue; }
                if (game.monster2.cellPrint(position)) { continue; }
                cellPrint(position);
            }
            System.out.println();
        }
    }

    void cellPrint(Int2 position) {
        int e = getAt(position);
        if (e == ROCK)       { System.out.print('#'+" "); }
        else if (e == EMPTY) { System.out.print('.'+" "); }
        else if (e == GOAL)  { System.out.print('G'+" "); }
        else                 { System.out.print('E'+' '); }
    }
}
```

```
//アクタークラスのプログラム
class Actor {
    Game game;
    Int2 position;
    int hp;
    char cellChar;

    Actor(Game game, Int2 position, int hp, char cellChar) {
        this.game = game;
        this.position = position;
        this.hp = hp;
        this.cellChar = cellChar;
    }

    boolean cellPrint(Int2 position) {
        if (this.position.equal(position)) {
            System.out.print(cellChar+" ");
            return true;
        }
        return false;
    }

    boolean moveTo(Int2 direction) {
        Int2 target = position.add(direction);

        if (game.map.getAt(target) == Map.ROCK) { return false; }
        if (target.equal(game.player.position)) { return false; }
        if (target.equal(game.monster1.position)) { return false; }
        if (target.equal(game.monster2.position)) { return false; }

        position = target;
        return true;
    }
}
```

```
//プレイヤークラスのプログラム
class Player extends Actor {
    static final int INIT_HP = 100;

    Player(Game game) {
        super(game, new Int2(1,1), INIT_HP, '@');
    }

    public String toString() {
        return "Player : "+position+" "+hp;
    }

    void attackTo(Int2 direction) {
        Int2 target = position.add(direction);
        if (target.equal(game.monster1.position)) {
            game.monster1.hp -= 20;
        }
        if (target.equal(game.monster2.position)) {
            game.monster2.hp -= 10;       // レッドモンスターは防御力が大きい
        }
    }

    void action() {
        Int2 direction;
        switch (Game.getKeyChar()) {
            case 'z': direction = new Int2(-1,+1); break;
            case 'x': direction = new Int2( 0,+1); break;
            case 'c': direction = new Int2(+1,+1); break;
            case 'a': direction = new Int2(-1, 0); break;
            case 'd': direction = new Int2(+1, 0); break;
            case 'q': direction = new Int2(-1,-1); break;
            case 'w': direction = new Int2( 0,-1); break;
            case 'e': direction = new Int2(+1,-1); break;
            default:  direction = new Int2( 0, 0);
        }

        if (!moveTo(direction)) { attackTo(direction); }
    }
}
```

```
//モンスタークラスのプログラム
class Monster extends Actor {
    static final int INIT_HP = 150;

    Monster(Game game, Int2 position) {
        super(game, position, INIT_HP, 'M');
    }

    public String toString() {
        return "Monster : "+position+" "+hp;
    }

    void attackTo(Int2 direction) {
        Int2 target = this.position.add(direction);
        if (target.equal(game.player.position)) {
            game.player.hp -= 30;
        }
    }

    void action() {
        Int2 direction = game.player.position.sub(position).signum();

        if (!moveTo(direction)) { attackTo(direction); }
    }
}
```

```
//レッドモンスタークラスのプログラム
class RedMonster extends Monster {
    RedMonster(Game game, Int2 position) {
        super(game, position);           // 親クラスのコンストラクタの呼び出し
        cellChar = 'R';            // 親の親のクラスのインスタンス変数への代入
    }

    @Override                                        // 上書きのアノテーション
    public String toString() {                  // オブジェクトを文字列に変換
        return "Red"+super.toString(); // "Red"の後に"Monster : ..."が続く
    }

    @Override                                        // 上書きのアノテーション
    void attackTo(Int2 direction) {
        Int2 target = this.position.add(direction);
        if (target.equal(game.player.position)) {
            game.player.hp -= 60;             // プレイヤーのダメージが大きい
        }
    }
}
```

```
//2 次元ベクトルクラスのプログラム
class Int2 {
    int x,y;

    Int2(int x, int y) { this.x = x; this.y = y; }

    Int2 add(Int2 a) { return new Int2(x+a.x,y+a.y); }
    Int2 sub(Int2 a) { return new Int2(x-a.x,y-a.y); }
    Int2 signum() { return new Int2(Integer.signum(x),Integer.signum(y)); }
    boolean equal(Int2 a) { return (x == a.x)&&(y == a.y); }

    public String toString(){ return "("+x+","+y+")"; }
}
```

1

1

第18章　様々なオブジェクトの作り方を用意する：メソッドの多重定義

この章ではオブジェクト指向とは直接関係しないメソッド（コンストラクタを含む）の**多重定義**（overload、オーバーロード）を解説します。この機能は、オブジェクト指向ではないプログラミング言語にも広く採用されており、みなさんもそれと気づかずに利用しているかもしれません。

18.1　簡単な例題

メソッドの多重定義とは、同じメソッド名であっても仮引数のデータ型、個数が異なれば、別のメソッドとして認識されることです。以下の例ではメソッド名 f の 3 種類のメソッドが多重定義されています。

```
class ClassA {
    void f() {
        System.out.println("f() is called");
    }
    void f(double d) {
        System.out.println("f(double d) is called");
    }
    void f(int i, char c) {
        System.out.println("f(int i, char c) is called");
    }
}
```

これらのメソッドを呼び出す以下のプログラム片：

```
    ...
    ClassA object = new ClassA();
    object.f(222.0);            // void f(double d) が選択される
    object.f();                 // void f() が選択される
    object.f(333,'!');          // void f(int i, char c) が選択される
    ...
```

を実行すると、メソッド名、実引数のデータ型と仮引数のデータ型、実引数の個数と仮引数の個数の情報の照合を行い、実際に呼び出すべきメソッドが適切に選択されます。このときの出力は以下の通りです。

```
f(double d) is called
f() is called
f(int i, char c) is called
```

18.1.1　多重定義の種明かし

多重定義では一見すると非常に高度な解析処理が行われているように思えますが、その仕掛けは単純です。たとえば上のクラス定義はコンパイラ内部では以下の等価なプログラムに置換されます。

```
class ClassA {
    void f() {
        System.out.println("f() is called");
    }
    void f_double(double d) {
        System.out.println("f(double d) is called");
    }
    void f_int_char(int i, char c) {
        System.out.println("f(int i, char c) is called");
    }
}
```

つまり、コンパイラ内部のメソッド名は、本来のメソッド名に引数のデータ型名を付加した名前になっており、メソッドを曖昧性なく選別できるのです。これに対応して、呼び出し側のプログラムも、コンパイラ内部では本来の関数名に引数のデータ型名の付いたプログラム：

```
    ...
    ClassA object = new ClassA();
    object.f_double(222.0);
    object.f();
    object.f_int_char(333,'!');
    ...
```

として認識されています。この単純な方法で、呼び出し側と呼び出される側のメソッドの適切なマッチングが行われています。

この仕組みは簡単ですが、プログラマにとっては全てのメソッド（関数）に無理矢理、異なる名前をつけるストレスが減ります。実際、プログラムが自然に読めるなど、効果は絶大です。多重定義は C++、Java に限らず最近のほとんどのプログラミング言語でごく普通に利用可能です。

ただし、多重定義にはひとつだけ問題があります。C++や Java は数値データの自動型変換（automatic type promotion[1]）をサポートしています。この自動変換と多重定義の相性が悪いのです。

たとえばコンパイラはプログラムの文脈に応じて int 型を double 型へ自動的に変換します。よって、多重定義の引数に int 型、 float 型、 double 型などが使われている場合、呼び出す関数を決定する過程が複雑になり、最悪の場合、決定できません。たとえば上の例題において

```
    ...
    object.f(222);  // 引数が int 型
    ...
```

の場合、整数 222 が double 型 222.0 に変換されてから、関数呼び出し object.f(222.0) が実行されます。

よって次のクラス定義の例：

```
class ClassB {
    void g(int a, double b) {
        System.out.println("g(int a, double b) is called");
    }
    void g(double a, int b) {
        System.out.println("g(double a, int b) is called");
    }
}
```

では、メソッド g() の引数は、double と int、または int と double であることが期待されますが、以下のメソッド呼び出し：

```
    ...
    (new ClassB()).g(1, 1);
    ...
```

では、二つの引数は共に int 型です。この場合、第 1 引数と第 2 引数のどちらの引数を double 型へ自動変換すればよいのかコンパイラは決定できず、コンパイルエラーとなります。

このように、数値型引数を多重定義に用いる場合には注意が必要です。

[1]日本語訳では自動型昇格の方が正確ですが、日本語として馴染みにくく感じるため、本文では「変換」を用いました。「昇格」は、値の幅が狭いデータを幅が広いデータへ変換することを言います。たとえば、char 型は $[-128, 127]$ の範囲の数を扱いますが、int 型は $[-2^{31}, 2^{31}-1]$ の範囲の数を扱うため、char < int と考えます。同様に、int < float < double という順序関係を考えます。「昇格」はより順位の高いデータへの型変換のことです。順位の低いデータへの自動的な型変換は行われません。

```
class Player extends Actor {
    static final int INIT_HP = 100;

    String name;                            // プレイヤーの名前を導入

    Player(Game game, String name) {        // 名前を与えるコンストラクタ
        super(game, new Int2(1,1), INIT_HP, '@');
        this.name = name;
    }

    Player(Game game) {                     // 名前を与えないコンストラクタ
        this(game, "Player");          // Player(game, "Player") を呼び出す
    }

    public String toString() {
        return name+" : "+position+" "+hp;   // 新たに名前を表示
    }

    void attackTo(Int2 direction) { ** 前章から変更なし ** }

    void action() { ** 前章から変更なし ** }
}
```

図 18.1: コンストラクタの多重定義を行うダンジョンゲームのプログラム（プレイヤー）

18.1.2　オーバロードとオーバーライド

オーバーロード（overload、多重定義）は前章のオーバーライド（override、上書き）と似た響きの言葉です。しかし内容は全く異なります。オーバーロード（辞書的意味は「過積載」）は、同じ名前のメソッドがひとつのクラスに重複して定義されており、それら全てのメソッドは呼び出し可能な状態を表す用語です。それに対してオーバーライド（辞書的意味は「上書き、無視」）は、メソッドがクラス階層の上下で重複して定義されていて、最下層のメソッドのみが呼び出し可能で、それ以外のメソッドは無視される状態を表します。内容が全く異なります。

なお、Java の教科書の中には多重定義を多相性の一種とみなすものがあります。その場合、この章の多重定義を**静的多相性**、次章、次々章の内容 — こちらが本来の多相性 — を**動的多相性**と立て分けます。

コンストラクタもメソッドの一種であり、多重定義可能です。この章ではプレイヤーとモンスターのコンストラクタを多重定義します。

18.2　プレイヤークラスの変更

図 18.1（280 ページ）がこの章の `Player` クラスの定義です。ここには二つのコンストラクタ：

```
    Player(Game game, String name) {   // 名前を与えるコンストラクタ
    ...
    Player(Game game) {                // 名前を与えないコンストラクタ
```

が定義されています。以下、その違いの解説です。

18.2.1　プレイヤーに名前を付ける

この章で新たに Player クラスにプレイヤーの名前を保持するインスタンス変数：

```
    String name;                              // プレイヤーの名前を導入
```

を導入します[2]。そしてプレイヤーの状態表示では、これまでの

Player : (1,1) 100

ではなく、たとえば名前が Taro の場合

Taro : (1,1) 100

と表示するように、 toString() を

```
        return name +" : "+position+" "+hp;      // 新たに名前を表示
```

に変更します（図 18.1 参照）。

次に、名前の初期設定をコンストラクタで行うために、新たに以下のコンストラクタを定義します。

```
    Player(Game game, String name) {         // 名前を与えるコンストラクタ
        super(game, new Int2(1,1), INIT_HP, '@');
        this.name = name;
    }
```

親クラスのコンストラクタ super(...) を呼び出した後に name を初期設定しています。

[2] この講義テキストでは「モンスターは名前を持たない」と考え、 name はアクターのインスタンス変数ではなく、プレイヤー固有のインスタンス変数とします。

```
Player(Game game) {
    this(game, "Player");
}

    Player(Game game, String name) {
        super(game, new Int2(1,1), INIT_HP, '@');
        this.name = name;
    }

        Actor(Game game, Int2 position, int hp, char cellChar) {
            this.game = game;
            this.position = position;
            this.hp = hp;
            this.cellChar = cellChar;
        }
```

図 18.2: コンストラクタの呼び出し順序（名前のないプレイヤーの生成の場合）

18.2.2　名前のないプレイヤーも可

プレイヤーの名前を設定できますが、しかし名前を設定しなくてもゲームができるように、以下のコンストラクタを多重定義します。

```
Player(Game game) {                    // 名前を与えないコンストラクタ
    this(game, "Player");          // Player(game, "Player") を呼び出す
}
```

ここに、 this が自分自身を表す変数であったことを思い出しましょう。9.3.1 項（128 ページ）の C++についての解説は Java でも同様に成り立ちます。Java では、

- コンストラクタの中で this に引数を与えることで別のコンストラクタに初期化を代行させることが可能です。

上の場合、“Player” をデフォルトの名前として初期化を行うことになります。図 18.2（282 ページ）に呼び出し関係を図示しましたので、参考にしてください。

18.3　ゲームクラスの変更

図 18.3（283 ページ）がこの章の Game クラスの定義です。変更箇所は以下の 1 行だけです。

```
    player = new Player(this, "Taro");            // プレイヤー名を追加
```

```
class Game {
    public static void main(String args[]) { ** 前章から変更なし ** }

    * インスタンス変数も前章から変更なし **

    Game() {
        map = new Map(this);
        player = new Player(this, "Taro");            // プレイヤー名を追加
        monster1 = new Monster(this, new Int2(1, Map.YSIZE-2));
        monster2 = new RedMonster(this, new Int2(Map.XSIZE-2, 1));
    }

    void print() { ** 前章から変更なし ** }

    void mainLoop() { ** 前章から変更なし ** }

    void turn() { ** 前章から変更なし ** }

    static char getKeyChar()  { ** 前章から変更なし ** }
}
```

図 18.3: コンストラクタの多重定義を行うダンジョンゲームのプログラム（ゲーム）

この場合、2 引数のコンストラクタが呼び出され、プレイヤーの名前が Taro に設定されます。

18.4　モンスタークラスの変更

図 18.4（284 ページ）がこの章の Monster クラスの定義です。ここでは二つのコンストラクタ：

```
    Monster(Game game, Int2 position, char c) { // 第 3 引数に表示文字を指定
    ...
    Monster(Game game, Int2 position) {         // 第 3 引数がない
```

が多重定義されています。

以下、それぞれの解説です。

18.4.1　マップ表示文字を与えることができる

コンストラクタ：

```
    Monster(Game game, Int2 position, char c) { // 第 3 引数に表示文字を指定
        super(game, position, INIT_HP, c);
    }
```

```
class Monster extends Actor {
    static final int INIT_HP = 150;

    Monster(Game game, Int2 position, char c) { // 第 3 引数に表示文字を指定
        super(game, position, INIT_HP, c);
    }

    Monster(Game game, Int2 position) {         // 第 3 引数がない
        this(game, position, 'M');              // 表示文字を M に設定
    }

    public String toString()  { ** 前章から変更なし ** }

    void attackTo(Int2 direction) { ** 前章から変更なし ** }

    void action() { ** 前章から変更なし ** }
}
```

図 18.4: コンストラクタの多重定義を行うダンジョンゲームのプログラム（モンスター）

はマップ表示文字を与えて初期化するものです。

18.4.2　マップ表示文字を省略するのも可

もうひとつのコンストラクタ：

```
    Monster(Game game, Int2 position) {         // 第 3 引数がない
        this(game, position, 'M');              // 表示文字を M に設定
    }
```

は、もしマップ表示文字が与えられなかったならば、'M' をデフォルト値としても用い、三引数のコンストラクタを呼び出します。本体では、this(game, position, 'M') を呼び出しています。これは、コンストラクタ Monster(game, position, 'M') の呼び出しを意味します。

18.5　レッドモンスタークラスの変更

図 18.5（285 ページ）がこの章の RedMonster クラスの定義です。Monster クラスの、表示文字を与える方のコンストラクタ Monster(game, position, 'R') を呼び出すことで前章でエレガントでないと述べた cellChar を 2 回設定する問題（17.2.1 項（259 ページ）参照）が解消しました。図 18.6（285 ページ）はレッドモンスターのコンストラクタから、モンスター、アクターのコンストラクタを順に読んでいる様子を図示したものです。

```
class RedMonster extends Monster {
    RedMonster(Game game, Int2 position) {
        super(game, position, 'R');// マップ表示文字を super に与えて初期化する
    }

    @Override
    public String toString() { ** 前章から変更なし ** }

    @Override
    void attackTo(Int2 direction) { ** 前章から変更なし ** }
}
```

図 18.5: コンストラクタの多重定義を行うダンジョンゲームのプログラム（レッドモンスター）

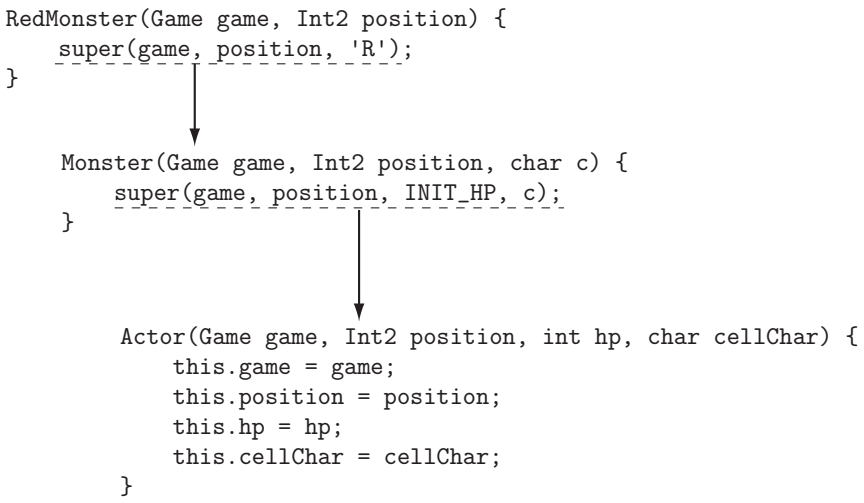


図 18.6: コンストラクタの呼び出し順序（レッドモンスターの生成の場合）

18.6　補足：C++の便利な機能

18.6.1　デフォルト引数

この章の Player クラスでは二つのコンストラクタ：

```
Player(Game game, String name) {   // 名前を与えるコンストラクタ
...
Player(Game game) {                // 名前を与えないコンストラクタ
```

が定義しました。C++では、**デフォルト引数**（default argument）と呼ばれる機能を用いることで、これと同じことをひとつのコンストラクタで済ませることが可能です。その場合、以下のようにコンストラクタ宣言します。

```
Player(Game game, String name = "Player") {
```

プログラム中で第 2 引数が省略された場合、C++コンパイラは自動的に第 2 引数に “Player” を補充して関数呼び出します。

便利な機能ですが、Java にはデフォルト引数の機能はありません。何故ならば、上に説明したように、代わりにメソッドを多重定義すれば済むからです。これは言語仕様を無駄に大きくしないという Java の設計方針に因るところです。C++と Java は共通するところが多いものの、C++にあって Java に採用されていない仕様も種々あります。次項の演算子関数もその例です。

18.6.2　演算子の多重定義

たとえば Python ではベクトル $(1, 2)$ とスカラー値 3 の以下のような演算が可能です。

$$(1, 2) + 3 = (1, 2) + (3, 3) = (4, 5)$$

$$3 + (1, 2) = (3, 3) + (1, 2) = (4, 5)$$

C++ではこれと同様のことを演算子関数のユーザー定義で可能です。Int2 型を用いて、プログラム例を示すと以下の通りです。これはクラスのメンバー関数の多重定義ではなく、グローバル関数（通常のクラスの外に定義する関数）の多重定義です。

```
//C++での演算子の多重定義、Java ではできない

Int2 operator + (Int2 a, int b) {
    return Int2(a.x+b, a.y+b);
}

Int2 operator + (int a, Int2 b) {
    return Int2(a+b.x, a+b.y);
}
```

二つの関数定義で引数の順番が異なることに注意してください。

Java にはこのような定義を行う機能は提供されていません。

18.7　付録：この章の全プログラムリスト

```
//ゲームクラスのプログラム
import java.io.*;

class Game {
    public static void main(String args[]) {
        Game game = new Game();
        game.mainLoop();
    }

    Map map;
    Player player;
    Monster monster1;
    RedMonster monster2;

    Game() {
        map = new Map(this);
        player = new Player(this, "Taro");            // プレイヤー名を追加
        monster1 = new Monster(this, new Int2(1, Map.YSIZE-2));
        monster2 = new RedMonster(this, new Int2(Map.XSIZE-2, 1));
    }

    void print() {
        map.print();
        System.out.println(player);
        System.out.println(monster1);
        System.out.println(monster2);
        System.out.println();
    }

    void mainLoop() {
        print();
        while (true) { turn(); }
    }

    void turn() {
        player.action();
        monster1.action();
        monster2.action();
        print();
    }

    static char getKeyChar() {
        char c = 0;
        System.out.print("? ");
        try {
            BufferedReader in = new BufferedReader(
                                new InputStreamReader(System.in));
            c = in.readLine().charAt(0);
        } catch(IOException e){
            System.out.println("ERROR:: getKeyChar()");
        }
        return c;
    }
}
```

```
//マップクラスのプログラム
class Map {
    static final int ROCK = 0;
    static final int EMPTY = 1;
    static final int GOAL = 2;

    static final int XSIZE = 13;
    static final int YSIZE = 11;

    Game game;
    int data[][]= new int[XSIZE][YSIZE];

    Map(Game game) {
        this.game = game;
        for (int x = 0; x < XSIZE; x++) {
            for (int y = 0; y < YSIZE; y++) {
                if ((x == 0)||(x == XSIZE-1)||
                   (y == 0)||(y == YSIZE-1)||
                   ((Math.abs(x-XSIZE/2) <= 1)&&
                    (Math.abs(y-YSIZE/2) <= 1))) {
                    data[x][y] = ROCK;
                }
                else { data[x][y] = EMPTY; }
            }
        }
        data[XSIZE-2][YSIZE-2] = GOAL;
    }

    int getAt(Int2 position) { return data[position.x][position.y]; }

    void print() {
        for (int y = 0; y < YSIZE; y++) {
            for (int x = 0; x < XSIZE; x++) {
                Int2 position = new Int2(x,y);
                if (game.player.cellPrint(position)) { continue; }
                if (game.monster1.cellPrint(position)) { continue; }
                if (game.monster2.cellPrint(position)) { continue; }
                cellPrint(position);
            }
            System.out.println();
        }
    }

    void cellPrint(Int2 position) {
        int e = getAt(position);
        if (e == ROCK)       { System.out.print('#'+" "); }
        else if (e == EMPTY) { System.out.print('.'+" "); }
        else if (e == GOAL)  { System.out.print('G'+" "); }
        else                 { System.out.print('E'+' '); }
    }
}
```

```
//アクタークラスのプログラム
class Actor {
    Game game;
    Int2 position;
    int hp;
    char cellChar;

    Actor(Game game, Int2 position, int hp, char cellChar) {
        this.game = game;
        this.position = position;
        this.hp = hp;
        this.cellChar = cellChar;
    }

    boolean cellPrint(Int2 position) {
        if (this.position.equal(position)) {
            System.out.print(cellChar+" ");
            return true;
        }
        return false;
    }

    boolean moveTo(Int2 direction) {
        Int2 target = position.add(direction);

        if (game.map.getAt(target) == Map.ROCK) { return false; }
        if (target.equal(game.player.position)) { return false; }
        if (target.equal(game.monster1.position)) { return false; }
        if (target.equal(game.monster2.position)) { return false; }

        position = target;
        return true;
    }
}
```

```
//プレイヤークラスのプログラム
class Player extends Actor {
    static final int INIT_HP = 100;

    String name;                            // プレイヤーの名前を導入

    Player(Game game, String name) {        // 名前を与えるコンストラクタ
        super(game, new Int2(1,1), INIT_HP, '@');
        this.name = name;
    }

    Player(Game game) {                     // 名前を与えないコンストラクタ
        this(game, "Player");          // Player(game, "Player") を呼び出す
    }

    public String toString() {
        return name+" : "+position+" "+hp;   // 新たに名前を表示
    }

    void attackTo(Int2 direction) {
        Int2 target = position.add(direction);
        if (target.equal(game.monster1.position)) {
            game.monster1.hp -= 20;
        }
        if (target.equal(game.monster2.position)) {
            game.monster2.hp -= 10;
        }
    }

    void action() {
        Int2 direction;
        switch (Game.getKeyChar()) {
            case 'z': direction = new Int2(-1,+1); break;
            case 'x': direction = new Int2( 0,+1); break;
            case 'c': direction = new Int2(+1,+1); break;
            case 'a': direction = new Int2(-1, 0); break;
            case 'd': direction = new Int2(+1, 0); break;
            case 'q': direction = new Int2(-1,-1); break;
            case 'w': direction = new Int2( 0,-1); break;
            case 'e': direction = new Int2(+1,-1); break;
            default:  direction = new Int2( 0, 0);
        }

        if (!moveTo(direction)) { attackTo(direction); }
    }
}
```

```
//モンスタークラスのプログラム

class Monster extends Actor {
    static final int INIT_HP = 150;

    Monster(Game game, Int2 position, char c) { // 第 3 引数に表示文字を指定
        super(game, position, INIT_HP, c);
    }

    Monster(Game game, Int2 position) {         // 第 3 引数がない
        this(game, position, 'M');              // 表示文字を M に設定
    }

    public String toString() {
        return "Monster : "+position+" "+hp;
    }

    void attackTo(Int2 direction) {
        Int2 target = this.position.add(direction);
        if (target.equal(game.player.position)) {
            game.player.hp -= 30;
        }
    }

    void action() {
        Int2 direction = game.player.position.sub(position).signum();

        if (!moveTo(direction)) { attackTo(direction); }
    }
}
```

```
//レッドモンスタークラスのプログラム
class RedMonster extends Monster {
    RedMonster(Game game, Int2 position) {
        super(game, position, 'R');// マップ表示文字を super に与えて初期化する
    }

    @Override
    public String toString() {
        return "Red"+super.toString();
    }

    @Override
    void attackTo(Int2 direction) {
        Int2 target = this.position.add(direction);
        if (target.equal(game.player.position)) {
            game.player.hp -= 60;
        }
    }
}
```

```
//2 次元ベクトルクラスのプログラム
class Int2 {
    int x,y;

    Int2(int x, int y) { this.x = x; this.y = y; }

    Int2 add(Int2 a) { return new Int2(x+a.x,y+a.y); }
    Int2 sub(Int2 a) { return new Int2(x-a.x,y-a.y); }
    Int2 signum() { return new Int2(Integer.signum(x),Integer.signum(y)); }
    boolean equal(Int2 a) { return (x == a.x)&&(y == a.y); }

    public String toString(){ return "("+x+","+y+")"; }
}
```

1

1

第19章　アクターがプレイヤーやモンスターとして振る舞う：多相性の原理

この章はこの講義テキストの中で最も分かりにくい章です。何を言いたいのか、初めは分からないかもしれません。ゆっくり進めていきます。

19.1　多相性とは

多相性は原語で polymorphism で、多く（poly-）の様相（-morph-) を表す性質（-ism）のことを意味します。オブジェクト指向では、オブジェクトが状況に応じて動的に多様な種類のメソッドを適切に呼び出すことを言います。

多相性を喩え話で説明してみます。

プリンタを考えましょう。プリンタにはインクジェットプリンタ、レーザープリンタなど様々な種類があります。しかしプリンタの種類は異なりますが、データを与えれば印刷物が出力できる機能は同じです。個々のプリンタではインクが異なりますし、カラーの調整も異なります。内部の印刷機構も異なりますから、全く同じ印刷の出来栄えにはならないでしょう。しかしそれぞれのプリンタは多少の違いはあるものの標準的に想定された範囲の印刷処理を行うはずです。その意味では、さまざまな種類があったとしても「プリンタ」と一括（くく）りにして扱っても問題なさそうです。どのプリンタに対しても「このデータを自分の方法でプリントしなさい」と命令すればよいのです。

別の例として乗り物を考えましょう。自動車、新幹線、飛行機など様々な種類の乗り物があります。「東京駅から新大阪駅まで荷物を届けよ」と命令すれば、自動車ならば高速道を通り、新幹線ならば鉄道を通り、飛行機＋自動車ならばまず自動車で東京駅から羽田まで陸送し、羽田から伊丹まで空輸し、伊丹から新大阪駅まで自動車で陸送します。それぞれで内部処理は異なりますが、東京駅から新大阪駅まで荷物を運ぶという機能があることは同じです。その意味で、どれも「乗り物」と一括りにできます。どの乗り物に対しても「この荷物を自分の方法で新大阪駅まで届けなさい」と命令すればよいのです。

このテキストのダンジョンゲームのプログラムの例で言えば、アクターにはプレイヤー、モンスター、レッドモンスターと 3 種類があります。いずれも次のターンの動作を行うメソッド `action()`

を持ち、その詳細は異なるものの、マップ上で移動し、攻撃を行う点では同じです。その意味では、「アクター」と一括りにして扱うことができそうです。どのアクターに対しても「自分の方法で action() を実行しなさい」と命令すればよいのです。オブジェクト指向の多相性の機能はそれを実現してくれます。

19.2　簡単な例題

上の内容の基礎となるオブジェクト指向プログラミングの機能を次の簡単なプログラムで説明します。

```
class Bird {                            // 鳥
    void canFly() { System.out.println("Bird can fly!"); }
}

class Penguin extends Bird {            // ペンギン
    @Override
    void canFly() { System.out.println("Penguin cannot fly!"); }

    void canSwim() { System.out.println("Penguin can swim!"); }
}
```

このクラス定義を利用する例を以下、場合分けして考えます。

1. まず、以下のプログラム片：

```
    Bird object = new Bird();

    object.canFly();
```

   を実行すると出力：

```
Bird can fly!
```

   を得ます。Bird の canFly() が実行されました。特段、問題のあるケースではありません。

2. 以下のプログラム片：

```
    Penguin object = new Penguin();

    object.canFly();
```

を実行すると出力：

```
Penguin cannot fly!
```

を得ます。親クラス Bird の canFly() を上書きしている Penguin の canFly() が実行されました。やはり問題のあるケースではありません。

3. 以下のプログラム片：

```
Penguin object = new Bird();
```

はコンパイルエラーになります。何故ならば、メソッド呼び出し

```
object.canSwim()
```

を考えましょう。object は Penguin の参照変数ですから、Penguin に定義されているメソッド canSwim() を呼び出すことができるはずです。しかし上のプログラムでは参照している実体は Bird ですから、オブジェクト実体は canSwim() を知りません（それが定義されていません）。このような明らかに不都合な実行は許されないため、コンパイラは “Penguin object = new Bird();” を不正と判断します。つまり、

- 子クラス型の参照変数は親クラスのオブジェクト実体を参照できません。

直感的は（ペンギンは鳥ですが）「鳥はペンギンではない」と理解することもできます。

4. 次がこの章のポイントです。以下のプログラム片：

```
Bird object = new Penguin();
object.canFly();
```

を実行すると出力：

```
Penguin cannot fly!
```

を得ます。変数とオブジェクト実体の関係は、上の 3. の逆パターンです。親クラスの変数が子クラスのオブジェクト実体を参照しています。これはコンパイルエラーになりません。何故ならば、子クラスのオブジェクトは親クラスのオブジェクトの性質を継承しているため、Penguin のオブジェクト実体は、 Bird のオブジェクトの全ての振る舞いに対応できるからです。そして、ここが最重要ポイントなのですが、メソッド呼び出し “object.canFly();” では Penguin のメソッドが実行されます。変数 object は Bird の参照変数ですが、 object の参照する実体が Penguin のオブジェクトだからです。**オブジェクト指向では、実際に実行されるメソッドは参照変数のクラスで決まるのではなく、実際に参照されているオブジェクト実体のクラスで決まります。**

4 番目の実行例をダンジョンゲームで確認しましょう。この確認の後、次章でこれを活用します。

```
class Game {
    public static void main(String args[]) { ** 前章から変更なし ** }

    Map map;
    Actor actor1;                       // 参照変数の型を Actor に変更
    Actor actor2;                       // 参照変数の型を Actor に変更
    Actor actor3;                       // 参照変数の型を Actor に変更

    Game() {
        map = new Map(this);
        actor1 = new Player(this, "Taro");    // 実体はそれぞれのオブジェクト
        actor2 = new Monster(this, new Int2(1, Map.YSIZE-2));
        actor3 = new RedMonster(this, new Int2(Map.XSIZE-2, 1));
    }

    void print() {
        map.print();
        System.out.println(actor1);  // Actor オブジェクトの文字列表示
        System.out.println(actor2);  // Actor オブジェクトの文字列表示
        System.out.println(actor3);  // Actor オブジェクトの文字列表示
        System.out.println();
    }

    void mainLoop() { ** 前章から変更なし ** }

    void turn() {
        actor1.action();                // Actor が action() を実行する。
        actor2.action();                // Actor が action() を実行する。
        actor3.action();                // Actor が action() を実行する。
        print();
    }

    static char getKeyChar() { ** 前章から変更なし ** }
}
```

図 19.1: 参照変数を Actor クラスに変更したダンジョンゲームのプログラム（ゲーム）

19.3　ゲームクラスの変更

図 19.1（301 ページ）がこの章の Game クラスのプログラムです。

前章までの

```
    Player  player;
    Monster monster1;
    RedMonster monster2;
```

を、以下のように変数のクラスを全て Actor クラスに変更しました。同時に変数名を actor1、actor2、actor3 に変更しました。

```
class Actor {
    ** インスタンス変数は前章と同じ **

    Actor(Game game, Int2 position, int hp, char cellChar) {
        ** 前章と同じ **
    }

    boolean cellPrint(Int2 position) { ** 前章と同じ ** }

    boolean moveTo(Int2 direction) { ** 変数名の変更を除き、前章と同じ ** }

    void action() {}                // アクターは action() を持つ必要がある！
}
```

図 19.2: 参照変数を Actor クラスに変更したダンジョンゲームのプログラム（アクター）

```
    Actor actor1;                       // 参照変数の型を Actor に変更
    Actor actor2;                       // 参照変数の型を Actor に変更
    Actor actor3;                       // 参照変数の型を Actor に変更
```

しかし、これら変数が実際に参照するオブジェクト実体のクラスはこれまで同様に Player、Monster、RedMonster です。よって

```
        actor1 = new Player(this, "Taro");    // 実体はそれぞれのオブジェクト
        actor2 = new Monster(this, new Int2(1, Map.YSIZE-2));
        actor3 = new RedMonster(this, new Int2(Map.XSIZE-2, 1));
```

19.4　アクタークラスの変更

図 19.2（302 ページ）はこの章のアクタークラスの定義です。

以下のメソッドを追加します。

```
    void action() {}                // アクターは action() を持つ必要がある！
```

分かるように、このメソッドの本体は空（から）です。しかし、このメソッド定義は必要です。これによって、アクタークラスおよびその全てのサブクラスには action() が定義されていることが保証され、 アクターを参照する変数 actor1、actor2、actor3 から action() を呼び出すことが可能となるからです。そして（ここが重要ですが）実際に実行されるメソッドはオブジェクト実体によって決まります。決して上の、本体が空（から）のメソッドではありません。

19.5　プレイヤー/モンスター/レッドモンスタークラスの変更

Player クラス、Monster クラス、 RedMonster クラスのプログラムについては、変数名の変更（player → actor1、monster1 → actor2、monster2 → actor3）のみです。

19.6　メソッド呼び出しの様子

図 19.3（304 ページ）は、この章のクラス階層です。赤字のメソッドは上書きに関係するメソッドを表します。Actor() に action() が追加されている点が前章までとは異なります。

さて、 Game クラスのメソッド turn() では、アクターオブジェクトに対してメソッド action() を呼び出します。

```
        actor1.action();                // Actor が action() を実行する。
        actor2.action();                // Actor が action() を実行する。
        actor3.action();                // Actor が action() を実行する。
```

それぞれの action() の動作を少し詳しく見ていきましょう。

まず、actor1 は Player オブジェクトですから、メソッド呼び出し actor1.action() では、図 19.3 の (2) の action() が呼び出されます。この action() の定義は以下の通りでした。

```
    void action() {
        Int2 direction;
        switch (Game.getKeyChar()) {
            中略
        }

        if (!moveTo(direction)) { attackTo(direction); }
    }
```

この中の moveTo() はアクタークラスの該当メソッド、attackTo() は Player クラスの該当メソッドが呼び出されます。

次に、actor2 は Monster オブジェクトですから、メソッド呼び出し actor2.action() では、図 19.3 の (3) の action() が呼び出されます。この action() の定義は以下の通りでした。

```
    void action() {
        Int2 direction = game.actor1.position.sub(position).signum();

        if (!moveTo(direction)) { attackTo(direction); }
    }
```

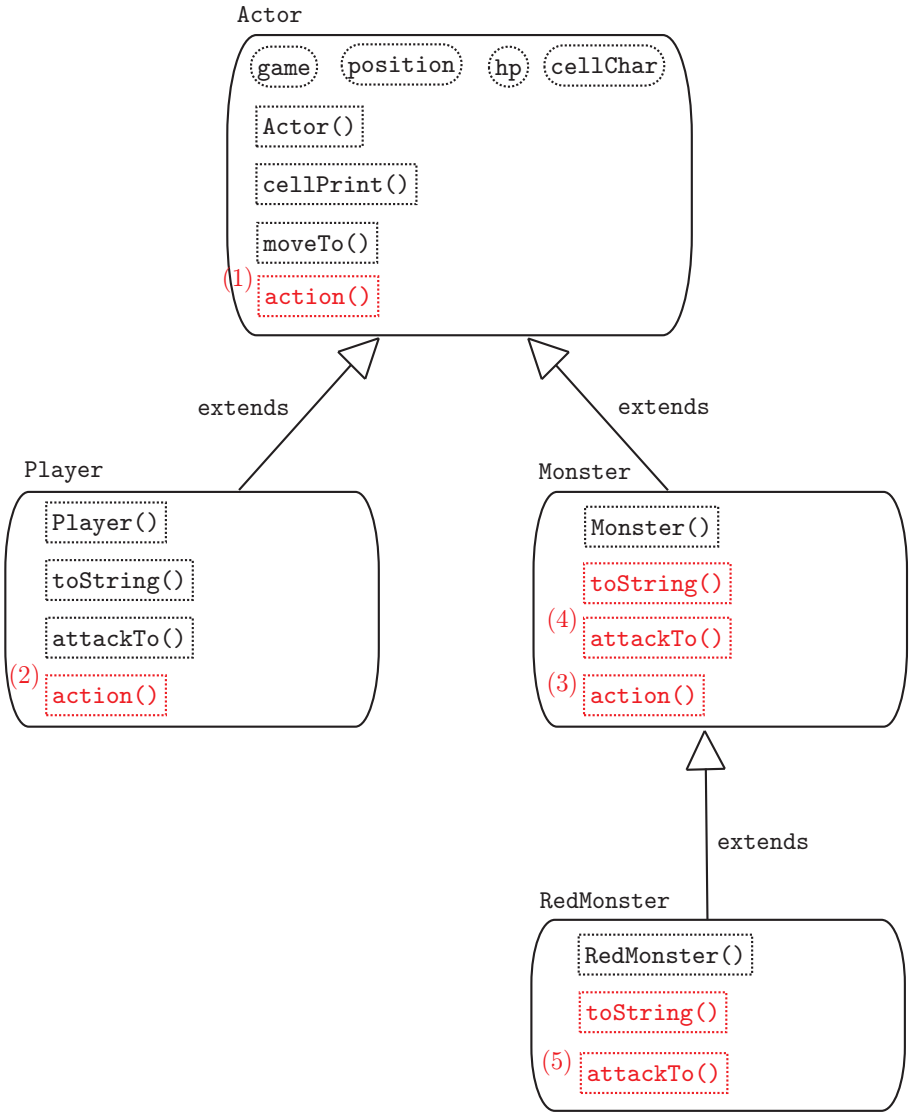


図 19.3: この章のプレイヤー、モンスター、レッドモンスターのクラス階層

この中の moveTo() はアクタークラスの該当メソッド、attackTo() は図 19.3 の (4) のメソッドが呼び出されます。

actor3 は RedMonster オブジェクトですから、メソッド呼び出し actor3.action() では、図 19.3 の (3) の action() が呼び出されます。 RedMonster クラスには action() の定義はないことに注意してください。この action() の定義は上に示した通りです。この中の moveTo() はアクタークラスの該当メソッド、attackTo() は 19.3 の (5) の、レッドモンスターのメソッドが呼び出されます。

なお、ここではアクタークラスそのもののオブジェクト実体を取り扱っていませんから、図 19.3 の (1) の action() が呼ばれることはありません。先に述べたように、アクタークラスのこのメソッドは、アクタークラスの全てのサブクラスのオブジェクトが action() を実行できることを保証するためだけに置かれています。このような、そこにメソッドが存在することだけがその役目であるようなメソッドについては第 21 章で詳細に議論します。

ところで、この節の説明を読むとメソッド呼び出しがクラス階層の上下を行き来し、「オブジェクト指向はなんとも複雑で面倒なものだ」と感じるかもしれません。それはプログラムの読み方がオブジェクト指向に馴染んでいないせいです。

ひとつのクラスとそのスーパークラスのみに限定して「このクラスのオブジェクトの振る舞いは...」とプログラムを読むと、局所的な理解で済むため、逆に非常に読みやすいプログラムになっています。今すぐ実感することは難しいかもしれませんが、後章でいくつもの例題をこなしていく内に自然にオブジェクト指向流のプログラムの読み方が身に付いてくるでしょう。

19.7　何故 toString() はうまく動作するのか？

14.1.1 項（200 ページ）では、クラスにメソッド public String toString() が定義されているとき、オブジェクトを文字列（ String）に変換すべき局面では自動的に toString() が呼び出しされる「約束」になっていると説明しました。その「約束」とは実は多相性です。ここで説明します。

19.7.1　Object クラス

Java には Object という名前のクラスがシステム定義されており、

- Object は暗黙に全てのクラスのスーパークラスである

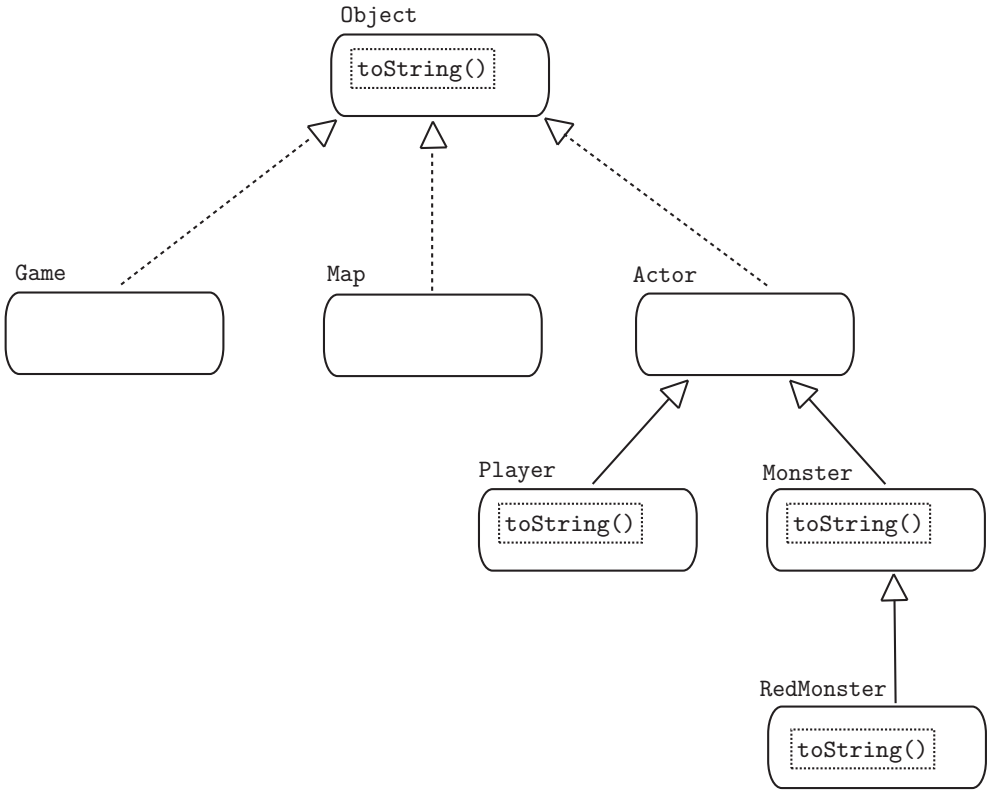


図 19.4: Object クラスを含むこの章のクラス階層：Object クラスへ入る点線の矢印は暗黙の継承を表す。

ように位置付けられています。図 19.4（306 ページ）はそれを図示したものです。Game、Map、Actor はダンジョンゲームのプログラムでは親クラスを持ちませんが、システム上は暗黙に Object クラスを親に持つと見なされます。

よって、全てのクラスは Object クラスの機能を暗黙に継承しています。その機能はオラクル社の Java Platform, Standard Edition の API 仕様:

https://docs.oracle.com/javase/jp/7/api/java/lang/Object.html

に説明されています。その仕様の中に Object クラスのインスタンスメソッド public String toString() について以下のように解説されています。

1. 当該オブジェクトの文字列表現を返すメソッドである。
2. 上書きされることが望ましい。
3. 上書きされていない場合には、「当該オブジェクトのクラス名 @ 当該オブジェクトのハッシュ

値[1]」の文字列表現を返す。

上の仕様の意味を以下で解説します。

19.7.2　toString() の呼び出し

今、図 19.1 の

```
System.out.println(actor1);
```

の動作を考えましょう。このとき、メソッド `println()` は以下のように動作する仕様になっています。

1. もし実引数が文字列ならば、単にその文字列を標準出力へ出力するだけです。
2. プリミティブならば、前章の多重定義によって、それぞれのデータ型に従う出力を行います。
3. `null` ならば、“`null`” を出力します。
4. さもなくば（文字列でもなく、プリミティブでもなく、`null` でもないならば）、引数を `Object` クラス（またはそのサブクラス）のインスタンスと見なし、そのインスタンス（オブジェクト実体）に `toString()` を適用し[2]、インスタンスの文字列表現を得た後に、その文字列を表示します。

4 番目に注目しましょう。多相性の原理によれば、`toString()` が上書きされている場合、上書きしたメソッドが呼び出されます。よって上のメソッド呼び出し `System.out.println(actor1)` では、`actor1` は `Player` オブジェクトですから `Player` クラスの `toString()` が自動的に呼び出されるのです。

これがオブジェクト毎に適切な `toString()` が呼び出される仕組みの種明かしです。

以上、`Player` クラス、`Monster` クラスの `toString()` が `Object` クラスの `toString()` を上書きしていたことが判明しましたから、この章以降、`Player` クラス、`Monster` クラスの `toString()` にも `@Override` アノテーションをつけることとします。

[1] ハッシュ値（hash value）は、データを何らかの整数値に置き換えた値のことです。データへの高速アクセス技術に用いられます。
[2]

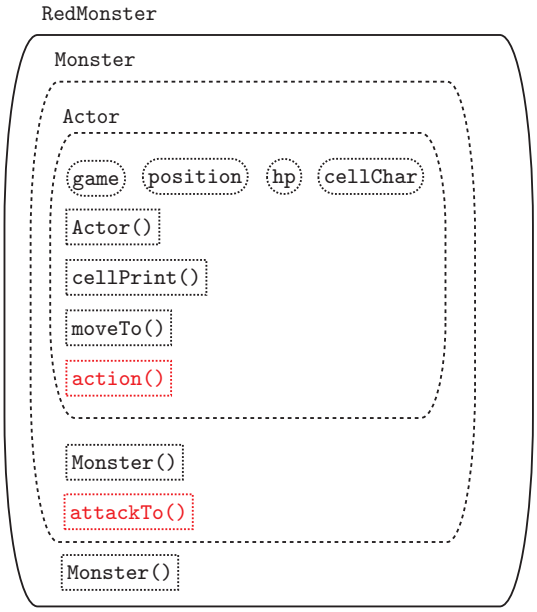


図 19.5: RedMonster オブジェクトの内部構造：最内部にアクター部分があり、その外側にモンスター部分、最外部にレッドモンスター部分がある構造です。よって、最内部だけに注目すればレッドモンスターはアクターとして振る舞います。同様にモンスターとして振る舞うこともできます。赤色のメソッドはサブクラスから上書きされたメソッドです。なお、Object クラスと toString() は省略しています。

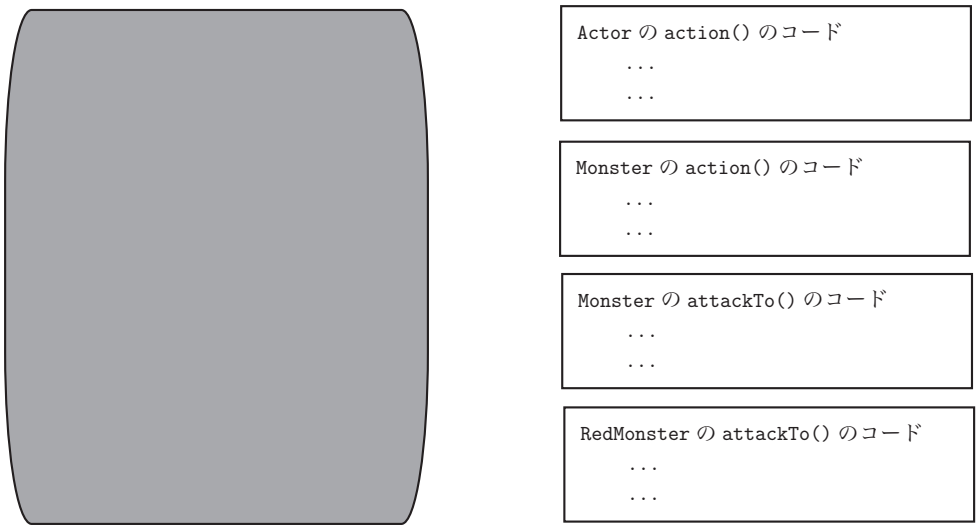
19.8　多相性の実装の仕組み

この章で説明した多相性 — オブジェクト毎にそのオブジェクトに適切なメソッドが自動的に呼び出される機能 — は言語処理系の中でどのように実装されているのでしょうか。

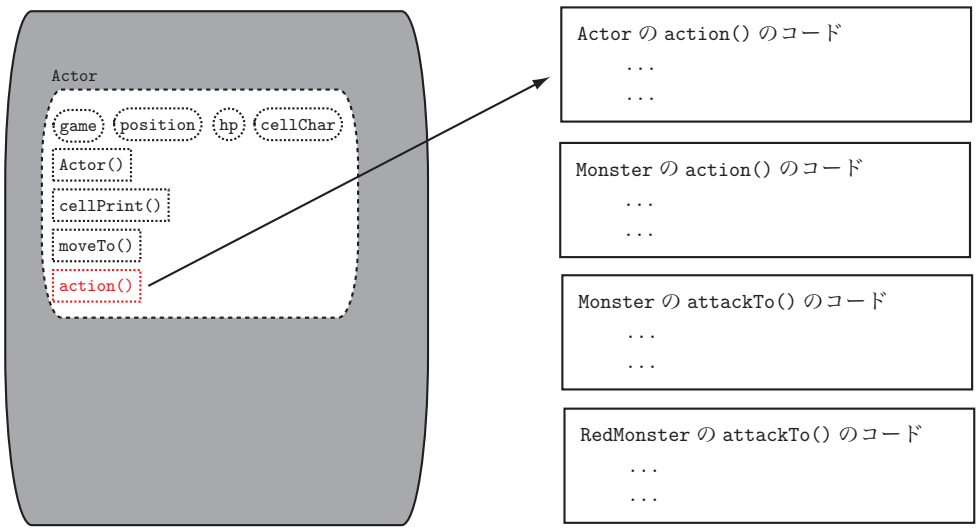
19.8.1　レッドモンスターオブジェクトの内部構造

レッドモンスターについて解説します。図 19.5（308 ページ）はレッドモンスターのオブジェクトの内部構造を図示したものです。ここに赤字に表示されるメソッドは上書きされているインスタンスメソッドです。これに注目しましょう。

図 19.3（304 ページ）のクラス階層図では、メソッド action() は モンスターにも定義されていました。しかし、図 19.5 ではモンスターには action() がありません。というのも、モンスターに定義した action() は親クラスの Actor() の action() を上書きするため、 モンスターには action() はありません。同様にレッドモンスターの attackTo() はモンスターの attackTo() を上書きするため、レッドモンスターには attackTo() はありません。

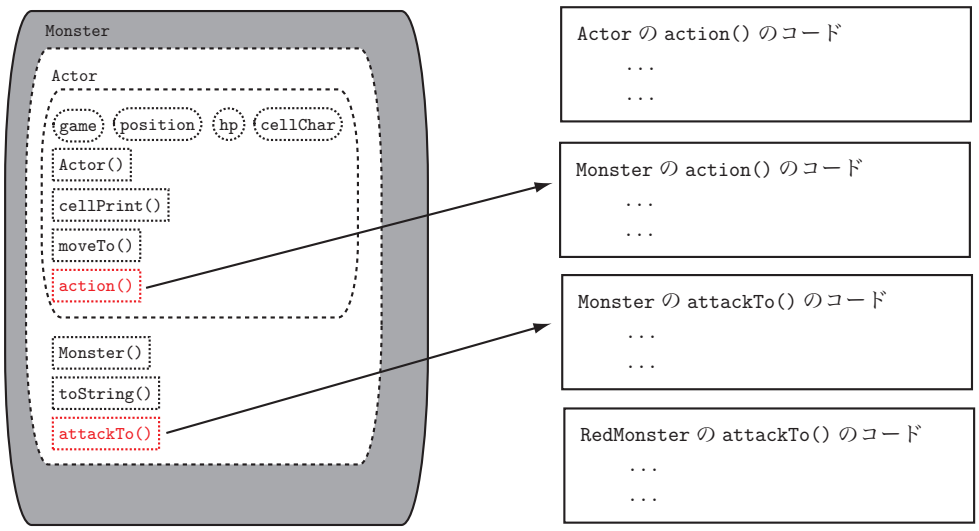


(a) メモリ領域の確保

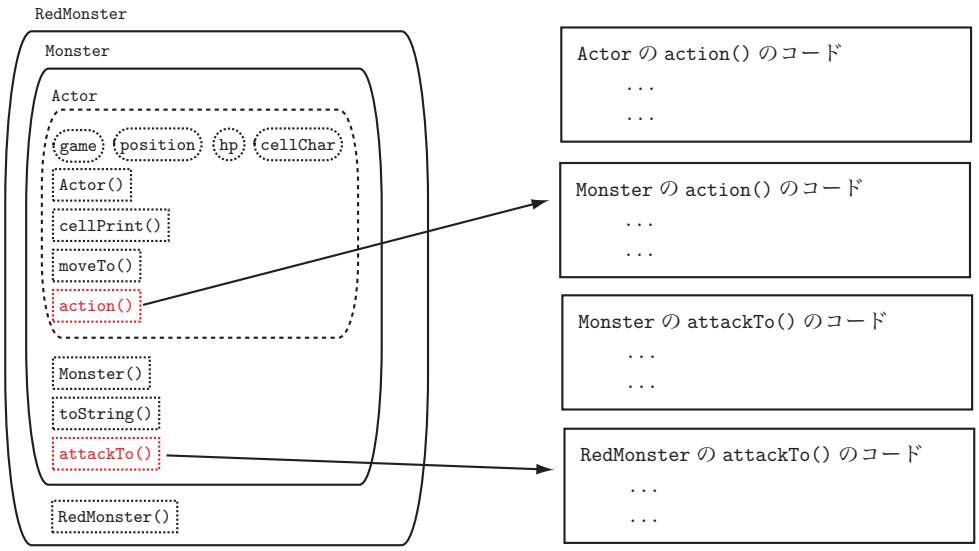


(b) アクター部分の初期化

図 19.6: RedMonster オブジェクトが構築される過程（その 1）：(a) オブジェクトのためのメモリを確保した直後です。内部は初期化されていません。(b) アクター部分が初期化された直後です。上書きされる action()（赤色）のポインタはこの時点では Actor の本来の action() を指しています。



(c) モンスター部分の初期化



(d) レッドモンスター部分の初期化

図 19.7: RedMonster オブジェクトが構築される過程（その 2）：(c) モンスター部分が初期化された直後です。 action() のポインタは Monster のコンストラクタによって Monster の action() に上書き（ポインタの付け替え）されています。メソッド attackTo()（赤字）もサブクラスで上書きされますが、この時点では Monster の本来のメソッドを指しています。(d) レッドモンスター部分が初期化された直後です。メソッド attackTo() が RedMonster のそれで上書きされています。

19.8.2　メソッドへのポインタ

7.8 節（98 ページ）では、C++のメンバー関数のコードはオブジェクト内には保持されないと述べました。コードをオブジェクト内に保持してない点は Java でも同様です。

しかし、上書きのあるメソッドでは実際に実行されるメソッドがオブジェクト実体に依存するため、メソッドの情報をオブジェクト内に保持する必要があります。C++/Java ではそれをメソッド（C++の場合にはメンバー関数）へのポインタを用いて実装しています[3]。

これを図 19.6（309 ページ）、図 19.7（310 ページ）を用いて解説します。

まず、図 19.6 (a) の左側は、動的メモリ領域にレッドモンスターに相当するメモリ領域が確保されたところを表します。図 19.6 (a) の右側は、メソッド群のコードです。これらは Java のプログラムがコンパイルされて所定のコード領域に置かれているものです。

次に、図 19.6 (b) の左側は、メモリ領域内の `Actor` オブジェクト相当部分が初期化されたところを表します。このとき、オブジェクト内には `action()` が呼ばれたときに実行すべきメソッドへのポインタがあります。この時点ではオブジェクト内の `action()` のポインタは `Actor` クラスの `action()` を指しています。

図 19.7 (c) の左側は、メモリ領域内の `Monster` オブジェクト相当部分が初期化されたところを表します。このとき、オブジェクト内の `action()` は、`Monster` クラスの `action()` を指すようにポインタの指す位置が上書きされます。また、`Monster` オブジェクト部分には `attackTo()` に関連するポインタが新たに作られ、 `Monster` クラスの `attackTo()` を指します。

次に、図 19.7 (d) の左側は、メモリ領域内の `RedMonster` オブジェクト相当部分が初期化されたところを表します。このとき、オブジェクト内の `attackTo()` は、`RedMonster` クラスの `attackTo()` を指すようにポインタの指す位置が上書きされます。

以上の 4 ステップで、`RedMonster` オブジェクトの初期化が完了します。結局、`action()` のポインタは `Monster` クラスのそれを指し、`attackTo()` のポインタは `RedMonster` クラスのそれを指すように設定されました。

この章のプログラムで扱う `Player` オブジェクト、`Monster` オブジェクト、`RedMonster` オブジェクトでは、上のような初期化によって、それぞれのポインタが適切なメソッドを指すように初期設定されるのです。これが多相性が適切に働く理由です。

ここでは、上書きされるメソッドのみに着目して説明しましたが、一般に全てのメソッドはそのサブクラスから上書きされる可能性があります。よって、オブジェクト内には全てのメソッドのポインタが用意されています。このようなポインタはテーブルで管理されており、一般に**仮想関数**

[3] このようなポインタを関数ポインタと呼びます。通常のデータを指すポインタと同等ですが、ポインタの指している先のコードを改変することはできません。

テーブル（virtual function table）と呼ばれています。「仮想関数」は元は C++ の用語です。

19.9　補足：C++における多相性

C++でも Java と同様に多相性を利用できます。

以下は、この章の冒頭の簡単な例題を C++に翻訳したものです。

```
class Bird {
  public:
    virtual void canFly() {                      // キーワード virtual に注意
        cout << "Bird can fly!" << endl;
    }
};

class Penguin : public Bird {
  public:
    void canFly() override {                 // アノテーション override に注意
        cout << "Penguin cannot fly!" << endl;
    }

    void canSwim() {
        cout << "Penguin can swim!" << endl
    }
};
```

ここに、キーワード `virtual` はそのメンバー関数について多相性を機能させるか否か — C++の言い方では、そのメンバー関数が仮想的（virtual）な否か — を指定するものです。C++では多相性の可否をプログラマが選択できます。否の場合、呼び出すメンバー関数はプログラムの文面のみから静的に決定されます。たとえば、この章のプログラムでは、

```
    Actor actor1;                          // 参照変数の型を Actor に変更
```

と宣言していますが、そうすると、メンバー関数 `action()` の多相性を機能させない場合、`actor1->action()` で呼び出されるメンバー関数は、そのオブジェクト実体に依らず、常に `Actor` クラスの `action()` です。呼び出す関数が静的に決まるため、仮想関数テーブルにポインタを用意する必要がなく、その分、メモリを節約できます。また、ポインタを介さないため、関数の呼び出しの手間[4]も軽減されます。C++は効率重視の言語であるため、細かいところまでプログラマが指定できる訳です。

[4]実際に実行する関数をポインタを手繰（たぐ）って求める操作です。手繰ることでメモリアクセス分のコストが増えます。

それに対して Java では全てのメソッドで多相性を機能させる仕様となっており、プログラマによる取捨選択はできません。Java は実行効率よりも使いやすさ、柔軟性を重視するためです。

19.10　付録：この章の全プログラムリスト

```
//ゲームクラスのプログラム
import java.io.*;

class Game {
    public static void main(String args[]) {
        Game game = new Game();
        game.mainLoop();
    }

    Map map;
    Actor actor1;                    // 参照変数の型を Actor に変更
    Actor actor2;                    // 参照変数の型を Actor に変更
    Actor actor3;                    // 参照変数の型を Actor に変更

    Game() {
        map = new Map(this);
        actor1 = new Player(this, "Taro");    // 実体はそれぞれのオブジェクト
        actor2 = new Monster(this, new Int2(1, Map.YSIZE-2));
        actor3 = new RedMonster(this, new Int2(Map.XSIZE-2, 1));
    }

    void print() {
        map.print();
        System.out.println(actor1);  // Actor オブジェクトの文字列表示
        System.out.println(actor2);  // Actor オブジェクトの文字列表示
        System.out.println(actor3);  // Actor オブジェクトの文字列表示
        System.out.println();
    }

    void mainLoop() {
        print();
        while (true) { turn(); }
    }

    void turn() {
        actor1.action();                // Actor が action() を実行する。
        actor2.action();                // Actor が action() を実行する。
        actor3.action();                // Actor が action() を実行する。
        print();
    }

    static char getKeyChar() {
        char c = 0;
        System.out.print("? ");
        try {
            BufferedReader in = new BufferedReader(
                                new InputStreamReader(System.in));
            c = in.readLine().charAt(0);
        } catch(IOException e){
            System.out.println("ERROR:: getKeyChar()");
        }
        return c;
    }
}
```

```
//マップクラスのプログラム
class Map {
    static final int ROCK = 0;
    static final int EMPTY = 1;
    static final int GOAL = 2;

    static final int XSIZE = 13;
    static final int YSIZE = 11;

    Game game;
    int data[][]= new int[XSIZE][YSIZE];

    Map(Game game) {
        this.game = game;
        for (int x = 0; x < XSIZE; x++) {
            for (int y = 0; y < YSIZE; y++) {
                if ((x == 0)||(x == XSIZE-1)||
                   (y == 0)||(y == YSIZE-1)||
                   ((Math.abs(x-XSIZE/2) <= 1)&&
                    (Math.abs(y-YSIZE/2) <= 1))) {

                    data[x][y] = ROCK;
                }
                else { data[x][y] = EMPTY; }
            }
        }
        data[XSIZE-2][YSIZE-2] = GOAL;
    }

    int getAt(Int2 position) { return data[position.x][position.y]; }

    void print() {
        for (int y = 0; y < YSIZE; y++) {
            for (int x = 0; x < XSIZE; x++) {
                Int2 position = new Int2(x,y);
                if (game.actor1.cellPrint(position)) { continue; }
                if (game.actor2.cellPrint(position)) { continue; }
                if (game.actor3.cellPrint(position)) { continue; }
                cellPrint(position);
            }
            System.out.println();
        }
    }

    void cellPrint(Int2 position) {
        int e = getAt(position);
        if (e == ROCK) { System.out.print('#'+" "); }
        else if (e == EMPTY) { System.out.print('.'+" "); }
        else if (e == GOAL) { System.out.print('G'+" "); }
        else { System.out.print('E'+' '); }
    }
}
```

```
//アクタークラスのプログラム
class Actor {
    Game game;
    Int2 position;
    int hp;
    char cellChar;

    Actor(Game game, Int2 position, int hp, char cellChar) {
        this.game = game;
        this.position = position;
        this.hp = hp;
        this.cellChar = cellChar;
    }

    boolean cellPrint(Int2 position) {
        if (this.position.equal(position)) {
            System.out.print(cellChar+" ");
            return true;
        }
        return false;
    }

    boolean moveTo(Int2 direction) {
        Int2 target = position.add(direction);

        if (game.map.getAt(target) == Map.ROCK) { return false; }
        if (target.equal(game.actor1.position)) { return false; }
        if (target.equal(game.actor2.position)) { return false; }
        if (target.equal(game.actor3.position)) { return false; }

        position = target;
        return true;
    }

    void action() {}                // アクターは action() を持つ必要がある！
}
```

```
//プレイヤークラスのプログラム
class Player extends Actor {
    static final int INIT_HP = 100;

    String name;

    Player(Game game, String name) {
        super(game, new Int2(1,1), INIT_HP, '@');
        this.name = name;
    }

    Player(Game game) { this(game, "Player"); }

    @Override
    public String toString() {
        return name+" : "+position+" "+hp;
    }

    void attackTo(Int2 direction) {
        Int2 target = position.add(direction);
        if (target.equal(game.actor2.position)) {
            game.actor2.hp -= 20;
        }
        if (target.equal(game.actor3.position)) {
            game.actor3.hp -= 10;
        }
    }

    @Override
    void action() {
        Int2 direction;
        switch (Game.getKeyChar()) {
            case 'z': direction = new Int2(-1,+1); break;
            case 'x': direction = new Int2( 0,+1); break;
            case 'c': direction = new Int2(+1,+1); break;
            case 'a': direction = new Int2(-1, 0); break;
            case 'd': direction = new Int2(+1, 0); break;
            case 'q': direction = new Int2(-1,-1); break;
            case 'w': direction = new Int2( 0,-1); break;
            case 'e': direction = new Int2(+1,-1); break;
            default:  direction = new Int2( 0, 0);
        }

        if (!moveTo(direction)) { attackTo(direction); }
    }
}
```

```
//モンスタークラスのプログラム
class Monster extends Actor {
    static final int INIT_HP = 150;

    Monster(Game game, Int2 position, char c) {
        super(game, position, INIT_HP, c);
    }

    Monster(Game game, Int2 position) { this(game, position, 'M'); }

    @Override
    public String toString() {
        return "Monster : "+position+" "+hp;
    }

    void attackTo(Int2 direction) {
        Int2 target = this.position.add(direction);
        if (target.equal(game.actor1.position)) {
            game.actor1.hp -= 30;
        }
    }

    @Override
    void action() {
        Int2 direction = game.actor1.position.sub(position).signum();

        if (!moveTo(direction)) { attackTo(direction); }
    }
}
```

```
//レッドモンスタークラスのプログラム
class RedMonster extends Monster {
    RedMonster(Game game, Int2 position) {
        super(game, position, 'R');
    }

    @Override
    public String toString() {
        return "Red"+super.toString();
    }

    @Override
    void attackTo(Int2 direction) {
        Int2 target = this.position.add(direction);
        if (target.equal(game.actor1.position)) {
            game.actor1.hp -= 60;
        }
    }
}
```

```
//2 次元ベクトルクラスのプログラム
class Int2 {
    int x,y;

    Int2(int x, int y) { this.x = x; this.y = y; }

    Int2 add(Int2 a) { return new Int2(x+a.x,y+a.y); }
    Int2 sub(Int2 a) { return new Int2(x-a.x,y-a.y); }
    Int2 signum() { return new Int2(Integer.signum(x),Integer.signum(y)); }
    boolean equal(Int2 a) { return (x == a.x)&&(y == a.y); }

    public String toString(){ return "("+x+","+y+")"; }
}
```

1

1

第20章　プレイヤー、モンスターをアクターの配列で一括管理する：多相性の活用

前章ではプレイヤー、モンスター、レッドモンスターを全て Actor オブジェクトとして参照しました。この章では一歩進めて、それらを全て Actor オブジェクトの配列を用いて一括管理します。この配列をこの講義テキストでは「アクター配列」と呼ぶことにし、Java の動的配列クラス ArrayList を用いて実装します。配列を用いることで、モンスターの数を自由に変えることができるようになります。

この章の内容は少し分量が多くなっています。丁寧に見ていきましょう。

20.1　簡単な例題： ArrayList の利用

Java の ArrayList は可変長配列とも呼ばれ、格納できる要素数を動的に変更できる便利な配列です。

20.1.1　基本機能

この講義では以下の機能を用います。

パッケージのインポート： ArrayList を利用するためには、プログラムの冒頭に以下を置きます。

```
import java.util.ArrayList;
```

データ型： ArrayList は**ジェネリック型**[1]（generic type、総称型）と呼ばれ、 ArrayList に格納するオブジェクトのクラス名 T を指定して、“ArrayList<T>” でデータ型を指定します。格納できるオブジェクトは、クラス T およびそのサブクラス（ここ重要！）のオブジェクトで

[1] Java のジェネリック型は、柔軟なプログラミングを可能にする多相性の 1 形態（パラメータ多相性）と見なされることがあります。

す。なお、ArrayList の配列要素のデータ型はオブジェクトに制限されるため、プリミティブ型を要素とする、たとえば ArrayList<int> のようなデータ型を作ることはできません[2]。

オブジェクトの生成： ArrayList のオブジェクトは以下で生成できます。

```
ArrayList<T> objects = new ArrayList<T>();
```

配列サイズを指定していませんが、問題ありません。オブジェクトが自動的に管理してくれますから、配列溢れが発生することはありません。初期のサイズは 10 です。

要素の追加： メソッド add(T t) で配列の末尾に T 型オブジェクト t を追加できます。

配列サイズ（要素数）の取得： メソッド size() で知ることができます。

要素の取得： メソッド get(int i) で i 番目の要素を知ることができます。このメソッドは配列から要素を削除することはしません。

その他の機能（要素の削除など）については各自で調べてください。

20.1.2　繰り返し処理（その 1）

さて、以下のクラス階層を例に考えます。

```
class ClassA {
    void f() {
        System.out.println("A");
    }
}

class ClassB extends ClassA {
    @Override
    void f() {
        System.out.println("B");
    }
}

class ClassC extends ClassA {
    @Override
    void f() {
        System.out.println("C");
    }
}
```

[2]プリミティブ型を要素としたい場合には、その型のラッパークラス（23.3 節参照）を用います。

以下では、ClassA およびそのサブクラスのオブジェクトを ArrayList<ClassA>に格納し、各要素についてメソッド実行しています。

```
....
ArrayList<ClassA> objects = new ArrayList<ClassA>();

objects.add(new ClassB());
objects.add(new ClassA());
objects.add(new ClassC());
objects.add(new ClassB());

for (int i = 0; i < objects.size(); i++) {
    objects.get(i).f();
}
...
```

プログラムの実行結果は以下のとおりです。

```
B
A
C
B
```

ループでは全てのオブジェクトに同じ処理を行なっているにもかかわらず、オブジェクト毎に出力が異なっていることに注意してください。これは多相性の典型的な（しかし面白くない）例です。

20.1.3　繰り返し処理（その 2、拡張 for 文）

Java では ArrayList のようなデータの並びを扱うオブジェクト[3]について、よりコンパクトな繰り返しの記述ができる、拡張 for 文が利用できます。上のプログラムをこの記法で記述すると以下の通りです。

```
....

for (ClassA a : objects) { // objects に含まれる各 a について繰り返す
    a.f();
}
...
```

for 文のカッコの中に、「要素のデータ型」、「要素を表す変数」、「：」、配列オブジェクトの順に記載し、for 文の本体では、要素を表す変数を持ちます。インデックスを用いる for 文よりも簡潔で読みやすいことを確認してください。この章のプログラムにはこの記法を用います。

[3]正確には Iterable インターフェースを実装するクラスのオブジェクトです。インターフェースについては次章において解説します。

20.1.4　繰り返し処理（その 3、Iterator の利用）

データの並びを扱うオブジェクトの繰り返し処理では繰り返し処理専用オブジェクト — 一般に**イテレータ**（iterator）と呼ばれます — も用いて抽象度を上げた記述も一般的です。以下は、上の繰り返し処理を記述したものです。

```
import java.util.Iterator;  // パッケージのインポートが必要

    ....

    Iterator<ClassA> iterator = objects.iterator(); // Iterator の取得
    while (iterator.hasNext()) {    // 次の要素があるか否かチェック
        ClassA a = iterator.next(); // 次の要素を取り出す
        a.f();
    }
    ...
```

この方法はデータの並びから要素を削除する場合に特に便利であることが知られています。

考察　余裕のある人は「要素を削除する場合に特に便利である」理由を調べなさい。

なお、イテレータはこの講義テキストのプログラムには用いません。

20.2　簡単な例題：乱数の利用

Java では Random クラスのオブジェクトを用いて簡単に乱数を利用できます。

パッケージのインポート： プログラムの冒頭に以下を置きます。

```
import java.util.Random;
```

乱数オブジェクトの生成： 式 new Random() で乱数を生成する機能を持つオブジェクトを作ることができます。引数にシード（乱数の種）を与えない場合、現在時刻をシードに用いますから、ゲーム実行の度に実行内容が変化します。コンストラクタ new Random(long seed) ではシードを引数で渡します。

乱数（整数）の取得： メソッド nextInt(n) は、0 から $n-1$ の間のランダムな整数値を戻します。

以下は、0 から 4 の乱数を 10 個求める簡単な利用例です。

```
import java.util.Random;

class test {
    public static void main(String str[]){
        Random r = new Random();  // 乱数を生成する基となるオブジェクトを作る

        for (int i = 0; i < 10; i++) {
            System.out.println(r.nextInt(5)); // 乱数を 1 個、求める
        }
    }
}
```

よりスマートな利用には Random クラスのサブクラスを作り、それを利用することも考えられますが、この講義のプログラムではそこまで凝る必要はないでしょう。

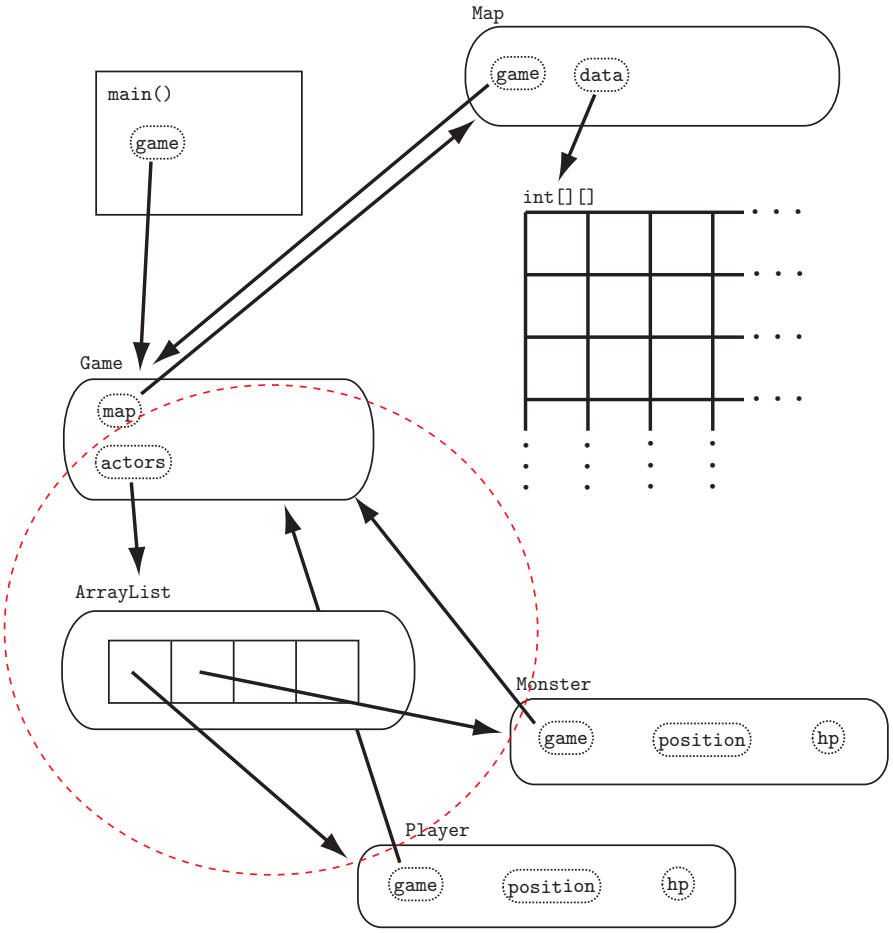


図 20.1: ArrayList を用いるプログラムの全体構造：点線赤枠が変更箇所

20.3　プログラムの構造

ダンジョンゲームの改造に入ります。図 20.1（328 ページ）はこの章のプログラムの構造です。

前章との大きな違いは、Game オブジェクトは、インスタンス変数であるアクター配列 actors を用いてプレイヤー、モンスターを一括管理する点です。actors は ArrayList<Actor> 型であり、Actor クラスとそのサブクラスのオブジェクトを要素として持つことができます。

実際のゲームのプログラムでは、アクター配列の各要素をループで処理していきます。よって、前章のプログラムにおいて actor1、actor2、actor3 について逐一処理していた部分をこの章では全てループに変更します。よって変更箇所は全体に及ぶものになります。

20.4　ゲームクラスの変更

図 20.2（329 ページ）は、この章の Game クラスの定義です。

```
class Game {
    public static void main(String args[]) { ** 前章から変更なし ** }

    static Random random = new Random();           // 乱数オブジェクト

    Map map;

    ArrayList<Actor> actors;                       // アクター配列
    final static int PLAYER_ID = 0;                // プレイヤーの登録位置
    final static int NUM_OF_MONSTERS = 4;          // モンスターの初期総数

    Game() {
        map = new Map(this);

        actors = new ArrayList<Actor>();           // アクター配列の初期化
        actors.add(new Player(this,"Taro"));       // プレイヤーの登録
        for (int i = 0; i < NUM_OF_MONSTER; i++) {
            if (Game.random.nextInt(2) == 0) {
                actors.add(new Monster(this));     // モンスターの登録
            }
            else {
                actors.add(new RedMonster(this));  // レッドモンスターの登録
            }
        }
    }

    void print() {
        map.print();
        for (Actor a : actors) { System.out.println(a); }//各アクターを表示
        System.out.println();
    }

    void mainLoop() { ** 前章から変更なし ** }

    void turn() {
        for (Actor a : actors) { a.action(); }// 各アクターの action() を実行
        print();
    }

    static char getKeyChar() { ** 前章から変更なし ** }
}
```

図 20.2: 参照変数を ArrayList に変更したダンジョンゲームのプログラム（ゲーム）

20.4.1　Random クラスの利用

図 20.2 の以下の宣言：

```
static Random random = new Random();            // 乱数オブジェクト
```

は、乱数オブジェクトを static な変数（クラス変数）として生成します。乱数オブジェクトはプログラム全体にひとつあれば十分なので、static にしました。乱数が必要な場面では全てこのオブジェクトを利用することにします。コンストラクタ内の以下が乱数を用いる場面に当たります。

```
            if (Game.random.nextInt(2) == 0) {
```

このとき、乱数として 0 または 1 が得られます。

20.4.2　アクター配列の作成

前章まで用いてきたプレイヤー、モンスター、レッドモンスターを参照する三つのインスタンス変数はこの章以降では使用しません。コンストラクタ内の以下の 2 行：

```
        actors = new ArrayList<Actor>();        // アクター配列の初期化
        actors.add(new Player(this,"Taro"));    // プレイヤーの生成・登録
```

は、アクター配列を作り、Player オブジェクトを生成し、登録しています。よって配列内の Player オブジェクトの位置は 0 番目です。プログラムではプレイヤーの位置を定数 PLAYER_ID に設定：

```
    final static int PLAYER_ID = 0;             // プレイヤーの登録位置
```

しています。数値 0 を直接、プログラム内に記述することは避けましょう。後日、プログラムを読み直したときに、あるいは他のプログラマがプログラムを読んだときに、その 0 が何を意味するのか解読できない可能性大です。

この章からモンスターの数を以下の定数に設定しています。とりあえず 4 とします。モンスターの数を増やすとマップはにぎやかになりますが、ゲームの難易度が上がります。

```
    final static int NUM_OF_MONSTERS = 4;           // モンスターの数
```

モンスター、レッドモンスターは以下の for 文で配列に追加登録します。乱数値が 0 ならば Monster オブジェクトを生成・登録し、1 ならば RedMonster オブジェクトを生成・登録します。つまり、モンスターとレッドモンスターが等確率で生成されます。

```
for (int i = 0; i < NUM_OF_MONSTER; i++) {
    if (Game.random.nextInt(2) == 0) {
        actors.add(new Monster(this));   // モンスターの登録
    }
    else {
        actors.add(new RedMonster(this));// レッドモンスターの登録
    }
}
```

モンスター数が可変のため、前章までのようにモンスターの初期位置を固定にすることはできません。そこで初期位置は乱数で決めるように変更します。それについてはアクタークラスのプログラムで解説します。

20.4.3　アクター配列の参照

メソッド print() では、アクター配列の各要素の内容を表示します。以下のプログラムパターン：

```
for (Actor a : actors) { ...  } // 各アクター a について繰り返す
```

は、この章のプログラムの頻出パターンです。ループによる一括処理ですが、プレイヤー、モンスター、レッドモンスター毎に異なる処理が行われるのは既に述べた通りです。

メソッド turn() の中の処理は前章では以下の通りでした。

```
actor1.action();
actor2.action();
actor3.action();
```

これを上記の頻出パターンに当てはめて変更します。

```
for (Actor a : actors) { a.action(); }// 各アクターの action() を実行
```

ここでもそれぞれのアクターの種別（プレイヤー、モンスター、レッドモンスター）毎に異なる action() の処理が行われます。

20.5　マップクラスの変更

図 20.3（333 ページ）はこの章の Map クラスの定義です。以下、主な変更箇所の説明です。

20.5.1　アクターの配列のマップ表示

アクター配列の導入に伴う変更部分は以下の 2 箇所です。

print() メソッド内の前章では以下の通りでした。このプログラム片では 3 体のアクターの表示を順に試みています。

```
                if (game.actor1.cellPrint(position)) { continue; };
                if (game.actor2.cellPrint(position)) { continue; };
                if (game.actor3.cellPrint(position)) { continue; };
```

これをこの章のプログラムでは以下のように配列の各アクターに関する for ループで行っています。

```
                for (Actor a : game.actors) { // 各アクターについて
                    if (a.cellPrint(position)) {
                                            // もしアクターが表示されたならば
                        continue x_level;           // x_level へループを脱出
                    }
                }
```

ここに continue x_level; は、ラベル x_level への脱出を行います（下記参照）。なお、この部分の 3 重ループ：

```
        for (int y = 0; y < YSIZE; y++) {
            x_level:
            for (int x = 0; x < XSIZE; x++) {
                ...
                for (Actor a : game.actors) {
                    if (...) { continue x_level; }
                }
                ...
            }
            ...
        }
```

の実行については以下を参考にしてください。

```
class Map {
    ** インスタンス変数は前章から変更なし **

    Map(Game game) { ** 前章から変更なし ** }

    int getAt(Int2 position) { ** 前章から変更なし ** }

    void print() {
        for (int y = 0; y < YSIZE; y++) {
            x_level:                                    // 脱出先のラベル
            for (int x = 0; x < XSIZE; x++) {
                Int2 position = new Int2(x,y);
                for (Actor a : game.actors) { // 各アクターについて
                    if (a.cellPrint(position)) {
                                                // もしアクターが表示されたならば
                        continue x_level;           // x_level ループを脱出
                    }
                }
                cellPrint(position);
                        // もしどのアクターも表示されなかったならば、セルを表示
            }
            System.out.println();
        }
    }

    void cellPrint(Int2 position) { ** 前章から変更なし ** }

    Int2 getRandomCellPosition() {
        while (true) {
            Int2 position = new Int2(Game.random.nextInt(XSIZE),
                                     Game.random.nextInt(YSIZE));
                                                        // 座標位置をランダムに求め
            if (getAt(position) != ROCK) { return position; }
                                                    // 岩でないならば戻り値とする
        }
    }
}
```

図 20.3: 参照変数を ArrayList に変更したダンジョンゲームのプログラム（マップ）

参考（ラベル付き break）：　ループからの途中脱出にはしばしば break を用います。たとえば

```
for (int i = 0; i < N; i++) {
    if (test(i)) { break; }
}
```

では、ループ本体を実行している最中に test(i) が成り立ったならば、N 回実行する前にループを脱出します。これはよくあるプログラミング手法です。

問題は以下のように多重ループの場合です。

```
for (int k = 0; k < L; k++) {
    boolean flag = false;          // フラグのデフォルト値を flase に設定
    for (int j = 0; j < M; j++) {
        for (int i = 0; i < N; i++) {
            if (test(i,j,k)) {
                flag = true;       // フラグを true に変更
                break;             // 最内ループを脱出
            }
        }
        if (flag) { break; } // フラグが true ならば最内ループを途中脱出して
                             // おり、続けて 2 番目のループも途中脱出する
    }
    // 最外ループは途中脱出しない
}
```

つまり、この例では test(i,j,k) が成り立ったならば、i,j の二重ループを途中脱出し、最外の k のループについては処理を継続します。Java 以前の言語では上のようにフラグ flag を用いて順に外側ループへ脱出するプログラムを書いていました。

この鬱陶（うっとう）しさを除くべく、Java では一気に多重ループを脱出できる **ラベル付き** break が導入されました。以下は上と等価なプログラムです。

```
for (int k = 0; k < L; k++) {
    j_level:                       // 処理はこのレベルに戻る
    for (int j = 0; j < M; j++) {
        for (int i = 0; i < N; i++) {
            if (test(i,j,k)) { break j_level; }
        }
    }
}
```

文“ break j_level; ”はラベル j_level が定義されているループ位置まで一気に脱出します。簡潔な記述になっていることを確認してください。

参考（ラベル付き continue）： これはラベル付き break の break を continue に替えたものです。

```
for (int k = 0; k < L; k++) {
    j_level:                        // 処理はこのレベルに戻る
    for (int j = 0; j < M; j++) {
        for (int i = 0; i < N; i++) {
            if (test(i,j,k)) { continue j_level; }
            perform(i,j,k);
        }
        perform(j,k);
    }
}
```

もし test(i,j,k) が成り立ったならば、perform(i,j,k); を実行せず、i のループを脱出し、さらに perform(j,k); も実行せず、j の値を 1 増やして j のループについては処理を継続します。

20.5.2　ランダムな座標位置の生成

メソッド getRandomCellPosition() は新たに導入したもので、マップ内の、岩が無いセルの位置をランダムにひとつ見つけて戻り値とします。メソッド本体では、while 文でランダムな位置の生成とその位置が空（から）か否かのテストを繰り返しています。

20.6　アクタークラスの変更

図 20.4（336 ページ）はこの章のアクタークラスの定義です。以下、主な変更箇所の説明です。

20.6.1　マップ上の初期位置をランダムに決めるコンストラクタ

この章では新たに座標位置を指定しないコンストラクタ：

```
Actor(Game game, int hp, char cellChar) {
```

を多重定義しました。位置が与えられない場合には新たに乱数で位置を決めるものとします。

乱数での位置決めは図 20.4 の二つのコンストラクタ本体の以下の部分です。

```
class Actor {
    ** インスタンス変数は前章から変更なし **

    Actor(Game game, Int2 position, int hp, char cellChar) {
        ** 前章から変更なし **
    }

    Actor(Game game, int hp, char cellChar) {
        this.game = game;
        this.hp = hp;
        this.cellChar = cellChar;
        while_loop:
        while (true) {
            position = game.map.getRandomCellPosition();
                                                    // ランダムに位置を求める
            for (Actor a : game.actors) {
                if (position.equal(a.position)) { continue while_loop; }
                                // position にアクター a が居るならば、再試行
            }
            return;      // 初期位置が確定したため、コンストラクタの実行を終了
        }
    }

    boolean cellPrint(Int2 position) { ** 前章から変更なし **}

    boolean moveTo(Int2 direction) {
        Int2 target = position.add(direction);

        if (game.map.getAt(target) == Map.ROCK) { return false; }

        for (Actor a : game.actors) { // 各アクターについて
            if (target.equal(a.position)) { return false; }
                            // もし移動希望先にアクターが居るならば、移動失敗
        }

        position = target;
        return true;
    }

    void action() {}
}
```

図 20.4: 参照変数を ArrayList に変更したダンジョンゲームのプログラム（アクター）

```
        position = game.map.getRandomCellPosition();
                                                    // ランダムに位置を求める
        for (Actor a : game.actors) {
            if (position.equal(a.position)) { continue while_loop; }
                                // position にアクター a が居るならば、再試行
        }
        return;
```

まず、 Map クラスの getRandomCellPosition() でその位置に岩がないセルの座標位置をランダムにひとつ求め、とりあえずインスタンス変数 position に代入します。これを候補位置とし、全てのアクター（このコンストラクタを呼ぶ時点でアクター配列に既に登録されているアクター）が候補位置に居ないかチェックします。居ないことが確認されたならば、コンストラクタの処理は終了です。

20.6.2　アクターの移動を行う moveTo() の変更

前章の moveTo() では、target に他のアクターが居ないか否かのチェックを以下で行いました。

```
    if (target.equal(game.actor1.position)) { return false; }
    if (target.equal(game.actor2.position)) { return false; }
    if (target.equal(game.actor3.position)) { return false; }
```

この章では、上の 3 回の処理をアクター配列に関するループに置き換えます。

```
    for (Actor a : game.actors) { // 各アクターについて
        if (target.equal(a.position)) { return false; }
                            // もし移動希望先にアクターが居るならば、移動失敗
    }
```

20.7　プレイヤークラスの変更

図 20.5（338 ページ）は Player クラスの変更です。

前章の attackTo()：

```
class Player extends Actor {
    ** インスタンス変数は前章から変更なし **

    Player(Game game, String name) { ** 前章から変更なし ** }

    Player(Game game) { ** 前章から変更なし ** }

    public String toString() { ** 前章から変更なし ** }

    void attackTo(Int2 direction) {
        Int2 target = position.add(direction);

        for (Actor a : game.actors) {             // 各アクターについて
            if (target.equal(a.position)) {       // a が target の位置に居て
                if (a instanceof RedMonster) {    // a が RedMonster ならば
                    a.hp -= 10;                     // hp を 10 減らす
                    return;
                }
                else if (a instanceof Monster) {   // a が Monster ならば
                    a.hp -= 20;                     // hp を 20 減らす
                    return;
                }
            }
        }
    }

    @Override
    void action() { ** 前章から変更なし ** }
}
```

図 20.5: 参照変数を ArrayList に変更したダンジョンゲームのプログラム（アクター）

```
        if (target.equal(game.actor2.position)) {
            game.actor2.hp -= 20;
        }
        if (target.equal(game.actor3.position)) {
            game.actor3.hp -= 10;
        }
```

では、actor2、actor3 それぞれについて、それが target の位置に存在するならば、hp を 20 または 10 減少させる処理を行います。この章ではアクター配列を用いるため、これを以下のように書き直します。

```
for (Actor a : game.actors) {              // 各アクターについて
    if (target.equal(a.position)) {        // a が target の位置に居て
        if (a instanceof RedMonster) {      // a が RedMonster ならば
            a.hp -= 10;                      // hp を 10 減らす
            return;
        }
        else if (a instanceof Monster) {    // a が Monster ならば
            a.hp -= 20;                      // hp を 20 減らす
            return;
        }
    }
}
```

つまり、各アクター a について、a の位置が target の位置にあり、a がレッドモンスターならば hp を 10 減少させ、a がモンスターならば hp を 20 減少させます。ここで「a がレッドモンスターならば」の判定に新規事項の演算子：instanceof を用いています。以下のその解説です。

20.7.1　オブジェクトの所属するクラスを求める：instanceof 演算子

オブジェクト obj がクラス ClassA またはそのサブクラスのオブジェクトであるか否か問い合わせるときには 2 項演算子 instanceof を用いて構文：

```
obj instanceof ClassA
```

で判定できます。サブクラスの場合も true になるため、たとえば先ほどのプログラム片：

```
        if (a instanceof RedMonster) {      // a が RedMonster ならば
            a.hp -= 10;                      // hp を 10 減らす
            return;
        }
        else if (a instanceof Monster) {    // a が Monster ならば
            a.hp -= 20;                      // hp を 20 減らす
            return;
        }
```

の二つの判定を入れ替えて

```
                if (a instanceof Monster) {          // a が Monster ならば
                    a.hp -= 20;                       // hp を 20 減らす
                    return;
                }
                else if (a instanceof RedMonster) { // a が RedMonster ならば
                    a.hp -= 10;                       // hp を 10 減らす
                    return;
                }
```

とすると、正しく動作しません。何故ならば、 Monster クラスのオブジェクトも RedMonster クラスのオブジェクトも、最初の判定 a instanceof Monster で true となってしますからです（レッドモンスターはモンスターの特殊形です）。

ところで、一般に instanceof を用いて「オブジェクトがもし... クラスに属するならば」という処理を行うことはオブジェクト指向プログラミングに反します。本来のオブジェクト指向の考え方では処理は各オブジェクトの自由に任せるべきです。instanceof を用いてオブジェクトのクラスを調べ、処理を選択する方法はオブジェクト指向的ではないと考えられています。第 24 章において上のプログラム部分を instanceof を使わない方式へ改造します。

20.8　モンスタークラスの変更

図 20.6（341 ページ）は Monster クラスの変更です。

変更点は二つです。

ひとつは、この章では引数に初期座標位置を必要としないコンストラクタを用います。既に述べたように初期座標位置は親クラス、アクターのコンストラクタ内で生成します。

もうひとつの変更点は attackTo() メソッドの内部処理です。前章の attackTo() ではプレイヤーを Game オブジェクト game のインスタンス変数 actor1）として参照していました（以下参照）。

```
        if (target.equal(game.player.position)) {
            game.player.hp -= 30;
        }
```

この章では Game オブジェクトのインスタンス変数であるアクター配列 actors の PLAYER_ID 番目の要素として参照します（以下参照）。

```
class Monster extends Actor {
    static final int INIT_HP = 150;

    Monster(Game game, char c) { super(game, INIT_HP, c); }
                                                        // 位置の引数を削除

    Monster(Game game) { this(game, 'M'); }        // デフォルトの文字の設定

    @Override
    public String toString() { ** 前章から変更なし ** }

    void attackTo(Int2 direction) {
        Int2 target = position.add(direction);
        Actor player = game.actors.get(game.PLAYER_ID);
                                                // Player オブジェクトを求める
        if (target.equal(player.position)) {
            player.hp -= 30;
            return;
        }
    }

    @Override
    void action() { ** 前章から変更なし ** }
}
```

図 20.6: 参照変数を ArrayList に変更したダンジョンゲームのプログラム（モンスター）

```
        Actor player = game.actors.get(game.PLAYER_ID);
        if (target.equal(player.position)) {
            player.hp -= 30;
            return;
        }
```

20.9　レッドモンスタークラスの変更

図 20.7（342 ページ）は Monster クラスの変更です。

RedMonster クラスの変更点は、 Monster クラスの変更点と同じです。ひとつは、引数に初期座標位置を必要としないコンストラクタを用いる点、もうひとつは、attackTo() メソッドの内部で PLAYER_ID を参照する点です。

```
class RedMonster extends Monster {
    RedMonster(Game game) { super(game, 'R'); }

    @Override
    public String toString() { ** 前章から変更なし ** }

    @Override
    void attackTo(Int2 direction) {
        Int2 target = position.add(direction);

        Actor player = game.actors.get(game.PLAYER_ID);
                                                // Player オブジェクトを求める
        if (target.equal(player.position)) {
            player.hp -= 60;
            return;
        }
    }
}
```

図 20.7: 参照変数を ArrayList に変更したダンジョンゲームのプログラム（レッドモンスター）

20.10　プログラムの実行

たとえば以下のような実行画面です。この場合、たまたまモンスターが 2 体、レッドモンスターが 2 体です。モンスターの初期位置はランダムに決まります。

```
# # # # # # # # # # # # #
# @ . . . . . . . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . R #
# . . M . # # # . . . . #
# . . . . # # # . . . . #
# . . . . # # # . . . . #
# . . . . . . . . . . . #
# . . M . . . . . . . . #
# . . . . . . R . . . G #
# # # # # # # # # # # # #
Taro : (1,1) 100
RedMonster : (7,9) 150
Monster : (3,4) 150
Monster : (3,8) 150
RedMonster : (11,3) 150

? c                          <---- 右下方向をキー入力
# # # # # # # # # # # # #
# . . . . . . . . . . . #
# . @ . . . . . . . R . #
# . M . . . . . . . . . #
# . . . . # # # . . . . #
# . . . . # # # . . . . #
# . . . . # # # . . . . #
# . M . . . . . . . . . #
# . . . . . R . . . . . #
# . . . . . . . . . . G #
# # # # # # # # # # # # #
Taro : (2,2) 100
RedMonster : (6,8) 150
Monster : (2,3) 150
Monster : (2,7) 150
RedMonster : (10,2) 150


以下同様
```

以下はモンスター数を 10 体（NUM_OF_MONSTERS = 10）に増やした場合です。どう足掻（あが）いても HP が正のままゴールに辿り着けそうにありません。

```
# # # # # # # # # # # # #
# @ . . . . . . R M . . #
# . . M . . . . . . . . #
# . . . . . . . . . . . #
# . M . . # # # . . . . #
# . . R . # # # . . . . #
# . R . . # # # . . R M #
# M . . . . . R . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . G #
```

```
# # # # # # # # # # # # #
Taro : (1,1) 100
Monster : (1,7) 150
Monster : (3,2) 150
Monster : (9,1) 150
RedMonster : (10,6) 150
RedMonster : (2,6) 150
Monster : (11,6) 150
RedMonster : (8,1) 150
RedMonster : (7,7) 150
RedMonster : (3,5) 150
Monster : (2,4) 150

? x                          <---- 下方向をキー入力
# # # # # # # # # # # # #
# . . . . . . . . . . . #
# @ M . . . . R M . . . #
# M . . . . . . . . . . #
# . . . . # # # . . . . #
# R . R . # # # . R M . #
# M . . . # # # . . . . #
# . . . . . . R . . . . #
# . . . . . . . . . . . #
# . . . . . . . . . . G #
# # # # # # # # # # # # #
Taro : (1,2) 100
Monster : (1,6) 150
Monster : (2,2) 150
Monster : (8,2) 150
RedMonster : (9,5) 150
RedMonster : (1,5) 150
Monster : (10,5) 150
RedMonster : (7,2) 150
RedMonster : (7,7) 150
RedMonster : (3,5) 150
Monster : (1,3) 150


以下同様
```

20.11　付録：この章の全プログラムリスト

```
//ゲームクラスのプログラム
import java.io.*;
import java.util.Random;
import java.util.ArrayList;

class Game {
    public static void main(String args[]) {
        Game game = new Game();
        game.mainLoop();
    }

    static Random random = new Random();           // 乱数オブジェクト

    Map map;

    ArrayList<Actor> actors;                       // アクター配列
    final static int PLAYER_ID = 0;                // プレイヤーの登録位置
    final static int NUM_OF_MONSTERS = 4;          // モンスターの初期総数

    Game() {
        map = new Map(this);

        actors = new ArrayList<Actor>();           // アクター配列の初期化
        actors.add(new Player(this,"Taro"));       // プレイヤーの登録
        for (int i = 0; i < NUM_OF_MONSTER; i++) {
            if (Game.random.nextInt(2) == 0) {
                actors.add(new Monster(this));     // モンスターの登録
            }
            else {
                actors.add(new RedMonster(this));  // レッドモンスターの登録
            }
        }
    }

    void print() {
        map.print();
        for (Actor a : actors) { System.out.println(a); }//各アクターを表示
        System.out.println();
    }

    void mainLoop() {
        print();
        while (true) { turn(); }
    }

    void turn() {
        for (Actor a : actors) { a.action(); }// 各アクターの action() を実行
        print();
    }

    static char getKeyChar() {
        char c = 0;
        System.out.print("? ");
        try {
            BufferedReader in = new BufferedReader(
                                new InputStreamReader(System.in));
```

```
            c = in.readLine().charAt(0);
        } catch(IOException e){
            System.out.println("ERROR:: getKeyChar()");
        }
        return c;
    }
}
```

```
//マップクラスのプログラム
class Map {
    static final int ROCK = 0;
    static final int EMPTY = 1;
    static final int GOAL = 2;

    static final int XSIZE = 13;
    static final int YSIZE = 11;

    Game game;
    int data[][]= new int[XSIZE][YSIZE];

    Map(Game game) {
        this.game = game;
        for (int x = 0; x < XSIZE; x++) {
            for (int y = 0; y < YSIZE; y++) {
                if ((x == 0)||(x == XSIZE-1)||
                   (y == 0)||(y == YSIZE-1)||
                   ((Math.abs(x-XSIZE/2) <= 1)&&
                    (Math.abs(y-YSIZE/2) <= 1))) {
                    data[x][y] = ROCK;
                }
                else { data[x][y] = EMPTY; }
            }
        }
        data[XSIZE-2][YSIZE-2] = GOAL;
    }

    int getAt(Int2 position) { return data[position.x][position.y]; }

    void print() {
        for (int y = 0; y < YSIZE; y++) {
            x_level:
            for (int x = 0; x < XSIZE; x++) {
                Int2 position = new Int2(x,y);
                for (Actor a : game.actors) { // 各アクターについて
                    if (a.cellPrint(position)) {
                                                // もしアクターが表示されたならば
                        continue x_level;           // x_level ヘループを脱出
                    }
                }
                cellPrint(position);
                        // もしどのアクターも表示されなかったならば、セルを表示
            }
            System.out.println();
        }
    }

    void cellPrint(Int2 position) {
        int e = getAt(position);
        if (e == ROCK)       { System.out.print('#'+" "); }
        else if (e == EMPTY) { System.out.print('.'+" "); }
        else if (e == GOAL)  { System.out.print('G'+" "); }
        else                 { System.out.print('E'+' '); }
    }

    Int2 getRandomCellPosition() {
        while (true) {
```

```
            Int2 position = new Int2(Game.random.nextInt(XSIZE),
                                     Game.random.nextInt(YSIZE));
                                                    // 座標位置をランダムに求め
            if (getAt(position) != ROCK) { return position; }
                                                   // 岩でないならば戻り値とする
        }
    }
}
```

```
//アクタークラスのプログラム
class Actor {
    Game game;
    Int2 position;
    int hp;
    char cellChar;

    Actor(Game game, Int2 position, int hp, char cellChar) {
        this.game = game;
        this.position = position;
        this.hp = hp;
        this.cellChar = cellChar;
    }

    Actor(Game game, int hp, char cellChar) {
        this.game = game;
        this.hp = hp;
        this.cellChar = cellChar;
        while_loop:
        while (true) {
            position = game.map.getRandomCellPosition();
                                                    // ランダムに位置を求める
            for (Actor a : game.actors) {
                if (position.equal(a.position)) { continue while_loop; }
                                // position にアクター a が居るならば、再試行
            }
            return;      // 初期位置が確定したため、コンストラクタの実行を終了
        }
    }

    boolean cellPrint(Int2 position) {
        if (this.position.equal(position)) {
            System.out.print(cellChar+" ");
            return true;
        }
        return false;
    }

    boolean moveTo(Int2 direction) {
        Int2 target = position.add(direction);

        if (game.map.getAt(target) == Map.ROCK) { return false; }

        for (Actor a : game.actors) { // 各アクターについて
            if (target.equal(a.position)) { return false; }
                            // もし移動希望先にアクターが居るならば、移動失敗
        }

        position = target;
        return true;
    }

    void action() {}
}
```

```
//プレイヤークラスのプログラム
class Player extends Actor {
    static final int INIT_HP = 100;

    String name;

    Player(Game game, String name) {
        super(game, new Int2(1,1), INIT_HP, '@');
        this.name = name;
    }

    Player(Game game) { this(game, "Player"); }

    public String toString() {
        return name+" : "+position+" "+hp;
    }

    void attackTo(Int2 direction) {
        Int2 target = position.add(direction);

        for (Actor a : game.actors) {             // 各アクターについて
            if (target.equal(a.position)) {       // a が target の位置に居て
                if (a instanceof RedMonster) {    // a が RedMonster ならば
                    a.hp -= 10;                     // hp を 10 減らす
                    return;
                }
                else if (a instanceof Monster) {   // a が Monster ならば
                    a.hp -= 20;                     // hp を 20 減らす
                    return;
                }
            }
        }
    }

    @Override
    void action() {
        Int2 direction;
        switch (Game.getKeyChar()) {
            case 'z': direction = new Int2(-1,+1); break;
            case 'x': direction = new Int2( 0,+1); break;
            case 'c': direction = new Int2(+1,+1); break;
            case 'a': direction = new Int2(-1, 0); break;
            case 'd': direction = new Int2(+1, 0); break;
            case 'q': direction = new Int2(-1,-1); break;
            case 'w': direction = new Int2( 0,-1); break;
            case 'e': direction = new Int2(+1,-1); break;
            default:  direction = new Int2( 0, 0);
        }

        if (!moveTo(direction)) { attackTo(direction); }
    }
}
```

```
//モンスタークラスのプログラム
class Monster extends Actor {
    static final int INIT_HP = 150;

    Monster(Game game, char c) { super(game, INIT_HP, c); }
                                                        // 位置の引数を削除

    Monster(Game game) { this(game, 'M'); }        // デフォルトの文字の設定

    @Override
    public String toString() {
        return "Monster : "+position+" "+hp;
    }

    void attackTo(Int2 direction) {
        Int2 target = position.add(direction);
        Actor player = game.actors.get(game.PLAYER_ID);
                                                // Player オブジェクトを求める
        if (target.equal(player.position)) {
            player.hp -= 30;
            return;
        }
    }

    @Override
    void action() {
        Actor player = game.actors.get(game.PLAYER_ID);
        Int2 direction = player.position.sub(position).signum();

        if (!moveTo(direction)) { attackTo(direction); }
    }
}
```

```
//レッドモンスタークラスのプログラム
class RedMonster extends Monster {
    RedMonster(Game game) { super(game, 'R'); }

    @Override
    public String toString() {
        return "Red"+super.toString();
    }

    @Override
    void attackTo(Int2 direction) {
        Int2 target = position.add(direction);

        Actor player = game.actors.get(game.PLAYER_ID);
                                                // Player オブジェクトを求める
        if (target.equal(player.position)) {
            player.hp -= 60;
            return;
        }
    }
}
```

```
//2 次元ベクトルクラスのプログラム
class Int2 {
    int x,y;

    Int2(int x, int y) { this.x = x; this.y = y; }
    Int2 add(Int2 a) { return new Int2(x+a.x,y+a.y); }
    Int2 sub(Int2 a) { return new Int2(x-a.x,y-a.y); }
    Int2 signum() { return new Int2(Integer.signum(x),Integer.signum(y)); }
    boolean equal(Int2 a) { return (x == a.x)&&(y == a.y); }

    public String toString(){ return "("+x+","+y+")"; }
}
```

1

第 20 章　プレイヤー、モンスターをアクターの配列で一括管理する：多相性の活用

1

第21章　アクターというオブジェクトは存在しない：抽象クラスとインターフェース

このテキストで取り扱っているアクターはプレイヤーとモンスターの共通する性質をまとめるために導入したクラスです。アクターのオブジェクト実体を作ることは想定されていません。これはちょうど、「人間」と「猫」に共通する性質を「哺乳類」としてまとめる場合に似ています。人間でもなく猫でもない純粋な哺乳類という生き物は世の中に存在しません[1]。このように概念を整理する上で有用ではあるものの、実際にオブジェクト実体が生成されることのないクラスを**抽象クラス**（abstract class）と呼びます。抽象クラスでないクラスは**具象クラス**（concrete class）です。

抽象クラスのメソッドの中には、その抽象クラスの全てのサブクラスから上書きされているメソッドがありえます。そのようなメソッドはどのような状況においても決して実行されることはありません。たとえば 19.4 節、図 19.2（302 ページ）で導入した `Actor` クラスのメソッド：

```
void action() {}                // アクターは action() を持つ必要がある！
```

は、サブクラスが `action()` を持つことを保証する上で必要ですが、このメソッド自体が実行されることはありません。実行されないならば、メソッドの存在のみを宣言すれば十分です。わざわざ実行されない空（から）のメソッド本体を記述することは無駄です。実際、Java ではそのような本体を持たないメソッドの宣言が許されており、それを**抽象メソッド**（abstract method）と呼んでいます。抽象メソッドを持つクラスは必ず抽象クラスでなければなりません[2]。

なお、抽象クラスは 14 章（199 ページ）の抽象データ型とは全く別の概念です。混同しないように注意してください。

次にこの抽象クラスのアイディアをさらに推し進め、

- そのクラスのメソッドは全て抽象メソッドである

[1]厳密に言えば「猫」という動物も存在しません。確実に存在すると言えるのは、個体を特定できる、たとえば山田家のミケや、7 号館の裏に住むノラなどになるのでしょう。しかし、この講義ではそのような「存在とは何か？」という議論に踏み込むことはしません。

[2]逆は一般には成り立ちません。抽象メソッドを持たない抽象クラスはあり得ます。

ようなクラスを考えることができます。これは抽象クラスの特殊形ですが、このようなクラスを Java では**インターフェース**（interface）と呼んでいます。「interface」の辞書的意味は「境界面」ですが、Java では「外部仕様」と意訳するほうがより近いでしょう。何故ならば、インターフェースには全てのメソッドが実装されておらず、メソッドの名前、引数の型と個数、戻り値の型のみが定義されている[3]からです。インターフェースは Java のオブジェクト指向において重要な役割を担うのですが、それについては後章で解説します。

21.1　抽象クラスの簡単な例題

抽象クラスを用いた簡単な例題を紹介します。

以下は `class` の直前に単にキーワード `abstract` を付けたクラス定義です。`abstract` で修飾されたことによって、抽象クラスとみなされ、このクラスのオブジェクトは生成できなくなります。

```
abstract class ClassA {  // 抽象クラス
    int a = 111;

    void f() {
        System.out.println(a);
    }

    void g() {
        System.out.println("g is called.");
    }
}
```

実際、プログラムの中に `new ClassA()` があると、コンパイルエラーとなります。

メソッド本体が記述されない抽象メソッドは、メソッドの宣言の直前にキーワード `abstract` を付けます。以下のプログラムでは `f()` が抽象メソッドです。

```
abstract class ClassA {
    int a = 111;

    abstract void f();    // 抽象メソッド

    void g() {
        System.out.println("g is called.");
    }
}
```

[3]メソッド（関数）の名前、引数の型と個数、戻り値の型の組み合わせを一般にシグニチャ（signiture）と呼びます。これはメソッドの最も単純な外部仕様に相当します。なお、戻り値の型を除外して考える場合もあります。

抽象クラスのサブクラスでは抽象メソッドを必ず上書きせねばなりません。たとえば以下は ClassA のサブクラスの例です。

```
class ClassB extends ClassA {
    int b = 222;

    @Override
    void f() {              // 抽象メソッドの実装
        System.out.println(a + " " + b);
    }
}
```

もし抽象メソッドを上書きしないならば、そのクラスも抽象クラスと見なされ、オブジェクトを生成できません。その場合、さらにその下のサブクラスにおいて抽象メソッドを上書きし、メソッド本体を実装する必要があります。たとえば以下では、ClassA のサブクラス ClassC も抽象クラスであり、そのサブクラスの ClassD が具象クラスです。

```
// 抽象クラスのサブクラスがさらに抽象クラスの場合
abstract class ClassC extends ClassA {
    int c = 333;
}

class ClassD extends classC {
    @Override
    void f() {              // 抽象メソッドの実装
        System.out.println(a + " " + c);
    }
}
```

21.2　インターフェースの簡単な例題

インターフェースを用いた簡単な例題を紹介します。

```
interface InterfaceA {  // インターフェース
    void f();
    void g();
}
```

宣言のみからなることに注意してください。上のインターフェースを抽象クラスとして等価記述すれば以下の通りです。

```
abstract class InterfaceA {
    public abstract void f();
    public abstract void g();
}
```

メソッドが public 修飾されていることに注意してください。上に述べたように、インターフェースはプログラムの外部仕様に相当します。外部仕様は外部へ公開することが原則ですから、

- インターフェースのメソッドは全て暗黙に public（外部へ公開する）修飾される

ものと約束されています。

インターフェースを利用するには、サブクラスを作った場合とほとんど同様に、実装クラスを定義します。クラスの継承の場合にはキーワード extends （拡張する）を用いましたが、インターフェースの実装の場合にはキーワード implements（実装する）を用います。なお、インターフェースの場合には、サブクラス、子クラスといった術語よりも **実装クラス**、**実装するクラス** という呼び方が好まれるようです。

```
class ClassB implements InterfaceA {
    int a = 111;
    int b = 222;

    @Override
    public void f() {
        System.out.println(a + " " + b);
    }

    @Override
    public void g() {
        System.out.println("g is called.");
    }
}
```

上に述べたように、オーバライドするメソッドは public 修飾する必要があります。

オブジェクトの作り方、使い方はクラスの場合と同様です。以下がその例です。

```
class Test {
    public static void main(String[] str) {
        InterfaceA obj = new ClassB();
        obj.f();
        obj.g();
    }
}
```

```
abstract class Actor {                          // abstract の追加
    ** インスタンス変数は前章から変更なし **

    Actor(Game game, Int2 position, int hp, char cellChar) {
        ** 前章から変更なし **
    }

    Actor(Game game, int hp, char cellChar) { ** 前章から変更なし ** }

    boolean cellPrint(Int2 position) { ** 前章から変更なし ** }

    boolean moveTo(Int2 direction) { ** 前章から変更なし ** }

    abstract void action();                 // abstract の追加、本体 {} の削除
}
```

図 21.1: 抽象クラスを導入したダンジョンゲームのプログラム（アクター）

obj で参照されているオブジェクトはサブクラスかつ具象クラスのオブジェクトであることに注意してください。インターフェースは抽象クラスですから、インターフェースそのもののオブジェクトを生成することはできません。

何故わざわざインターフェースなどという代物を導入したのでしょうか。直接の理由は多重継承の代替策です。これについては 23 章で概説しますし、27 章以降でうんざりするほどインターフェースと付き合うことになります。

21.3　アクタークラスの変更

図 21.1（360 ページ）はアクタークラスの変更です。そこに注釈されている 2 箇所のみの変更です。

まず、クラスを abstract 修飾しました。次に、前章まで

```
    void action() {}
```

であった定義の空（から）の本体 {} を無くし、abstract 修飾しました。

プログラム中に他の変更は一切ありません。プログラムの実行も前章から変更ありません。ただし、この変更によって Actor クラスそのもののオブジェクトは今後、生成できなくなります。

この章の解説は以上です。

21.4　C++における抽象クラス

以下は、21.1 節（357 ページ）の例題でした。

```
abstract class ClassA {
    int a = 111;

    abstract void f();    // 抽象メソッド

    void g() {
        System.out.println("g is called.");
    }
}
```

これを C++に書き直すと以下の通りです。

```
class ClassA {    // C++にキーワード abstract はありません。
  public:
    int a = 111;

    virtual void f() = 0; // abstract の代わりに = 0 を用いる

    void g() {
        cout << "g is called." << endl;
    }
}
```

図 19.9（312 ページ）で述べたように、メンバー関数を修飾する `virtual` はそのメンバー関数に多相性を持たせることを指定するキーワードです。多相性を持つメンバー関数を仮想関数と呼びます。本体を持たない仮想関数は C++でも宣言可能です。その場合、上のプログラム例のように“`= 0`”を付けます[4]。このような関数を C++では **純粋仮想関数**（pure virtual function）と呼びます。Java の抽象メソッドに相当するものです。

C++には抽象クラスを宣言するキーワードはなく、純粋仮想関数を持つクラスが抽象クラスと見なされます。

[4]“`= 0`”はなんとも奇妙な構文ですが、本体が空（から）、つまりヌル=`NULL` = 0 であることから決められた構文なのでしょう（たぶん）。

21.5　付録：この章の全プログラムリスト

```
//ゲームクラスのプログラム
import java.io.*;
import java.util.Random;
import java.util.ArrayList;

class Game {
    public static void main(String args[]) {
        Game game = new Game();
        game.mainLoop();
    }

    static Random random = new Random();

    Map map;
    ArrayList<Actor> actors;
    final static int PLAYER_ID = 0;
    final static int NUM_OF_MONSTERS = 4;

    Game() {
        map = new Map(this);
        actors = new ArrayList<Actor>();
        actors.add(new Player(this,"Taro"));
        for (int i = 0; i < NUM_OF_MONSTERS; i++) {
            if (random.nextInt(2) == 0) {
                actors.add(new Monster(this));
            }
            else {
                actors.add(new RedMonster(this));
            }
        }
    }

    void print() {
        map.print();
        for (Actor a : actors) { System.out.println(a); }
        System.out.println();
    }

    void mainLoop() {
        print();
        while (true) { turn(); }
    }

    void turn() {
        for (Actor a : actors) { a.action(); }
        print();
    }

    static char getKeyChar() {
        char c = 0;
        System.out.print("? ");
        try {
            BufferedReader in = new BufferedReader(
                                new InputStreamReader(System.in));
            c = in.readLine().charAt(0);
        } catch(IOException e){
```

```
            System.out.println("ERROR:: getKeyChar()");
        }
        return c;
    }
}
```

```
//マップクラスのプログラム
class Map {
    static final int ROCK = 0;
    static final int EMPTY = 1;
    static final int GOAL = 2;

    static final int XSIZE = 13;
    static final int YSIZE = 11;

    Game game;
    int data[][]= new int[XSIZE][YSIZE];

    Map(Game game) {
        this.game = game;
        for (int x = 0; x < XSIZE; x++) {
            for (int y = 0; y < YSIZE; y++) {
                if ((x == 0)||(x == XSIZE-1)||
                   (y == 0)||(y == YSIZE-1)||
                   ((Math.abs(x-XSIZE/2) <= 1)&&
                    (Math.abs(y-YSIZE/2) <= 1))) {
                    data[x][y] = ROCK;
                }
                else { data[x][y] = EMPTY; }
            }
        }
        data[XSIZE-2][YSIZE-2] = GOAL;
    }

    int getAt(Int2 position) { return data[position.x][position.y]; }

    void print() {
        for (int y = 0; y < YSIZE; y++) {
            x_level:
            for (int x = 0; x < XSIZE; x++) {
                Int2 position = new Int2(x,y);

                for (Actor a : game.actors) {
                    if (a.cellPrint(position)) {
                        continue x_level;
                    }
                }
                cellPrint(position);
            }
            System.out.println();
        }
    }

    void cellPrint(Int2 position) {
        int e = getAt(position);
        if (e == ROCK)       { System.out.print('#'+" "); }
        else if (e == EMPTY) { System.out.print('.'+" "); }
        else if (e == GOAL)  { System.out.print('G'+" "); }
        else                 { System.out.print('E'+' '); }
    }

    Int2 getRandomCellPosition() {
        while (true) {
            Int2 position = new Int2(Game.random.nextInt(XSIZE), Game.random.nextInt(YSIZE));
```

```

            if (getAt(position) != ROCK) { return position; }
        }
    }
}
```

```
//アクタークラスのプログラム
abstract class Actor {                        // abstract の追加
    Game game;
    Int2 position;
    int hp;
    char cellChar;

    Actor(Game game, Int2 position, int hp, char cellChar) {
        this.game = game;
        this.position = position;
        this.hp = hp;
        this.cellChar = cellChar;
    }

    Actor(Game game, int hp, char cellChar) {
        this.game = game;
        this.hp = hp;
        this.cellChar = cellChar;
        while_loop:
        while (true) {
            position = game.map.getRandomCellPosition();
            for (Actor a : game.actors) {
                if (position.equal(a.position)) { continue while_loop; }
            }
            return;
        }
    }

    boolean cellPrint(Int2 position) {
        if (this.position.equal(position)) {
            System.out.print(cellChar+" ");
            return true;
        }
        return false;
    }

    boolean moveTo(Int2 direction) {
        Int2 target = position.add(direction);

        if (game.map.getAt(target) == Map.ROCK) { return false; }

        for (Actor a : game.actors) {
            if (target.equal(a.position)) { return false; }
        }

        position = target;
        return true;
    }

    abstract void action();               // abstract の追加、本体 {} の削除
}
```

```
//プレイヤークラスのプログラム
class Player extends Actor {
    static final int INIT_HP = 100;

    String name;

    Player(Game game, String name) {
        super(game, new Int2(1,1), INIT_HP, '@');
        this.name = name;
    }

    Player(Game game) { this(game, "Player"); }

    public String toString() {
        return name+" : "+position+" "+hp;
    }

    void attackTo(Int2 direction) {
        Int2 target = position.add(direction);

        for (Actor a : game.actors) {
            if (target.equal(a.position)) {
                if (a instanceof RedMonster) {
                    a.hp -= 10;
                    return;
                }
                else if (a instanceof Monster) {
                    a.hp -= 20;
                    return;
                }
            }
        }
    }

    @Override
    void action() {
        Int2 direction;
        switch (Game.getKeyChar()) {
            case 'z': direction = new Int2(-1,+1); break;
            case 'x': direction = new Int2( 0,+1); break;
            case 'c': direction = new Int2(+1,+1); break;
            case 'a': direction = new Int2(-1, 0); break;
            case 'd': direction = new Int2(+1, 0); break;
            case 'q': direction = new Int2(-1,-1); break;
            case 'w': direction = new Int2( 0,-1); break;
            case 'e': direction = new Int2(+1,-1); break;
            default:  direction = new Int2( 0, 0);
        }

        if (!moveTo(direction)) { attackTo(direction); }
    }
}
```

```
//モンスタークラスのプログラム
class Monster extends Actor {
    static final int INIT_HP = 150;

    Monster(Game game, char c) { super(game, INIT_HP, c); }

    Monster(Game game) { this(game, 'M'); }

    @Override
    public String toString() {
        return "Monster : "+position+" "+hp;
    }

    void attackTo(Int2 direction) {
        Int2 target = position.add(direction);

        Actor player = game.actors.get(game.PLAYER_ID);
        if (target.equal(player.position)) {
            player.hp -= 30;
            return;
        }
    }

    @Override
    void action() {
        Actor player = game.actors.get(game.PLAYER_ID);
        Int2 direction = player.position.sub(position).signum();

        if (!moveTo(direction)) { attackTo(direction); }
    }
}
```

```
//レッドモンスタークラスのプログラム
class RedMonster extends Monster {
    RedMonster(Game game) { super(game, 'R'); }

    @Override
    public String toString() {
        return "Red"+super.toString();
    }

    @Override
    void attackTo(Int2 direction) {
        Int2 target = position.add(direction);

        Actor player = game.actors.get(game.PLAYER_ID);
        if (target.equal(player.position)) {
            player.hp -= 60;
            return;
        }
    }
}
```

```
//2 次元ベクトルクラスのプログラム
class Int2 {
    int x,y;

    Int2(int x, int y) { this.x = x; this.y = y; }
    Int2 add(Int2 a) { return new Int2(x+a.x,y+a.y); }
    Int2 sub(Int2 a) { return new Int2(x-a.x,y-a.y); }
    Int2 signum() { return new Int2(Integer.signum(x),Integer.signum(y)); }
    boolean equal(Int2 a) { return (x == a.x)&&(y == a.y); }

    public String toString(){ return "("+x+","+y+")"; }
}
```

第 21 章　アクターというオブジェクトは存在しない：抽象クラスとインターフェース

1

1

第22章　床にポーションとポイズンを置く：抽象クラスの利用

前章まではマップの構成要素であるセルは int 型であり、セルの種別を保持するだけの受動的な役割しか果たしていませんでした。この章ではセルをオブジェクトに格上げし、そのセルにアクターが移動したとき（または、そのセルをアクターが踏んだとき）にアクターに様々な作用を及ぼす能動的な役割を付与します。

この章は前章、前々章の内容を確認する意図で作りました。新規事項はありません。

22.1　セルのクラス階層

以下は、この章で導入するセルのクラスの説明です。これらは図 22.1（374 ページ）に示すクラス階層の上にあります。

Cell ... クラス階層の頂上にある抽象クラスで、全てのセルの性質を宣言しているものです。

SimpleCell ... 罠ではない、単純な仕組みのセルを表す抽象クラスです。Cell の抽象メソッドの一部を実装しています。

TrapCell ... 初期状態では見えない、罠を表す抽象クラスです。罠の可視/不可視の状態を保持するインスタンス変数 visible を有します。Cell の抽象メソッドの一部を実装しています。

RockCell ... 岩を表すセルです。マップ上ではこれまで同様 '#' で表示します。

EmptyCell ... 最もシンプルな空（から）のセルです。マップ上ではこれまで同様 '.' で表示します。

PotionCell ... ポーション（回復薬）が載っているセルです。マップ上では 'o' で表示します。アクターがこのセルを踏むと、そのアクターの HP が増えます。ただし効果は 1 回限りとし、直後に EmptyCell に置き換わります。

GoalCell ... プレイヤーがこのセルに入るとゲームは成功裏に終了です。マップ上では これまで同様 'G' で表示します。

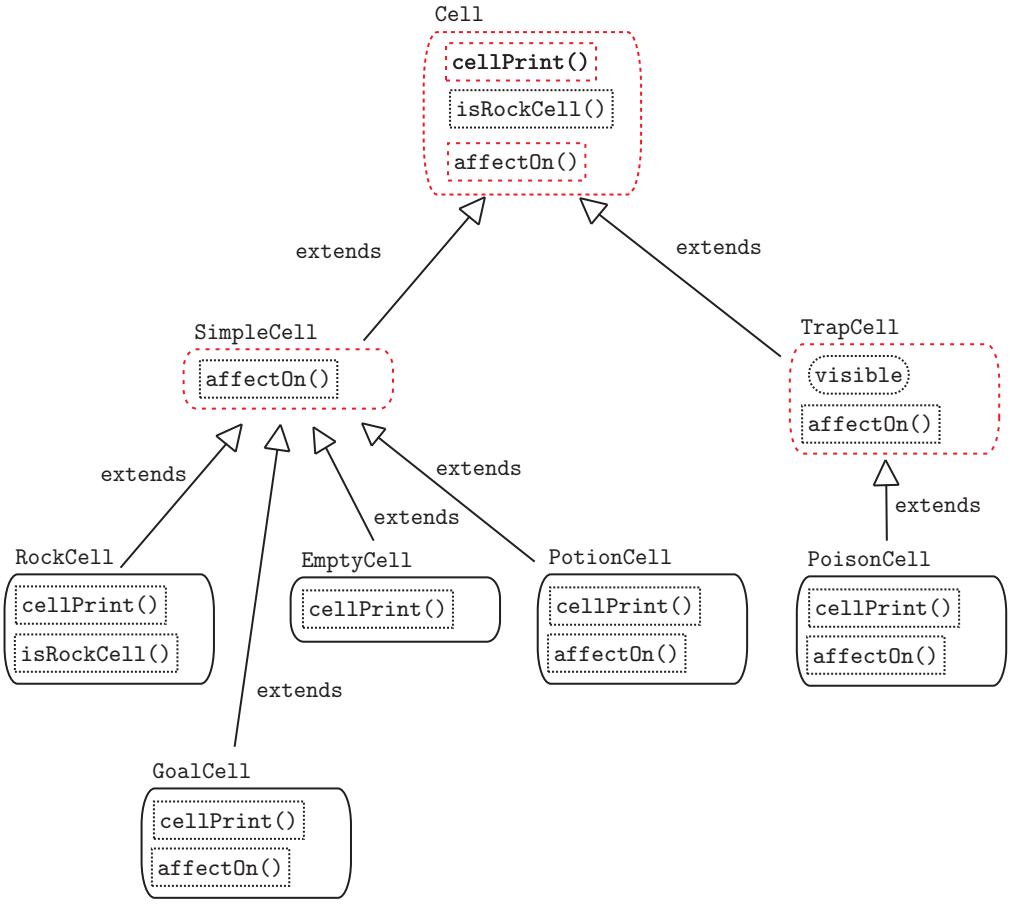


図 22.1: 様々なセルのクラス階層：赤い点線の囲みは、抽象クラスまたは抽象メソッドを表します。

`PoisonCell` ... ポイズン（毒）が載っている罠セルです。 初期状態では不可視です。アクターがこのセルを踏むと、可視に変化し、そのアクターの HP が減ります。以後、可視の状態で、このセルを踏むアクターの HP を繰り返し減らします。罠発覚後、マップ上では ‘x’ で表示します。

22.2　セルの抽象クラス

図 22.2（375 ページ）は、三つの抽象クラスのプログラムです。

`Cell` クラスではメソッド `cellPrint()` と `affectOn()` が抽象メソッドであることを確認してください。メソッド `cellPrint()` は、`Map` オブジェクトのメソッド `print()` においてセルを表示するときに呼び出されるメソッドです。セルの種類毎に異なる表示を行います。唯一実装されている：

```
abstract class Cell {
    abstract void cellPrint();              // マップの表示に用いる

    boolean isRockCell() { return false; } // デフォルトでは岩ではない

    abstract Cell affectOn(Actor a);        // アクターへの影響を記述する
}

abstract class SimpleCell extends Cell {
    @Override
    Cell affectOn(Actor a) { return this; }
                                        // アクターへ影響せず、セルを置換しない
}

abstract class TrapCell extends Cell {
    boolean visible = false;            // 初期値では不可視

    @Override
    Cell affectOn(Actor a) {
        visible = true;         // アクターがこのセルを踏んだならば、可視へ変更
        return this;            // セルを置換しない
    }
}
```

図 22.2: インターフェースを導入したダンジョンゲームのプログラム（セル、その 1）

```
    boolean isRockCell() { return false; } // デフォルトでは岩ではない
```

は、ほとんどのセルのデフォルトの実装を表します。RockCell のみデフォルトの動作を上書きし、true を戻します。メソッド Cell affectOn(Actor a) には、アクター a がこのセルを踏んだときの a への作用を実装します。戻り値がセルである理由は、たとえばポーションのあるセルでは、そのポーションは消費されたならば、セルはポーションの無いセルに置き換えることを想定しているためです。メソッドは置き換えるべきセルを戻り値として返します。置き換える必要がない場合には自分自身 this を返します。

さて、抽象クラスのサブクラスが抽象クラスであっても構いません。抽象クラス SimpleCell クラスは Cell クラスのサブクラスであり、 affectOn() を実装していますが、 cellPrint() は未実装です。TrapCell クラスも同様です。

SimpleCell クラスの

```
    Cell affectOn(Actor a) { return this; }
                                        // アクターへ影響せず、セルを置換しない
```

は、単に自分自身を返しているだけです。返されたセルがマップ上の同じ位置の新たな構成要素となりますが、自分自身が新たな構成要素となるため、結局、何も変化しません。

TrapCell クラスの

```
    Cell affectOn(Actor a) {
        visible = true;         // アクターがこのセルを踏んだならば、可視へ変更
        return this;            // セルを置換しない
    }
```

も自分自身を返す点では同様ですが、アクターがこのセルを踏んだならばセルは可視になる（罠が明らかになる）ため、visible = true; を実行します。

22.3　セルの具象クラス

図 22.3（377 ページ）、図 22.4（378 ページ） は、抽象クラスの下の具象クラスのプログラムです。

22.3.1　RockCell クラス、EmptyCell クラス

RockCell クラスの

```
    public boolean isRockCell() { return true; } // このオブジェクトは岩である
```

は、親クラスの定義を上書きするものです。デフォルトでは false ですが、RockCell クラス のみ true を返します。

22.3.2　EmptyCell クラス

特に説明すべき事項はありません。

22.3.3　PotionCell

PotionCell クラスの

```
        a.hp += 50;                                 // アクターの HP を 50 増やす
        return new EmptyCell(); // ポーションを消費したら、EmptyCell に置換
```

```
class RockCell extends SimpleCell {             // 岩のセル
    @Override
    void cellPrint() { System.out.print("# "); } // # で表示することを実装

    @Override
    boolean isRockCell() { return true; }        // デフォルトの実装を上書き
}

class EmptyCell extends SimpleCell {            // 空（から）のセル
    @Override
    void cellPrint() { System.out.print(". "); } // . で表示することを実装
}

class PotionCell extends SimpleCell {          // ポーションの置かれているセル
    @Override
    void cellPrint() { System.out.print("o "); } // o で表示することを実装

    @Override
    Cell affectOn(Actor a) {
        a.hp += 50;                              // アクターの HP を 50 増やす
        return new EmptyCell();                  // 空のセルに置き換わる
    }
}

class GoalCell extends SimpleCell {             // ゴールセル
    @Override
    void cellPrint() { System.out.print("G "); } // G で表示することを実装

    @Override
    Cell affectOn(Actor a) {
        if (a instanceof Player) {               // プレイヤーならば
            System.out.println("YOU WIN");       // YOU WIN を表示して
            System.exit(0);                      // ゲームを終了
        }
        return this;
    }
}
```

図 22.3: インターフェースを導入したダンジョンゲームのプログラム（セル、その 2）

```
class PoisonCell extends TrapCell {            // 毒のあるセル
    @Override
    void cellPrint() {
        // 可視ならば x、不可視ならば . で表示
        if (visible) { System.out.print("x "); }
        else { System.out.print(". "); }
    }

    @Override
    Cell affectOn(Actor a) {
        a.hp -= 10;                            // アクターの HP を 10 減らす
        return super.affectOn(a);              // 親クラスのメソッドを実行
    }
}
```

図 22.4: インターフェースを導入したダンジョンゲームのプログラム（セル、その 3）

は、アクターを回復させた後、マップ上のこの位置のセルを EmptyCell に置き換えることを意味します。置き換え部分については後節の Actor クラスのプログラムで解説します。

22.3.4　GoalCell クラス

このセルを踏んだアクターがプレイヤーの場合（a instanceof Player）には、

```
System.out.println("YOU WIN");        // YOU WIN を表示して
System.exit(0);                       // ゲームを終了
```

によって、成功のメッセージを出力し、プログラムを強制終了します（荒っぽいですが）。

22.3.5　PoisonCell

PoisonCell クラスの cellPrint()：

```
    void cellPrint() {
        if (visible) { System.out.print("x "); }
        else { System.out.print(". "); }
    }
```

では、visible が初期値の false ならば、出力は EmptyCell と同じ ‘.’ ですが、一旦、アクターがこのセルを踏むと visible は true となり、罠が ‘x’ で表示されます。

affectOn() ：

```
Cell affectOn(Actor a) {
    a.hp -= 10;
    return super.affectOn(a);
}
```

では、アクターの HP を 10 減じ、スーパークラス、すなわち TrapCell クラスの affectOn() を呼び出します。visible の値の更新はスーパークラスで行なっています（図 22.2（375 ページ）参照）。

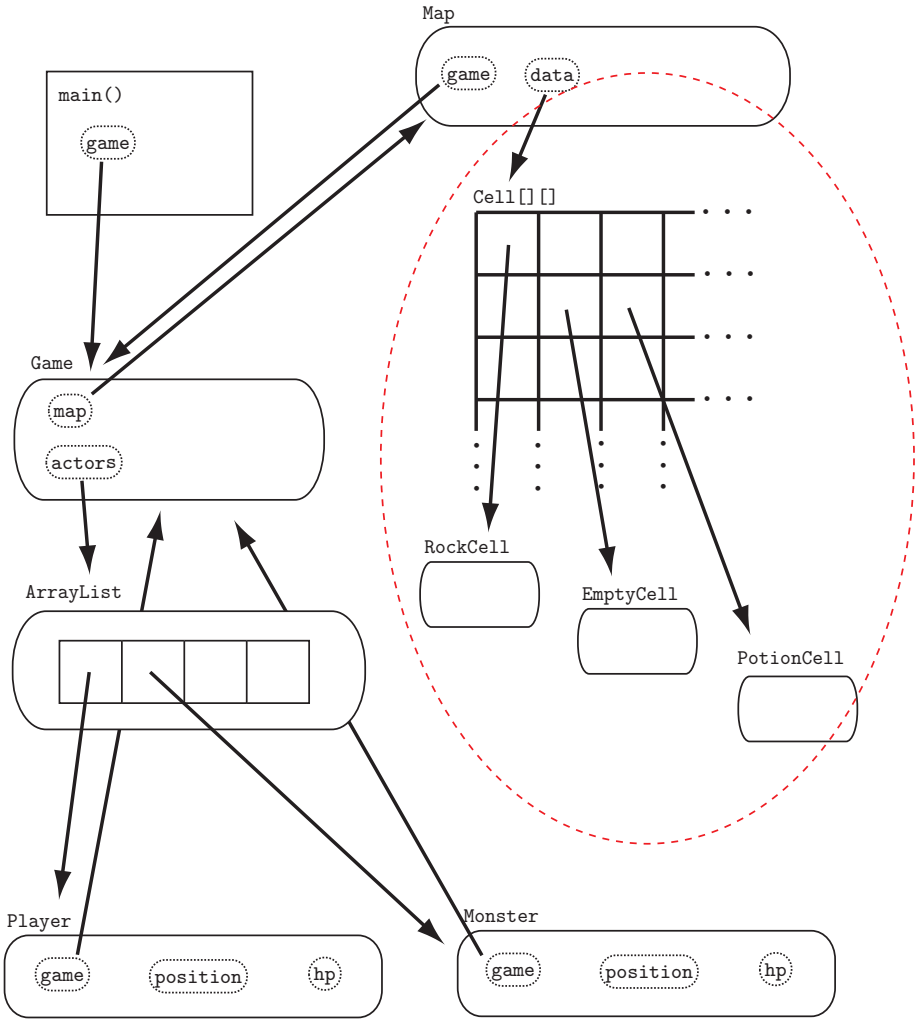


図 22.5: Cell のクラス群を用いるプログラムの全体構造

22.4　プログラムの構造

この章のプログラムの構造は図 22.5（380 ページ）のようになります。赤い点線の枠が変更箇所です。

前章まではセルは int 型でしたが、この章で Cell 型に変更しました。これに伴ってゲームのプログラムが一部変更になります。

22.5　ゲームクラスの変更

ほぼマップオブジェクト内の変更のため、大局的なゲームクラスのレベルでのプログラムの変更はありません。

```
class Map {
    // 定数 ROCK、EMPTY、GOAL の宣言は削除

    static final int XSIZE = 13;
    static final int YSIZE = 11;

    Game game;
    Cell data[][]= new Cell[XSIZE][YSIZE];            // int から Cell へ変更

    Map(Game game) {
        this.game = game;
        ** 中略（前章と同じ） **
                    data[x][y] = new RockCell();               // 岩
                }
                else {
                    switch (Game.random.nextInt(5)) {       // 乱数で選択
                    case 0: data[x][y] = new PotionCell();   // ポーション
                            break;
                    case 1: data[x][y] = new PoisonCell();     // ポイズン
                            break;
                    default: data[x][y] = new EmptyCell();     // 空のセル
                    }
                }
            }
        }
        data[XSIZE-2][YSIZE-2] = new GoalCell();               // ゴール
    }

    Cell getAt(Int2 position) { ** 前章から変更なし ** }

    void setAt(Int2 position, Cell cell) {     // セルを position の位置に設定
        data[position.x][position.y] = cell;
    }
```

図 22.6: Cell のクラス群を用いるダンジョンゲームのプログラム（マップ、その 1）

22.6　マップクラスの変更

この章のマップクラスのプログラムは図 22.6（381 ページ）、図 22.7（382 ページ）の通りです。前章まで個々のセルは単なる int 型データに過ぎませんでしたが、この章からセルには Cell オブジェクト（の参照）が格納されます。つまり、セルは高機能化されます。

これに伴い、以下の変更があります。

まず、前章の

```
int data[][]= new int[XSIZE][YSIZE];
```

を以下へ変更します。これは Cell オブジェクトへの参照の 2 次元配列です。

```
    void print() {
        ** 中略（この部分は前章と同じ） **
                for (Actor a : game.actors) {    // ここも前章から変更なし
                    if (a.cellPrint(position)) { continue x_level; }
                }
                getAt(position).cellPrint();// 前章では cellPrint(position);
            }
            System.out.println();
        }
    }

    Int2 getRandomCellPosition() {
        while (true) {
            Int2 position = new Int2(Game.random.nextInt(XSIZE),
                                     Game.random.nextInt(YSIZE));

            if (!getAt(position).isRockCell()) { return position; }
                                                // 岩が無ければ戻り値とする
        }
    }
}
```

図 22.7: Cell のクラス群を用いるダンジョンゲームのプログラム（マップ、その 2）

```
    Cell data[][]= new Cell[XSIZE][YSIZE];            // int から Cell へ変更
```

コンストラクタ内のマップデータの初期設定では、これまで Map.ROCK を設定していたセルに new RockCell() を設定します。同じく Map.GOAL を設定していたセルに new GoalCell() を設定します。Map.EMPTY を設定していたセルには以下のプログラム部分で、new PotionCell()、new PoisonCell()、new EmptyCell() を乱数で設定します。それぞれの出現確率は 1/5、1/5、3/5 です。

```
                    switch (Game.random.nextInt(5)) {       // 乱数で選択
                    case 0: data[x][y] = new PotionCell();   // ポーション
                            break;
                    case 1: data[x][y] = new PoisonCell();     // ポイズン
                            break;
                    default: data[x][y] = new EmptyCell();     // 空のセル
                    }
```

数値 5 を小さくすれば（ただし 3 以上）、ポーションセル、ポイズンセルの出現確率が上がりますし、数値を大きくすればポーションセル、ポイズンセルの出現確率は下がります。

次にマップデータの更新を便利にするため、以下のメソッドを導入します。

```
    Cell getAt(Int2 position) { ** 前章から変更なし ** }

    void setAt(Int2 position, Cell cell) {     // セルを position の位置に設定
        data[position.x][position.y] = cell;
    }
```

getAt() は前章からの変更版です。setAt() はこの章から新規に導入するもので、ゲッター getAt() と対をなすセッターです（14.3 節（202 ページ）参照）。外部から直接 data にアクセスさせないために用意しました。

メソッド print() では、前章までセルの表示に cellPrint(position) を用いていましたが、この章からはセルオブジェクト getAt(position) にメソッド cellPrint() を適用します。この章以降、cellPrint(position) は削除します。

メソッド getRandomCellPosition() では、前章まで

```
        getAt(position) != ROCK
```

であった式を

```
        !getAt(position).isRockCell()
```

へ変更します。

```
abstract class Actor {
    Game game;
    Int2 position;
    int hp;
    char cellChar;

    Actor(Game game, Int2 position, int hp, char cellChar) {
        ** 前章から変更なし **
    }

    Actor(Game game, int hp, char cellChar) { ** 前章から変更なし ** }

    boolean cellPrint(Int2 position) { ** 前章から変更なし ** }

    boolean moveTo(Int2 direction) {
        Int2 target = position.add(direction);

        Cell cell = game.map.getAt(target);     // 移動希望先のセルを求める
        if (cell.isRockCell()) { return false; }  // セルが岩ならば移動不可

        for (Actor a : game.actors) {                 // 前章から変更なし
            if (target.equal(a.position)) { return false; }
        }

        Cell newCell = cell.affectOn(this);  // セルからこのアクターへの影響
        game.map.setAt(target, newCell);     // 新しいセルをマップに配置
        position = target;
        return true;
    }

    abstract void action();
}
```

図 22.8: Cell のクラス群を用いるダンジョンゲームのプログラム（アクター）

22.7　アクタークラスの変更

この章のアクタークラスのプログラムは図 22.8（384 ページ）の通りです。

変更箇所はメソッド moveTo() のみです。まず、以下では、移動希望先のセルオブジェクトを取得し、それが岩ならば移動しないことをチェックします。

```
        Cell cell = game.map.getAt(target);     // 移動希望先のセルを求める
        if (cell.isRockCell()) { return false; }  // セルが岩ならば移動不可
```

次に、移動希望先に他のアクターがいないことをチェックします。ここは前章から変更ありません。

そして、

```
Cell newCell = cell.affectOn(this);  // セルからこのアクターへの影響
```

によって、セルから当該アクターオブジェクトへ影響を与えます。メソッド affectOn はセルオブジェクト（の参照）を戻します。ほとんどの場合、その位置のセルが変更されることはありませんから、cell と同じオブジェクトが戻されますが、PotionCell の場合には回復薬は一回の使い切りですから、EmptyCell が戻されます。

プレイヤークラス、モンスタークラス、レッドモンスタークラスのプログラムに変更はありません。

22.8　プログラムの実行例

この章のプログラムの実行は、たとえば以下のように進みます。

```
# # # # # # # # # # # # #
# @ . . o . . . o . o . #
# . . o . . . . o . o . #
# . . . . . . . . . . . #
# . o . . # # # . . R o #
# . . . . # # # . R . . #
# . . . . # # # o o . o #
# o o . o . . . M o . . #
# o . o . . . o . . . o #
# . . . o . . . o . o M #
# # # # # # # # # # # # #
Taro : (1,1) 100
RedMonster : (10,4) 150
RedMonster : (9,5) 150
Monster : (8,7) 150
Monster : (11,9) 150

? d                          <---- 右方向をキー入力
# # # # # # # # # # # # #
# . @ . o . . . o . o . #
# . . o . . . . o . o . #
# . . . . . . . R . . #
# . o . . # # # R . . o #
# . . . . # # # . o . . #
# . . . . # # # o o . o #
# o o . o . . . M o . . #
# o . o . . . o . . M o #
# . . . o . . . o . o G #
# # # # # # # # # # # # #
Taro : (2,1) 100
RedMonster : (9,3) 140       <---- 毒で-10
RedMonster : (8,4) 150
Monster : (8,7) 150
Monster : (10,8) 140         <---- 毒で-10

? c                          <---- 右下方向をキー入力
# # # # # # # # # # # # #
# . . . o . . . o . o . #
# . . @ . . . . R . o . #
# . . . . . . R . x . . #     <---- 毒
# . o . . # # # . . . o #
# . . . . # # # . o . . #
# . . . . # # # o o . o #
# o o . o . . . M M . . #
# o . o . . . o . . x o #     <---- 毒
# . . . o . . . o . o G #
# # # # # # # # # # # # #
Taro : (3,2) 150             <---- 回復薬で 50
RedMonster : (8,2) 190       <---- 回復薬で 50
RedMonster : (7,3) 150
Monster : (8,7) 150
Monster : (9,7) 190          <---- 回復薬で 50

? e                          <---- 右上方向をキー入力
```

```
# # # # # # # # # # # # # #
# . . . @ . . R o . o . #
# . . . . . R . . . o . #
# . . . . . . . . x . . #
# . o . . # # # . . . o #
# . . . . # # # . o . . #
# . . . . # # # M o . o #
# o o . o . . . M . . . #
# o . o . . . o . . x o #
# . . . o . . . o . o G #
# # # # # # # # # # # # #
Taro : (4,1) 200            <---- 回復薬で 50
RedMonster : (7,1) 190
RedMonster : (6,2) 150
Monster : (8,7) 150
Monster : (8,6) 240         <---- 回復薬で 50


以下、同様
```

22.9　付録：この章の全プログラムリスト

```
//ゲームクラスのプログラム
import java.io.*;
import java.util.Random;
import java.util.ArrayList;

class Game {
    public static void main(String args[]) {
        Game game = new Game();
        game.mainLoop();
    }

    static Random random = new Random();

    Map map;
    ArrayList<Actor> actors;
    final static int PLAYER_ID = 0;
    final static int NUM_OF_MONSTERS = 4;

    Game() {
        map = new Map(this);
        actors = new ArrayList<Actor>();
        actors.add(new Player(this,"Taro"));
        for (int i = 0; i < NUM_OF_MONSTERS; i++) {
            if (random.nextInt(2) == 0) {
                actors.add(new Monster(this));
            }
            else {
                actors.add(new RedMonster(this));
            }
        }
    }

    void print() {
        map.print();
        for (Actor a : actors) { System.out.println(a); }
        System.out.println();
    }

    void mainLoop() {
        print();
        while (true) { turn(); }
    }

    void turn() {
        for (Actor a : actors) { a.action(); }
        print();
    }

    static char getKeyChar() {
        char c = 0;
        System.out.print("? ");
        try {
            BufferedReader in = new BufferedReader(
                                new InputStreamReader(System.in));
            c = in.readLine().charAt(0);
        } catch(IOException e){
```

```
            System.out.println("ERROR:: getKeyChar()");
        }
        return c;
    }
}
```

```
//マップクラスのプログラム
class Map {
    static final int XSIZE = 13;
    static final int YSIZE = 11;

    Game game;
    Cell data[][]= new Cell[XSIZE][YSIZE];            // int から Cell へ変更

    Map(Game game) {
        this.game = game;
        for (int x = 0; x < XSIZE; x++) {
            for (int y = 0; y < YSIZE; y++) {
                if ((x == 0)||(x == XSIZE-1)||
                   (y == 0)||(y == YSIZE-1)||
                   ((Math.abs(x-XSIZE/2) <= 1)&&
                    (Math.abs(y-YSIZE/2) <= 1))) {
                    data[x][y] = new RockCell();               // 岩
                }
                else {
                    switch (Game.random.nextInt(5)) {       // 乱数で選択
                    case 0: data[x][y] = new PotionCell();   // ポーション
                            break;
                    case 1: data[x][y] = new PoisonCell();     // ポイズン
                            break;
                    default: data[x][y] = new EmptyCell();     // 空のセル
                    }
                }
            }
        }
        data[XSIZE-2][YSIZE-2] = new GoalCell();               // ゴール
    }

    Cell getAt(Int2 position) { return data[position.x][position.y]; }

    void setAt(Int2 position, Cell cell) {     // セルを position の位置に設定
        data[position.x][position.y] = cell;
    }

    void print() {
        for (int y = 0; y < YSIZE; y++) {
            x_level:
            for (int x = 0; x < XSIZE; x++) {
                Int2 position = new Int2(x,y);

                for (Actor a : game.actors) {
                    if (a.cellPrint(position)) {
                        continue x_level;
                    }
                }
                getAt(position).cellPrint();// 前章では cellPrint(position);
            }
            System.out.println();
        }
    }

    Int2 getRandomCellPosition() {
        while (true) {
            Int2 position = new Int2(Game.random.nextInt(XSIZE),
```

```
                                        Game.random.nextInt(YSIZE));

                if (!getAt(position).isRockCell()) { return position; }
                                                   // 岩が無ければ戻り値とする
            }
        }
    }
```

```
//アクタークラスのプログラム
abstract class Actor {
    Game game;
    Int2 position;
    int hp;
    char cellChar;

    Actor(Game game, Int2 position, int hp, char cellChar) {
        this.game = game;
        this.position = position;
        this.hp = hp;
        this.cellChar = cellChar;
    }

    Actor(Game game, int hp, char cellChar) {
        this.game = game;
        this.hp = hp;
        this.cellChar = cellChar;
        while_loop:
        while (true) {
            position = game.map.getRandomCellPosition();
            for (Actor a : game.actors) {
                if (position.equal(a.position)) { continue while_loop; }
            }
            return;
        }
    }

    boolean cellPrint(Int2 position) {
        if (this.position.equal(position)) {
            System.out.print(cellChar+" ");
            return true;
        }
        return false;
    }

    boolean moveTo(Int2 direction) {
        Int2 target = position.add(direction);

        Cell cell = game.map.getAt(target);     // 移動希望先のセルを求める
        if (cell.isRockCell()) { return false; }  // セルが岩ならば移動不可

        for (Actor a : game.actors) {
            if (target.equal(a.position)) { return false; }
        }

        Cell newCell = cell.affectOn(this);  // セルからこのアクターへの影響
        game.map.setAt(target, newCell);     // 新しいセルをマップに配置
        position = target;
        return true;
    }

    abstract void action();
}
```

```
//プレイヤークラスのプログラム
class Player extends Actor {
    static final int INIT_HP = 100;

    String name;

    Player(Game game, String name) {
        super(game, new Int2(1,1), INIT_HP, '@');
        this.name = name;
    }

    Player(Game game) { this(game, "Player"); }

    public String toString() {
        return name+" : "+position+" "+hp;
    }

    void attackTo(Int2 direction) {
        Int2 target = position.add(direction);

        for (Actor a : game.actors) {
            if (target.equal(a.position)) {
                if (a instanceof RedMonster) {
                    a.hp -= 10;
                    return;
                }
                else if (a instanceof Monster) {
                    a.hp -= 20;
                    return;
                }
            }
        }
    }

    @Override
    void action() {
        Int2 direction;
        switch (Game.getKeyChar()) {
            case 'z': direction = new Int2(-1,+1); break;
            case 'x': direction = new Int2( 0,+1); break;
            case 'c': direction = new Int2(+1,+1); break;
            case 'a': direction = new Int2(-1, 0); break;
            case 'd': direction = new Int2(+1, 0); break;
            case 'q': direction = new Int2(-1,-1); break;
            case 'w': direction = new Int2( 0,-1); break;
            case 'e': direction = new Int2(+1,-1); break;
            default:  direction = new Int2( 0, 0);
        }

        if (!moveTo(direction)) { attackTo(direction); }
    }
}
```

```
//モンスタークラスのプログラム
class Monster extends Actor {
    static final int INIT_HP = 150;

    Monster(Game game, char c) { super(game, INIT_HP, c); }

    Monster(Game game) { this(game, 'M'); }

    @Override
    public String toString() {
        return "Monster : "+position+" "+hp;
    }

    void attackTo(Int2 direction) {
        Int2 target = position.add(direction);

        Actor player = game.actors.get(game.PLAYER_ID);
        if (target.equal(player.position)) {
            player.hp -= 30;
            return;
        }
    }

    @Override
    void action() {
        Actor player = game.actors.get(game.PLAYER_ID);
        Int2 direction = player.position.sub(position).signum();

        if (!moveTo(direction)) { attackTo(direction); }
    }
}
```

```
//レッドモンスタークラスのプログラム
class RedMonster extends Monster {
    RedMonster(Game game) { super(game, 'R'); }

    @Override
    public String toString() {
        return "Red"+super.toString();
    }

    @Override
    void attackTo(Int2 direction) {
        Int2 target = position.add(direction);

        Actor player = game.actors.get(game.PLAYER_ID);
        if (target.equal(player.position)) {
            player.hp -= 60;
            return;
        }
    }
}
```

```
//2 次元ベクトルクラスのプログラム
class Int2 {
    int x,y;

    Int2(int x, int y) { this.x = x; this.y = y; }
    Int2 add(Int2 a) { return new Int2(x+a.x,y+a.y); }
    Int2 sub(Int2 a) { return new Int2(x-a.x,y-a.y); }
    Int2 signum() { return new Int2(Integer.signum(x),Integer.signum(y)); }
    boolean equal(Int2 a) { return (x == a.x)&&(y == a.y); }

    public String toString(){ return "("+x+","+y+")"; }
}
```

```
//セルクラスのプログラム
abstract class Cell {
    abstract void cellPrint();              // マップの表示に用いる

    boolean isRockCell() { return false; } // デフォルトでは岩ではない

    abstract Cell affectOn(Actor a);        // アクターへの影響を記述する
}

abstract class SimpleCell extends Cell {
    @Override
    Cell affectOn(Actor a) { return this; }
                                        // アクターへ影響せず、セルを置換しない
}

abstract class TrapCell extends Cell {
    boolean visible = false;             // 初期値では不可視

    @Override
    Cell affectOn(Actor a) {
        visible = true;         // アクターがこのセルを踏んだならば、可視へ変更
        return this;            // セルを置換しない
    }
}

class RockCell extends SimpleCell {             // 岩のセル
    @Override
    void cellPrint() { System.out.print("# "); } // # で表示することを実装

    @Override
    boolean isRockCell() { return true; }        // デフォルトの実装を上書き
}

class EmptyCell extends SimpleCell {            // 空（から）のセル
    @Override
    void cellPrint() { System.out.print(". "); } // . で表示することを実装
}

class PotionCell extends SimpleCell {          // ポーションの置かれているセル
    @Override
    void cellPrint() { System.out.print("o "); } // o で表示することを実装

    @Override
    Cell affectOn(Actor a) {
        a.hp += 50;                              // アクターの HP を 50 増やす
        return new EmptyCell();                  // 空のセルに置き換わる
    }
}

class GoalCell extends SimpleCell {             // ゴールセル
    @Override
    void cellPrint() { System.out.print("G "); } // G で表示することを実装

    @Override
    Cell affectOn(Actor a) {
        if (a instanceof Player) {               // プレイヤーならば
            System.out.println("YOU WIN");       // YOU WIN を表示して
            System.exit(0);                      // ゲームを終了
```

```
        }
        return this;
    }
}

class PoisonCell extends TrapCell {            // 毒のあるセル
    @Override
    void cellPrint() {
        // 可視ならば x、不可視ならば . で表示
        if (visible) { System.out.print("x "); }
        else { System.out.print(". "); }
    }

    @Override
    Cell affectOn(Actor a) {
        a.hp -= 10;                            // アクターの HP を 10 減らす
        return super.affectOn(a);              // 親クラスのメソッドを実行
    }
}
```

1

1

第23章　クラスに関する話題

この章ではダンジョンプログラムを改造しません。オブジェクト指向に関連する話題 — ここまでの解説で「後章で述べる」と残しておいた内容 — を紹介します。

23.1　has-a 関係と is-a 関係

オブジェクト指向を学んでもすぐには自力でクラスの階層を設計できそうにありません。その設計のヒントとして以下がよく知られています。

これから作ろうとするプログラムに関係する概念が、A、B、... といくつか頭に浮かんできたとしましょう。

もし二つの概念 A と B の間に

- 「A は B を持つ（A has a B）」が成り立つならば、A のオブジェクトは B をインスタンス変数（メンバー変数）として持つようにプログラム化する

と、比較的素直にプログラム化できます。

たとえば、「プレイヤーは HP を（属性として）持つ」が成り立つと考え、5 章において `Player` クラスにメンバー変数 `hp` を持たせました。同じく「マップはゲーム（の情報）を持つ」が成り立つと考え、9 章では `Map` オブジェクトにメンバー変数 `game` を持たせました。身近な例では、「自動車はエンジンを持つ」が成り立つと考え、自動車オブジェクトにエンジンに相当するインスタンス変数を持たせることなども素直な発想です。

この「... は ... を持つ（... has a ...）」を **has-a 関係**（has-a relationship）と呼びます。

もうひとつのヒントは、もし二つの概念 A と B の間に

- 「A は B である（A is a B）」、「A は B の特殊形である」、「A は B の拡張である」などが成り立つならば、A クラスは B クラスのサブクラスであるようにプログラム化する

と、これも比較的素直にプログラム化できます。

たとえば、「プレイヤーはアクターである」が成り立つと考え、15 章において Player クラスを Actor クラスのサブクラスとしました。また、「ポイズンセルは罠セルであり、罠セルはセルである」が成り立つと考え、前章では PoisonCell は TrapCell のサブクラスとし、TrapCell は Cell のサブクラスとしました。身近な例だと、「バスは自動車である」が成り立つと考え、バスクラスを自動車クラスのサブクラスにすることなどもありそうです。「猫は哺乳類である」もその例です。

この「... は ... である（... is a ...）」を **is-a 関係**（is-a relationship）と呼びます。

has-a 関係と is-a 関係はクラス階層設計の分かりやすいヒントなのですが、プログラムが常にこれに従う訳ではありません。

たとえば「バスは自動車である」という表明を「バスは自動車（としての性質）を持つ」と無理やり解釈すれば、バスは自動車をインスタンス変数（メンバー変数）として持つようなプログラミングも可能です。これは、is-a 関係を無理やり has-a 関係へ置き直したものですが、逆も可能です。「自動車はエンジンを持つ」という表明を「自動車はエンジン（にその他のパーツを加えて拡張したもの）である」と解釈すれば、has-a 関係は is-a 関係に置換されます。

この講義テキストの「罠セル（TrapCell）はセル（Cell）である」について具体例を示します。以下は、前章のこの二つのクラスの関連箇所のプログラムです。

```
abstract class Cell {
    abstract void cellPrint();

    boolean isRockCell() { return false; }

    abstract Cell affectOn(Actor a);
}

abstract class TrapCell extends Cell {
    boolean visible = false;

    @Override
    Cell affectOn(Actor a) {
        visible = true;
        return this;
    }
}
```

これを、「罠セル（TrapCell）はセル（Cell）（としての性質）を持つ」の表明に合うように改造します。以下では、TrapCell クラスが、Cell クラスのインスタンス変数 cell を持つことに注意してください。

```
class Cell {
    boolean isRockCell() { return false; }
}

abstract class TrapCell {
    Cell cell = new Cell();                  // Cell オブジェクトの追加
    boolean visible = false;

    abstract void cellPrint();  // 元々Cell にあった抽象メソッドを移動

    boolean isRockCell() { return cell.isRockCell(); } // 処理を cell に任せる

    Cell affectOn(Actor a) {
        visible = true;
        return this;
    }
}
```

やはり前者のプログラムが読みやすいのですが、後者のプログラムが全く読みにくいというほどでもありません。実は後者のプログラムは、**委譲**（delegate、デリゲート）と呼ばれるプログラミング技法に関連します[1]。興味のある人は以下などを参考にしてください。

Java による委譲によるクラス継承と Adapter パターン例

https://tomosoft.jp/design/?p=3444

クラス階層をどうのように設計するべきか、これは非常に難しい問題であって、プログラマには広範な知識、経験、深い洞察力が必要とされます。講義テキストの 1 章にも述べましたが、この講義ではそれを論じていく時間はありません。

[1] 正確に言えば、「関係する」だけであり、委譲そのものではありません。この辺も「誤解が多い」としばしばネット上で熱く語られています。

23.2　多重継承

クラス継承では、親クラスの性質を子クラスが受け継ぎます。であるならば、複数の親の性質を受け継ぐことで多様な性質を集積させたクラスを生成できると期待されます。これを**多重継承**（multiple inheritance）と呼びます。

よく利用される例として「水陸両用車」があります[2]。これについて小さな例題プログラムを作ってみます。ここに travelTime は、東京から大阪までの所要時間のつもりです。

```
abstract class  Vehicle {                // 乗り物の抽象クラス
    abstract double travelTime();        // 東京から大阪までの所要時間
}

class Car extends Vehicle {              // 自動車
    double travelTime() { return 6.0; }  // 6 時間
}

class Ship extends Vehicle {             // 船
    double travelTime() { return 24.0; } // 24 時間
}

class AmphibiousVehicle extends Car, Ship { //水陸両用車
}
```

さて、上の定義の元、以下を実行したときに所用時間はどう求められるべきでしょうか。

```
    ...
    Vehicle v = new AmphibiousVehicle();
    System.out.println(v.travelTime());
    ...
```

Car、Ship 両方にメソッド travelTime() が定義されているところが問題です。これは Java だけの問題ではなく、オブジェクト指向そのものの持つ継承に関する大きな問題であり、様々なプログラミング言語でそれぞれの解決法が提案されています。

Java における解決法は「多重継承は言語仕様として導入しない」という割り切りです。ですから、上の

[2]水陸両用車が多重継承の例として相応しく無いという意見は多々あり、たとえば以下を参照してください。継承そのものについても深く論じてあります。

継承にかかわる諸問題
https://iwatam-server.sakura.ne.jp/software/devintro/inheritance/x85.html

```
class AmphibiousVehicle extends Car, Ship { //水陸両用車
```

の多重継承の部分は Java ではコンパイルエラーになります。多重継承はプログラミング言語の研究としては面白いテーマなのですが、実用上はそこに注力する価値が低いため、Java はあっさり放棄した訳です。ただし、多重継承の代替策としてクラスの継承とインターフェース（21.2 節（358 ページ）参照）の実装を併用する方法が提案されています。Java のこの方向性（適切な制限と代替策の提示）は多くのプログラマに支持されています。上の水陸両用車の例で言えば、以下のような代替プログラムが可能です。

```
abstract class  Vehicle {                // 乗り物の抽象クラス
    abstract double travelTime();        // 東京から大阪までの所要時間
}

class Car extends Vehicle {              // 自動車
    double travelTime() { return 6.0; }  // 6 時間
}

interface Ship {                         // 船
    double travelTimeByShip();           // 船での所要時間
}

class AmphibiousVehicle extends Car implements Ship { //水陸両用車
    public double travelTimeByShip() { return 24.0; } // 船では 24 時間
}

    ...
    AmphibiousVehicle av = new AmphibiousVehicle();
    System.out.println(av.travelTime());        // 自動車としての所要時間
    System.out.println(av.travelTimeByShip());  // 船としての所要時間
    ...
```

ところで、C++ は多重継承を真面目に言語仕様に組み込んでいます。しかし案の定、複雑な仕様になっています。オブジェクト指向の入門としてそれを取り上げる余裕はないため、このテキストでは解説を省略します。

表 23.1: プリミティブ型とそのラッパークラス

プリミティブ型	ラッパークラス
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

23.3　ラッパークラス

整数値 1、浮動小数点数値 3.1415926e00 などのプリミティブ型データをオブジェクトのように扱うことはできません。たとえば 20.1 節（323 ページ）の `ArrayList<T>` の要素はオブジェクト型に限定されています。そうなるとプリミティブ型は可変長配列の恩恵に預かることはできません。

この問題を解決するために、プリミティブ型データをそれと同じ情報量を持つオブジェクトへ変換する手法がしばしば用いられます。たとえば以下のクラス定義は `int` をオブジェクト型へ変換するクラス定義です。

```
class MyInt {
    int value;

    MyInt(int value) {
        this.value = value;
    }

    int getValue() { return value; }
}
```

このようなクラスを**ラッパークラス**（wrapper class）と呼びます。wrap は「包む」という意味です。`int` をオブジェクトの中に包むからです。一般に `int` を包むだけがラッパークラスではなく

- そのまま利用するには少し障害があるようなデータ、オブジェクトに一皮被せて使いやすくする

ようなクラスは全てラッパークラスであり、プログラミングの様々な局面で利用されています。

ところで、数値のラッパークラスが必要であることは Java のプログラミング経験者ならば誰で

もすぐに気づくことですから、Java ではそれらが標準装備されています。表 23.1 がその一覧です。一般に Java のラッパークラスと言えば、表 23.1 のクラス群を指すことがほとんどです。

たとえば表中の Integer クラスは本質的には上の MyInt と同等ですが、実用に便利なインスタンスメソッド、クラスメソッドが満載されています。以下のプログラムは Integer クラス の利用例です。

```
class Test {
    public static void main(String[] strs) {
        Integer w = Integer.valueOf(123); // int から Integer を作成

        int i = w.intValue();             // Integer から int を作成

        System.out.println(i);
    }
}
```

実は上と同等のことは以下のプログラムでも可能です。

```
class Test {
    public static void main(String[] strs) {
        Integer w = 123;                  // int から Integer を作成

        int i = w;                        // Integer から int を作成

        System.out.println(i);
    }
}
```

文 Integer w = 123; の左辺は Integer、右辺は int ですから、本来は代入できないのですが、このような状況では Java は int を Integer へ自動変換します。これを**オートボクシング**（auto-boxing）と呼びます。自動（オート）でボックス化する（int データをオブジェクトの箱に入れる）という意味です。文 int i = w; はその逆です。これを**オートアンボクシング**（auto-unboxing）と呼びます[3]。自動（オート）でボックス化を解除する（int データをオブジェクトの箱から出す）という意味です。オートボクシング/オートアンボクシングを活用すれば、int をオブジェクトのように扱う/Integer オブジェクトを int のように扱うことができます。

ワッパークラスの機能の詳細は省略します。興味のある人は、「java ラッパークラス」をネット検索してください。

[3]「オートアンボクシング」はしばしば「アンボクシング」と短く呼ばれますから注意してください。

1

1

第24章　攻撃処理をエレガントにプログラム化する：ダブルディスパッチ法の利用

20.7.1 項（339 ページ）の以下のプログラム片を思い出しましょう。

```
if (a instanceof RedMonster) {     // a が RedMonster ならば
    a.hp -= 10;                     // hp を 10 減らす
    return;
}
else if (a instanceof Monster) {   // a が Monster ならば
    a.hp -= 20;                     // hp を 20 減らす
    return;
}
```

instanceof 演算子を用いてオブジェクトのクラスを判定し、攻撃を選別する処理が作られていました。しかし、オブジェクトが属するクラスを判定しながら処理を進める方法は、オブジェクト指向に相応しくないプログラムと考えられます。オブジェクト指向では多相性を使えば instanceof に頼らないエレガントな処理ができるからです。

この章では上の問題を**ダブルディスパッチ法**（double dispatch method）で解決します。「ディスパッチ」とは辞書では「送り出す」「分配する」などの意味ですが、オブジェクト指向プログラミングでは多相性を利用して実際に実行すべきメソッドを特定することをディスパッチ（あるいはシングルディスパッチ）と言います。ここではそれを二重（ダブル）に行います。

オブジェクト指向のプログラムパターンを類型化する**デザインパターン**（Design Pattern、たとえば Wikipedia：デザインパターン（ソフトウェア）を参照）では、この章の方法を **ビジターパターン**（visitor pattern）と呼んでいます。この章の内容は、オブジェクト指向の特徴的な利用法のひとつの紹介にもなっています。

24.1　簡単な例題：ジャンケンのプログラム

グー（Goo）、チョキ（Choki）、パー（Par）のオブジェクトを作り、さらにオブジェクト間の勝ち負けを判定するプログラムを作ってみましょう。

```
abstract class Janken {
    public static void main(String[] str) {
        Janken j1 = new Goo();                  // グーオブジェクト
        Janken j2 = new Choki();                // チョキオブジェクト
        System.out.println(j1.win(j2));         // グーはチョキに勝つ？
    }

    String win(Janken a) {              // 勝ちを判定するメソッド
        if (this instanceof Goo) {
            if (a instanceof Goo) { return "EVEN"; }       // グーとあいこ
            else if (a instanceof Choki) { return "YES"; } // チョキに勝つ
            else /* Par */ { return "NO"; }                // パーに勝てない
        }
        else if (this instanceof Choki) {
            if (a instanceof Goo) { return "NO"; }          // グーに勝てない
            else if (a instanceof Choki) { return "EVEN"; } // チョキとあいこ
            else /* Par */ { return "YES"; }                // パーに勝つ
        }
        else /* Par */ {
            if (a instanceof Goo) { return "YES"; }        // グーに勝つ
            else if (a instanceof Choki) { return "NO"; }  // チョキに勝てない
            else /* Par */ { return "EVEN"; }              // パーとあいこ
        }
    }
}

class Goo extends Janken { }                      // グークラス

class Choki extends Janken { }                    // チョキクラス

class Par extends Janken { }                      // パークラス
```

図 24.1: チョキ、パー、パーによるジャンケンプログラム（素朴な実装）

24.1.1　素朴なプログラム

図 24.1（411 ページ）は instanceof 演算子を多用する素朴なプログラムです。

メソッド win() の実装では、まず this（メソッドの主体）のクラスを instanceof で判別し、次に引数 a のクラスを判別し、勝ち負けの結果を返答しています。多相性を活用しない典型的なプログラムです。なお、グー、チョキ、パーのクラスは単にそれらがジャンケンクラスのサブクラスとして宣言されていることだけが重要で、グー、チョキ、パーのクラスに中身はありません。もちろんジャンケンというオブジェクトは存在しませんから、ジャンケンクラスは抽象クラスです。

24.1.2　シングルディスパッチを用いるプログラム

図 24.2（413 ページ）は 2 番目のプログラムです。このプログラムでは、Janken クラスのメソッド win() はグー、チョキ、パーの各クラスから上書きされています。よって、メソッド呼び出し j1.win(j2) において、オブジェクト j1 について実際に実行されるメソッドは多相性によってグー、チョキ、パーの中のひとつに特定できます。しかし j2 のクラスは不明です[1]から、各クラスの win() の中では instanceof を用いて引数 a（＝ j2）のクラスを判別し、勝ち負けの結果を返答しています。

このプログラムでは j1 についてのみシングルディスパッチ法を用いるプログラムです。

[1] もちろんプログラム全体を眺めている私たちは、Janken j2 = new Choki(); が実行されることをを知っているので、j2 がチョキクラスのオブジェクトであることも把握しています。しかし、Java のコンパイラはプログラムを局所的にしか解析できないため、j2 のクラスを静的には知ることができません。

```
abstract class Janken {
    public static void main(String[] str) {
        Janken j1 = new Goo();                  // グーオブジェクト
        Janken j2 = new Choki();                // チョキオブジェクト
        System.out.println(j1.win(j2));         // グーはチョキに勝つ？
    }

    abstract String win(Janken a);              // 勝ちを判定するメソッド
}

class Goo extends Janken {                      // グークラス
    @Override
    String win(Janken a) {
        if (a instanceof Goo) { return "EVEN"; }       // グーとあいこ
        else if (a instanceof Choki) { return "YES"; } // チョキに勝つ
        else /* Par */ { return "NO"; }               // パーに勝てない
    }
}

class Choki extends Janken {                    // チョキクラス
    @Override
    String win(Janken a) {
        if (a instanceof Goo) { return "NO"; }          // グーに勝てない
        else if (a instanceof Choki) { return "EVEN"; } // チョキとあいこ
        else /* Par */ { return "YES"; }               // パーに勝つ
    }
}

class Par extends Janken {                     // パークラス
    @Override
    String win(Janken a) {
        if (a instanceof Goo) { return "YES"; }         // グーに勝つ
        else if (a instanceof Choki) { return "NO"; }   // チョキに勝てない
        else /* Par */ { return "EVEN"; }              // パーとあいこ
    }
}
```

図 24.2: チョキ、パー、パーによるジャンケンプログラム（シングルディスパッチによる実装）

24.1.3　ダブルディスパッチ法を用いたプログラム

図 24.3（415 ページ）はダブルディスパッチによるプログラムです。Janken クラスのメソッド win() はグー、チョキ、パーの各クラスから上書きされている点は 2 番目のプログラムと同じです。よって、メソッド呼び出し j1.win(j2) において、実際に実行されるメソッドは多相性によってグー、チョキ、パーの中のひとつに特定できます。

図 24.3 の場合、j1 はグーですから、グークラスの win() が実行されます。そしてその本体では以下の通り、

```
String win(Janken a) { return a.lose(this); } // 勝ち判定を負け判定に変換
```

主体のオブジェクトと引数のオブジェクトを入れ替えて、今度はメソッド lose() を呼びます。ここで、メソッド lose() はジャンケンクラスにおいて 3 種類が多重定義されていることに注目してください。

```
abstract String lose(Goo b);          // グーとの負けを判定するメソッド
abstract String lose(Choki b);        // チョキとの負けを判定するメソッド
abstract String lose(Par b);          // パーとの負けを判定するメソッド
```

メソッド呼び出し a.lose(this) の引数 this のクラスは、この場合、グークラスと確定していますから、そうすると、lose(Goo b) が呼び出されます。

さて、lose(Goo b) は上書きされていますから、メソッド呼び出し a.lose(this) の主体オブジェクト a のクラスに応じて呼び出すメソッドのクラスが自動的に決まります。この場合、a のクラスはチョキですから、そうすると、メソッド実行 a.lose(this) では、チョキクラスの

```
@Override String lose(Goo b) { return "YES"; }        // グーに負ける
```

が呼び出され、答えが確定します。

ポイントは、2 回のディスパッチでオブジェクトのクラスを確定している点です。そして、1 回目で確定したクラスの情報は多重定義の仕組みを利用して次の動作に利用されている点です。結果、図 24.3 のプログラムには if 文も instanceof もありません。

```
abstract class Janken {
    public static void main(String[] str) {
        Janken j1 = new Goo();                  // グーオブジェクト
        Janken j2 = new Choki();                // チョキオブジェクト
        System.out.println(j1.win(j2));         // グーはチョキに勝つ？
    }

    abstract String win(Janken a);              // 勝ちを判定するメソッド

    abstract String lose(Goo b);         // グーとの負けを判定するメソッド
    abstract String lose(Choki b);       // チョキとの負けを判定するメソッド
    abstract String lose(Par b);         // パーとの負けを判定するメソッド
}

class Goo extends Janken {                      // グークラス
    @Override
    String win(Janken a) { return a.lose(this); } // 勝ち判定を負け判定に変換

    @Override String lose(Goo b) { return "EVEN"; }       // グーとあいこ
    @Override String lose(Choki b) { return "NO"; }       // チョキに負けない
    @Override String lose(Par b) { return "YES"; }        // パーに負ける
}

class Choki extends Janken {                    // チョキクラス
    @Override
    String win(Janken a) { return a.lose(this); } // 勝ち判定を負け判定に変換

    @Override String lose(Goo b) { return "YES"; }        // グーに負ける
    @Override String lose(Choki b) { return "EVEN"; }     // チョキとあいこ
    @Override String lose(Par b) { return "NO"; }         // パーに負けない
}

class Par extends Janken {                      // パークラス
    @Override
    String win(Janken a) { return a.lose(this); } // 勝ち判定を負け判定に変換

    @Override String lose(Goo b) { return "NO"; }         // グーに負けない
    @Override String lose(Choki b) { return "YES"; }      // チョキに負ける
    @Override String lose(Par b) { return "EVEN"; }       // パーとあいこ
}
```

図 24.3: チョキ、パー、パーによるジャンケンプログラム（ダブルディスパッチによる実装）

24.2　ゲームクラス、マップクラスの変更

ゲームプログラムの本章の変更は攻撃部分のみのため、ゲームクラス、マップクラスの変更はありません。

24.3　アクターとそのサブクラスのクラス構造

アクター以下のクラスのプログラムを示す前に、図 24.4（417 ページ）にこの章のアクター、プレイヤー、モンスター、レッドモンスターのクラス関係を示します。変更されるメソッド、新規導入されるメソッドは赤枠のものです。

この章ではダブルディスパッチを実現するために、二つのメソッドを新規導入します。今、オブジェクト a がオブジェクト b を攻撃するとき、ここではまずメソッド呼び出し：

```
b.takeDamageFrom(a); // b は a からダメージを受ける
```

を行います。そしてこのメソッドの中で、a と b を逆転させたメソッド呼び出し：

```
a.giveDamageTo(b); // a は b へダメージを与える。
```

を行います。これによって、a と b のクラスが確定するため、このメソッドの中でオブジェクト b の HP を指定値だけ減らします。以上の変更によってメソッド attackTo() の内容はアクターの全てのクラスのオブジェクトで同一となるため、メソッド attackTo() はアクタークラスへ移動します。

以下、具体的にコードを見ていきましょう。

24.4　アクタークラスの変更

図 24.5（418 ページ）は、変更されたアクタークラスのプログラムです。このメソッドでは、

1. まず、ターゲットの位置 target を求め、
2. 各アクター a について
   (a) a がターゲットの位置にいるならば
   (b) a は this からダメージを受け、

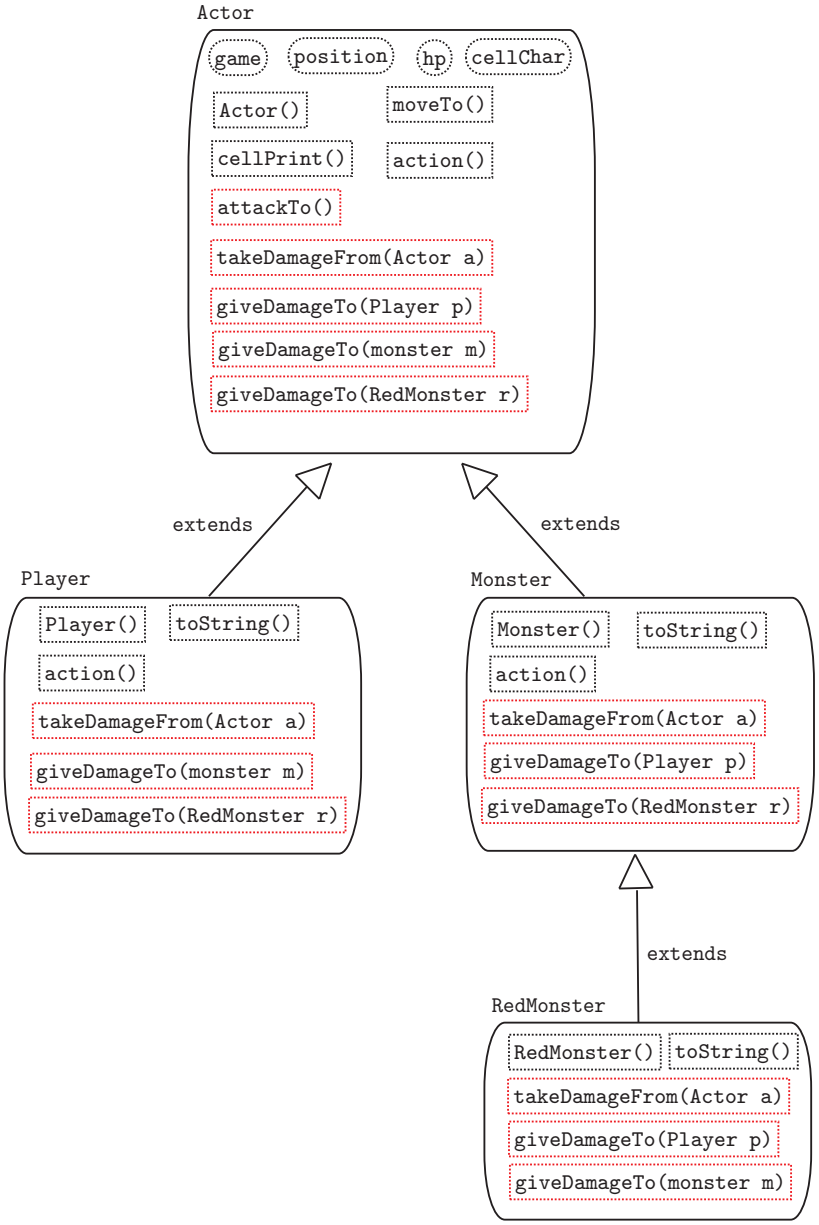


図 24.4: この章のアクター以下のクラス構造：

```
abstract class Actor {
    ** インスタンス変数に変更なし **

    Actor(Game game, Int2 position, int hp, char cellChar) {
        ** 前章から変更なし **
    }

    Actor(Game game, int hp, char cellChar) { ** 前章から変更なし ** }

    boolean cellPrint(Int2 position) { ** 前章から変更なし ** }

    boolean moveTo(Int2 direction) { ** 前章から変更なし ** }

    abstract void action();

    void attackTo(Int2 direction) {                // サブクラスから移動
        Int2 target = position.add(direction);     // ターゲットの位置の計算
        for (Actor a : game.actors) {   // それぞれのアクターについて
            if (target.equal(a.position)) {// ターゲットの位置に a がいるならば
                a.takeDamageFrom(this);      // a が this からダメージを受ける
                return;                      // メソッドを脱出
            }
        }
    }

    abstract void takeDamageFrom(Actor a);   // ダメージを受ける

    void giveDamageTo(Player p) {}           // Player にダメージを与える
    void giveDamageTo(Monster m) {}          // Monster にダメージを与える
    void giveDamageTo(RedMonster r) {}       // Redmonster にダメージを与える
}
```

図 24.5: 攻撃をダブルディスパッチに変更したダンジョンゲームのプログラム（アクター）

(c) メソッドを終了（脱出）します。繰り返しはこの時点で中断します

処理のポイントは、以下のメソッドで a にダメージを与える点です。

```
a.takeDamageFrom(this);      // a が this からダメージを受ける
```

なお、メソッド takeDamageFrom() はアクタークラスには実装されておらず、サブクラスにおいて主体と引数を入れ替えて giveDamageTo() を呼び出します。サブクラスにおいて giveDamageTo() が上書きされていない場合、以下のいずれかのメソッドを実行し、結局、ダメージを与えることはありません。

```
    void giveDamageTo(Player p) {}          // Player にダメージを与える
    void giveDamageTo(Monster m) {}         // Monster にダメージを与える
    void giveDamageTo(RedMonster r) {}      // Redmonster にダメージを与える
```

上書きによって所定のダメージを与えます。

```
class Player extends Actor {
    ** インスタンス変数などは前章から変更なし **

    Player(Game game, String name) { ** 前章から変更なし ** }

    Player(Game game) { ** 前章から変更なし ** }

    public String toString() { ** 前章から変更なし ** }

    @Override
    void takeDamageFrom(Actor a) { a.giveDamageTo(this); }
                                        // オブジェクトと引数を逆転させる

    @Override
    void giveDamageTo(Monster m) { m.hp -= 20; } // モンスターへ 20 のダメージ

    @Override
    void giveDamageTo(RedMonster r) { r.hp -= 15; } // レッドモンスターへは 15

    @Override
    void action() { ** 前章から変更なし ** }
}
```

図 24.6: 攻撃をダブルディスパッチに変更したダンジョンゲームのプログラム（プレイヤー）

24.5　プレイヤークラスの変更

図 24.5（418 ページ）が、この章のプレイヤーのプログラムです。attackTo() が無くなったため、かなりすっきりしました。

以下はダブルディスパッチのために必須です。

```
    @Override
    void takeDamageFrom(Actor a) { a.giveDamageTo(this); }
                                        // オブジェクトと引数を逆転させる
```

このメソッドが実行された場合、このメソッドは Player クラスに属するため、this が Player オブジェクトであることは明らかです。よって、メソッド呼び出し：a.giveDamageTo(this) ではメソッド： giveDamageTo(Player p) が呼び出されます。もし a がモンスターならば、多相性によって、モンスタークラスの giveDamageTo(Player p) が適切に呼び出されます。

```
class Monster extends Actor {
    ** 定数宣言は前章から変更なし **

    Monster(Game game, char c) { ** 前章から変更なし ** }

    Monster(Game game) { ** 前章から変更なし ** }

    @Override
    public String toString() { ** 前章から変更なし ** }

    @Override
    void takeDamageFrom(Actor a) { a.giveDamageTo(this); }

    @Override
    void giveDamageTo(Player p) { p.hp -= 30; } // プレイヤーへ 30 のダメージ

    @Override
    void giveDamageTo(RedMonster r) { r.hp -= 5; } // レッドモンスターへは 5

    @Override
    void action() { ** 前章から変更なし ** }
}
```

図 24.7: 攻撃をダブルディスパッチに変更したダンジョンゲームのプログラム（モンスター）

24.6　モンスタークラスの変更

図 24.7（421 ページ）がモンスタークラスのプログラムです。基本的に、プレイヤークラスのプログラムの変更と同じです。

なお、プレイヤークラスにあった giveDamageTo(Monster m) がモンスタークラスに無いのは、ここでは、モンスターがモンスターを偶然攻撃してしまった場合のダメージは 0 と考えるからです。なお、レッドモンスターを攻撃した場合には 5 だけダメージを与えることができる設定にしました（この辺、いい加減です）。

```
class RedMonster extends Monster {
    RedMonster(Game game) { ** 前章から変更なし ** }

    @Override
    void takeDamageFrom(Actor a) { a.giveDamageTo(this); }

    @Override
    void giveDamageTo(Player p) { p.hp -= 60; } // プレイヤーへ 60 のダメージ

    @Override
    void giveDamageTo(Monster m) { m.hp -= 7; } // モンスターへは 7 のダメージ
}
```

図 24.8: 攻撃をダブルディスパッチに変更したダンジョンゲームのプログラム（レッドモンスター）

24.7　レッドモンスタークラスの変更

レッドモンスタークラスのプログラム（図 24.8（422 ページ）参照）についても同様です。

ダブルディスパッチ法の利点は、if 文、instanceof に頼らず、場合分けができることです。しかもプログラムの拡張は容易です。たとえば新たにブルーモンスターを導入する場合には、アクタークラスに

```
    void giveDamageTo(BlueMonster b) {}      // BlueMonster にダメージを与える
```

を追加定義します。そしてそのサブクラスには必要に応じて、このメソッドを上書きすればよいのです。

24.8　付録：この章の全プログラムリスト

```
//ゲームクラスのプログラム
import java.io.*;
import java.util.Random;
import java.util.ArrayList;

class Game {
    public static void main(String args[]) {
        Game game = new Game();
        game.mainLoop();
    }

    static Random random = new Random();

    Map map;
    ArrayList<Actor> actors;
    final static int PLAYER_ID = 0;
    final static int NUM_OF_MONSTERS = 4;

    Game() {
        map = new Map(this);
        actors = new ArrayList<Actor>();
        actors.add(new Player(this,"Taro"));
        for (int i = 0; i < NUM_OF_MONSTERS; i++) {
            if (random.nextInt(2) == 0) {
                actors.add(new Monster(this));
            }
            else {
                actors.add(new RedMonster(this));
            }
        }
    }

    void print() {
        map.print();
        for (Actor a : actors) { System.out.println(a); }
        System.out.println();
    }

    void mainLoop() {
        print();
        while (true) { turn(); }
    }

    void turn() {
        for (Actor a : actors) { a.action(); }
        print();
    }

    static char getKeyChar() {
        char c = 0;
        System.out.print("? ");
        try {
            BufferedReader in = new BufferedReader(
                                new InputStreamReader(System.in));
            c = in.readLine().charAt(0);
        } catch(IOException e){
```

```
            System.out.println("ERROR:: getKeyChar()");
        }
        return c;
    }
}
```

```
//マップクラスのプログラム
class Map {
    static final int XSIZE = 13;
    static final int YSIZE = 11;

    Game game;
    Cell data[][]= new Cell[XSIZE][YSIZE];

    Map(Game game) {
        this.game = game;
        for (int x = 0; x < XSIZE; x++) {
            for (int y = 0; y < YSIZE; y++) {
                if ((x == 0)||(x == XSIZE-1)||
                   (y == 0)||(y == YSIZE-1)||
                   ((Math.abs(x-XSIZE/2) <= 1)&&
                    (Math.abs(y-YSIZE/2) <= 1))) {
                    data[x][y] = new RockCell();
                }
                else {
                    switch (Game.random.nextInt(5)) {
                    case 0: data[x][y] = new PotionCell();
                            break;
                    case 1: data[x][y] = new PoisonCell();
                            break;
                    default: data[x][y] = new EmptyCell();
                    }
                }
            }
        }
        data[XSIZE-2][YSIZE-2] = new GoalCell();
    }

    Cell getAt(Int2 position) { return data[position.x][position.y]; }

    void setAt(Int2 position, Cell cell) {
        data[position.x][position.y] = cell;
    }

    void print() {
        for (int y = 0; y < YSIZE; y++) {
            x_level:
            for (int x = 0; x < XSIZE; x++) {
                Int2 position = new Int2(x,y);

                for (Actor a : game.actors) {
                    if (a.cellPrint(position)) {
                        continue x_level;
                    }
                }
                getAt(position).cellPrint();
            }
            System.out.println();
        }
    }

    Int2 getRandomCellPosition() {
        while (true) {
            Int2 position = new Int2(Game.random.nextInt(XSIZE), Game.random.nextInt(YSIZE));
```

```
1
2               if (!getAt(position).isRockCell()) { return position; }
3           }
4       }
5   }
```

```
//アクタークラスのプログラム
abstract class Actor {
    Game game;
    Int2 position;
    int hp;
    char cellChar;

    Actor(Game game, Int2 position, int hp, char cellChar) {
        this.game = game;
        this.position = position;
        this.hp = hp;
        this.cellChar = cellChar;
    }

    Actor(Game game, int hp, char cellChar) {
        this.game = game;
        this.hp = hp;
        this.cellChar = cellChar;
        while_loop:
        while (true) {
            position = game.map.getRandomCellPosition();
            for (Actor a : game.actors) {
                if (position.equal(a.position)) { continue while_loop; }
            }
            return;
        }
    }

    boolean cellPrint(Int2 position) {
        if (this.position.equal(position)) {
            System.out.print(cellChar+" ");
            return true;
        }
        return false;
    }

    boolean moveTo(Int2 direction) {
        Int2 target = position.add(direction);

        Cell cell = game.map.getAt(target);
        if (cell.isRockCell()) { return false; }

        for (Actor a : game.actors) {
            if (target.equal(a.position)) { return false; }
        }

        Cell newCell = cell.affectOn(this);
        game.map.setAt(target, newCell);
        position = target;
        return true;
    }

    abstract void action();

    void attackTo(Int2 direction) {                // サブクラスから移動
        Int2 target = position.add(direction);     // ターゲットの位置の計算
        for (Actor a : game.actors) {   // それぞれのアクターについて
            if (target.equal(a.position)) {// ターゲットの位置に a がいるならば
```

```
                a.takeDamageFrom(this);      // a が this からダメージを受ける
                return;                      // メソッドを脱出
            }
        }
    }

    abstract void takeDamageFrom(Actor a);   // ダメージを受ける

    void giveDamageTo(Player p) {}           // Player にダメージを与える
    void giveDamageTo(Monster m) {}          // Monster にダメージを与える
    void giveDamageTo(RedMonster r) {}       // Redmonster にダメージを与える
}
```

```
//プレイヤークラスのプログラム
class Player extends Actor {
    static final int INIT_HP = 100;

    String name;

    Player(Game game, String name) {
        super(game, new Int2(1,1), INIT_HP, '@');
        this.name = name;
    }

    Player(Game game) { this(game, "Player"); }

    public String toString() {
        return name+" : "+position+" "+hp;
    }

    @Override
    void takeDamageFrom(Actor a) { a.giveDamageTo(this); }
                                            // オブジェクトと引数を逆転させる

    @Override
    void giveDamageTo(Monster m) { m.hp -= 20; } // モンスターへ 20 のダメージ

    @Override
    void giveDamageTo(RedMonster r) { r.hp -= 15; } // レッドモンスターへは 15

    @Override
    void action() {
        Int2 direction;
        switch (Game.getKeyChar()) {
            case 'z': direction = new Int2(-1,+1); break;
            case 'x': direction = new Int2( 0,+1); break;
            case 'c': direction = new Int2(+1,+1); break;
            case 'a': direction = new Int2(-1, 0); break;
            case 'd': direction = new Int2(+1, 0); break;
            case 'q': direction = new Int2(-1,-1); break;
            case 'w': direction = new Int2( 0,-1); break;
            case 'e': direction = new Int2(+1,-1); break;
            default:  direction = new Int2( 0, 0);
        }

        if (!moveTo(direction)) { attackTo(direction); }
    }
}
```

```
//モンスタークラスのプログラム
class Monster extends Actor {
    static final int INIT_HP = 150;

    Monster(Game game, char c) { super(game, INIT_HP, c); }

    Monster(Game game) { this(game, 'M'); }

    @Override
    public String toString() {
        return "Monster : "+position+" "+hp;
    }

    @Override
    void takeDamageFrom(Actor a) { a.giveDamageTo(this); }

    @Override
    void giveDamageTo(Player p) { p.hp -= 30; } // プレイヤーへ 30 のダメージ

    @Override
    void giveDamageTo(RedMonster r) { r.hp -= 5; } // レッドモンスターへは 5

    @Override
    void action() {
        Actor player = game.actors.get(game.PLAYER_ID);
        Int2 direction = player.position.sub(position).signum();

        if (!moveTo(direction)) { attackTo(direction); }
    }
}
```

```
//レッドモンスタークラスのプログラム
class RedMonster extends Monster {
    RedMonster(Game game) { super(game, 'R'); }

    @Override
    public String toString() {
        return "Red"+super.toString();
    }

    @Override
    void takeDamageFrom(Actor a) { a.giveDamageTo(this); }

    @Override
    void giveDamageTo(Player p) { p.hp -= 60; } // プレイヤーへ 60 のダメージ

    @Override
    void giveDamageTo(Monster m) { m.hp -= 7; } // モンスターへは 7 のダメージ
}
```

```
//2 次元ベクトルクラスのプログラム
class Int2 {
    int x,y;

    Int2(int x, int y) { this.x = x; this.y = y; }
    Int2 add(Int2 a) { return new Int2(x+a.x,y+a.y); }
    Int2 sub(Int2 a) { return new Int2(x-a.x,y-a.y); }
    Int2 signum() { return new Int2(Integer.signum(x),Integer.signum(y)); }
    boolean equal(Int2 a) { return (x == a.x)&&(y == a.y); }

    public String toString(){ return "("+x+","+y+")"; }
}
```

```
//セルクラスのプログラム
abstract class Cell {
    abstract void cellPrint();

    boolean isRockCell() { return false; }

    abstract Cell affectOn(Actor a);
}

abstract class SimpleCell extends Cell {
    @Override
    Cell affectOn(Actor a) { return this; }
}

abstract class TrapCell extends Cell {
    boolean visible = false;

    @Override
    Cell affectOn(Actor a) {
        visible = true;
        return this;
    }
}


class RockCell extends SimpleCell {
    @Override
    void cellPrint() { System.out.print("# "); }

    @Override
    boolean isRockCell() { return true; }
}

class EmptyCell extends SimpleCell {
    @Override
    void cellPrint() { System.out.print(". "); }
}

class PotionCell extends SimpleCell {
    @Override
    void cellPrint() { System.out.print("o "); }

    @Override
    Cell affectOn(Actor a) {
        a.hp += 50;
        return new EmptyCell();
    }
}

class GoalCell extends SimpleCell {
    @Override
    void cellPrint() { System.out.print("G "); }

    @Override
    Cell affectOn(Actor a) {
        if (a instanceof Player) {
            System.out.println("YOU WIN");
            System.exit(0);
```

```
        }
        return this;
    }
}

class PoisonCell extends TrapCell {
    @Override
    void cellPrint() {
        if (visible) { System.out.print("x "); }
        else { System.out.print(". "); }
    }

    @Override
    Cell affectOn(Actor a) {
        a.hp -= 10;
        return super.affectOn(a);
    }
}
```

第 24 章　攻撃処理をエレガントにプログラム化する：ダブルディスパッチ法の利用

1

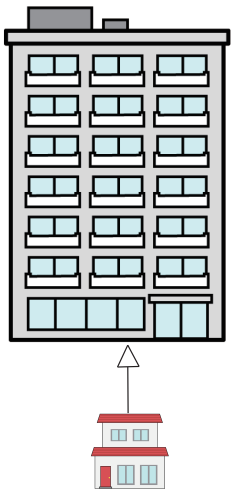
1

第25章　ゲームをグラフィカルにする準備：Java AWT/Swingを試してみる

Javaには GUI（Graphical Use Interface）を支援するツールキットが揃っています。**AWT**（Abstract Window Toolkit）は、1995 年から提供されている java の最初の GUI ツールキットです。**Swing**[1]は、AWT の中核部はそのままに全体がより洗練した仕組みで構築されたツールキットです。次章からダンジョンゲームを GUI 化しますが、この章では準備運動のつもりで簡単なプログラムを使って AWT/Swing の機能を確認します。

オブジェクト指向でないプログラミング言語では、言語の提供する様々な機能をユーザが関数を呼び出すことで利用するのが一般的です。しかし、オブジェクト指向プログラミング言語では、様々な機能がクラスとして用意されています。ユーザはそのクラスのサブクラスをユーザ定義し、親クラスの性質を継承して機能を利用します。

15 章、図 15.1（221 ページ）においてクラス階層のイメージを示しました。ここまではせいぜい 図 (b) の 2 階建て程度の家に住むイメージでしたが、ここからは、図 (c) のインテリジェントビルの機能（Java が提供している様々な機能）を利用していきます。下図はその活用イメージです。Java の提供するクラスを継承し、またインターフェースを実装し、様々な機能を活用していきます。



[1]名前の由来は不明です。

なお、この講義では AWT/Swing を詳細に学ぶことはしません。興味を持った人は自ら調べてください。

25.1　JFrame を使う

まず、Java のプログラムでウインドウを 1 枚開いてみます。図 25.1（439 ページ）がプログラムの実行の様子、図 25.2（439 ページ）がそのプログラムです。以下、プログラムの解説です。

プログラムの冒頭にはパッケージ読み込みのため、以下を付けます。

```
import java.awt.*;
import javax.swing.*;
```

この節のタイトルの JFrame がトップレベルの GUI 部品です。これは 1 枚のウインドウに対応します。Java の main メソッドでは以下のように JFrame オブジェクトをひとつ作成し、そのパラメータを設定していきます。

```
        JFrame frame = new JFrame("JFrame Test"); // jFrame オブジェクトを作る
```

コンストラクタの引数 "JFrame Test"はウインドウのタイトルです。以下：

```
        frame.setBounds(100,100,400,400);          // 初期位置、サイズを設定
```

では、ウインドウの左上をディスプレイ上の $(100, 100)$ の位置（単位は画素）として、ウインドウの大きさを $(400, 400)$ の大きさ（単位は画素）に設定します。次に、以下：

```
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                                                  // ウインドウを閉じるときの処理を設定
```

ではウインドウが閉じられるときの動作を設定します。EXIT_ON_CLOSE はウインドウが閉じられるときにアプリケーションも同時に強制終了することを意味します。他に、DO_NOTHING_ON_CLOSE、HIDE_ON_CLOSE、DISPOSE_ON_CLOSE が指定可能です。詳細は自ら調べてください。そして最後に以下でウインドウを表示します。これを忘れると、コンピュータディスプレイにウインドウは表示されず、そのままプログラムは終了します。決して忘れないようにしましょう。

図 25.1: AWT/Swing で 1 枚のウインドウを表示してみた例

```
import java.awt.*;
import javax.swing.*;

class Test {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JFrame Test"); // jFrame オブジェクトを作る
        frame.setBounds(100,100,400,400);         // 初期位置、サイズを設定
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                                              // ウインドウを閉じるときの処理を設定
        frame.setVisible(true);                   // ウインドウを表示する
    }
}
```

図 25.2: 図 25.1 を表示するプログラム

```
        frame.setVisible(true);                    // ウインドウを表示する
```

このプログラムではウインドウが表示された後はアプリケーションプログラムは何の動作も行わず、単に終了を待つだけです。

なお、JFrame オブジェクトを上の要領で二つ生成し、表示すると、ひとつのプログラムで 2 枚のウインドウをディスプレイ上に表示させることができます。

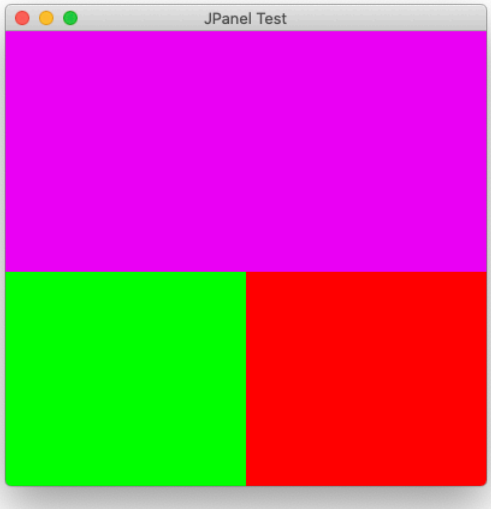


図 25.3: ウインドウに 3 枚のパネルを配置した例

25.2　JPanel を使う

JFrame はウインドウの管理を行いますが、ウインドウ内を領域毎に分割したり、その上に図形を描画したり、その上にボタンを配置したりはできない設計になっています。そのような役割を担う最も汎用的な部品はパネル JPanel です。

JFrame には複数枚の JPanel を配置できます。図 25.3（441 ページ）は、JPanel 3 枚を JFrame に貼り付けた場合の実行の様子です。図 25.4（442 ページ）は構成図です。図 25.5（442 ページ）はそのプログラムです。

3 枚のパネルの区別が付くように、ここではそれぞれのパネルに紫、赤、緑の背景色を付けました。Swing では様々な形状、様々な大きさのディスプレイへ柔軟に対応するために、フレーム内のパネルの位置をプログラマが厳密に指定することはできません。ラフなレイアウトのみを指定することができ、実際の配置は描画環境に応じて Swing が自動的に決定します。そこで図 25.5 のプログラムでは 3 枚のパネルを北側、東側、西側と指定してフレームに配置しています。

図 25.5 プログラムを解説すると、以下は JPanel オブジェクトを生成します。

```
        JPanel panel0 = new JPanel();                    // パネルの生成
```

以下は、パネルサイズを指定します。

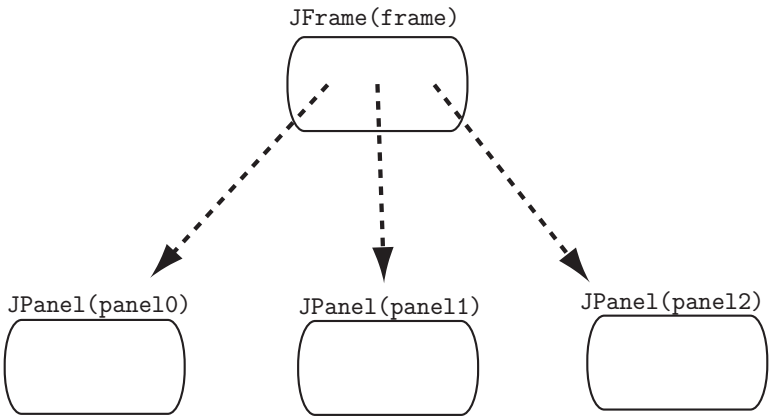


図 25.4: 図 25.3 における AWT/Swing オブジェクト（コンポーネント）の構成

```
import java.awt.*;
import javax.swing.*;

class Test {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JPanel Test");
        frame.setBounds(100,100,400,400);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JPanel panel0 = new JPanel();                    // パネルの生成
        panel0.setPreferredSize(new Dimension(200, 200)); // パネルサイズ
        panel0.setBackground(new Color(200,0,240));       // 明るい紫
        frame.add(panel0, BorderLayout.NORTH);
                    // 北側に配置したいと指定して、フレームにパネルを配置

        JPanel panel1 = new JPanel();                    // パネルの生成
        panel1.setPreferredSize(new Dimension(200, 200)); // パネルサイズ
        panel1.setBackground(Color.RED);                  // 赤
        frame.add(panel1, BorderLayout.EAST);
                // 東側に配置したいと指定して、フレームにパネルを追加配置

        JPanel panel2 = new JPanel();                    // パネルの生成
        panel2.setPreferredSize(new Dimension(200, 200)); // パネルサイズ
        panel2.setBackground(Color.GREEN);                // 緑
        frame.add(panel2, BorderLayout.WEST);
                // 西側に配置したいと指定して、フレームにパネルを追加配置

        frame.setVisible(true);
    }
}
```

図 25.5: 図 25.3 を表示するプログラム

```
panel0.setPreferredSize(new Dimension(200, 200)); // パネルサイズ
```

ただし、メソッド名に Preferred（優先された）と付いているように、あくまでも希望のサイズを指定できるだけです。実際の描画サイズは Swing のシステムが決定します。Dimension クラスの解説は省略します。気になる人は自ら調べてください。

以下は、パネルの背景色を指定します。

```
panel0.setBackground(new Color(200,0,240));       // 明るい紫
```

上では RGB カラーで指定していますが、2 番目、3 番目のパネルの場合：

```
panel1.setBackground(Color.RED);                  // 赤
...
panel2.setBackground(Color.GREEN);                // 緑
```

のように色名で指定することもできます。やはり、Color クラスの詳細は述べません。気になる人は自ら調べてください。

以下は、設定を終えたパネルをフレームに配置します。

```
frame.add(panel0, BorderLayout.NORTH);
            // 北側に配置したいと指定して、フレームにパネルを配置
```

このとき、このフレームがパネルの中で北側（上側）に配置されるようにレイアウトの指定を付けています。指定できるレイアウトは NORTH 以外には SOUTH、CENTER、WEST、EAST です。

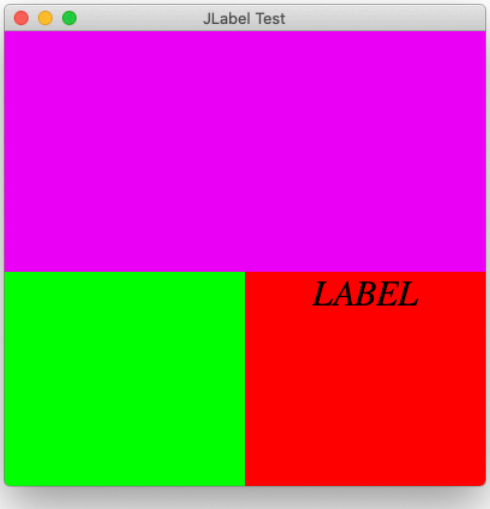


図 25.6: パネルにラベルを配置し、ラベルに LABEL を表示した例

25.3　JPanel に JLabel を貼る

JPanel の上には様々な GUI 部品 — これを一般に**ウィジェット**（widget）と総称します — を配置することができます。ここではラベル JLabel を置いてみましょう。ラベルはテキストを表示するための最も単純なウィジェットです。

図 25.6（444 ページ）はラベルを赤いパネル（panel1）に配置した場合の実行の様子です。図 25.7（445 ページ）は構成図です。図 25.8（445 ページ）はそのプログラムです。panel1 に JLabel を配置する処理を行なっています。

プログラムの以下：

```
        JLabel label = new JLabel("LABEL");                    // ラベルの生成
```

でラベルオブジェクトを生成します。コンストラクタには表示する文字列を与えます。以下はフォントの設定です。

```
        label.setFont(new Font(Font.SERIF, Font.ITALIC, 30));
```

Font オブジェクトを引数で与えます。Font のコンストラクタの各引数の意味は以下の通りです。

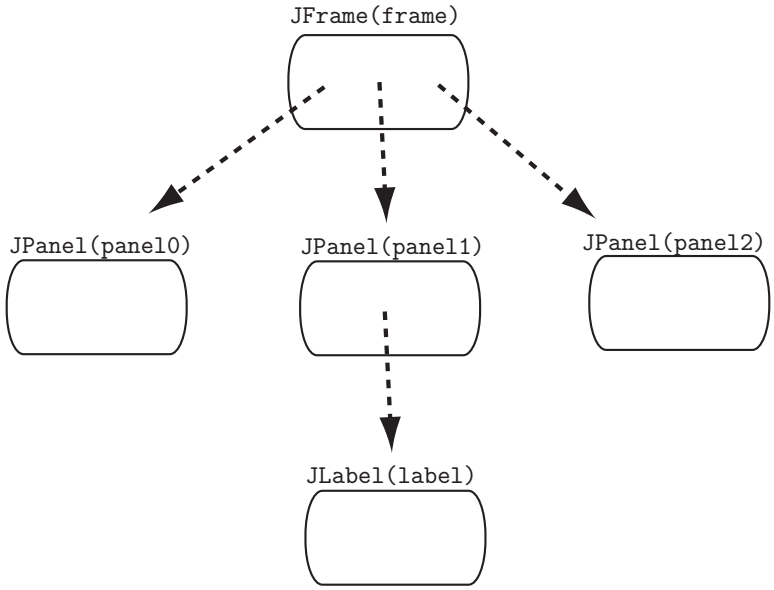


図 25.7: 図 25.6 における AWT/Swing オブジェクト（コンポーネント）の構成

```
import java.awt.*;
import javax.swing.*;

class Test {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JLabel Test");
        ...
        中略（前節のプログラムと同じ）
        ...

        JLabel label = new JLabel("LABEL");                    // ラベルの生成
        label.setFont(new Font(Font.SERIF, Font.ITALIC, 30));
                                                              // フォントの設定
        panel1.add(label, BorderLayout.EAST);        // label1 を panel1 に配置

        ...
        中略（前節のプログラムと同じ）
        ...
    }
}
```

図 25.8: 図 25.6 を表示するプログラム

フォント名 第一引数にはフォント名を与えます。フォント名は、使用するフォント名を文字列（たとえば "Times"）を与えるか、あるいは Font クラスに標準定義されているフォント名（たとえば Font.SERIF、Font.MONOSPACED など）を与えるかのいずれかの方法が可能です。前者のフォントを物理フォント、後者を論理フォントと呼びます。物理フォントの場合、実行環境にそれがインストールされていない場合に不都合です。論理フォントでは、実行環境にインストールされているフォントの中から論理フォント名に最も近い物理フォントをシステムが自動的に選択してくれ、手間要らずです。

スタイル 標準（PLAIN）、斜体（ITALIC）、太文字（BOLD）を与えます。斜体で太文字の場合 ITALIC|BOLD と bit 和として与えます。

サイズ フォントサイズをポイントで与えます。

最後にラベルオブジェクトをパネルに配置します。

```
        panel1.add(label, BorderLayout.EAST);       // label1 を panel1 に配置
```

上の設定ではパネルの東側（右側）に配置するように指示していますが、実行結果では上の方に配置されており、なかなか思った通りにはなりません（講義担当者の理解不足が原因ですが）が、あまり気にせず、先に進みましょう。

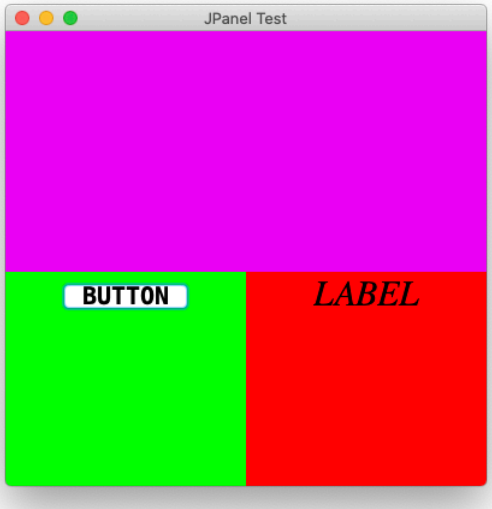


図 25.9: パネルにボタンを配置し、ボタンに BUTTON を表示した例

25.4　JPanel に JButton を貼る

前節と同様にして、パネル（panel2）の上にはボタン（JButton）を置いてみます。

図 25.9（447 ページ）はボタンを緑のパネルに配置した場合の実行の様子です。図 25.10（448 ページ）は構成図です。図 25.11（448 ページ）はそのプログラムです。

プログラム中のボタンの生成、ボタン上の文字のフォントの設定、ボタンのパネルへの登録は前節と同様です。なお、この章ではボタンを置いてみるだけとします。ボタンの機能の利用は後章で行います。

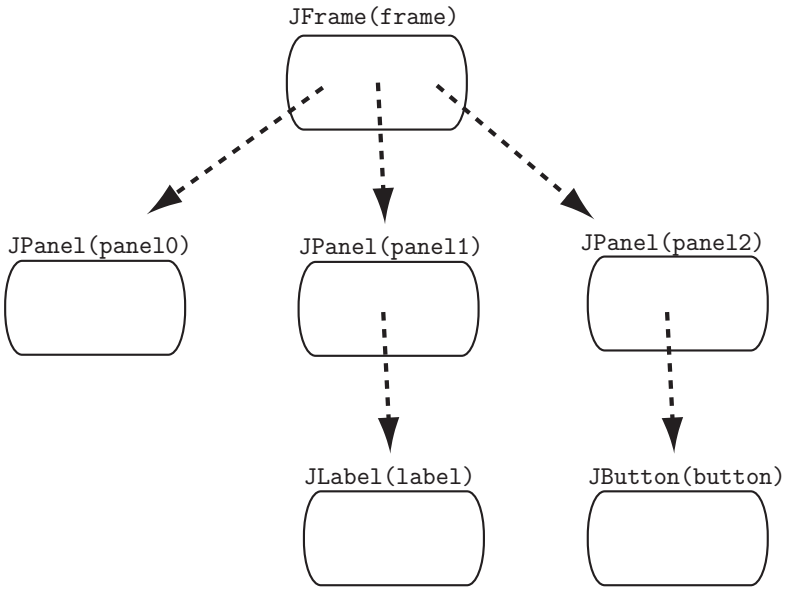


図 25.10: 図 25.9 における AWT/Swing オブジェクト（コンポーネント）の構成

```
import java.awt.*;
import javax.swing.*;

class Test {
    public static void main(String[] args) {
        ...
        中略（前節のプログラムと同じ）
        ...

        JButton button = new JButton("BUTTON");              // ボタンの生成
        button.setFont(new Font(Font.MONOSPACED, Font.BOLD, 20));
                                                            // フォントの設定
        panel2.add(button, BorderLayout.WEST);               // ボタンの配置

        frame.setVisible(true);
    }
}
```

図 25.11: fig:swing4output を表示するプログラム

25.5　JPanel を継承して図形を描画する

JPanel の上には図形を描画することもできます。そのための方法として、サブクラスを用いて JPanel のメソッドを上書きし、そのメソッドに描画処理を記述します。上書きの機能をこのように用いるのはオブジェクト指向の常套手段です。

図 25.12（450 ページ）(a) は、猫のイラスト画像[2]、塗りつぶされた円、円弧を描画した例です。図 25.13（451 ページ）はプログラムの構成図です。描画を行うために、panel0 のオブジェクト実体を JPanel を継承する MyPanel オブジェクトに変更し、JPanel のメソッド paintComponent() を MyPanel クラスで上書きします。図 25.14（452 ページ）が実際のプログラムです。

25.5.1　paintComponent() による描画

この節のポイントは、MyPanel クラスにおいて paintComponent() が上書き定義される点です。

このメソッド paintComponent() は Swing の各 GUI 部品を描画および再描画するメソッドです。たとえばウインドウが初期化された、ウインドウサイズが変更された、ディスプレイ上の複数のウインドウの重なり順が変更された、などの状況が発生したときに、コンピュータディスプレイ上の描画の乱れを修正するためにウインドウ内の部品を再描画する必要があります。その描画を担当するメソッドが paintComponent() です。このメソッドの実行については

- paintComponent() は Java のウインドウ管理システムから自動的に呼び出される

と約束されています。よって paintComponent() を上書きすれば、上書きしたサブクラスのメソッドが実行され、プログラマの意図する描画が可能です。逆に言えば、上書きしなければ親クラスに記述された標準的な再描画処理のみが行われます。

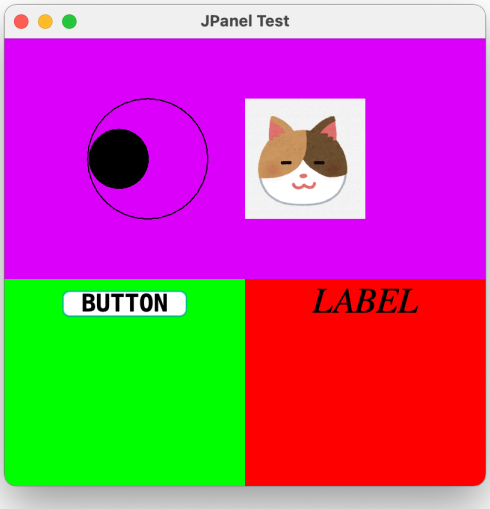
図 25.14 の以下のメソッド定義のヘッダー部：

```
public void paintComponent(Graphics g) {     // 上書きする描画用メソッド
```

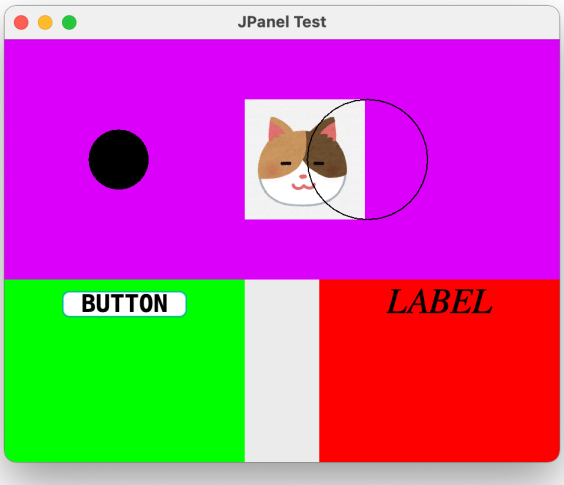
が示すように、メソッド paintComponent() の引数は Graphics g です。この g はグラフィックスコンテキストと呼ばれ、プログラムの指示をグラフィックス装置へ伝える仲介役のオブジェクトです。このオブジェクトの中身、出処を知る必要はありません。プログラマが g に対して描画操作を行えば、パネル上にそれが反映される仕組みです。

図 25.14 のメソッド本体の冒頭の

[2]猫のイラストは「かわいいフリー素材集：いらすとや」、https://www.irasutoya.com からダウンロードしました。無償で利用できることに感謝いたします。



(a) プログラム実行直後



(b) ウインドウを何度か再描画を行った後（円が右へ移動する）

図 25.12: 猫のイラスト画像、円（塗りつぶし）と円（線のみ）を描画した例：再描画の度に円（線のみ）が右側へ移動します。

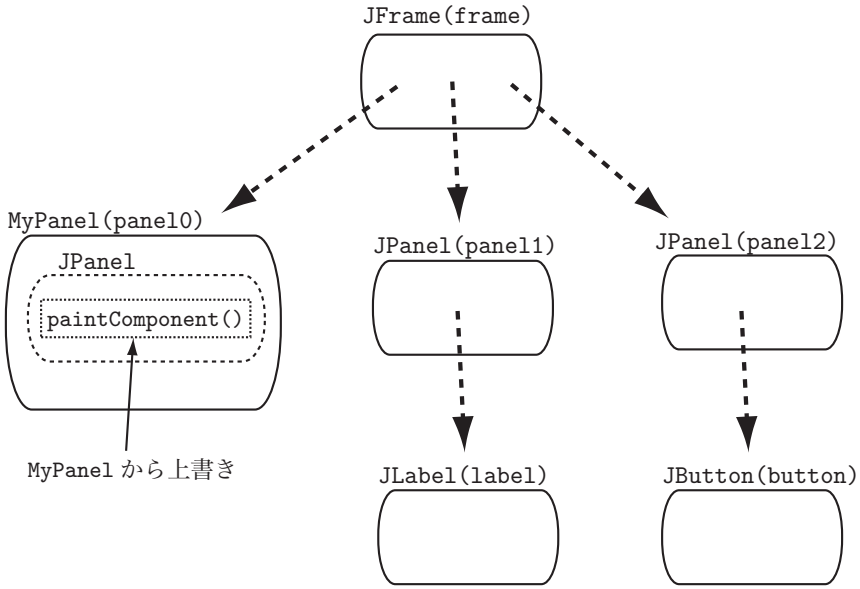


図 25.13: 図 25.12 における AWT/Swing オブジェクト（コンポーネント）の構成

```
super.paintComponent(g);     // とりあえずスーパークラスのメソッドを実行
```

は、AWT/Swing では paintComponent() を上書きする場合にはとりあえずスーパークラスの同じメソッド paintComponent() を呼び出すべきであるという慣用に従ったものです。というのも、paintComponent() の実行に際して、スーパークラス JPanel は JPanel なりに実行すべき処理（背景描画やこのパネルに配置されている全ての GUI 部品の再描画など[3]）があり、その処理を済ませてからユーザ独自の処理を行う方が一般的だからです[4]。

さて、MyPanel での独自処理は以下です。

```
g.drawImage(image, 200, 50, 100, 100, this); // 猫の画像の描画
g.drawOval( 50+i, 50, 100, 100);             // 円弧の描画
g.fillOval( 70, 75,  50,  50);               // 円の塗り潰し
```

g.drawImage() は第 1 引数で与えた画像オブジェクトを第 2〜5 引数で指定する矩形領域に描画するメソッドです。第 1 引数の画像オブジェクトのデータは、描画の度に作成していたのでは非

[3]一般プログラマはその詳細を知る必要はないとされています、
[4]スーパークラスの paintComponent() は後から呼び出した方がよい例外的な場合もありますが、特殊ですから省略します。

```
import java.awt.*;
import javax.swing.*;

class Test {
    public static void main(String[] args) {
        ...
        中略（前節のプログラムと同じ）
        ...

        JPanel panel0 = new MyPanel();

        ...
        中略（前節のプログラムと同じ）
        ...
    }
}

class MyPanel extends JPanel {
    int i = 0;                    // paintComponent() が呼び出された回数を保持

    final Image image = Toolkit.getDefaultToolkit().getImage("cat.png");
                                                  // 猫の画像をプログラム中に取り込む

    @Override
    public void paintComponent(Graphics g) {       // 上書きする描画用メソッド
        super.paintComponent(g);    // とりあえずスーパークラスのメソッドを実行
        g.drawImage(image, 200, 50, 100, 100, this); // 猫の画像の描画
        g.drawOval( 50+i, 50, 100, 100);           // 円弧の描画
        g.fillOval( 70, 75,  50,  50);             // 円の塗り潰し
        i++;                                       // 呼び出された回数を増やす
    }
}
```

図 25.14: 図 25.12 を表示するプログラム

効率ですから、メンバー変数としてあらかじめ作成しておきます。それが以下です。画像ファイル cat.png はプログラムと同じフォルダーに格納しておきます[5]。

```
    static final Image image
                    = Toolkit.getDefaultToolkit().getImage("cat.png");
```

static 修飾されたクラス変数 image の初期化はプログラム実行後にこのクラスが使用され始める[6]最初のタイミングで行われます。g.drawImage() の第 6 引数にはこの画像描画のオブザーバー

[5] .png ファイル以外に.jpeg、.gif ファイルが利用できます。.gif の場合、gif アニメーションの描画も可能です。
[6] しばしば「クラスがロードされるとき」と表現されます。

を指定します。通常、第 6 引数には画像描画のキャンバスを提供するオブジェクト（この場合は、MyPanel オブジェクト）を指定して問題ありません[7]。この場合は `this`（自分自身）です。

次の `drawOval()` は楕円の円弧を描画するメソッド、`fillOval()` は楕円の中を塗りつぶすメソッドです。第 1、2 引数は楕円に外接する四辺形の左上の座標値、第 3、4 引数はその四辺形の横、縦の大きさです。この場合、第 3、4 引数は同じ値ですから、円が描画されます。よって、`g.drawOval( 50+i, 50, 100, 100)` は、座標 $(50 + i + 100/2, 50 + 100/2) = (100 + i, 100)$ を中心とする半径 $100/2 = 50$ の円弧を描き、`g.fillOval( 70, 75, 50, 50)` は、座標 $(70 + 50/2, 75 + 50/2) = (95, 100)$ を中心とする半径 $50/2 = 25$ の円を塗りつぶす描画を行います。

インスタンス変数 `i` は描画動作をチェックするために導入してみました。メソッド `paintComponent()` が呼び出される度に `i` が増え、楕円の位置が右側へ移動するはずです。実際に、マウス操作でウインドウサイズを少し変更した後の描画が、図 25.12（450 ページ） (b) です。円弧が右側へ移動しています。

[7]オブザーバーの役割について詳しく知りたい場合は、“Java AWT 入門”、http://wisdom.sakura.ne.jp/system/java/awt/index.html の 13 章、14 章を参考にしてください。

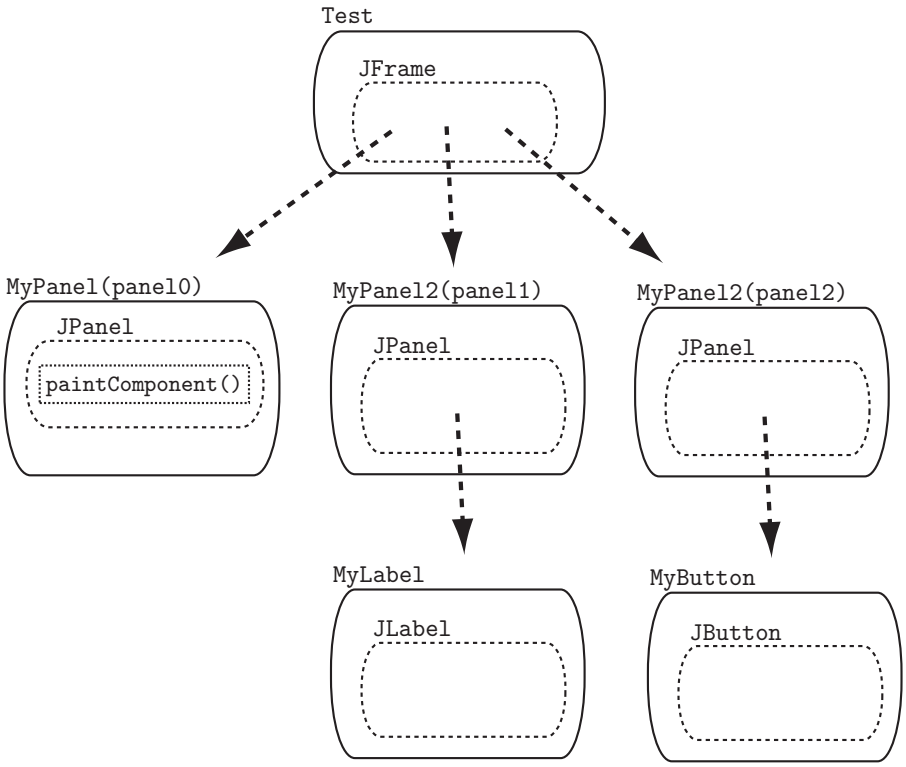


図 25.15: 部品全てにサブクラスを作るプログラムの AWT/Swing オブジェクト（コンポーネント）の構成

25.6　全ての GUI 部品を継承してみる

前節では描画部分についてサブクラスを用いました。この節では描画にとどまらず、関係する全てのクラスについてサブクラスを定義し、設定の多くをサブクラスに記述するように変更します。実は AWT/Swing を用いる多くの Java のプログラムはこのような作りになっています。

図 25.15（454 ページ）は、この節のプログラムの構成です。図 25.13（451 ページ）と比較してください。

図 25.16（455 ページ）は `JFrame` のサブクラスである `Test` クラスの定義です。前章までの `JFrame` の設定がクラスのコンストラクタに記述されていることに注意してください。その結果、`main` メソッドの本文はたった 1 行：

```
        new Test();
```

で済んでいます。最上位メソッドの中がゴチャゴチャせず、すっきりしています。`Test` オブジェクトを変数から参照していませんが、このオブジェクトはガベージコレクションの対象とはなりま

```
import java.awt.*;
import javax.swing.*;

class Test extends JFrame {
    public static void main(String[] args) {
        new Test();
    }

    Test() {
        super("Test");
        setBounds(100,100,400,400);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JPanel panel0 = new MyPanel();
        add(panel0, BorderLayout.NORTH);

        JPanel panel1 = new MyPanel2(Color.RED, new MyLabel(),
                                     BorderLayout.EAST);
        add(panel1, BorderLayout.EAST);

        JPanel panel2 = new MyPanel2(Color.GREEN, new MyButton(),
                                     BorderLayout.WEST);
        add(panel2, BorderLayout.WEST);

        setVisible(true);
    }
}
```

図 25.16: 既存クラスをサブクラスで拡張する（その 1、その 2 へ続く）

せん。というのも、このオブジェクトは AWT/Swing のウインドウ管理システムからプログラムの文面からは見えない形で参照されているためです。プログラム全体の実行を終了しない限り、このオブジェクトは削除されません。

図 25.17（456 ページ）は MyPanel、MyPanel2、MyLabel、MyButton のクラスの定義です。やはり設定を全てコンストラクタ内に記述しています。

```
class MyPanel extends JPanel {
    int i = 0;

    final Image image = Toolkit.getDefaultToolkit().getImage("Player.png");

    MyPanel() {
        setPreferredSize(new Dimension(200, 200));
        setBackground(new Color(200,0,240));
    }

    @Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.drawImage(image, 200, 50, 100, 100, this);
        g.drawOval( 50+i, 50, 100, 100);
        g.fillOval( 70, 75,  50,  50);
        i++;
    }
}
class MyPanel2 extends JPanel {
    MyPanel2(Color color, Component compnent, String layout) {
        setPreferredSize(new Dimension(200, 200));
        setBackground(color);
        add(compnent, layout);
    }
}

class MyLabel extends JLabel {
    MyLabel() {
        super("LABEL");
        setFont(new Font(Font.SERIF, Font.ITALIC, 30));
    }
}

class MyButton extends JButton {
    MyButton() {
        super("BUTTON");
        setFont(new Font(Font.MONOSPACED, Font.BOLD, 20));
    }
}
```

図 25.17: 既存クラスをサブクラスで拡張する（その 2）

25.7　補足：参考文献

AWT/Swing の GUI 部品のまとめサイトとして以下が便利です。参考にしてください。

Java プログラム開発の入門・基礎・応用、初心者から中級者まで
Java プログラミングの総合サポートサイト/Javax.Swing とは
`http://java.it-manual.com/application/swing.html`

AWT/Swing の裏事情や Java のより進んだ理解には以下が勉強になります。

AWT と Swing のペイントの仕組み
上質な描画コード (ペイントコード) がアプリケーションの実行性能を支える
`http://alga.no.coocan.jp/paint.html`

1

1

第26章　ゲームをグラフィカルにする：AWT/Swingの利用

この章では AWT/Swing をダンジョンゲームに適用し、グラフィカルなゲーム作成を目指します。基本的な方法は前章で述べた通りです。マップ、アクターなどのクラスを GUI 部品のサブクラスとして拡張します。

26.1　モデル・ビュー・コントローラ

ユーザインターフェースが大きな比重を成すソフトウェアについて、その構造を論じる視点として**モデル・ビュー・コントローラ**（Model-View-Controller、以下 MVC）と呼ばれる概念が知られています。MVC では、プログラム構造を三つの部分に分け、それらの関係性で構造を論じます。三つの部分（モデル、ビュー、コントローラ）は以下のように説明できます（図 26.1（461 ページ）参照）。

モデル ... プログラムの中核となるデータとその操作を行う部分です。このテキストのダンジョンゲームで言えば、マップ、アクターのデータとその更新操作部分に相当します。

ビュー ... モデルのデータをコンピュータディスプレイ等へ出力する部分です。このテキストのダンジョンゲームでは前章までは CUI を用いてきましたが、この章ではそれを GUI に置き換えます。

コントローラ ... ユーザからの入力をモデルに伝える部分です。ダンジョンゲームではキーボードからの文字の入力に相当します。

プログラムの各部分は上の三つのいずれかに分類できることが理想ですが、現実のプログラムではそう単純に割り切れるものではないため、多くの場合、MVC はプログラミングにひとつの指針を与えるものと理解されています。

実はこの講義テキストのダンジョンゲームは当初より MVC を意識して作られています。図 26.1 には MVC の各部へダンジョンゲームのメソッドを対応付けています。多くのメソッドはモデルの更新に利用されています。プリント関係のメソッドはビューに関連しています。コントローラに相当するのは `Game` クラスの `getKeyChar()` のみです。

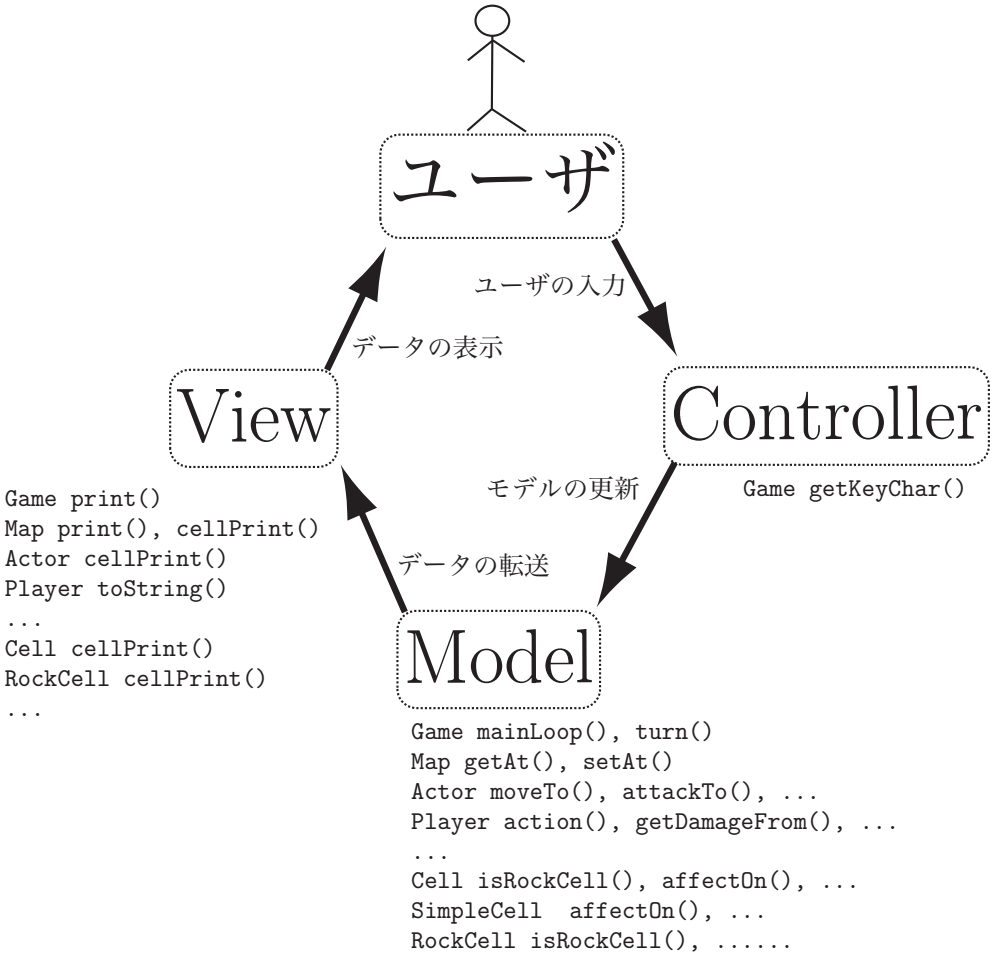


図 26.1: Model-View-Controller とダンジョンゲームのメソッド群の関係

図 26.1 を見ると、ダンジョンゲームの中核部（モデル）を変えずに表示のみを入れ替えるにはビューを入れ替えればよいことが分かります。つまり、ビューに関連する `Game` クラスの `print()`、`Map` クラスの `print()`、`cellPrint()`、`Actor` クラスの `cellPrint()` などを AWT/Swing を利用するものに入れ替えれば良い訳です。コントローラ部分は次章において変更します。

26.2　プログラムの構造

図 26.2（462 ページ）は、この章のプログラムの実行の様子です。既に 1.3 節（4 ページ）で紹介済みです。この表示を実現するために、プログラム構造を図 26.3（463 ページ）のように変更します。

ゲームウインドウは `JFrame` で生成するものとします。これは前章の方法と同じです。ウインドウは 2 枚の `JPanel` からなり、上方の `JPanel` にマップを表示し、下方の `JPanel` にアクターの状

図 26.2: AWT/Swing を用いたダンジョンゲームの実行の様子

態を文字列で表示するものとします。

以下、各変更箇所を説明します。

1. まず、Game クラスを JFrame クラスのサブクラスとします。これは、Game オブジェクトがゲームのモデル部分を統括するオブジェクトであるため、それをゲームのビューを管理する JFrame と一体化させることは都合が良いためです。図 26.3 では Game オブジェクトの中に JFrame オブジェクトが含まれる関係となっています（同様の図はたとえば前章、図 25.15（454 ページ）を参照）。

2. 次に、Map クラスを JPanel クラスのサブクラスとします。Map オブジェクトはマップおよびそれを構成するセル群を保持するオブジェクトです。その情報はゲームのマップ画面の表示に直結しますから、Map オブジェクトを JPanel オブジェクト（図 26.2 の上方の JPanel）と一体化させます。図 26.3 では Map オブジェクトの中に JPanel オブジェクトが含まれる関係となっています。

3. アクター配列 actors も 図 26.2 の下方の JPanel と一体化させた方が分かりやすいのですが、アクター配列は図 26.3 では単なる ArrayList に過ぎず、これと JPanel を一体化するには、新たにクラスを定義する必要があります。これは返って面倒になりそうです。そこで、

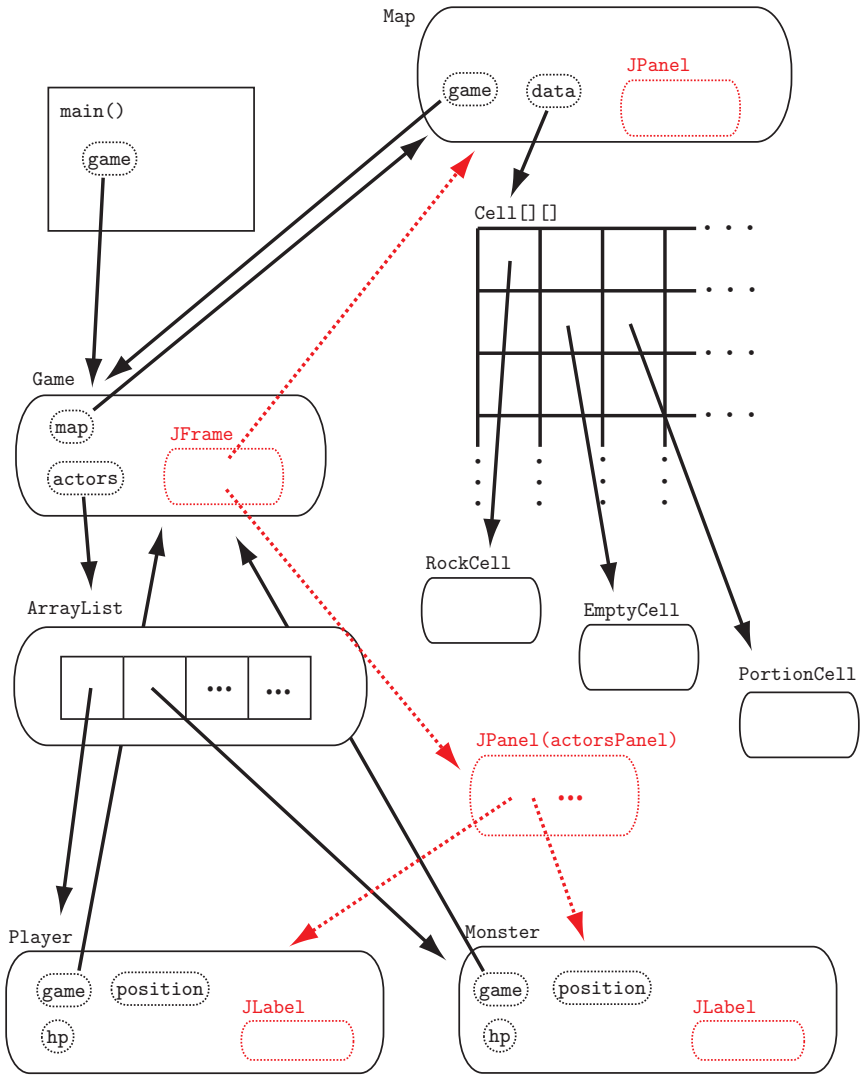


図 26.3: AWT/Swing を用いるプログラムの全体構造

アクター配列 actors に対応する JPanel はアクター配列とは別に作ることにします。Game クラスのコンストラクタではこの JPanel オブジェクトを actorsPanel という変数で参照します（後節のプログラムを参照）。

4. 各アクターの位置、HP をウインドウ上に表示せねばなりませんが、その表示は JLabel に任せることとします（25.3 節（444 ページ）参照）。そこで、Actor クラスを JLabel クラスのサブクラスとします。

5. 上のように定義する JFrame オブジェクト（Game オブジェクトの中核部分）に Map オブジェクト（中核部分が JPanel オブジェクト）と actorsPanel を配置します。

6. `JPanel` オブジェクトである `actorsPanel` に、アクター配列に保持されている各アクターオブジェクト（中核部分が `JLabel` オブジェクト）を配置します。

以上がこの章の AWT/Swing 関係のオブジェクトの構造です。

```
class Game extends JFrame {          // JFrame クラスのサブクラスとして宣言
    public static void main(String args[]) {  ** 前章から変更なし ** }

    ** インスタンス変数なども前章から変更なし **

    Game() {
        super("Dungeon Game");                      // JFrame 部分の初期化

        ** map, actors の初期化は前章と同じ **

        setBounds(100, 100, Cell.SIZE*Map.XSIZE+5+200,   // ウインドウの
                            Cell.SIZE*Map.YSIZE+25+200); // 設定
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  //

        add(map, BorderLayout.NORTH);    // map オブジェクトを JFrame に配置

        JPanel actorsPanel = new JPanel();      // もう一枚の JPanel の生成
        actorsPanel.setLayout(new GridLayout(10,1));  // 縦一列にレイアウト
        for (Actor a : actors) { actorsPanel.add(a); }
                                                // 各アクターをパネルに配置
        add(actorsPanel, BorderLayout.SOUTH);// actorsPanel を JFrame に配置

        setVisible(true);
    }

    void print() { repaint(); }       // JFrame へウインドウの再描画の依頼

    void mainLoop() {  ** 前章から変更なし ** }

    void turn() {  ** 前章から変更なし ** }

    static char getKeyChar() {  ** 前章から変更なし ** }
}
```

図 26.4: AWT/Swing を利用するダンジョンゲームのプログラム（ゲーム）

26.3　ゲームクラスの変更

図 26.4（465 ページ）は、この章のゲームクラスのプログラムです。

まず、第一に Game クラスを JFrame クラスのサブクラスとします。

```
class Game extends JFrame {          // JFrame クラスのサブクラスとして宣言
```

次にコンストラクタの冒頭で、JFrame クラスのコンストラクタを呼び出します。ウインドウのラベルは "Dungeon Game" とします。

```
        super("Dungeon Game");                      // JFrame 部分の初期化
```

そして、インスタンス変数を前章同等に初期化した後で、ウインドウの設定を行います。以下：

```
setBounds(100, 100, Cell.SIZE*Map.XSIZE+5+200,   // ウインドウの
                    Cell.SIZE*Map.YSIZE+25+200); // 設定
```

はウインドウサイズの設定ですが、ここに Cell.SIZE は、ひとつのセルを正方形で描画する（図 26.2（462 ページ）参照）場合の正方形の一辺の画素サイズです。Cell クラスに新たに定義しています（26.9 節（477 ページ）参照）。

　そして以下のメソッド呼び出しで、Map オブジェクト（JPanel のサブクラスのオブジェクト）をこのオブジェクトの JFrame オブジェクト部分の上方（北側）に配置します。

```
add(map, BorderLayout.NORTH);     // map オブジェクトを JFrame に配置
```

次に、各アクターの JLabel を表示するための JPanel を変数 actorsPanel で参照します。

```
JPanel actorsPanel = new JPanel();      // もう一枚の JPanel の生成
```

そして、このパネルに貼り付ける GUI 部品（ここでは各アクターに対応する JLabel）が縦一列に表示されるように以下の設定を行います。

```
actorsPanel.setLayout(new GridLayout(10,1));  // 縦一列にレイアウト
```

ここにオブジェクト new GridLayout(m,n) は貼り付ける GUI 部品を左上を起点として、縦 m 行、横 n 列の並びに配列するレイアウトの指定です。JPanel への add() の順番は、$(1,1)$、$(1,2)$、...、$(1,n)$、$(2,1)$、... の順番です。上のプログラムでは $n=1$ ですから、配置は上から順に　$(1,1)$、$(2,1)$、$(3,1)$、... となります。

　そして、以下では各アクター（JLabel のサブクラス）をパネルに順に配置登録し、パネルを JFrame（ゲームクラスのスーパークラス）に配置します。

```
for (Actor a : actors) { actorsPanel.add(a); }
                                          // 各アクターをパネルに配置
add(actorsPanel, BorderLayout.SOUTH);// actorsPanel を jFrame に配置
```

26.3.1　再描画依頼メソッド：repaint()

前章（25.5.1 項（449 ページ））で描画および再描画が必要になった場合には Java のウインドウ管理システムによってメソッド paintComponent() が自動的に呼び出されると述べました。「必要になった場合」として 25.5.1 項では「ウインドウが初期化された、ウインドウサイズが変更された、ディスプレイ上の複数のウインドウの重なり順が変更された」などを挙げましたが、さらに「必要になった場合」として

- プログラマが JFrame に対して repaint()　を呼び出したとき

も再描画を行うと約束されています。これはプログラマによる強制再描画の指示です。

Java のウインドウ管理システムは描画および再描画の発動条件を取りまとめて管理しており、実際にシステムが必要と判断した場合に paintComponent() を呼び出します。

図 26.5（468 ページ）にこの章のプログラムにおける paintComponent() と repaint() に関連する呼び出しをプログラム構成図に記載しました。

Game オブジェクトはメソッド print() から repaint() を呼び出しています。そうすると、ウインドウ管理システムが再描画をまず JFrame オブジェクト（Game オブジェクトの中核部分）に指示します。JFrame オブジェクトは自身が保持する GUI 部品、Map オブジェクト と actorsPanel に対して paintComponent() を呼び出します。Map オブジェクト は自身の描画処理を行います。この詳細について次節で述べます。actorsPanel は自身が保持する GUI 部品、アクターオブジェクトのそれぞれに対して paintComponent() を呼び出します。このように GUI 部品の包含関係に基づいて paintComponent() が連鎖的に呼び出されていきます。

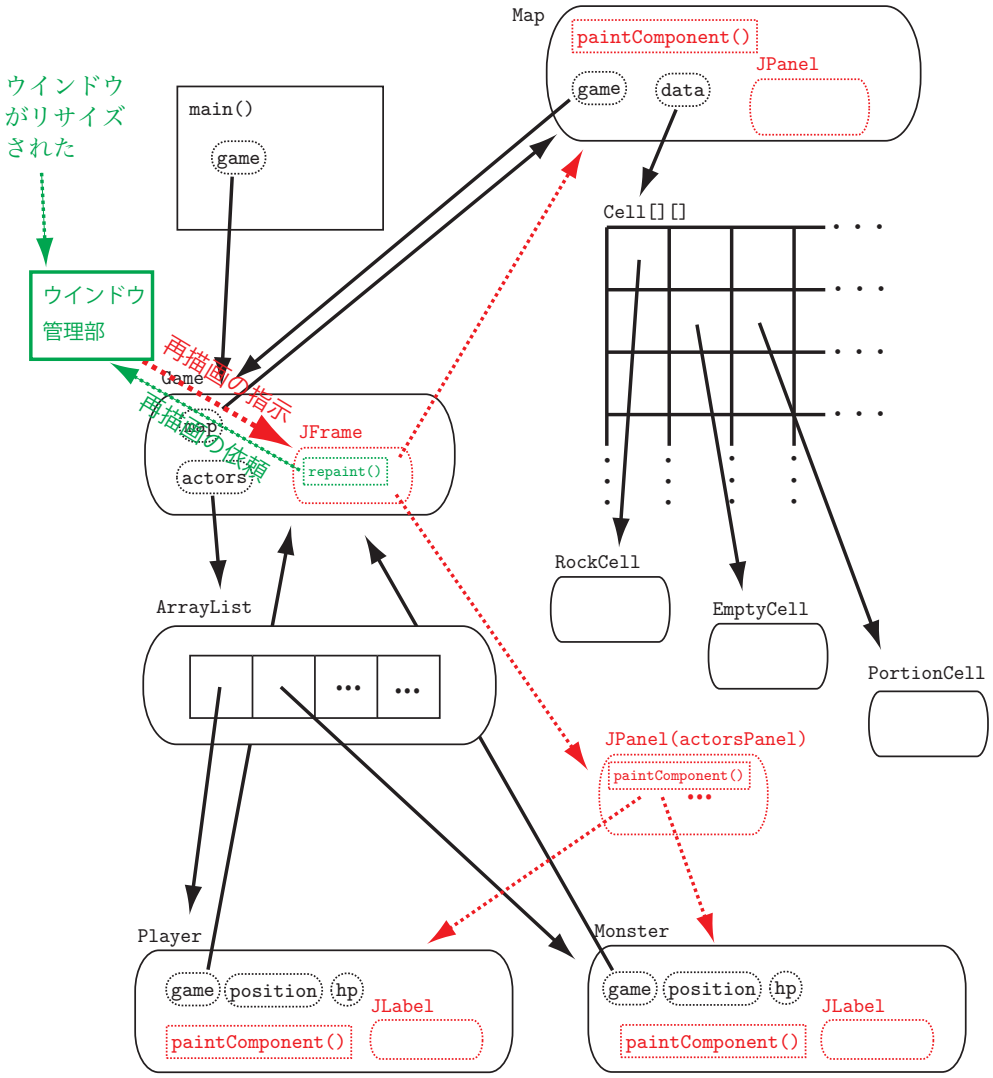


図 26.5: AWT/Swing を用いるプログラムの描画手順：repaint() と paintComponent() の呼び出し

```
class Map extends JPanel {                          // サブクラス化
    ** インスタンス変数など前章から変更なし **

    Map(Game game) {
        ** Cell の初期化部分は前章から変更なし **

        setPreferredSize(new Dimension(Cell.SIZE * XSIZE,
                                       Cell.SIZE * YSIZE));
                                                   // パネルの希望サイズを設定
    }

    Cell getAt(Int2 position) { ** 前章から変更なし ** }

    void setAt(Int2 position, Cell cell) { ** 前章から変更なし ** }

    @Override
    public void paintComponent(Graphics g) {     // JPanel のメソッドを上書き
        super.paintComponent(g);         // とりあえず JPanel のメソッドを実行
        for (int y = 0; y < YSIZE; y++) {
            for (int x = 0; x < XSIZE; x++) {
                Int2 position = new Int2(x, y);
                getAt(position).paintAt(g, position); // Cell をパネルに表示
                for (Actor a : game.actors) {
                    a.paintAt(g, position);       // 各アクターをパネルに表示
                }
            }
        }
    }

    Int2 getRandomCellPosition() { ** 前章から変更なし ** }
}
```

図 26.6: AWT/Swing を利用するダンジョンゲームのプログラム（マップ）

26.4　マップクラスの変更

26.2 節（461 ページ）で述べたように、マップクラスを JPanel のサブクラスとします。

コンストラクタ内の

```
        setPreferredSize(new Dimension(Cell.SIZE * XSIZE,
                                       Cell.SIZE * YSIZE));
                                                   // パネルの希望サイズを設定
```

は、マップを表示（描画）するために必要なサイズ（画素数）の設定です。セルの一辺の長さ（= Cell.SIZE）× セルの縦横の数で計算します。

前章まであったメソッド print() はこの章の paintComponent(Graphics g) に置き換えます。メソッド paintComponent(Graphics g) はスーパークラス JPlane のメソッドの上書きで

す。前節で述べたように、Game オブジェクトの print() が呼び出されたときに、このメソッドは AWT/Swing のシステムから自動的に呼び出されます。図 26.7（471 ページ）のように、ここでは Map オブジェクトの paintComponent(Graphics g) からセルおよびアクターのメソッド paintAt() を呼び出すこととします。

セルへのメソッド呼び出し：

```
getAt(position).paintAt(g, position); // Cell をパネルに表示
```

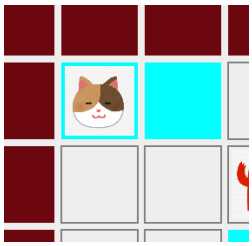
はセルを JPanel に描画する処理です。セルのメソッド paintAt() については後述します。

アクターへのメソッド呼び出し：

```
a.paintAt(g, position);        // 各アクターをパネルに表示
```

は、アクターを JPanel に描画する処理です。メソッド paintAt() については後述します。

ここで、セルの描画とアクターの描画の順序が前々章までと逆であることに注意してください。前々章までは、もし (x, y) の位置にアクターが居るならば、それを（キャラクターとして）表示し、セルを表示しません。もし (x, y) の位置にアクターが居ないならば、セルを表示します。この章では、まずセルを描画し、次いで同じ位置のアクターを描画します。アクターの描画はセルの描画を画像として上書きするのですが、これによって、たとえば以下のようなポーションセル（水色の正方形）の上に居るプレイヤー（猫）が描画できることになります。



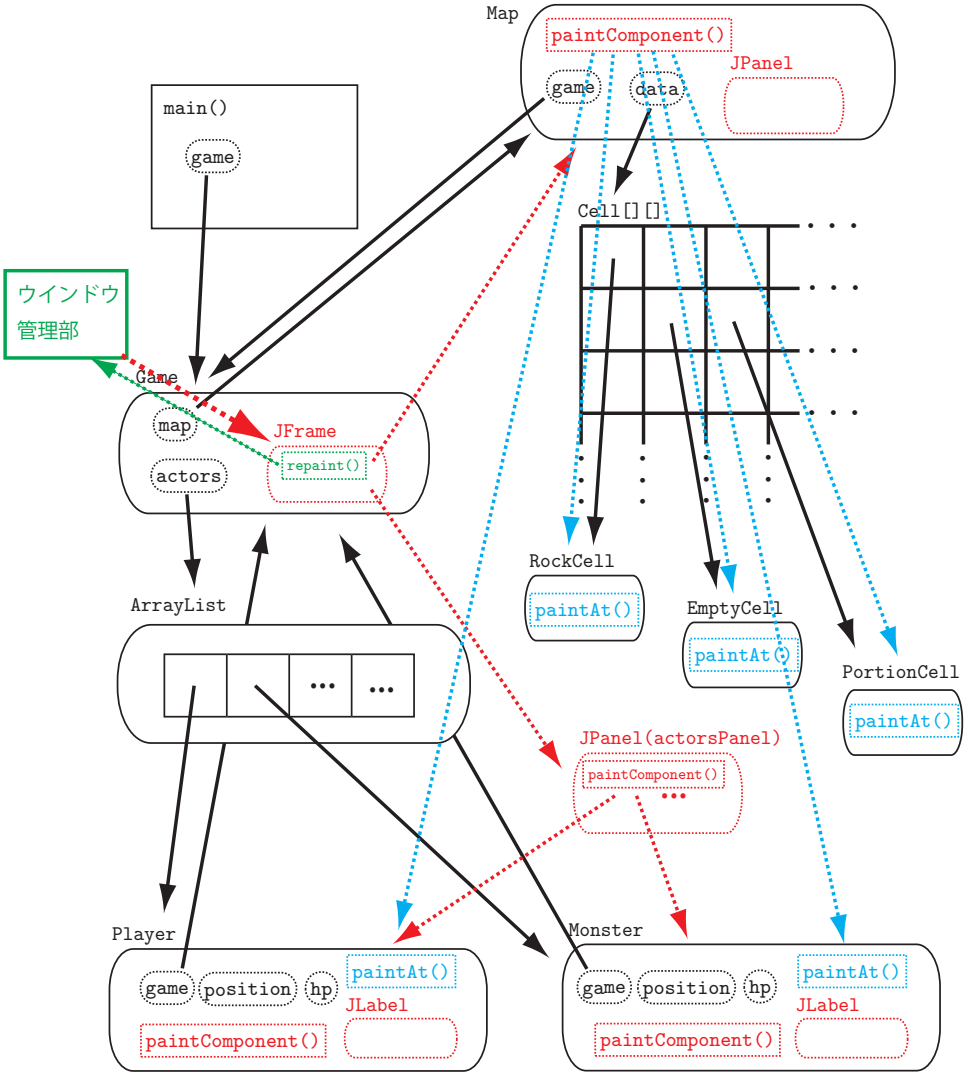


図 26.7: AWT/Swing を用いるプログラムの描画手順：`paintAt()` の呼び出し（青い点線矢印に注目）

26.5　アクタークラスの変更

図 26.8（473 ページ）がこの章のアクタークラスのプログラムです。26.2 節（461 ページ）で述べたように、アクタークラスは JLabel のサブクラスとします。

インスタンス変数 cellChar は 16 章（238 ページ）以来、アクターの表示に用いてきましたが、もはや CUI は用いませんから削除します。これに伴い、コンストラクタの引数が減るなどの変更があります。

メソッド paintComponent(Graphics g) はスーパークラス JLabel のメソッドの上書きです。前節で述べたように、このメソッドは AWT/Swing のシステムから自動的に呼び出されます。

paintComponent(Graphics g) の中の

```
        setText(toString());            // ラベルの内容をアクターの状態で更新
```

は、引数であるアクターの文字列表示をラベルのバッファに設定します。図 26.2（462 ページ）の下方に表示されているアクターの名前、位置、HP がこれに相当します。設定された文字列は自動的にパネル上に反映されます[1]。

マップ上の各セルを描画する paintAt() は、アクターが引数 position の位置に居るときに画像を描画するメソッドです。

```
    void paintAt(Graphics g, Int2 position) { // アクターをマップに表示
        if (this.position.equal(position)) {
            g.drawImage(getImage(),                          // 画像
            ...
```

上のプログラム片に続く引数部分：

```
                        position.x*Cell.SIZE + Cell.SIZE*2/24,// x 座標
                        position.y*Cell.SIZE + Cell.SIZE*2/24,// y 座標
                        Cell.SIZE*20/24,                      // 水平サイズ
                        Cell.SIZE*20/24,                      // 垂直サイズ
```

は、セルを縦横それぞれ 24 分割した中に図 26.9（474 ページ）のように画像を描画することを意図しています。

メソッド getImage() はこのオブジェクトの描画像を求めるメソッドで、すぐ下に

[1] 設定された文字列がパネル上に反映されるタイミングがズレて、反映が遅れる場合があるかもしれません（調査中です）。

```
abstract class Actor extends JLabel {   // サブクラス化
    Game game;
    Int2 position;
    int hp;
    ** cellChar を削除 **

    Actor(Game game, Int2 position, int hp) { // cellChar を削除
        super(" ");                           // ラベルの初期値は " "
        ** 以下、前章から変更なし **
    }

    Actor(Game game, int hp) {                // cellChar を削除
        super(" ");                           // ラベルの初期値は " "
        ** 以下、前章から変更なし **
    }

    @Override
    public void paintComponent(Graphics g) { // アクターの状態の出力
        super.paintComponent(g);         // とりあえず JLabel のメソッドを実行
        setText(toString());           // ラベルの内容をアクターの状態で更新
    }

   void paintAt(Graphics g, Int2 position) { // アクターをマップに表示
        if (this.position.equal(position)) {
            g.drawImage(getImage(),                          // 画像
                        position.x*Cell.SIZE + Cell.SIZE*2/24,// x 座標
                        position.y*Cell.SIZE + Cell.SIZE*2/24,// y 座標
                        Cell.SIZE*20/24,                     // 水平サイズ
                        Cell.SIZE*20/24,                     // 垂直サイズ
                        game.map);                          // オブザーバー
        }
    }

    abstract Image getImage();         // アクターの画像を求める抽象メソッド

    boolean moveTo(Int2 direction) { ** 前章から変更なし ** }

    abstract void action();

    void attackTo(Int2 direction) { ** 前章から変更なし ** }

    abstract void takeDamageFrom(Actor a);
    void giveDamageTo(Player p) {}
    void giveDamageTo(Monster m) {}
    void giveDamageTo(RedMonster r) {}
}
```

図 26.8: AWT/Swing を利用するダンジョンゲームのプログラム（アクター）

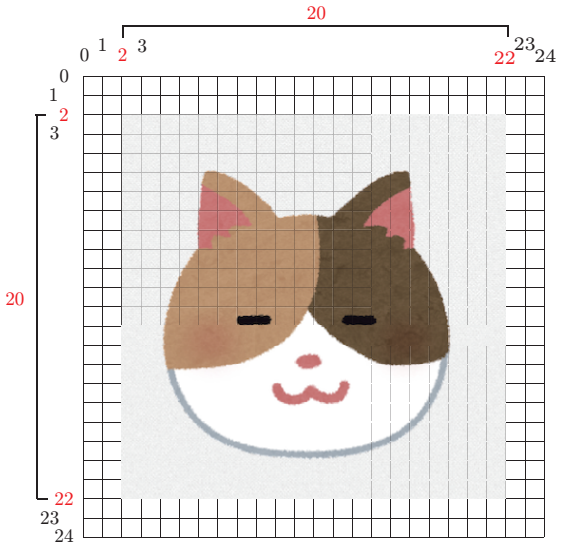


図 26.9: セル上でのプレイヤーの描画サイズ（モンスター、レッドモンスターも同様）

```
abstract Image getImage();          // アクターの画像を求める抽象メソッド
```

と、抽象メソッドとして宣言されています。その実装はサブクラスの Player、Monster、RedMonster がそれぞれ担当します。

その他の変更はありません。

```
class Player extends Actor {
    ** 以下の部分を除いてほとんど変更なし **

    static final Image image1
                    = Toolkit.getDefaultToolkit().getImage("Player1.png");
    static final Image image2
                    = Toolkit.getDefaultToolkit().getImage("Player2.png");

    @Override
    Image getImage() {                          // 画像オブジェクト取得の実装
        if (hp > INIT_HP/2) { return image1; } // 元気ならば
        else { return image2; }                 // 弱ってきたならば
    }
}
```

図 26.10: AWT/Swing を利用するダンジョンゲームのプログラム（プレイヤー）

26.6　プレイヤークラスの変更

図 26.10（475 ページ）はこの章のプレイヤークラスのプログラムです。主な変更箇所（アクタークラスの `cellChar` の削除に因るものを省略します）はメソッド `getImage()` の実装です。このメソッドはプレイヤーの画像を `Image` オブジェクトとして返す役割があります。ここでは、

```
        if (hp > INIT_HP/2) { return image1; } // 元気ならば
        else { return image2; }                 // 弱ってきたならば
```

によって、もしプレイヤーの HP が初期 HP の半分より上ならば元気な猫 `image1` を、さもなくば影の薄い、助けを求めている猫 `image2` を返すようにしています。

26.7　モンスタークラスの変更

図 26.11（476 ページ）はこの章のモンスタークラスのプログラムです。プレイヤー同様、主な変更箇所は、メソッド `getImage()` の実装です。ここでは画像 を用います。

```
class Monster extends Actor {
    ** 以下の部分を除いてほとんど変更なし **

    static final Image image
                    = Toolkit.getDefaultToolkit().getImage("Monster.png");

    @Override
    Image getImage() { return image; }    // 画像オブジェクト取得の実装
}
```

図 26.11: AWT/Swing を利用するダンジョンゲームのプログラム（モンスター）

```
class RedMonster extends Monster {
    ** 以下の部分を除いてほとんど変更なし **

    static final Image image
                    = Toolkit.getDefaultToolkit().getImage("RedMonster.png");

    @Override
    Image getImage() { return image; }   // 画像オブジェクト取得の実装
}
```

図 26.12: AWT/Swing を利用するダンジョンゲームのプログラム（レッドモンスター）

26.8　レッドモンスタークラスの変更

図 26.12（476 ページ）はこの章のレッドモンスタークラスのプログラムです。ここでは画像 を用います。

ところで、クラス-サブクラス関係にある Monster クラスと RedMonster クラスの両方で同じ名前のクラス変数 image を用いています。これは、15 章（220 ページ）の「クラスの継承関係の上下で同じ名前の変数名を使うことは避けるべき」と述べたことに違反するのですが、とは言え、あえて別の名前を使用するほどの必要性が認められないため、ここでは image を重複使用しました。

```
abstract class Cell {
    static final int SIZE = 48; // セルを描画する正方形の一辺の長さ

    boolean isRockCell() { return false; }  // 変更なし

    abstract Cell affectOn(Actor a);        // 変更なし

    abstract void paintAt(Graphics g, Int2 position); // 抽象メソッド

    void drawCell(Graphics g, Int2 position) {  // 正方形の枠線を描く
        g.drawRect(position.x*SIZE + SIZE/24,
                   position.y*SIZE + SIZE/24,
                   SIZE*22/24,
                   SIZE*22/24);
    }

    void fillCell(Graphics g, Int2 position) {  // 正方形の内部を塗る
        g.fillRect(position.x*SIZE + SIZE/24,
                   position.y*SIZE + SIZE/24,
                   SIZE*22/24,
                   SIZE*22/24);
    }
}

abstract class SimpleCell extends Cell { ** 前章から変更なし ** }

abstract class TrapCell extends Cell { ** 前章から変更なし ** }
```

図 26.13: AWT/Swing を利用するダンジョンゲームのプログラム（セル、その 1）

26.9　セルクラスの変更

図 26.13（477 ページ）はセルの抽象クラスのプログラムです。

前章までのメソッド `cellPrint()` は削除します。代わりに抽象メソッド `paintAt()` を用います。

メソッド `drawCell()` は、枠線のみを引く `EmptyCell` の描画に用いられます。ひとつのセル内の細かい寸法は図 26.14（478 ページ）を参考にしてください。メソッド `fillCell()` は、枠内を塗りつぶす `RockCell`、`PotionCell`、`PoisonCell`、`GoalCell` の描画に用いられます。セル内の寸法は図 26.14 と同じです。

`drawCell()` と `fillCell()` は各サブクラスの `paintAt()` から呼び出されます。

図 26.15（478 ページ）に `RockCell`、`PoisonCell` のプログラムを載せます。その他のセルについては省略しました。章末のプログラムを参照してください。

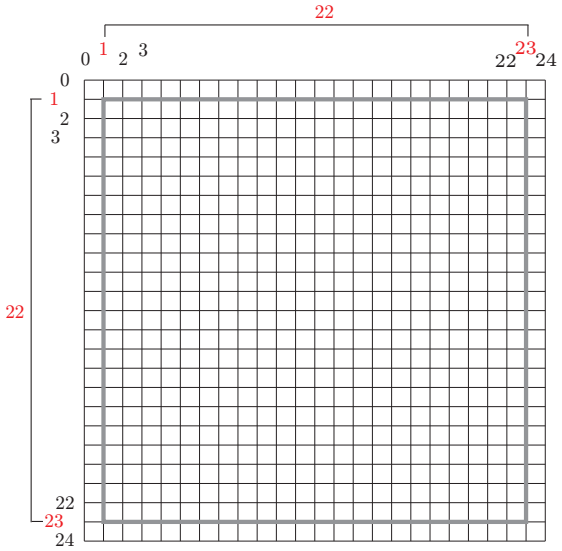


図 26.14: セル上での床の描画サイズ（ゴール、岩、ポーション、ポイズンの各セルも同様）

```
class RockCell extends SimpleCell {
    @Override
    void paintAt(Graphics g, Int2 position) {
        g.setColor(new Color(100, 20, 20)); // 茶色
        fillCell(g, position);
    }

    @Override
    boolean isRockCell() { ** 前章から変更なし }
}

class PoisonCell extends TrapCell {
    @Override
    void paintAt(Graphics g, Int2 position) {
        if (visible) {
            g.setColor(new Color(0, 128, 0)); // 緑色
            fillCell(g, position);
        } else {
            g.setColor(new Color(128, 128, 128));
            drawCell(g, position);
        }
    }

    @Override
    Cell affectOn(Actor a) { ** 前章から変更なし }
}
```

図 26.15: AWT/Swing を利用するダンジョンゲームのプログラム（セル、その 2）具象クラスの一部を掲載、その他も同様。付録のプログラムを参照のこと

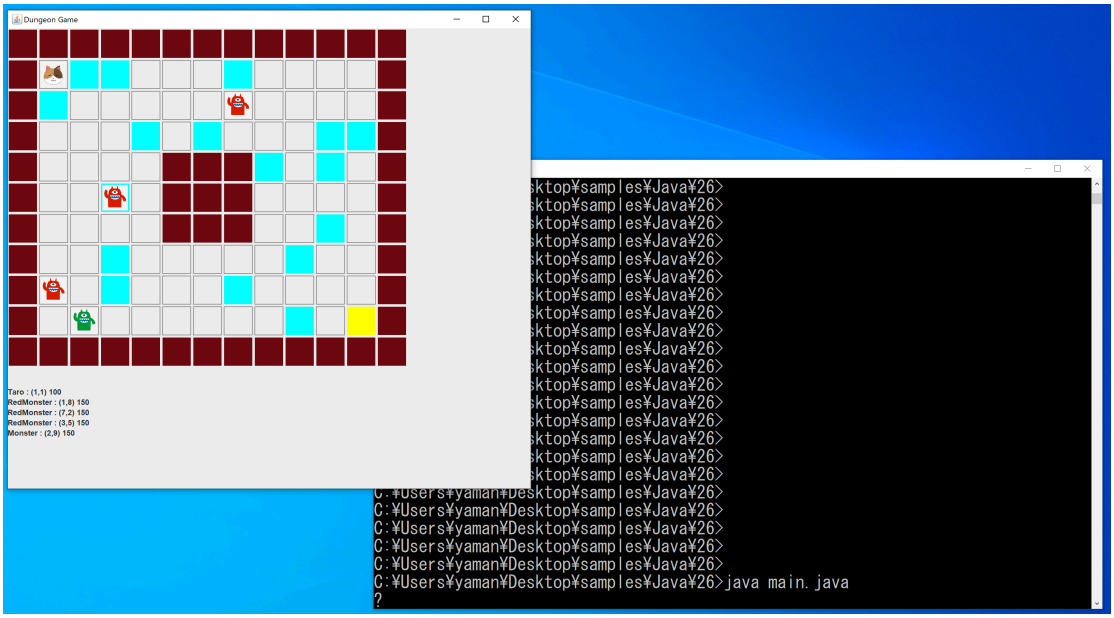


図 26.16: ゲーム実行直後のディスプレイの様子

26.10　プログラムの実行例

図 26.16（479 ページ）はコマンドプロンプトからプログラムを立ち上げた直後のディスプレイの様子です。

ゲームの入力方法は前章から変更していませんから、ここでたとえば右へ移動するキー（'d' とエンターキー）を打鍵してみましょう。しかしプレイヤーは動きません。なぜ、動かないのでしょうか。

26.10.1　キーボード入力のフォーカス

コンピュータディスプレイには通常、複数のウインドウが開かれており、それぞれのウインドウはユーザからの入力待ち状態にあります。入力にはマウスカーソルの移動、マウスのクリック、キーボード入力などがありますが、そのような入力情報は同時に全てのウインドウへ伝えられているのではなく、特定のウインドウが情報を受け取るように約束されているのが、現在のコンピュータのウインドウ管理システムの動作です。このとき、「そのウインドウは**フォーカス**（focus）を持っている」と言います。フォーカスを別のウインドウへ移動させる方法は OS や入力の種類によって様々です。しかし、キーボード入力に限って言えば、フォーカスを与えたいウインドウを最前面に表示させることでフォーカスが移動します。

上のような事項はわざわざ明文化して説明するまでもなく、ウインドウシステムを少し触ってみ

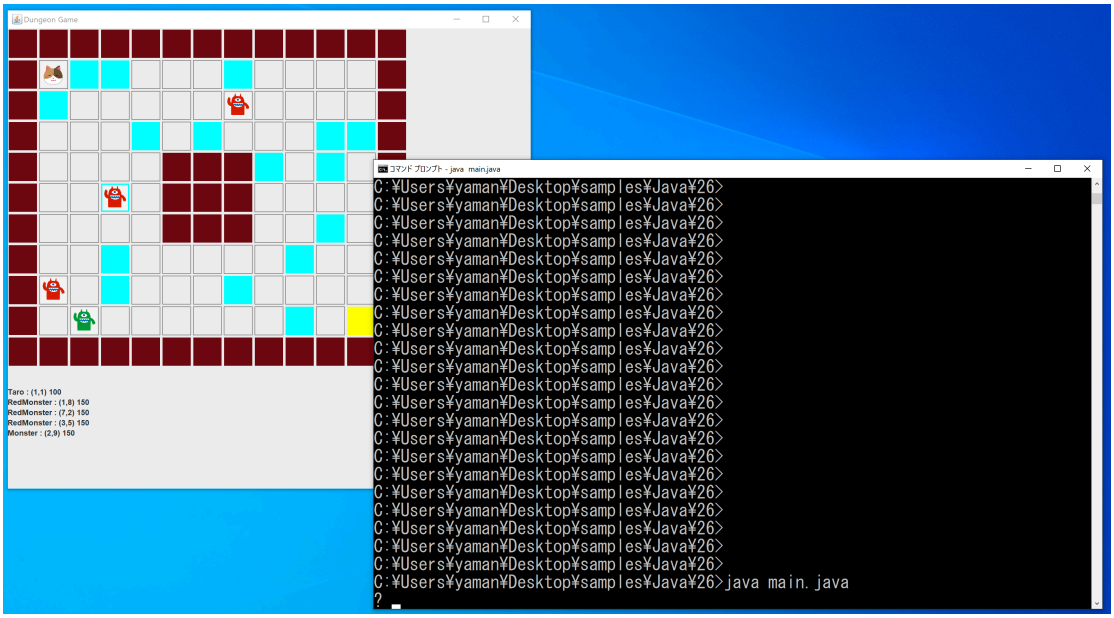


図 26.17: フォーカスをコマンドプロンプトへ移動させた状況

れば自然に体得できるものですが、ウインドウを使ったプログラムを開発する上でははっきりと認識しておく必要があります。

前置きが長くなりましたが、図 26.16 の状況ではキーボード入力のフォーカスはゲーム画面にあり、コマンドプロンプトにありません。ゲームを進めるためにはコマンドプロンプトにフォーカスを移動させる必要がありますから、コマンドプロンプトのウインドウをクリックして最前面へ移動させます。図 26.17（480 ページ）はその直後の様子です。

そうすると、コマンドプロンプト、すなわち実行中のゲームプログラムの標準入力はキーボード入力を受け取りますから、たとえば 'd' とエンターキーを入力した後にゲームが 1 ターン進んだ後のディスプレイの様子は図 26.18（481 ページ）です。

ゲームを終了させるときにはコマンドプロンプト画面で control-c を打鍵（control キーと c キーを同時に打鍵）するか、あるいはゲームウインドウの右上の × をクリックします。そうすると図 26.4（465 ページ）で以下の設定：

```
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  //
```

を行なっていたため、ウインドウが閉じると同時にプログラム全体も強制終了します。

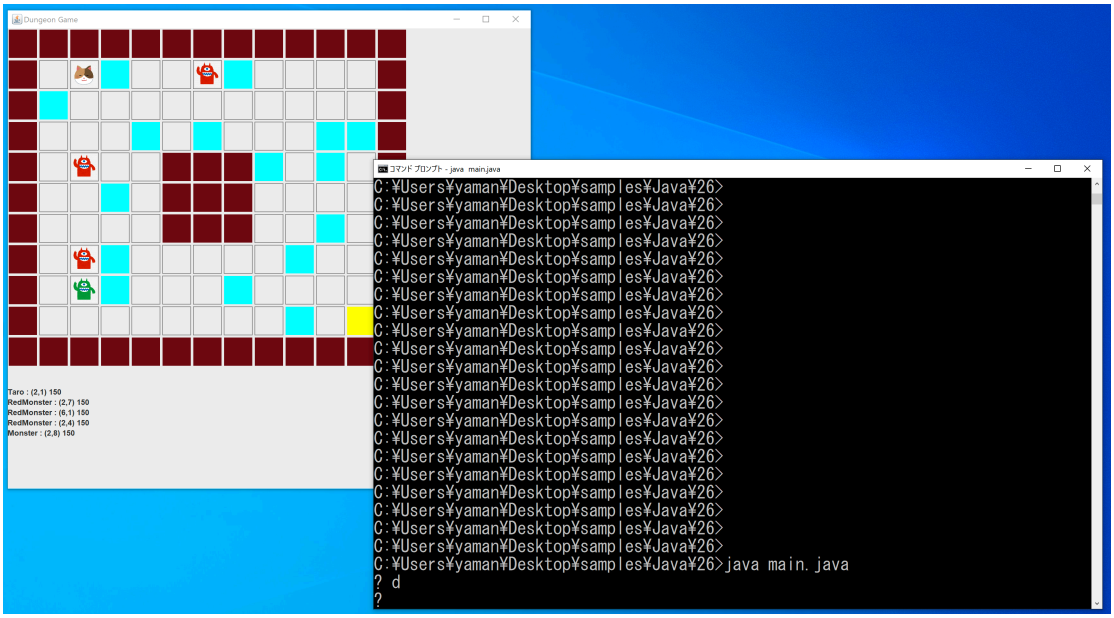


図 26.18: キー 'd' を入力した直後のゲームの様子

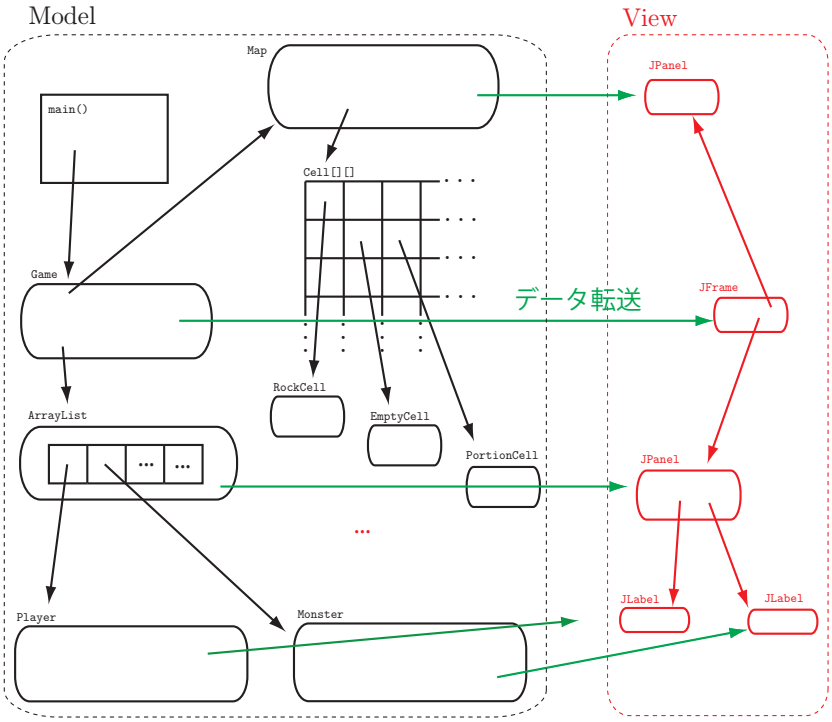


図 26.19: モデルとビューをオブジェクトとして分離する場合のプログラム構造：このテキストでは採用しませんが、分かりやすい構造です。なお、右側の JFrame、JPanel などは実際にはサブクラス化する必要があります（25.6 節（454 ページ）参照）。

26.11　補足：もうひとつのプログラム構造

ダンジョンゲームに AWT/Swing の GUI 部品を追加する方法として図 26.19（482 ページ）のように、モデルとビューを分離する方法も考えられます。

つまり、前章までのプログラムからプリント関係を削除し、代わりに JFrame、JPanel（のサブクラス）のオブジェクトをモデルから分離して管理し、ゲームの表示をそちらに完全に任せます。うまく行きそうですし、分かりやすく、また将来、ビュー部分を入れ替える可能性があるならば、この方法ならばビュー部分の置換が容易そうです。

しかし、図 26.19 からも分かるように、実は Game オブジェクトを頂点とするモデル部分の構造と、JFrame オブジェクト（のサブクラス）を頂点とするビュー部分の構造はほぼ同じ形状ですから、モデルとビューをクラス内に同居させてもプログラムが不自然に複雑になることもありません。また分離したことでモデルからビューへのデータ転送処理をプログラム化する必要もあります。その辺を考慮し、講義テキストでは分離する方法が採用しませんでした。

26.12　付録：この章の全プログラムリスト

```
//ゲームクラスのプログラム
import java.io.*;
import java.util.Random;
import java.util.ArrayList;
import javax.swing.*;
import java.awt.*;

class Game extends JFrame {          // JFrame クラスのサブクラスとして宣言
    public static void main(String args[]) {
        Game game = new Game();
        game.mainLoop();
    }

    static Random random = new Random();

    Map map;
    ArrayList<Actor> actors;
    final static int PLAYER_ID = 0;
    final static int NUM_OF_MONSTERS = 4;

    Game() {
        super("Dungeon Game");                      // JFrame 部分の初期化

        map = new Map(this);
        actors = new ArrayList<Actor>();
        actors.add(new Player(this, "Taro"));
        for (int i = 0; i < NUM_OF_MONSTERS; i++) {
            if (random.nextInt(2) == 0) {
                actors.add(new Monster(this));
            }
            else {
                actors.add(new RedMonster(this));
            }
        }

        setBounds(100, 100, Cell.SIZE*Map.XSIZE+5+200,   // ウインドウの
                            Cell.SIZE*Map.YSIZE+25+200); // 設定
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  //

        add(map, BorderLayout.NORTH);    // map オブジェクトを JFrame に配置

        JPanel actorsPanel = new JPanel();      // もう一枚の JPanel の生成
        actorsPanel.setLayout(new GridLayout(10,1));  // 縦一列にレイアウト
        for (Actor a : actors) { actorsPanel.add(a); }
                                                // 各アクターをパネルに配置
        add(actorsPanel, BorderLayout.SOUTH);// actorsPanel を JFrame に配置

        setVisible(true);
    }

    void print() { repaint(); }       // JFrame へウインドウの再描画の依頼

    void mainLoop() {
        print();
        while (true) { turn(); }
    }
```

```
1
2       void turn() {
3           for (Actor a : actors) { a.action(); }
4           print();
5       }
6
7       static char getKeyChar() {
8           char c = 0;
9           System.out.print("? ");
10          try {
11              BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
12              c = in.readLine().charAt(0);
13          } catch (IOException e) {
14              System.out.println("ERROR:: getKeyChar()");
15          }
16          return c;
17      }
18  }
```

```
//マップクラスのプログラム
class Map extends JPanel {                          // サブクラス化
    static final int XSIZE = 13;
    static final int YSIZE = 11;

    Game game;
    Cell data[][] = new Cell[XSIZE][YSIZE];

    Map(Game game) {
        this.game = game;
        for (int x = 0; x < XSIZE; x++) {
            for (int y = 0; y < YSIZE; y++) {
                if ((x == 0)||(x == XSIZE - 1)||
                    (y == 0) || (y == YSIZE - 1)||
                    ((Math.abs(x - XSIZE / 2) <= 1)&&
                     (Math.abs(y - YSIZE / 2) <= 1))) {
                    data[x][y] = new RockCell();
                }
                else {
                    switch (Game.random.nextInt(5)) {
                    case 0: data[x][y] = new PotionCell();
                            break;
                    case 1: data[x][y] = new PoisonCell();
                            break;
                    default: data[x][y] = new EmptyCell();
                    }
                }
            }
        }
        data[XSIZE-2][YSIZE-2] = new GoalCell();

        setPreferredSize(new Dimension(Cell.SIZE * XSIZE,
                                       Cell.SIZE * YSIZE));
                                                   // パネルの希望サイズを設定
    }

    Cell getAt(Int2 position) { return data[position.x][position.y]; }

    void setAt(Int2 position, Cell cell) {
        data[position.x][position.y] = cell;
    }

    @Override
    public void paintComponent(Graphics g) {     // JPanel のメソッドを上書き
        super.paintComponent(g);         // とりあえず JPanel のメソッドを実行
        for (int y = 0; y < YSIZE; y++) {
            for (int x = 0; x < XSIZE; x++) {
                Int2 position = new Int2(x, y);
                getAt(position).paintAt(g, position); // Cell をパネルに表示
                for (Actor a : game.actors) {
                    a.paintAt(g, position);       // 各アクターをパネルに表示
                }
            }
        }
    }

    Int2 getRandomCellPosition() {
        while (true) {
```

```
            Int2 position = new Int2(Game.random.nextInt(XSIZE),
                                     Game.random.nextInt(YSIZE));

            if (!getAt(position).isRockCell()) { return position; }
        }
    }
}
```

```
1  //アタークラスのプログラム
2  abstract class Actor extends JLabel {   // サブクラス化
3      Game game;
4      Int2 position;
5      int hp;
6
7      Actor(Game game, Int2 position, int hp) { // cellChar を削除
8          super(" ");                           // ラベルの初期値は " "
9          this.game = game;
10         this.position = position;
11         this.hp = hp;
12     }
13
14     Actor(Game game, int hp) {
15         super(" ");
16         this.game = game;
17         this.hp = hp;
18         while_loop: while (true) {
19             position = game.map.getRandomCellPosition();
20             for (Actor a : game.actors) {
21                 if (position.equal(a.position)) {
22                     continue while_loop;
23                 }
24             }
25             return;
26         }
27     }
28
29     @Override
30     public void paintComponent(Graphics g) { // アクターの状態の出力
31         super.paintComponent(g);         // とりあえず JLabel のメソッドを実行
32         setText(toString());           // ラベルの内容をアクターの状態で更新
33     }
34
35    void paintAt(Graphics g, Int2 position) { // アクターをマップに表示
36         if (this.position.equal(position)) {
37             g.drawImage(getImage(),                          // 画像
38                         position.x*Cell.SIZE + Cell.SIZE*2/24,// x 座標
39                         position.y*Cell.SIZE + Cell.SIZE*2/24,// y 座標
40                         Cell.SIZE*20/24,                     // 水平サイズ
41                         Cell.SIZE*20/24,                     // 垂直サイズ
42                         game.map);                          // オブザーバー
43         }
44     }
45
46     abstract Image getImage();         // アクターの画像を求める抽象メソッド
47
48     boolean moveTo(Int2 direction) {
49         Int2 target = position.add(direction);
50
51         Cell cell = game.map.getAt(target);
52         if (cell.isRockCell()) { return false; }
53
54         for (Actor a : game.actors) {
55             if (target.equal(a.position)) { return false; }
56         }
57
58         Cell newCell = cell.affectOn(this);
```

```
        game.map.setAt(target, newCell);
        position = target;
        return true;
    }

    abstract void action();

    void attackTo(Int2 direction) {
        Int2 target = position.add(direction);
        for (Actor a : game.actors) {
            if (target.equal(a.position)) {
                a.takeDamageFrom(this);
                return;
            }
        }
    }

    abstract void takeDamageFrom(Actor a);
    void giveDamageTo(Player p) {}
    void giveDamageTo(Monster m) {}
    void giveDamageTo(RedMonster r) {}
}
```

```
//プレイヤークラスのプログラム
class Player extends Actor {
    static final int INIT_HP = 100;
    String name;

    Player(Game game, String name) {
        super(game, new Int2(1, 1), INIT_HP);
        this.name = name;
    }

    Player(Game game) { this(game, "Player"); }

    public String toString() { return name+" : "+position+" "+hp; }

    @Override
    void takeDamageFrom(Actor a) { a.giveDamageTo(this); }

    @Override
    void giveDamageTo(Monster m) { m.hp -= 20; }

    @Override
    void giveDamageTo(RedMonster r) { r.hp -= 15; }

    @Override
    void action() {
        Int2 direction;
        switch (Game.getKeyChar()) {
            case 'z': direction = new Int2(-1, +1); break;
            case 'x': direction = new Int2( 0, +1); break;
            case 'c': direction = new Int2(+1, +1); break;
            case 'a': direction = new Int2(-1,  0); break;
            case 'd': direction = new Int2(+1,  0); break;
            case 'q': direction = new Int2(-1, -1); break;
            case 'w': direction = new Int2( 0, -1); break;
            case 'e': direction = new Int2(+1, -1); break;
            default: direction = new Int2(0, 0);
        }

        if (!moveTo(direction)) { attackTo(direction); }
    }

    static final Image image1
                   = Toolkit.getDefaultToolkit().getImage("Player1.png");
    static final Image image2
                   = Toolkit.getDefaultToolkit().getImage("Player2.png");

    @Override
    Image getImage() {                      // 画像オブジェクト取得の実装
        if (hp > INIT_HP/2) { return image1; } // 元気ならば
        else { return image2; }                // 弱ってきたならば
    }
}
```

```
//モンスタークラスのプログラム
class Monster extends Actor {
    static final int INIT_HP = 150;

    Monster(Game game) { super(game, INIT_HP); }

    @Override
    public String toString() { return "Monster : "+position+" "+hp; }

    @Override
    void takeDamageFrom(Actor a) { a.giveDamageTo(this); }

    @Override
    void giveDamageTo(Player p) { p.hp -= 30; }

    @Override
    void giveDamageTo(RedMonster r) { r.hp -= 5; }

    @Override
    void action() {
        Actor player = game.actors.get(game.PLAYER_ID);
        Int2 direction = player.position.sub(position).signum();

        if (!moveTo(direction)) {
            attackTo(direction);
        }
    }

    static final Image image
                   = Toolkit.getDefaultToolkit().getImage("Monster.png");

    @Override
    Image getImage() { return image; }    // 画像オブジェクト取得の実装
}
```

```
//レッドモンスタークラスのプログラム
class RedMonster extends Monster {
    RedMonster(Game game) { super(game); }

    @Override
    public String toString() { return "Red" + super.toString(); }

    @Override
    void takeDamageFrom(Actor a) { a.giveDamageTo(this); }

    @Override
    void giveDamageTo(Player p) { p.hp -= 60; }

    @Override
    void giveDamageTo(Monster m) { m.hp -= 7; }

    static final Image image
                   = Toolkit.getDefaultToolkit().getImage("RedMonster.png");

    @Override
    Image getImage() { return image; }   // 画像オブジェクト取得の実装
}
```

```
//2 次元ベクトルクラスのプログラム
class Int2 {
    int x,y;

    Int2(int x, int y) { this.x = x; this.y = y; }
    Int2 add(Int2 a) { return new Int2(x+a.x,y+a.y); }
    Int2 sub(Int2 a) { return new Int2(x-a.x,y-a.y); }
    Int2 signum() { return new Int2(Integer.signum(x),Integer.signum(y)); }
    boolean equal(Int2 a) { return (x == a.x)&&(y == a.y); }

    public String toString(){ return "("+x+","+y+")"; }
}
```

```
//セルクラスのプログラム
abstract class Cell {
    static final int SIZE = 48; // セルを描画する正方形の一辺の長さ

    boolean isRockCell() { return false; }  // 変更なし

    abstract Cell affectOn(Actor a);        // 変更なし

    abstract void paintAt(Graphics g, Int2 position); // 抽象メソッド

    void drawCell(Graphics g, Int2 position) {  // 正方形の枠線を描く
        g.drawRect(position.x*SIZE + SIZE/24,
                   position.y*SIZE + SIZE/24,
                   SIZE*22/24,
                   SIZE*22/24);
    }

    void fillCell(Graphics g, Int2 position) {  // 正方形の内部を塗る
        g.fillRect(position.x*SIZE + SIZE/24,
                   position.y*SIZE + SIZE/24,
                   SIZE*22/24,
                   SIZE*22/24);
    }
}

abstract class SimpleCell extends Cell {
    @Override
    Cell affectOn(Actor a) { return this; }
}

abstract class TrapCell extends Cell {
    boolean visible = false;

    @Override
    Cell affectOn(Actor a) {
        visible = true;
        return this;
    }
}

class RockCell extends SimpleCell {
    @Override
    void paintAt(Graphics g, Int2 position) {
        g.setColor(new Color(100, 20, 20)); // 茶色
        fillCell(g, position);
    }

    @Override
    boolean isRockCell() { return true; }
}

class EmptyCell extends SimpleCell {
    @Override
    void paintAt(Graphics g, Int2 position) {
        g.setColor(new Color(128, 128, 128)); // 灰色
        drawCell(g, position);
    }
}
```

```
1
2  class PotionCell extends SimpleCell {
3      @Override
4      void paintAt(Graphics g, Int2 position) {
5          g.setColor(new Color(0, 255, 255)); // 空色
6          fillCell(g, position);
7      }
8
9      @Override
10     Cell affectOn(Actor a) {
11         a.hp += 50;
12         return new EmptyCell();
13     }
14 }
15
16 class GoalCell extends SimpleCell {
17     @Override
18     void paintAt(Graphics g, Int2 position) {
19         g.setColor(new Color(255, 255, 0)); // 黄色
20         fillCell(g, position);
21     }
22
23     @Override
24     Cell affectOn(Actor a) {
25         if (a instanceof Player) {
26             System.out.println("YOU WIN");
27             System.exit(0);
28         }
29         return this;
30     }
31 }
32
33 class PoisonCell extends TrapCell {
34     @Override
35     void paintAt(Graphics g, Int2 position) {
36         if (visible) {
37             g.setColor(new Color(0, 128, 0)); // 緑色
38             fillCell(g, position);
39         } else {
40             g.setColor(new Color(128, 128, 128));
41             drawCell(g, position);
42         }
43     }
44
45     @Override
46     Cell affectOn(Actor a) {
47         a.hp -= 10;
48         return super.affectOn(a);
49     }
50 }
```

1

1

第27章　エンターキーを使わない文字入力を実装する：KeyListenerの利用

前章でゲームの表示部（MVC のビュー）を GUI 化しましたが、入力部（MVC のコントローラ）はまだ CUI のまま、コマンドプロンプトからの入力でした。よって文字入力の度にエンターキーの打鍵が必要です。また、フォーカスをゲームウインドウからコマンドプロンプト へ切り替える必要もあります。

このような不都合をこの章と次章で解決します。ユーザからの入力をゲームウインドウと一体化する改造です。この章ではキー入力を扱います。次章ではさらにボタン入力を扱います。

改造のポイントは**イベントリスナー**（event listener ＝イベントを監視しているソフトウェア部品）の利用です。イベントとは、マウスのクリック、マウスカーソルの移動、キー入力などのコンピュータの状態の変化です。この章では、数あるイベントの中から `KeyListener`（キーリスナー＝キー入力を監視するソフトウェア部品）を用います。

なお、イベント回りの処理方式はシステムによって様々です。Java AWT/Swing の方式はこのテキストで述べる通りですが、かつての Java（1990 年代半ば）では AWT だけでイベント処理を行なっていました。他にも、たとえば iPhone のアプリ（iOS アプリ）のプログラミングはここで述べる方法と異なります[1]。統一されているとは言えない状況です。このテキストで述べる方式はイベント処理がどのようなものかを学習する上で役に立ちますが、イベント処理の**一方式に過ぎない**と理解してください。

27.1　簡単な例題（その 1）

図 27.1（498 ページ）は、簡単なキーリスナーのサンプルプログラムです。

プログラムではまず、パッケージ読み込みのため、以下を追加します。

```
import java.awt.event.*;
```

[1]iOS の場合、Java ではなく Swift でプログラミングします。言語が違えばオブジェクト指向技術の使い方も違ってきます。

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

class Test {
    public static void main(String[] args) {
        JFrame frame = new JFrame("KeyListener Test");        // Frame の設定
        frame.setBounds(100,100,400,400);                     // Frame の設定
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // Frame の設定
        frame.setVisible(true);                               // Frame の設定

        frame.addKeyListener(new MyKeyListener());      // キーリスナーの登録
    }
}

class MyKeyListener implements KeyListener {          // キーリスナーの実装
    @Override
    public void keyPressed(KeyEvent e) {    // キーが押されたときに呼び出される
        System.out.println("get "+e.getKeyChar());    // 受け取った通知を表示
    }

    @Override
    public void keyReleased(KeyEvent e){}             // 何もしない実装

    @Override
    public void keyTyped(KeyEvent e){}                // 何もしない実装
}
```

図 27.1: KeyListener 理解のサンプルプログラム（その 1）

JFrame オブジェクト frame は前章、前々章と同じように作成します。そして

```
        frame.addKeyListener(new MyKeyListener());      // キーリスナーの登録
```

によって、キーリスナーオブジェクト new MyKeyListener() を frame に登録しておきます。登録できるキーリスナーオブジェクトは、インターフェース KeyListener を実装するクラスのオブジェクトに限られます。

　さて、いよいよインターフェースを用います。インターフェースの復習は 21.2 節（358 ページ）です。このインターフェース KeyListener は以下の様に定義されています。

```
interface KeyListener {
    void keyPressed(KeyEvent e);
    void keyReleased(KeyEvent e);
    void keyTyped(KeyEvent e);
}
```

これを同等の抽象クラスに書き換えれば以下の通りです。

```
abstract class KeyListener {
    public abstract void keyPressed(KeyEvent e);
    public abstract void keyReleased(KeyEvent e);
    public abstract void keyTyped(KeyEvent e);
}
```

よって、 JFrame オブジェクトに登録するキーリスナーオブジェクトは上の三つ全ての抽象メソッドを実装するクラスのオブジェクトでなければなりません。図 27.1 の MyKeyListener クラスはまさにそのようなクラスになっています。ここに、三つのメソッドの意味は以下の通りです。

void keyPressed(KeyEvent e) ... キーが押されたことが感知されたならば、JFrame オブジェクトに登録されているキーリスナーオブジェクトのこのメソッドが実行されます。引数 e は押されたキーに関する情報を持つオブジェクトです。

void keyReleased(KeyEvent e) ... キーが（押された状態から）元に戻されたことが感知されたならば、登録されているキーリスナーオブジェクトのこのメソッドが実行されます。引数 e はキーに関する情報を持つオブジェクトです。

void keyTyped(KeyEvent e) ... キーボードから 1 文字が入力されたことが感知されたならば、登録されているキーリスナーオブジェクトのこのメソッドが実行されます。引数 e はキーに関する情報を持つオブジェクトです。このメソッドの動作は少し複雑です。たとえばシフトキーを押しながら ‘s’ を押して、結果として大文字 ‘S’ が入力されたときには、‘S’ の 1 文字の入力があったと感知されます.。この 1 文字の入力の間に keyPressed()、ketReleased() はそれぞれ 2 回感知されます。keyPressed()、ketReleased() が素朴なイベントの感知であるのに対して keyTyped() は複合的なイベントの感知になりますから使い方がやや難しいかもしれません。keyTyped() を利用するときには、しっかりと仕様を調査してください。

図 27.1 の MyKeyListener では keyPressed() の中で入力された文字を標準出力へ表示する処理：

```
        System.out.println("get "+e.getKeyChar());     // 受け取った通知を表示
```

を行なっています。ここに e.getKeyChar() は、キー情報オブジェクト e から押されたキー文字を求めるメソッド呼び出しです。図 27.1 の ketReleased()、keyTyped() は何もしない本体を実

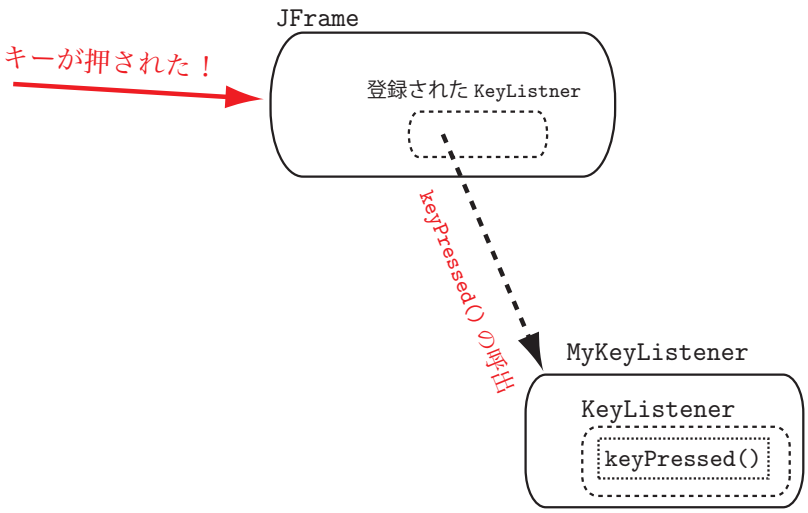


図 27.2: JFrame と KeyListener の関係（その 1）

装しています[2]。

図 27.2（500 ページ）は一連の図式化です。JFrame オブジェクトの中に登録されたキーリスナーオブジェクトを保持するバッファエリアがあり、図では MyKeyListener オブジェクト（の参照）が保持されています。MyKeyListener オブジェクトの中核部には KeyListener オブジェクトの部分があり、そこに keyPressed() が定義されています。そして、この JFrame オブジェクトがキー入力フォーカスを持っているときにキーの打鍵があると、その情報が自動的に JFrame オブジェクトへ伝えられ、 JFrame オブジェクトに登録されたキーリスナーオブジェクトの keyPressed() を呼び出します。キーリスナーオブジェクトが複数の場合には全てのオブジェクトで keyPressed() が呼び出されます。

図 27.3（501 ページ）はプログラムの実行の様子です。アプリケーションウインドウが最前面にあり、フォーカスを持っていることを確認してください。キーボードの ‘a’、‘b’、‘c’ のキーを押した（このとき、エンターキーは不要です）ため、コマンドプロンプト に “get a”、“get b”、“get c” が表示されています。

このキーリスナーを応用すれば前章のフォーカスの問題は解決します。ただし、ダンジョンゲームに組み込むには図 27.1 のプログラムやや素朴です。そこでもう少し図 27.1 のプログラムを洗練していきます。

[2]何もしないとしても定義する必要があります。

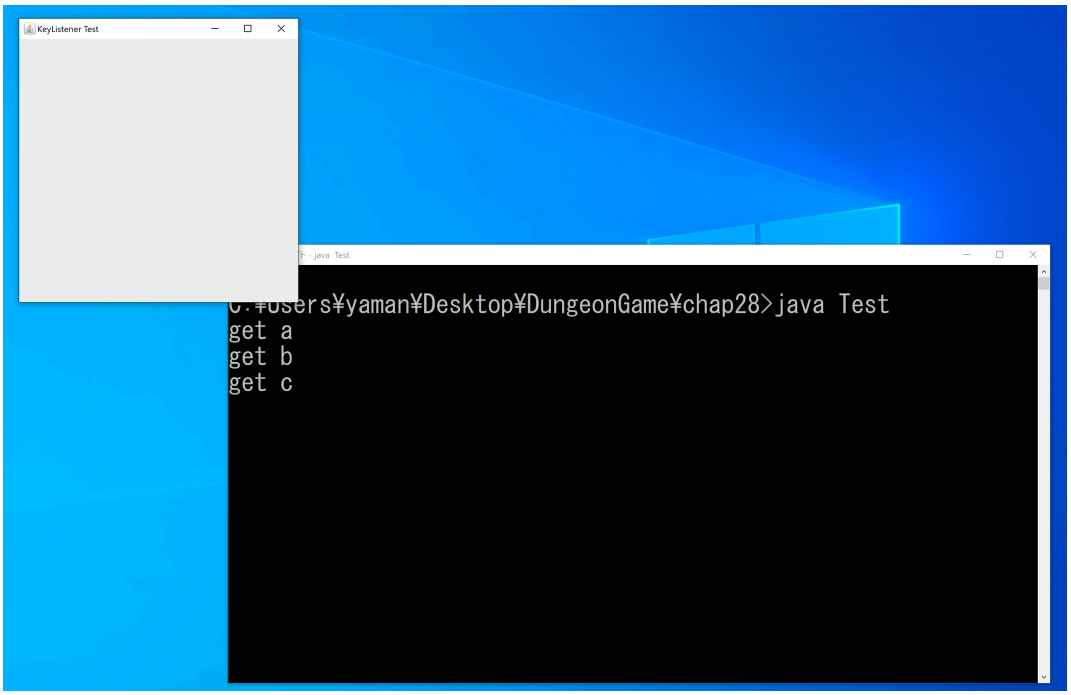


図 27.3: 図 27.1 のプログラムの実行の様子

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

class Test extends JFrame {                          // サブクラス化
    public static void main(String[] args) {
        new Test();             // main() での処理はオブジェクトを作るだけでよい
    }

    Test() {                    // コンストラクタで大半の処理を行う
        super("KeyListener Test");
        setBounds(100,100,400,400);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);

        addKeyListener(new MyKeyListener());
    }
}

class MyKeyListener implements KeyListener {
    @Override
    public void keyPressed(KeyEvent e) {
        System.out.println("get "+e.getKeyChar());
    }

    @Override
    public void keyReleased(KeyEvent e){}

    @Override
    public void keyTyped(KeyEvent e){}
}
```

図 27.4: KeyListener 理解のサンプルプログラム（その 2）

27.2　簡単な例題（その 2）

前節のプログラムを図 27.4（502 ページ）のように改造しましょう。ポイントは Test クラスを JFrame クラスのサブクラスとしている点です。これと同様のプログラムの改造を、図 25.14（452 ページ）から図 25.16（455 ページ）に改造したときに行いました。

図 27.5（503 ページ）はその図式です。

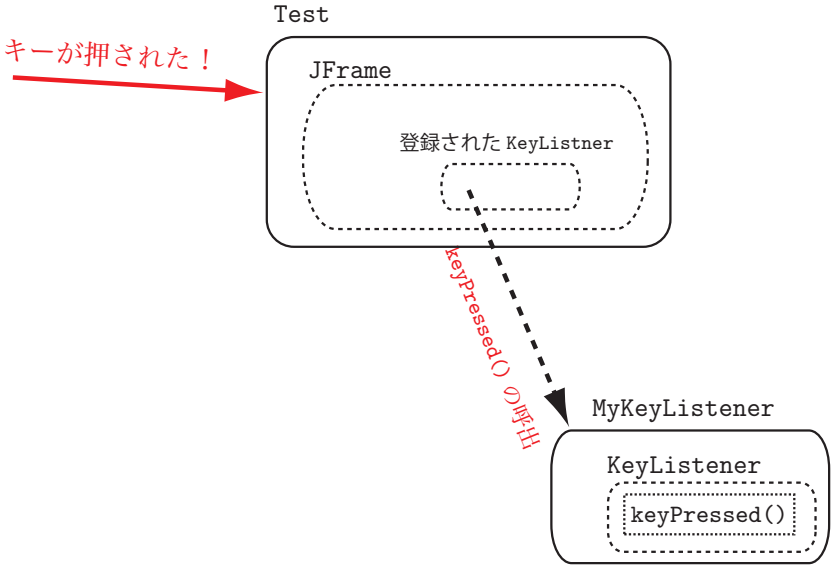


図 27.5: JFrame と KeyListener の関係（その 2）

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

class Test extends JFrame implements KeyListener { // JFrame かつ KeyListener
    public static void main(String[] args) {
        new Test();
    }

    Test() {
        super("KeyListener Test");
        setBounds(100,100,400,400);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);

        addKeyListener(this);    // 自分自身を登録
    }

    @Override
    public void keyPressed(KeyEvent e) {
        System.out.println("get "+e.getKeyChar());
    }

    @Override
    public void keyReleased(KeyEvent e){}

    @Override
    public void keyTyped(KeyEvent e){}
}
```

図 27.6: KeyListener 理解のサンプルプログラム（その 3）

27.3　簡単な例題（その 3）

さらなる改造は Test クラスを JFrame クラスのサブクラス、かつ KeyListener インターフェースの実装クラスに変更する改造です。

まず、以下に注意してください。

```
class Test extends JFrame implements KeyListener { // JFrame かつ KeyListener
```

Test オブジェクトに JFrame オブジェクトと KeyListener オブジェクトの両方の役割を押し込めた形になっています[3]。図 27.7（505 ページ）はその図式化です。

以下も注意すべき点です。JFrame に KeyListener オブジェクトを登録するところですが、自分自身（this）を登録しています。

[3]関連内容（多重継承）については 23.2 節（404 ページ）で解説しました。そちらを復習してください。

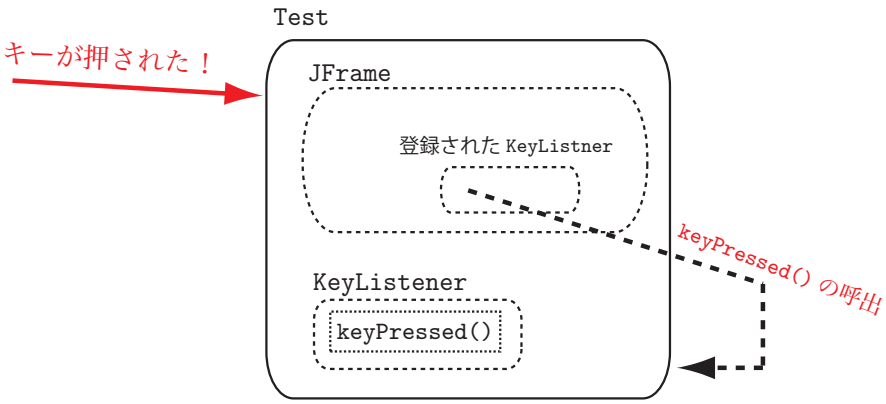


図 27.7: JFrame と KeyListener の関係（その 3）

```
addKeyListener(this);     // 自分自身を登録
```

キー入力イベントが発生すると、JFrame から自分自身の keyPressed() メソッドが呼び出されることになります。前節までは、キー入力処理を二つのクラスで行いましたが、ひとつのクラスにまとめた訳です。それほど大きくない処理ならば、まとめても問題ないでしょう。

ここまで示した三つのプログラム例はいずれも同じ動作を行うものです。どのプログラムが優れているか議論の分かれるところですが、一般には 3 番目のプログラムが好まれます。キー入力自体は小さな処理ですから、その小さな処理のためにわざわざクラスをひとつ新設するのは大袈裟と考えられるためです。

ダンジョンゲームにおいても 3 番目の手法を適用します。

27.4　プログラムの構造

図 27.8（506 ページ）はこの章のプログラムの構造です。かなり複雑になってきました。赤い部分が、前節の図 27.7 の構造をそのまま Game オブジェクトへ実装した部分です。つまり、ゲームオブジェクトは KeyListener を実装し、かつゲームオブジェクトに含まれる JFrame オブジェクトにゲームオブジェクト（自分自身）を KeyListener として登録します。

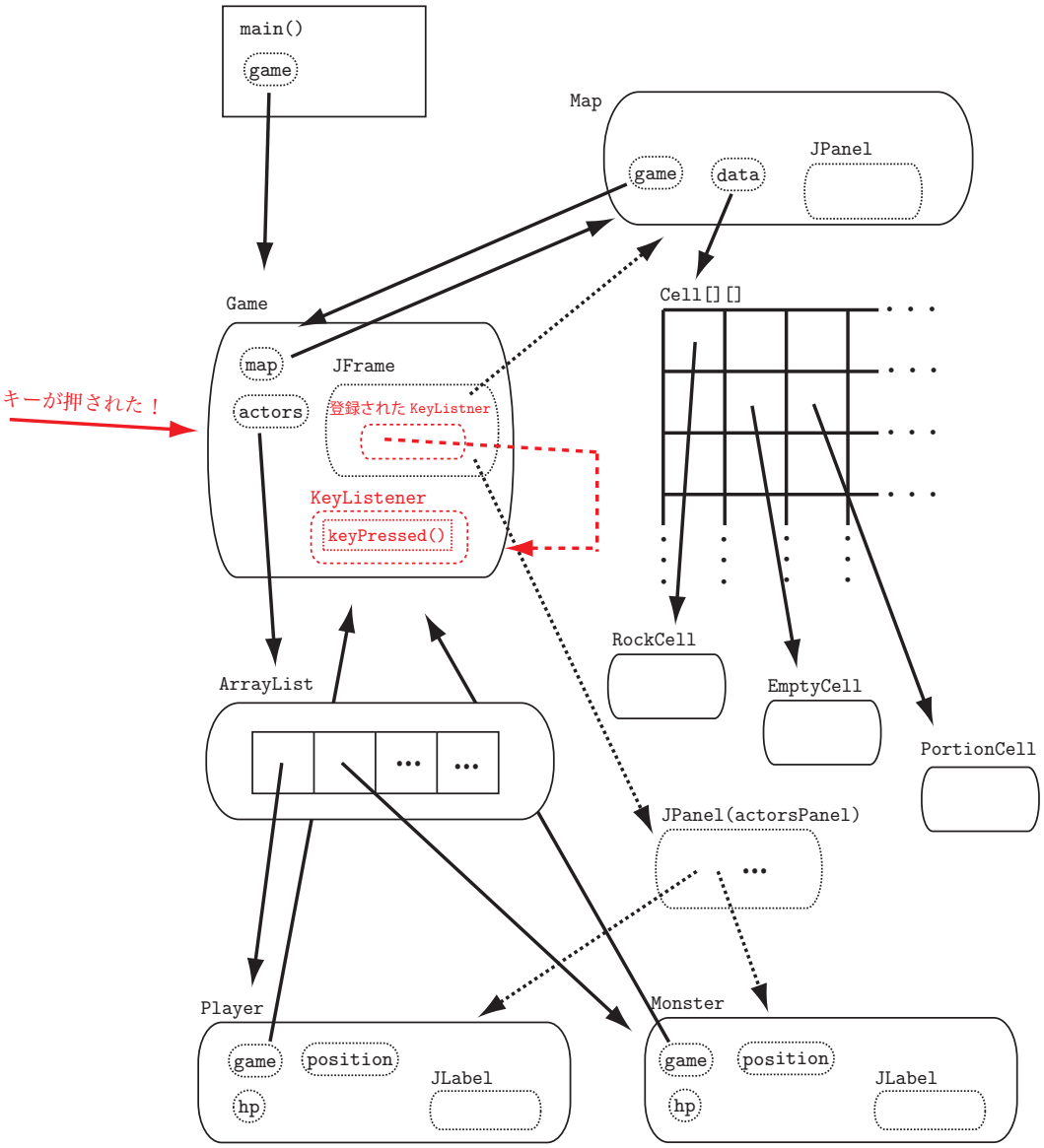


図 27.8: キーリスナーを用いるプログラムの全体構造：キーが押されたことを Game オブジェクトが感知し、自分自身へ伝えます。結果、KeyPressed() が実行されます。

```
class Game extends JFrame implements KeyListener {//KeyListener の実装の追加
    public static void main(String args[]) {
        Game game = new Game();
        // game.mainLoop(); の削除
    }

    ** インスタンス変数などは前章から変更なし **

    Game() {
        ** 以下を追加する以外は前章から変更なし **

        addKeyListener(this);                // KeyListener（自分）の登録
    }

    void print() { ** 前章から変更なし ** }

    void turn() { ** 前章から変更なし ** }            // mainLoop() は削除

    static char keyChar;                              // 入力キー文字を保持

    static char getKeyChar() { return keyChar; }      // 入力文字を取得

    static void setKeyChar(char c) { keyChar = c; }  // 入力文字を保存

    @Override
    public void keyPressed(KeyEvent e) {
        setKeyChar(e.getKeyChar());  // キーを保存し、
        turn();                      // 1 ターンを実行
    }

    @Override
    public void keyReleased(KeyEvent e) {} // 何もしない

    @Override
    public void keyTyped(KeyEvent e) {}    // 何もしない
}
```

図 27.9: KeyListener を利用するダンジョンゲームのプログラム（ゲーム）

27.5　ゲームクラスの変更

図 27.9（507 ページ）が、この章のゲームクラスのプログラムです。

まず、KeyListener の実装をクラス宣言に加えます。

```
class Game extends JFrame implements KeyListener {//KeyListener の実装の追加
```

次にゲームオブジェクト自身をキーリスナーに登録します。

```
        addKeyListener(this);                // KeyListener（自分）の登録
```

以下の二つのクラスメソッドは、クラス変数 keyChar のゲット、セットの専用メソッドです。入力したキー文字は一旦、 keyChar に保持することとします。

```
    static char getKeyChar() { return keyChar; }     // 入力文字を取得
    static void setKeyChar(char c) { keyChar = c; }  // 入力文字を保存
```

なお、これら変数、メソッドを static で修飾することの是非について講義担当者は迷いました。実は static 修飾はインスタンスメソッドとクラスメソッドの違い（C++で言えば、通常のメンバー関数と静的メンバー関数の違い）を理解する目的で導入しました。結論を言えば static 修飾は不要です。しかし、前章までの経緯からそのまま static 修飾を維持することとします。

さて、メソッド keyPressed() では、KeyEvent 型引数 e から文字を取り出し[4]、それを クラス変数 keyChar に保存し、その後に turn() を実行します。

```
        setKeyChar(e.getKeyChar());  // キーを保存し、
        turn();                      // 1 ターンを実行
```

これまで turn() は mainLoop() から実行していましたが、その役割を keyPressed() に置換した訳です。キーが押される度に turn() が実行されるように仕組みを変えます。

ゲームクラス以外のクラスに変更はありません。

27.6　プログラムの実行について

ゲームを立ち上げた後、コマンドプロンプト のウインドウには触らず、プレイヤーの移動を指示するキー（‘q’、‘w’、...、‘c’）を押してください。もはやエンターキーの入力は不要です。プレイヤーがサクサク動きます。

[4] e.getKeyChar() はキー情報 e から押されたキーの文字を取り出すメソッド getKeyChar() の呼び出しです。

27.7　付録：この章の全プログラムリスト

```
//ゲームクラスのプログラム
import java.io.*;
import java.util.Random;
import java.util.ArrayList;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class Game extends JFrame implements KeyListener {//KeyListener の実装の追加
    public static void main(String args[]) {
        Game game = new Game();
    }

    static Random random = new Random();

    Map map;
    ArrayList<Actor> actors;
    final static int PLAYER_ID = 0;
    final static int NUM_OF_MONSTERS = 4;

    Game() {
        super("Dungeon Game");

        map = new Map(this);
        actors = new ArrayList<Actor>();
        actors.add(new Player(this, "Taro"));
        for (int i = 0; i < NUM_OF_MONSTERS; i++) {
            if (random.nextInt(2) == 0) {
                actors.add(new Monster(this));
            }
            else {
                actors.add(new RedMonster(this));
            }
        }

        setBounds(100, 100, Cell.SIZE * Map.XSIZE + 5 + 200, Cell.SIZE * Map.YSIZE + 25 + 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        add(map, BorderLayout.NORTH);

        JPanel actorsPanel = new JPanel();
        actorsPanel.setLayout(new GridLayout(10, 1));
        for (Actor a : actors) { actorsPanel.add(a); }
        add(actorsPanel, BorderLayout.SOUTH);

        setVisible(true);

        addKeyListener(this);                // KeyListener（自分）の登録
    }

    void print() { repaint(); }

    void turn() {
        for (Actor a : actors) { a.action(); }
        print();
    }

```

```
    static char keyChar;                              // 入力キー文字を保持

    static char getKeyChar() { return keyChar; }     // 入力文字を取得

    static void setKeyChar(char c) { keyChar = c; }  // 入力文字を保存

    @Override
    public void keyPressed(KeyEvent e) {
        setKeyChar(e.getKeyChar());  // キーを保存し、
        turn();                      // 1 ターンを実行
    }

    @Override
    public void keyReleased(KeyEvent e) {} // 何もしない

    @Override
    public void keyTyped(KeyEvent e) {}    // 何もしない
}
```

```
//マップクラスのプログラム
class Map extends JPanel {
    static final int XSIZE = 13;
    static final int YSIZE = 11;

    Game game;
    Cell data[][] = new Cell[XSIZE][YSIZE];

    Map(Game game) {
        this.game = game;
        for (int x = 0; x < XSIZE; x++) {
            for (int y = 0; y < YSIZE; y++) {
                if ((x == 0)||(x == XSIZE-1)||
                    (y == 0)||(y == YSIZE-1)||
                    ((Math.abs(x-XSIZE/2) <= 1)&&
                     (Math.abs(y-YSIZE/2) <= 1))) {
                    data[x][y] = new RockCell();
                }
                else {
                    switch (Game.random.nextInt(5)) {
                    case 0: data[x][y] = new PotionCell();
                            break;
                    case 1: data[x][y] = new PoisonCell();
                            break;
                    default: data[x][y] = new EmptyCell();
                    }
                }
            }
        }
        data[XSIZE-2][YSIZE-2] = new GoalCell();

        setPreferredSize(new Dimension(Cell.SIZE*XSIZE, Cell.SIZE*YSIZE));
    }

    Cell getAt(Int2 position) { return data[position.x][position.y]; }

    void setAt(Int2 position, Cell cell) {
        data[position.x][position.y] = cell;
    }

    @Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        for (int y = 0; y < YSIZE; y++) {
            for (int x = 0; x < XSIZE; x++) {
                Int2 position = new Int2(x, y);
                getAt(position).paintAt(g, position);
                for (Actor a : game.actors) {
                    a.paintAt(g, position);
                }
            }
        }
    }

    Int2 getRandomCellPosition() {
        while (true) {
            Int2 position = new Int2(Game.random.nextInt(XSIZE),
                                     Game.random.nextInt(YSIZE));
```

```

            if (!getAt(position).isRockCell()) { return position; }
        }
    }
}
```

```
//アクタークラスのプログラム
abstract class Actor extends JLabel {
    Game game;
    Int2 position;
    int hp;

    Actor(Game game, Int2 position, int hp) {
        super(" ");
        this.game = game;
        this.position = position;
        this.hp = hp;
    }

    Actor(Game game, int hp) {
        super(" ");
        this.game = game;
        this.hp = hp;
        while_loop: while (true) {
            position = game.map.getRandomCellPosition();
            for (Actor a : game.actors) {
                if (position.equal(a.position)) {
                    continue while_loop;
                }
            }
            return;
        }
    }

    @Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        setText(toString());
    }

    void paintAt(Graphics g, Int2 position) {
        if (this.position.equal(position)) {
            g.drawImage(getImage(),
                        position.x*Cell.SIZE + Cell.SIZE*2/24,
                        position.y*Cell.SIZE + Cell.SIZE*2/24,
                        Cell.SIZE*20/24,
                        Cell.SIZE*20/24,
                        game.map);
        }
    }

    abstract Image getImage();

    boolean moveTo(Int2 direction) {
        Int2 target = position.add(direction);

        Cell cell = game.map.getAt(target);
        if (cell.isRockCell()) { return false; }

        for (Actor a : game.actors) {
            if (target.equal(a.position)) { return false; }
        }

        Cell newCell = cell.affectOn(this);
```

```
        game.map.setAt(target, newCell);
        position = target;
        return true;
    }

    abstract void action();

    void attackTo(Int2 direction) {
        Int2 target = position.add(direction);
        for (Actor a : game.actors) {
            if (target.equal(a.position)) {
                a.takeDamageFrom(this);
                return;
            }
        }
    }

    abstract void takeDamageFrom(Actor a);
    void giveDamageTo(Player p) {}
    void giveDamageTo(Monster m) {}
    void giveDamageTo(RedMonster r) {}
}
```

```
//プレイヤークラスのプログラム
class Player extends Actor {
    static final int INIT_HP = 100;
    String name;

    Player(Game game, String name) {
        super(game, new Int2(1, 1), INIT_HP);
        this.name = name;
    }

    Player(Game game) { this(game, "Player"); }

    public String toString() { return name+" : "+position+" "+hp; }

    @Override
    void takeDamageFrom(Actor a) { a.giveDamageTo(this); }

    @Override
    void giveDamageTo(Monster m) { m.hp -= 20; }

    @Override
    void giveDamageTo(RedMonster r) { r.hp -= 15; }

    @Override
    void action() {
        Int2 direction;
        switch (Game.getKeyChar()) {
            case 'z': direction = new Int2(-1, +1); break;
            case 'x': direction = new Int2( 0, +1); break;
            case 'c': direction = new Int2(+1, +1); break;
            case 'a': direction = new Int2(-1,  0); break;
            case 'd': direction = new Int2(+1,  0); break;
            case 'q': direction = new Int2(-1, -1); break;
            case 'w': direction = new Int2( 0, -1); break;
            case 'e': direction = new Int2(+1, -1); break;
            default: direction = new Int2(0, 0);
        }

        if (!moveTo(direction)) { attackTo(direction); }
    }

    static final Image image1 = Toolkit.getDefaultToolkit().getImage("Player1.png");
    static final Image image2 = Toolkit.getDefaultToolkit().getImage("Player2.png");

    @Override
    Image getImage() {
        if (hp > INIT_HP/2) { return image1; }
        else { return image2; }
    }
}
```

```
//モンスタークラスのプログラム
class Monster extends Actor {
    static final int INIT_HP = 150;

    Monster(Game game) { super(game, INIT_HP); }

    @Override
    public String toString() { return "Monster : "+position+" "+hp; }

    @Override
    void takeDamageFrom(Actor a) { a.giveDamageTo(this); }

    @Override
    void giveDamageTo(Player p) { p.hp -= 30; }

    @Override
    void giveDamageTo(RedMonster r) { r.hp -= 5; }

    @Override
    void action() {
        Actor player = game.actors.get(game.PLAYER_ID);
        Int2 direction = player.position.sub(position).signum();

        if (!moveTo(direction)) {
            attackTo(direction);
        }
    }

    static final Image image = Toolkit.getDefaultToolkit().getImage("Monster.png");

    @Override
    Image getImage() { return image; }
}
```

```
//レッドモンスタークラスのプログラム
class RedMonster extends Monster {
    RedMonster(Game game) { super(game); }

    @Override
    public String toString() { return "Red" + super.toString(); }

    @Override
    void takeDamageFrom(Actor a) { a.giveDamageTo(this); }

    @Override
    void giveDamageTo(Player p) { p.hp -= 60; }

    @Override
    void giveDamageTo(Monster m) { m.hp -= 7; }

    static final Image image = Toolkit.getDefaultToolkit().getImage("RedMonster.png");

    @Override
    Image getImage() { return image; }
}
```

```
//2 次元ベクトルクラスのプログラム
class Int2 {
    int x,y;

    Int2(int x, int y) { this.x = x; this.y = y; }
    Int2 add(Int2 a) { return new Int2(x+a.x,y+a.y); }
    Int2 sub(Int2 a) { return new Int2(x-a.x,y-a.y); }
    Int2 signum() { return new Int2(Integer.signum(x),Integer.signum(y)); }
    boolean equal(Int2 a) { return (x == a.x)&&(y == a.y); }

    public String toString(){ return "("+x+","+y+")"; }
}
```

```
//セルクラスのプログラム
abstract class Cell {
    static final int SIZE = 48;

    boolean isRockCell() { return false; }

    abstract Cell affectOn(Actor a);

    abstract void paintAt(Graphics g, Int2 position);

    void drawCell(Graphics g, Int2 position) {
        g.drawRect(position.x*SIZE + SIZE/24,
                   position.y*SIZE + SIZE/24,
                   SIZE*22/24,
                   SIZE*22/24);
    }

    void fillCell(Graphics g, Int2 position) {
        g.fillRect(position.x*SIZE + SIZE/24,
                   position.y*SIZE + SIZE/24,
                   SIZE*22/24,
                   SIZE*22/24);
    }
}

abstract class SimpleCell extends Cell {
    @Override
    Cell affectOn(Actor a) { return this; }
}

abstract class TrapCell extends Cell {
    boolean visible = false;

    @Override
    Cell affectOn(Actor a) {
        visible = true;
        return this;
    }
}

class RockCell extends SimpleCell {
    @Override
    void paintAt(Graphics g, Int2 position) {
        g.setColor(new Color(100, 20, 20));
        fillCell(g, position);
    }

    @Override
    boolean isRockCell() { return true; }
}

class EmptyCell extends SimpleCell {
    @Override
    void paintAt(Graphics g, Int2 position) {
        g.setColor(new Color(128, 128, 128));
        drawCell(g, position);
    }
}
```

```
1
2  class PotionCell extends SimpleCell {
3      @Override
4      void paintAt(Graphics g, Int2 position) {
5          g.setColor(new Color(0, 255, 255));
6          fillCell(g, position);
7      }
8
9      @Override
10     Cell affectOn(Actor a) {
11         a.hp += 50;
12         return new EmptyCell();
13     }
14 }
15
16 class GoalCell extends SimpleCell {
17     @Override
18     void paintAt(Graphics g, Int2 position) {
19         g.setColor(new Color(255, 255, 0));
20         fillCell(g, position);
21     }
22
23     @Override
24     Cell affectOn(Actor a) {
25         if (a instanceof Player) {
26             System.out.println("YOU WIN");
27             System.exit(0);
28         }
29         return this;
30     }
31 }
32
33 class PoisonCell extends TrapCell {
34     @Override
35     void paintAt(Graphics g, Int2 position) {
36         if (visible) {
37             g.setColor(new Color(0, 128, 0));
38             fillCell(g, position);
39         } else {
40             g.setColor(new Color(128, 128, 128));
41             drawCell(g, position);
42         }
43     }
44
45     @Override
46     Cell affectOn(Actor a) {
47         a.hp -= 10;
48         return super.affectOn(a);
49     }
50 }
```

1

1

第28章　ボタンでプレイヤーのアクションを入力する：ボタンとActionListenerの利用

この章では、キーボードに追加してボタンでプレイヤーのアクションを入力する改造を行います。前章で学んだキーリスナーの手法をアクションリスナー（ActionListener＝アクションを監視するソフトウェア部品）に応用します。基本的なプログラミング手法は前章と同じです。

28.1　簡単な例題（その1）

図28.1（524ページ）は、簡単なアクションリスナーのプログラムです。このプログラムではTestクラスがJFrameクラスを継承しており、プログラムの全体構造は前章の図27.4（502ページ）に相当します。

ボタンを作り、パネルに配置する処理は25.4節（447ページ）と同じです。ボタンのGUI部品JButtonにはActionListenerオブジェクトすことができます。ボタンが押されたときには、登録されているActionListenerオブジェクトのメソッドactionPerformed()が自動的に呼び出されます。以下は、そのActionListenerの登録部分です。

```
        button.addActionListener(new MyActionListener());
                                                // ボタンにアクションリスナーを登録
```

インターフェースActionListenerの定義は以下のとおりです。

```
interface ActionListener {
    void actionPerformed(ActionEvent e)
}
```

よってJButtonオブジェクトに登録するリスナーオブジェクトは上の抽象メソッドを実装するクラスのオブジェクトでなければなりません。

図28.1のMyActionListenerではactionPerformed()によって以下の処理（）を実行します。

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

class Test extends JFrame {
    public static void main(String[] args) {
        new Test();
    }

    Test() {
        super("ActionListener Test");
        setBounds(100,100,400,400);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JPanel panel = new JPanel();           // ボタンを配置するパネルを生成

        JButton button = new JButton("0");        // 0 が表示されたボタンを生成
        button.addActionListener(new MyActionListener());
                                              // ボタンにアクションリスナーを登録
        panel.add(button);                              // ボタンをパネルに配置
        add(panel);                                  // フレームにパネルを配置
        setVisible(true);
    }
}

class MyActionListener implements ActionListener {    // 実装クラス
    @Override
    public void actionPerformed(ActionEvent e){
        System.out.println("get");                   // 標準出力へ get を表示
    }
}
```

図 28.1: ActionListener 理解のサンプルプログラム（その 1）

```
        System.out.println("get");                   // 標準出力へ get を表示
```

図 28.2（525 ページ）は、一連の図式化です。JButton オブジェクトにはアクションリスナーオブジェクトを登録することができます。図では MyActionListener オブジェクト（の参照）が登録されています。MyActionListener オブジェクトの中核部には ActionListener オブジェクトの部分があり、そこに actionPerformed() が定義されています。そして、この JButton オブジェクトが押されると、登録されているアクションリスナーオブジェクトの actionPerformed() が実行されます。

図 28.3（526 ページ）はプログラムの実行の様子です。ウインドウ上のボタンをマウスでクリックする度にコマンドプロンプトに “get” が表示されています。

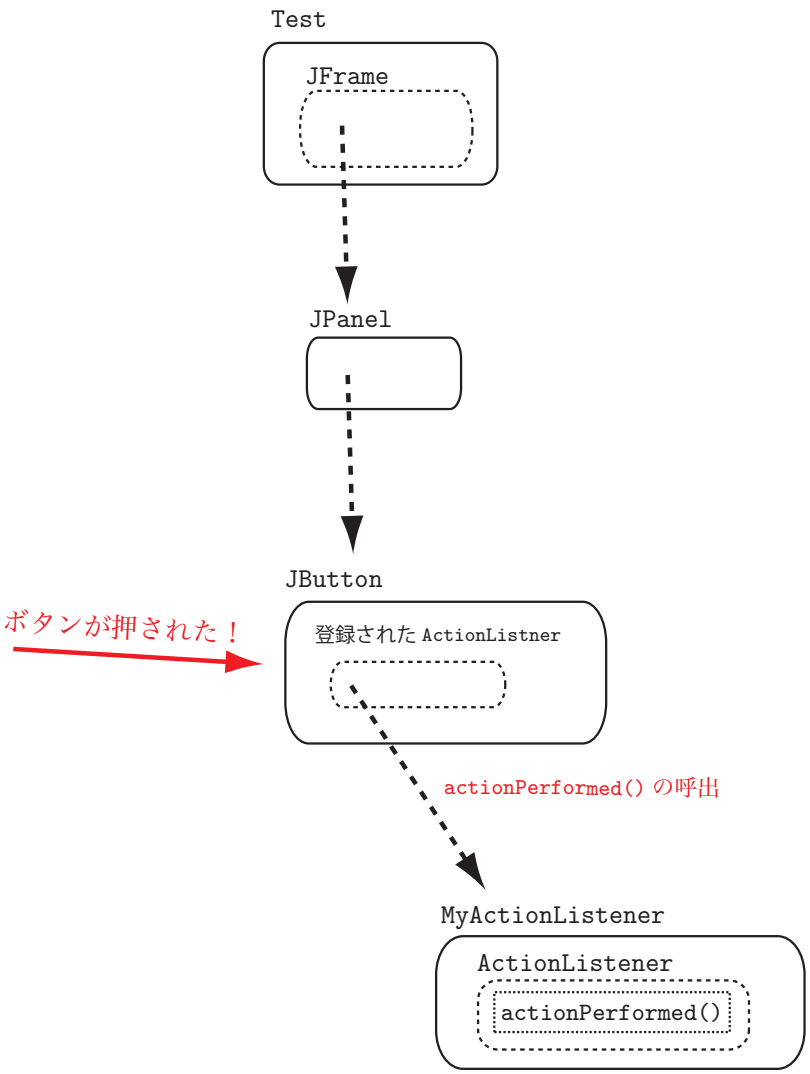


図 28.2: JButton と ActionListener の関係（その 1）

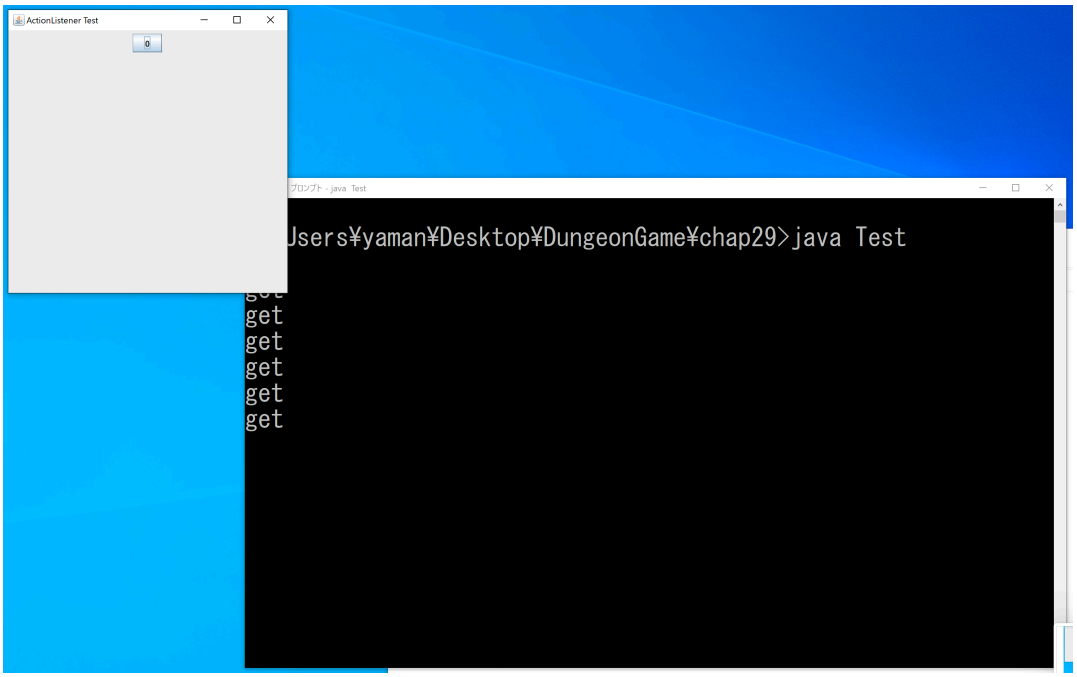


図 28.3: 図 28.1 のプログラムの実行の様子

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

class Test extends JFrame {
    public static void main(String[] args) {
        new Test();
    }

    Test() {
        super("ActionListener Test");
        setBounds(100,100,400,400);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JPanel panel = new JPanel();

        MyButton button = new MyButton();        // ボタンオブジェクトの生成

        panel.add(button);
        add(panel);
        setVisible(true);
    }
}

class MyButton extends JButton implements ActionListener {     // 継承と実装
    MyButton() {
        super("0");
        addActionListener(this);   // 自分自身を登録
    }

    @Override
    public void actionPerformed(ActionEvent e){
        System.out.println("get");
    }
}
```

図 28.4: ActionListener 理解のサンプルプログラム（その 2）

28.2　簡単な例題（その 2）

前章において JFrame と KeyListener を一体化したように、JButton と ActionListener を一体化しましょう。これによってボタン回りの構造を単純化できます。

図 28.4（527 ページ）がそのプログラムです。図 28.5（528 ページ）はその図式です。

ポイントは、以下のように、MyButton が JButton と ActionListener の合併になっている点です。

```
class MyButton extends JButton implements ActionListener {     // 継承と実装
```

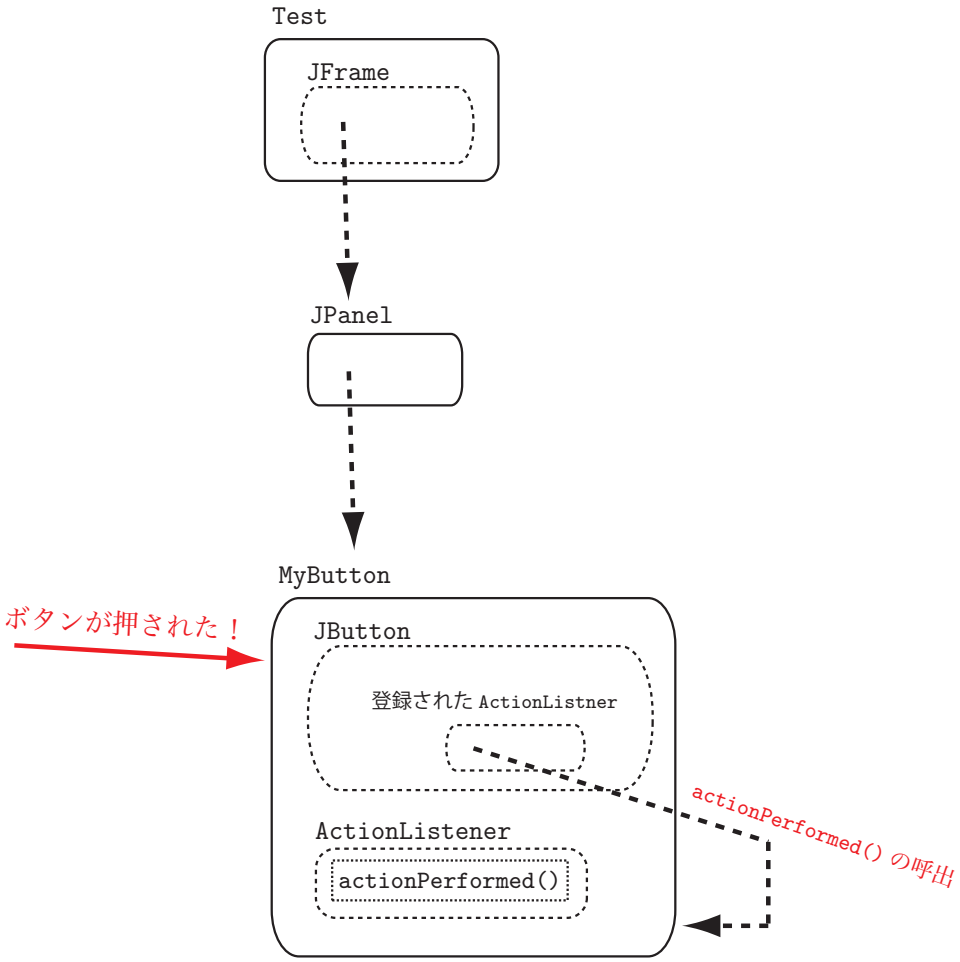


図 28.5: JButton と ActionListener の関係（その 2）

28.3　簡単な例題（その 3）

この節ではパネルに複数のボタンを張る改造を行います。

図 28.6（530 ページ）がそのプログラムです。

まず、二つのボタンに異なる処理を行わせるために、MyButton オブジェクトにインスタンス変数 c を導入し、コンストラクタで初期化します。

ボタンが押されたときには、以下を実行することでどのボタンが押されたか、プログラムで把握できます。

```
System.out.println("get "+c);
```

図 28.8（531 ページ）はプログラムの実行の様子です。0 と 1 のボタンを交互い押すと、その情報がコマンドプロンプトに表示されます。

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

class Test extends JFrame {
    public static void main(String[] args) {
        new Test();
    }

    Test() {
        super("ActionListener Test");
        setBounds(100,100,400,400);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JPanel panel = new JPanel();

        MyButton button0 = new MyButton('0');      // 0 を引数とするボタン
        MyButton button1 = new MyButton('1');      // 1 を引数とするボタン

        panel.add(button0);                        // ボタンの登録
        panel.add(button1);
        add(panel);
        setVisible(true);
    }
}

class MyButton extends JButton implements ActionListener {
    char c;
    MyButton(char c) {
        super(String.valueOf(c));               // ボタンの上に c を表示
        this.c = c;                             // インスタンス変数の初期化
        addActionListener(this);                // 自分をリスナーとして登録
    }

    @Override
    public void actionPerformed(ActionEvent e){
        System.out.println("get "+c);           // get c を表示
    }
}
```

図 28.6: ActionListener 理解のサンプルプログラム（その 3）

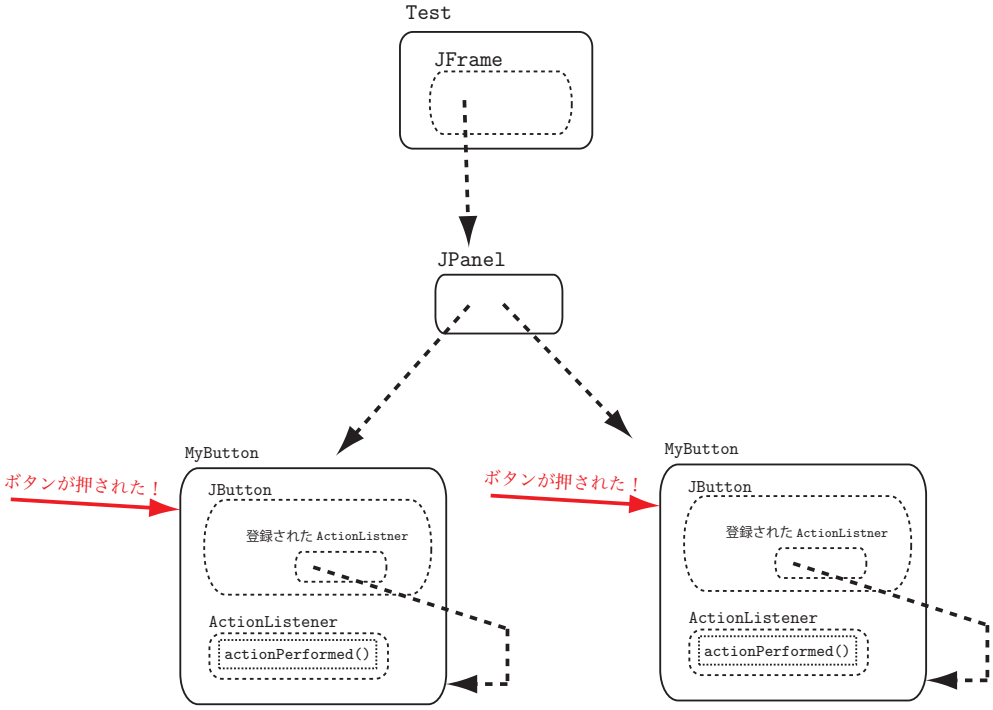


図 28.7: JButton と ActionListener の関係（その 3）

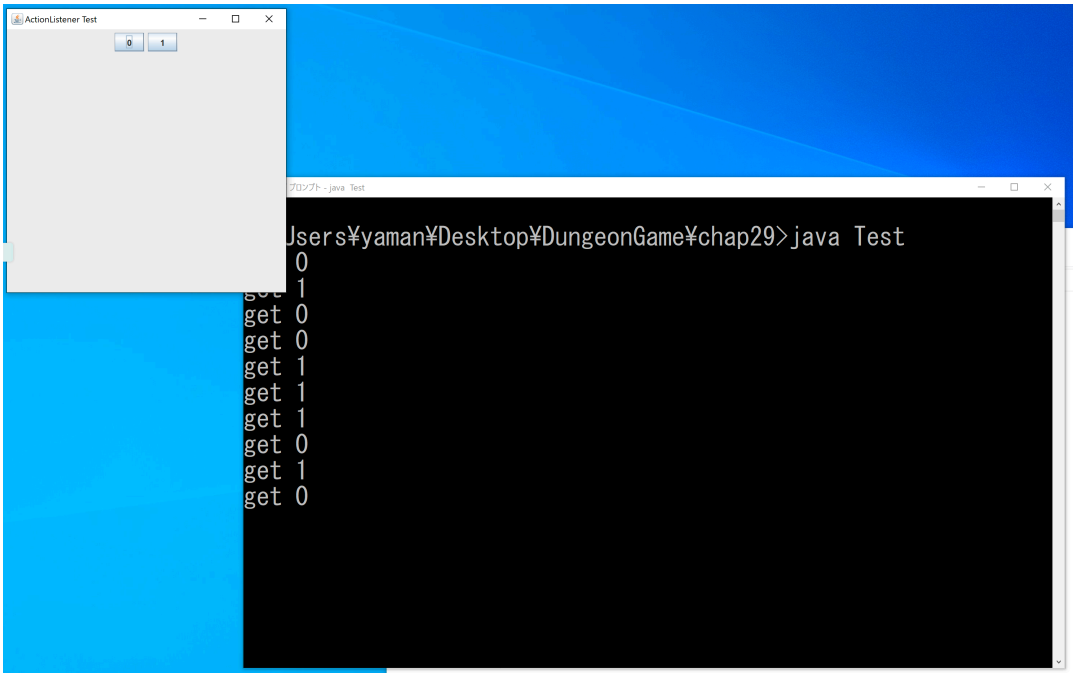


図 28.8: 図 28.6 のプログラムの実行の様子

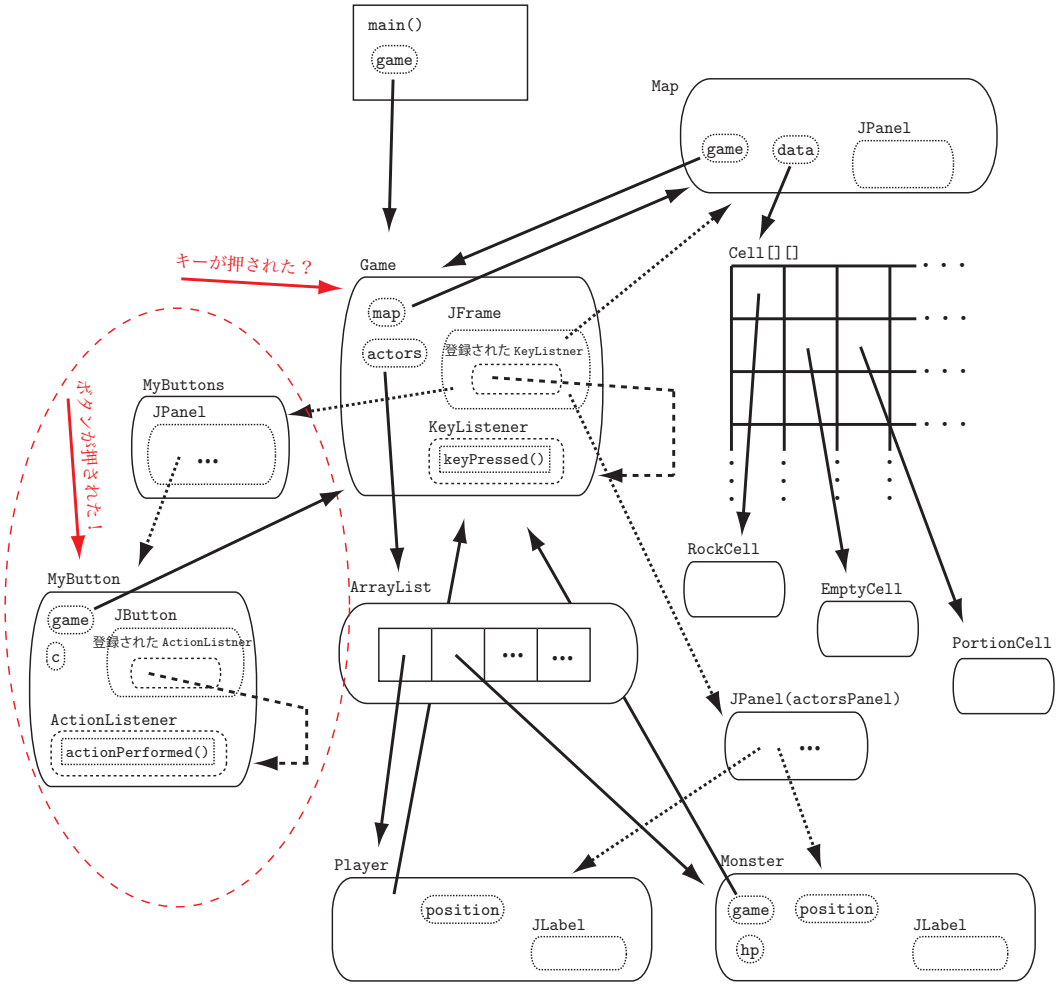


図 28.9: ボタンを用いるプログラムの全体構造：JFrame に MyButtons を配置し、MyButtons に 9 個の MyButton を配置します。 MyButton は自分のボタンが押されたならば、その自分自身の ActionPerformed() を実行します。

28.4　プログラムの構造

かなり複雑になってきていますが、図 27.8（506 ページ）はこの章のプログラムの構造です。赤い部分が前節の図 27.7 の構造をそのまま Game オブジェクトへ実装します。

28.5　ボタンクラスの導入

まず、図 28.10（533 ページ）が、この章で作るゲームの実行の様子です。画面右側に 9 個のボタンが 3 × 3 に並んでいます。このボタン群を配置するために専用のパネルをもちいることとします。図 28.11（534 ページ）はそのパネルクラス（MyButtons クラス）のプログラムです。コンストラ

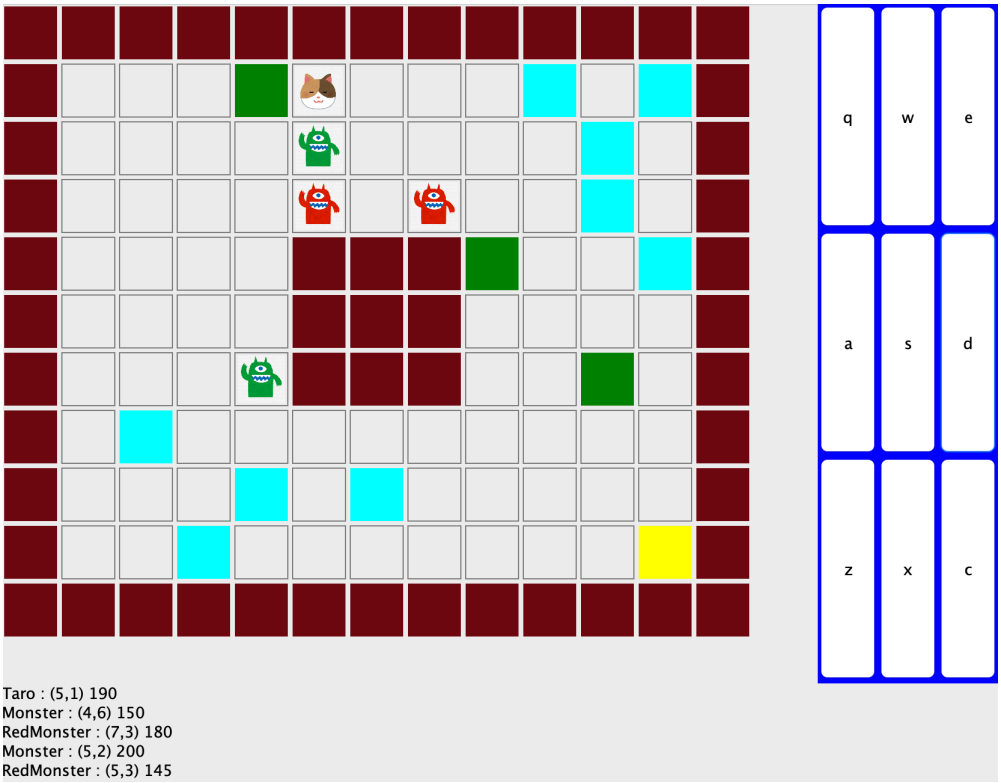


図 28.10: この章のプログラムの実行の様子

クタのみから成っています。コンストラクタの中では、パネルの基本設定の後、9 個の `MyButton` オブジェクトを生成し、パネルに配置しています。

`MyButton` クラスのプログラムは図 28.12（534 ページ）の通りです。インスタンス変数として、`game`（ゲームオブジェクトへの参照）、`c`（このボタンで入力したと見なす文字）を持ちます。自分自身を `ActionListener` として登録します。ボタンが押された時の実行メソッド `actionPerformed()` はボタンに設定された文字を `keyChar` へ設定し、ゲームの 1 ターンを実行します。

```
class MyButtons extends JPanel {
    MyButtons(Game game) {
        setPreferredSize(new Dimension(150, 150));  // パネルのサイズ
        setBackground(Color.BLUE);                      // 背景色は青
        setLayout(new GridLayout(3,3));        // 3 x 3 にボタンを配置

        add(new MyButton(game,'q'));    // 'q' の入力に対応するボタン
        add(new MyButton(game,'w'));    // 以下、同上
        add(new MyButton(game,'e'));
        add(new MyButton(game,'a'));
        add(new MyButton(game,'s'));
        add(new MyButton(game,'d'));
        add(new MyButton(game,'z'));
        add(new MyButton(game,'x'));
        add(new MyButton(game,'c'));
    }
}
```

図 28.11: ActionListener を利用するダンジョンゲームのプログラム（ボタン用パネル）

```
class MyButton extends JButton implements ActionListener {
    Game game;                          // ゲームオブジェクトへの参照
    char c;                             // 入力文字
    MyButton(Game game, char c) {
        super(String.valueOf(c));
        this.game = game;
        this.c = c;
        addActionListener(this);        //リスナー（自分）の登録
    }

    @Override
    public void actionPerformed(ActionEvent e){
        game.setKeyChar(c);             // 文字を設定し、
        game.turn();                    // 1 ターンを実行
    }
}
```

図 28.12: ActionListener を利用するダンジョンゲームのプログラム（ボタン）

```
class Game extends JFrame implements KeyListener {
    public static void main(String args[]) { ** 前章から変更なし ** }

    ** インスタンス変数などは前章から変更なし **

    Game() {
        ** 前半部は前章から変更なし **

        add(map, BorderLayout.WEST); // NORTH を WEST に変更

        add(new MyButtons(this), BorderLayout.EAST); // ボタン用パネルの配置

        ** 以下、前章から変更なし **
    }

    void print() { ** 前章から変更なし ** }

    void turn() { ** 前章から変更なし ** }

    static char keyChar;
    static char getKeyChar() { ** 前章から変更なし ** }
    static void setKeyChar(char c) { ** 前章から変更なし ** }

    @Override
    public void keyPressed(KeyEvent e) { ** 前章から変更なし ** }

    @Override
    public void keyReleased(KeyEvent e){}

    @Override
    public void keyTyped(KeyEvent e){}
}
```

図 28.13: ActionListener を利用するダンジョンゲームのプログラム（ゲーム）

28.6　ゲームクラスの変更

図 28.13（535 ページ）が、この章のゲームクラスのプログラムです。前章からの変更はわずかに以下のみです。

```
        add(map, BorderLayout.WEST); // NORTH を WEST に変更

        add(new MyButtons(this), BorderLayout.EAST); // ボタン用パネルの配置
```

ゲームクラスの変更は以上です。その他のクラスのプログラムに変更はありません。

28.7　フォーカスをボタンが取得

26.10.1 項（479 ページ）において、キーボードからの入力は特定のウインドウ（一般には最前面のウインドウ）へ伝えられると述べました。このフォーカスの問題がこの章でも再浮上します。

とりあえず前節まで説明したプログラムをコンパイルして動かしてみましょう。

この章のプログラムは前章のプログラムにボタン関連のプログラムを追加しただけですから、予定ではキーボードを使った動作の入力も、ボタンを使ったボタンの入力もできるはずです。しかし、実際には、ボタンでの入力はできるのですが、キーボードからの入力は受け付けられません。原因は、

- キー入力のフォーカスをボタン（`MyButton`）が優先取得してしまい、ゲームオブジェクト（`JFrame` のサブクラスのオブジェクト）がキー入力を感知できない

状態になるためです。ちょっと面倒な話です。この解決法として、キー入力のフォーカスをゲームオブジェクトに強制移動させるなどの方法が考えらますが、フォーカスの移動は意外に面倒です。最も簡単な方法は以下です。

- ボタン（`MyButton`）が持っているキー入力のフォーカスは移動させず放置し、
- ボタンのキーリスナーとしてゲームオブジェクトを登録し、ボタンが感知するキー入力イベントをゲームオブジェクトへ伝える。

「ボタンのキーリスナーとしてゲームオブジェクトを登録」することはプログラム全体の構造から言えば本末転倒な感じもしますが、この方法はプログラムを 1 行追加するだけでうまく行きます。図 28.12（534 ページ）の `MyButton` のコンストラクタの末尾に以下を追加するだけです。

```
addKeyListener(game);    // game をキーリスナーとしてボタンに登録
```

また、ゲームクラスのコンストラクタにあった以下の一文はもはや不要になります。

```
addKeyListener(this);
```

この辺の技術的な詳細は以下の解説などを参考にしてください。

GUI 部品にフォーカスを奪われた場合の対処

https://www.atmarkit.co.jp/ait/articles/0501/19/news131.html

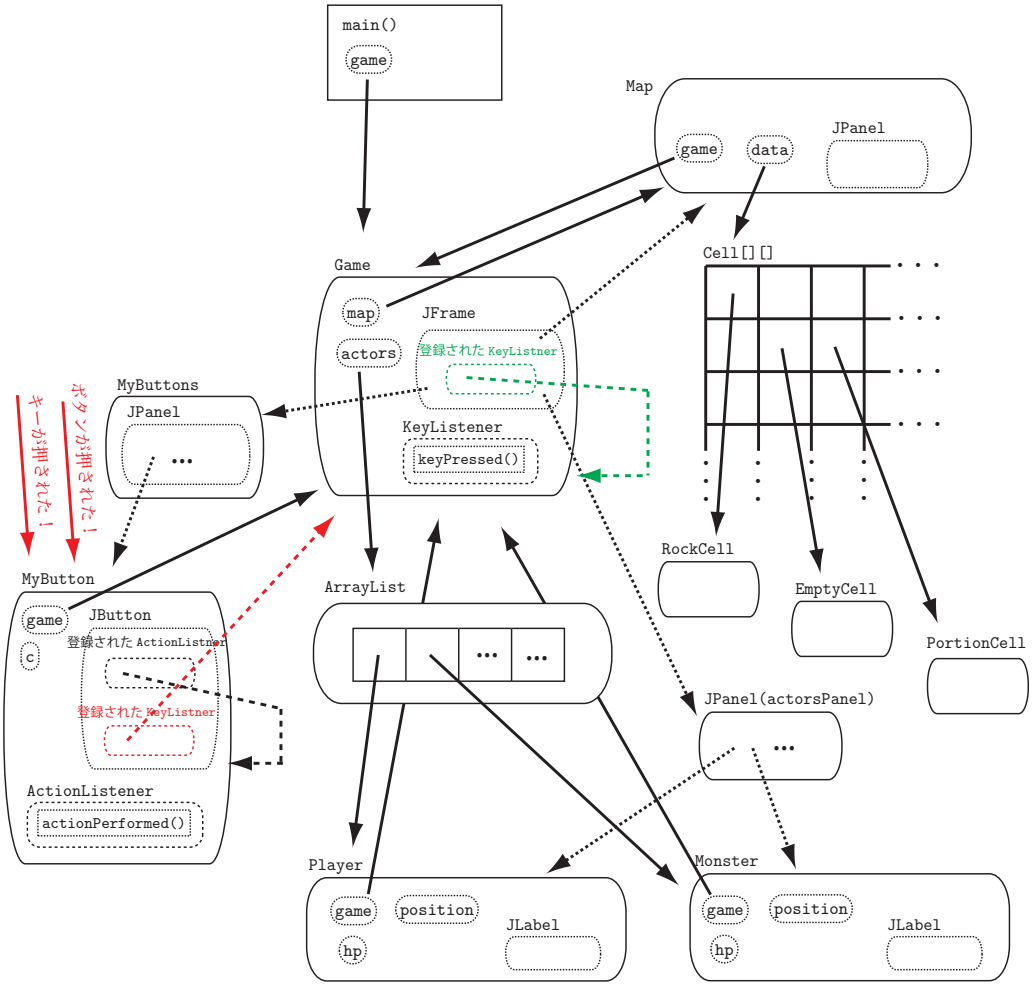


図 28.14: ボタンを用いるプログラムの全体構造：キー入力フォーカスはボタンが取得してしまうため、KeyListener（＝ Game オブジェクト）をボタンに登録してフォーカスの問題を解決します。

28.8　補足：その他のリスナー

プレイヤーの移動・攻撃の入力には、マウスで直接マップをクリックする方法がより直感的です。それには MouseListener を用いればよいでしょう。

Java のリスナーには他にも WindowListener、MouseMotionListener、TextListener、ItemListener があります。基本的な利用方法は、前章、この章で解説した内容と同じです。余裕のある人は自ら調べてください。

28.9　補足：イベント駆動プログラム

この章のプログラムのように、キー入力やボタン操作で処理が進むタイプのプログラムを**イベント駆動**（event driven）プログラムと呼びます。プログラムではイベント毎に処理内容が独立して決まっており、あるイベントが発生したならば、それに対応する処理を行い、終了後はイベント待ち状態に入るというループを繰り返していきます。

これに対して、前々章までのプログラム、あるいは「プログラミング概論/演習 I,II」で主に学んだ入出力と計算の処理の順序が固定されて実行される種類のプログラムを**フロー駆動**（flow driven）プログラムと呼びます。キー入力を受け付ける点では前々章までのプログラムはこの章のプログラムと同じです。両者の違いは、前々章のプログラムでは処理全体が大きなループ（入力 → 処理 → 出力 → 入力 → ...）を成しており、その中にキー入力部分が組み込まれている点に対して、この章のプログラムでは通常はイベント待ち状態にあり、キー入力イベントが発生したときにその処理が行われる点です。

イベント駆動プログラムの基本構造はアプリケーションに依らず、似た構造（イベント発生 → イベント感知 → イベント処理 → ...、という構造）になっています。そこで、イベント駆動プログラムの開発では、全体構造やイベント回りの共通部分を事前に作り込んだ **アプリケーションフレームワーク**（application framework）[1]が半完成品としてベンダーなどから提供される場合が大半です。Java の AWT/Swing もアプリケーションフレームワークと見なされます。ユーザはそのフレームワークにアプリケーションプログラムの固有の処理を追加すれば短期間でプログラムが完成させることができます。仮にアプリケーションフレームワークを用いずにイベント駆動プログラムを作ろうとするならば、ウインドウシステム、オペレーティングシステムに関する膨大な知識が必要となります。

[1] アプリケーションフレームワークはオブジェクト指向プログラミングにやや遅れ、1980 年台半ばから利用されるようになりました。

28.10　付録：この章の全プログラムリスト

```
//ゲームクラスのプログラム
import java.io.*;
import java.util.Random;
import java.util.ArrayList;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class Game extends JFrame implements KeyListener {
    public static void main(String args[]) {
        Game game = new Game();
    }

    static Random random = new Random();

    Map map;
    ArrayList<Actor> actors;
    final static int PLAYER_ID = 0;
    final static int NUM_OF_MONSTERS = 4;

    Game() {
        super("Dungeon Game");

        map = new Map(this);
        actors = new ArrayList<Actor>();
        actors.add(new Player(this, "Taro"));
        for (int i = 0; i < NUM_OF_MONSTERS; i++) {
            if (random.nextInt(2) == 0) {
                actors.add(new Monster(this));
            }
            else {
                actors.add(new RedMonster(this));
            }
        }

        setBounds(100, 100, Cell.SIZE * Map.XSIZE + 5 + 200, Cell.SIZE * Map.YSIZE + 25 + 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        add(map, BorderLayout.WEST); // NORTH を WEST に変更

        add(new MyButtons(this), BorderLayout.EAST); // ボタン用パネルの配置

        JPanel actorsPanel = new JPanel();
        actorsPanel.setLayout(new GridLayout(10, 1));
        for (Actor a : actors) { actorsPanel.add(a); }
        add(actorsPanel, BorderLayout.SOUTH);

        setVisible(true);

//        addKeyListener(this);
    }

    void print() { repaint(); }

    void turn() {
        for (Actor a : actors) { a.action(); }
        print();
```

```
    }

    static char keyChar;
    static char getKeyChar() { return keyChar; }
    static void setKeyChar(char c) { keyChar = c; }

    @Override
    public void keyPressed(KeyEvent e) {
        setKeyChar(e.getKeyChar());
        turn();
    }

    @Override
    public void keyReleased(KeyEvent e){}

    @Override
    public void keyTyped(KeyEvent e){}
}
```

```
//ボタンクラスのプログラム
class MyButtons extends JPanel {
    MyButtons(Game game) {
        setPreferredSize(new Dimension(150, 150));  // パネルのサイズ
        setBackground(Color.BLUE);                      // 背景色は青
        setLayout(new GridLayout(3,3));        // 3 x 3 にボタンを配置

        add(new MyButton(game,'q'));    // 'q' の入力に対応するボタン
        add(new MyButton(game,'w'));    // 以下、同上
        add(new MyButton(game,'e'));
        add(new MyButton(game,'a'));
        add(new MyButton(game,'s'));
        add(new MyButton(game,'d'));
        add(new MyButton(game,'z'));
        add(new MyButton(game,'x'));
        add(new MyButton(game,'c'));
    }
}

class MyButton extends JButton implements ActionListener {
    Game game;                          // ゲームオブジェクトへの参照
    char c;                             // 入力文字
    MyButton(Game game, char c) {
        super(String.valueOf(c));
        this.game = game;
        this.c = c;
        addActionListener(this);        //リスナー（自分）の登録
    }

    @Override
    public void actionPerformed(ActionEvent e){
        game.setKeyChar(c);             // 文字を設定し、
        game.turn();                    // 1 ターンを実行
    }
}
```

```
//マップクラスのプログラム
class Map extends JPanel {
    static final int XSIZE = 13;
    static final int YSIZE = 11;

    Game game;
    Cell data[][] = new Cell[XSIZE][YSIZE];

    Map(Game game) {
        this.game = game;
        for (int x = 0; x < XSIZE; x++) {
            for (int y = 0; y < YSIZE; y++) {
                if ((x == 0)||(x == XSIZE-1)||
                    (y == 0)||(y == YSIZE-1)||
                    ((Math.abs(x-XSIZE/2) <= 1)&&
                     (Math.abs(y-YSIZE/2) <= 1))) {
                    data[x][y] = new RockCell();
                }
                else {
                    switch (Game.random.nextInt(5)) {
                    case 0: data[x][y] = new PotionCell();
                             break;
                    case 1: data[x][y] = new PoisonCell();
                             break;
                    default: data[x][y] = new EmptyCell();
                    }
                }
            }
        }
        data[XSIZE-2][YSIZE-2] = new GoalCell();

        setPreferredSize(new Dimension(Cell.SIZE*XSIZE, Cell.SIZE*YSIZE));
    }

    Cell getAt(Int2 position) { return data[position.x][position.y]; }

    void setAt(Int2 position, Cell cell) {
        data[position.x][position.y] = cell;
    }

    @Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        for (int y = 0; y < YSIZE; y++) {
            for (int x = 0; x < XSIZE; x++) {
                Int2 position = new Int2(x, y);
                getAt(position).paintAt(g, position);
                for (Actor a : game.actors) {
                    a.paintAt(g, position);
                }
            }
        }
    }

    Int2 getRandomCellPosition() {
        while (true) {
            Int2 position = new Int2(Game.random.nextInt(XSIZE),
                                     Game.random.nextInt(YSIZE));
```

```

            if (!getAt(position).isRockCell()) { return position; }
        }
    }
}
```

```
//アクタークラスのプログラム
abstract class Actor extends JLabel {
    Game game;
    Int2 position;
    int hp;

    Actor(Game game, Int2 position, int hp) {
        super(" ");
        this.game = game;
        this.position = position;
        this.hp = hp;
    }

    Actor(Game game, int hp) {
        super(" ");
        this.game = game;
        this.hp = hp;
        while_loop: while (true) {
            position = game.map.getRandomCellPosition();
            for (Actor a : game.actors) {
                if (position.equal(a.position)) {
                    continue while_loop;
                }
            }
            return;
        }
    }

    @Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        setText(toString());
    }

    void paintAt(Graphics g, Int2 position) {
        if (this.position.equal(position)) {
            g.drawImage(getImage(),
                        position.x*Cell.SIZE + Cell.SIZE*2/24,
                        position.y*Cell.SIZE + Cell.SIZE*2/24,
                        Cell.SIZE*20/24,
                        Cell.SIZE*20/24,
                        game.map);
        }
    }

    abstract Image getImage();

    boolean moveTo(Int2 direction) {
        Int2 target = position.add(direction);

        Cell cell = game.map.getAt(target);
        if (cell.isRockCell()) { return false; }

        for (Actor a : game.actors) {
            if (target.equal(a.position)) { return false; }
        }

        Cell newCell = cell.affectOn(this);
```

```
        game.map.setAt(target, newCell);
        position = target;
        return true;
    }

    abstract void action();

    void attackTo(Int2 direction) {
        Int2 target = position.add(direction);
        for (Actor a : game.actors) {
            if (target.equal(a.position)) {
                a.takeDamageFrom(this);
                return;
            }
        }
    }

    abstract void takeDamageFrom(Actor a);
    void giveDamageTo(Player p) {}
    void giveDamageTo(Monster m) {}
    void giveDamageTo(RedMonster r) {}
}
```

```
//プレイヤークラスのプログラム
class Player extends Actor {
    static final int INIT_HP = 100;
    String name;

    Player(Game game, String name) {
        super(game, new Int2(1, 1), INIT_HP);
        this.name = name;
    }

    Player(Game game) { this(game, "Player"); }

    public String toString() { return name+" : "+position+" "+hp; }

    @Override
    void takeDamageFrom(Actor a) { a.giveDamageTo(this); }

    @Override
    void giveDamageTo(Monster m) { m.hp -= 20; }

    @Override
    void giveDamageTo(RedMonster r) { r.hp -= 15; }

    @Override
    void action() {
        Int2 direction;
        switch (Game.getKeyChar()) {
            case 'z': direction = new Int2(-1, +1); break;
            case 'x': direction = new Int2( 0, +1); break;
            case 'c': direction = new Int2(+1, +1); break;
            case 'a': direction = new Int2(-1,  0); break;
            case 'd': direction = new Int2(+1,  0); break;
            case 'q': direction = new Int2(-1, -1); break;
            case 'w': direction = new Int2( 0, -1); break;
            case 'e': direction = new Int2(+1, -1); break;
            default: direction = new Int2(0, 0);
        }

        if (!moveTo(direction)) { attackTo(direction); }
    }

    static final Image image1 = Toolkit.getDefaultToolkit().getImage("Player1.png");
    static final Image image2 = Toolkit.getDefaultToolkit().getImage("Player2.png");

    @Override
    Image getImage() {
        if (hp > INIT_HP/2) { return image1; }
        else { return image2; }
    }
}
```

```
//モンスタークラスのプログラム
class Monster extends Actor {
    static final int INIT_HP = 150;

    Monster(Game game) { super(game, INIT_HP); }

    @Override
    public String toString() { return "Monster : "+position+" "+hp; }

    @Override
    void takeDamageFrom(Actor a) { a.giveDamageTo(this); }

    @Override
    void giveDamageTo(Player p) { p.hp -= 30; }

    @Override
    void giveDamageTo(RedMonster r) { r.hp -= 5; }

    @Override
    void action() {
        Actor player = game.actors.get(game.PLAYER_ID);
        Int2 direction = player.position.sub(position).signum();

        if (!moveTo(direction)) {
            attackTo(direction);
        }
    }

    static final Image image = Toolkit.getDefaultToolkit().getImage("Monster.png");

    @Override
    Image getImage() { return image; }
}
```

```
//レッドモンスタークラスのプログラム
class RedMonster extends Monster {
    RedMonster(Game game) { super(game); }

    @Override
    public String toString() { return "Red" + super.toString(); }

    @Override
    void takeDamageFrom(Actor a) { a.giveDamageTo(this); }

    @Override
    void giveDamageTo(Player p) { p.hp -= 60; }

    @Override
    void giveDamageTo(Monster m) { m.hp -= 7; }

    static final Image image = Toolkit.getDefaultToolkit().getImage("RedMonster.png");

    @Override
    Image getImage() { return image; }
}
```

```
//2 次元ベクトルクラスのプログラム
class Int2 {
    int x,y;

    Int2(int x, int y) { this.x = x; this.y = y; }
    Int2 add(Int2 a) { return new Int2(x+a.x,y+a.y); }
    Int2 sub(Int2 a) { return new Int2(x-a.x,y-a.y); }
    Int2 signum() { return new Int2(Integer.signum(x),Integer.signum(y)); }
    boolean equal(Int2 a) { return (x == a.x)&&(y == a.y); }

    public String toString(){ return "("+x+","+y+")"; }
}
```

```
//セルクラスのプログラム
abstract class Cell {
    static final int SIZE = 48;

    boolean isRockCell() { return false; }

    abstract Cell affectOn(Actor a);

    abstract void paintAt(Graphics g, Int2 position);

    void drawCell(Graphics g, Int2 position) {
        g.drawRect(position.x*SIZE + SIZE/24,
                   position.y*SIZE + SIZE/24,
                   SIZE*22/24,
                   SIZE*22/24);
    }

    void fillCell(Graphics g, Int2 position) {
        g.fillRect(position.x*SIZE + SIZE/24,
                   position.y*SIZE + SIZE/24,
                   SIZE*22/24,
                   SIZE*22/24);
    }
}

abstract class SimpleCell extends Cell {
    @Override
    Cell affectOn(Actor a) { return this; }
}

abstract class TrapCell extends Cell {
    boolean visible = false;

    @Override
    Cell affectOn(Actor a) {
        visible = true;
        return this;
    }
}

class RockCell extends SimpleCell {
    @Override
    void paintAt(Graphics g, Int2 position) {
        g.setColor(new Color(100, 20, 20));
        fillCell(g, position);
    }

    @Override
    boolean isRockCell() { return true; }
}

class EmptyCell extends SimpleCell {
    @Override
    void paintAt(Graphics g, Int2 position) {
        g.setColor(new Color(128, 128, 128));
        drawCell(g, position);
    }
}
```

```

class PotionCell extends SimpleCell {
    @Override
    void paintAt(Graphics g, Int2 position) {
        g.setColor(new Color(0, 255, 255));
        fillCell(g, position);
    }

    @Override
    Cell affectOn(Actor a) {
        a.hp += 50;
        return new EmptyCell();
    }
}

class GoalCell extends SimpleCell {
    @Override
    void paintAt(Graphics g, Int2 position) {
        g.setColor(new Color(255, 255, 0));
        fillCell(g, position);
    }

    @Override
    Cell affectOn(Actor a) {
        if (a instanceof Player) {
            System.out.println("YOU WIN");
            System.exit(0);
        }
        return this;
    }
}

class PoisonCell extends TrapCell {
    @Override
    void paintAt(Graphics g, Int2 position) {
        if (visible) {
            g.setColor(new Color(0, 128, 0));
            fillCell(g, position);
        } else {
            g.setColor(new Color(128, 128, 128));
            drawCell(g, position);
        }
    }

    @Override
    Cell affectOn(Actor a) {
        a.hp -= 10;
        return super.affectOn(a);
    }
}
```

1

1

第29章　モンスターが勝手に動き出す：マルチスレッドの利用

通常のコンピュータプログラムの実行は、プログラムの上の1本の線に沿って進んでいきます。この線をしばしば**スレッド**（thread＝糸）あるいは**シングルスレッド**（single thread）と呼びます。これを拡張適用し、複数のプログラムが同時並列的[1]に実行される様子を**マルチスレッド**（multi-thread）と呼びます。

ここまで検討してきたダンジョンゲームでは、それがどんなに複雑な多相性を用いようともイベント駆動を用いようともシングルスレッドのプログラム実行でした。それをマルチスレッド化するのがこの章の目的です。これによってゲームは、モンスターがプレイヤーの動作（ユーザの入力操作）を待たずに勝手に動き出すリアルタイムシミュレーションゲームへと変貌します。

Java ではプログラマがマルチスレッドを簡単に利用できる仕組みが提供されています。まず、その簡単な紹介を行います。そしてゲームへ応用します。なお、この講義テキストは並列プログラミングの学習を主目的とするものではありませんから、それに関連する話題、たとえば**同期**（synchonization）、**スレッドセーフ**（thread safe）などの事項に触れる余裕はありません。たとえば以下を参考にしてください。

マルチスレッドと並行処理をわかりやすく説明します

`https://blog.framinal.life/entry/2020/04/05/204055`

29.1　簡単な例題（その1）

図 29.1（555 ページ）は最も単純なマルチスレッドのプログラム例です。

プログラムではまず、パッケージ読み込みのため、以下を付けます。

```
import java.lang.Thread.*;
```

まず `MyThread` クラスから見ていきます。

[1]正確に言えば、マルチスレッドの動作は並列的（parallel）ではなく、並行的（concurrent）と言います。並列実行はプロセッサを複数個持つ並列コンピュータでの動作であり、並行実行はプロセッサを 1 個（概念上の 1 個であって、現実には数個のプロセッサを持つものも含みます）しか持たないコンピュータでの擬似並列動作です。

```
import java.lang.Thread.*;

class Test {
    public static void main(String[] args) {
        MyThread t1 = new MyThread("WAHAHA");      // スレッドオブジェクトの
        MyThread t2 = new MyThread("WOOO");        // 生成

        t1.start();                                // スレッドの開始
        t2.start();

        System.out.println("WEEEE");
    }
}

class MyThread extends Thread {
    String cry;                                    // 叫び声

    MyThread(String cry) { this.cry = cry; }

    @Override
    public void run() { System.out.println(cry); } // 叫ぶ
}
```

図 29.1: スレッドを用いるサンプルプログラム

そのスーパークラスである Thread クラスはスレッドの実行制御に関する様々な機能を持つクラスであり、そのオブジェクトは一つのスレッドに対応します。ただし、Thread クラスそのもののオブジェクトを実行しても、何も行わず、すぐに実行を終了するだけです。有意なことを行うにはサブクラスでその処理を上書きする必要があります。そこで、以下のように MyThread クラスを Thread クラスのサブクラスとして宣言します。

```
class MyThread extends Thread {
    String cry;                                    // 叫び声
```

インスタンス変数 cry にはスレッド実行時に表示させたい文字列を登録しておくこととします。コンストラクタでは単に文字列を受け取り、 cry を初期化します。なお、ここには親クラス Thread のコンストラクタの呼び出しは明示されていませんから、デフォルトコンストラクタが自動的に呼び出されます。メソッド run() は Thread クラスに定義されているもので、このメソッドがスレッドとして実行されます。そこでユーザプログラムではスレッドに行わせたい処理を上書きして記述します。図 29.1 では

```
    public void run() { System.out.println(cry); } // 叫ぶ
```

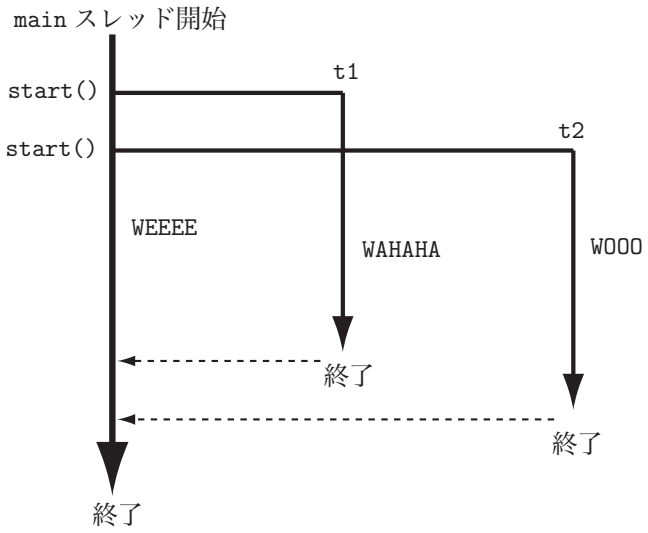


図 29.2: 図 29.1 のプログラムの実行の様子：main スレッド開始後、t1 が実行を開始し、次に t2 が実行を開始します。3 本のスレッドはそれぞれ叫び声を上げます。main スレッドは子スレッドの終了を待って自身も終了します。

と、叫び声を表示させる処理を行なっています。

以上でスレッドの準備はできました。

プログラムの実行では初めに main スレッド — Java プログラムの開始直後に main メソッドから実行が始まるスレッド — が起動します。これは通常の動作です。そして、Test クラスの main メソッドでは二つの MyThread オブジェクト t1、t2 を生成します。

```
        MyThread t1 = new MyThread("WAHAHA");      // スレッドオブジェクトの
        MyThread t2 = new MyThread("WOOO");        // 生成
```

そして以下：

```
        t1.start();                                // スレッドの開始
        t2.start();
```

によって main メソッドとは別に二つのスレッドの実行が開始されます。main スレッドから起動される t1、t2 は、main スレッドの**子スレッド**（child thread）と呼ばれます。逆に、maian スレッドは t1、t2 の**親スレッド**（parent thread ）です。スレッドが実行を始めると、start() メソッドから自動的に上書きされた run() メソッドが呼ばれます。

図 29.2（556 ページ）は、図 29.1 のプログラムの実行の様子（スレッドの図）を図示したものです。それぞれのスレッドがそれぞれの叫び声を上げて終了します。叫ぶタイミングは実行の度に微妙に異なります。なお、親スレッドは全ての子スレッドの終了を待ってから自身を終了します。

注意： main スレッドから直接、以下のメソッド呼び出しを行なってもマルチスレッドにはなりません。

```
t1.run();
t2.run();
```

これは単に main スレッドから二つの run() を順に呼び出しているだけです。

29.2　簡単な例題（その 2）

通常、スレッド実行は前節の図 29.1 のようにあっさりと終わるものではなく、run() メソッドの中でループを作って長時間動き続けるようにプログラミングします。あっさりと終わるような小さな単純な処理にわざわざスレッドを使うメリットはないからです。

そこで、この節のプログラム（図 29.3（558 ページ））の MyThread クラスの run() メソッドにはループの中で叫び続けるプログラムを用意しました。また繰り返し叫ぶ時間の間隔を保持するインスタンス変数 duration も導入しました。単位は 1 ミリ秒です。コンストラクタの第 2 引数でその値を渡します。

MyThread クラスに新規に導入されたもうひとつのインスタンス変数 flag は run() メソッドの本体：

```
while (flag) {                  // flag が true である限り
    System.out.println(cry);
    Test.sleep(duration);       // 一時停止
}
```

の終了を制御するためのものです。flag の初期値は true ですが、この変数値が false になると、while ループを脱出し、スレッドを終了します。

ループ中で呼び出されている Test.sleep() は、当該スレッドの実行を引数の時間だけ停止させます。このメソッドは実質的には Thread クラスのクラスメソッド sleep() を呼び出しているだけなのですが、Thread クラスの sleep() は例外（exception）を発生する可能性がありますから、以下：

```
import java.lang.Thread.*;

class Test {
    public static void main(String[] args) {
        MyThread t1 = new MyThread("WAHAHA", 500);  // 0.5 秒の停止
        MyThread t2 = new MyThread("WOOO", 1700);   // 1.7 秒の停止

        t1.start();
        t2.start();

        for (int i = 0; i < 5; i++) {
            System.out.println("WEEEE");
            Test.sleep(1000);                       // 1.0 秒の停止
        }

        t1.flag = false;                    // スレッドのフラグ変更
        t2.flag = false;                    // スレッドのフラグ変更
    }

    static void sleep(int time) {
        try { Thread.sleep(time); }            // 例外をチェックして
        catch (Exception e){ System.exit(1); } // キャッチ
    }
}

class MyThread extends Thread {
    String cry;                          // 叫び声
    int duration;                        // 時間間隔
    boolean flag = true;                 // ループ制御用フラグ

    MyThread(String cry, int duration) {
        this.cry = cry;
        this.duration = duration;
    }

    @Override
    public void run() {
        while (flag) {                 // flag が true である限り
            System.out.println(cry);
            Test.sleep(duration);      // 一時停止
        }
    }
}
```

図 29.3: スレッドを用いるサンプルプログラム

```
        try { Thread.sleep(time); }            // 例外をチェックして
        catch (Exception e){ System.exit(1); } // キャッチ
```

のように、try〜catch 文で囲む必要があり、これがプログラムの可読性が低下させると考え、メソッドを一皮被せました。

さて、main メソッドでは、以下のように、スレッド t1 は 0.5 秒毎に、スレッド t2 は 1.7 秒毎に叫び声を上げるように設定します。

```
        MyThread t1 = new MyThread("WAHAHA", 500);  // 0.5 秒の停止
        MyThread t2 = new MyThread("WOOO", 1700);   // 1.7 秒の停止
```

main スレッドは 1.0 秒毎に 5 回叫びます（以下、参照）。

```
        for (int i = 0; i < 5; i++) {
            System.out.println("WEEEE");
            Test.sleep(1000);
        }
```

そして、子スレッド t1、t2 を終了させるために、それぞれのフラグを変更します[2]。

```
        t1.flag = false;                    // スレッドのフラグ変更
        t2.flag = false;                    // スレッドのフラグ変更
```

図 29.4（560 ページ）は実行の様子です。実際のプログラム実行では各スレッドの叫び声が一定間隔で起こることを確認できます。

[2]直接、スレッドの実行を止める方法がありそうに思うかもしれません。実際、以前の Java の Thread クラスには stop() メソッドがありましたし、現在も利用可能です。しかし、スレッドを stop() で止める方法は乱暴すぎるとの理由から、stop() の使用は推奨されていません。

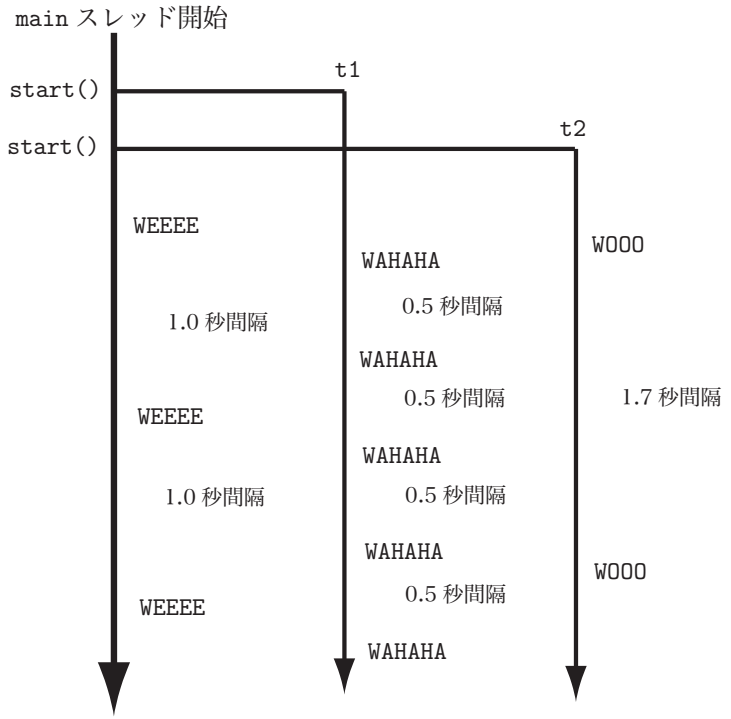


図 29.4: 図 29.3 のプログラムの実行の様子：main スレッド開始後、t1 が実行を開始し、次に t2 が実行を開始します。3 本のスレッドはループの中で叫び続けます。

29.3　簡単な例題（その 3）

図 29.5（562 ページ）のプログラムは、図 29.3（558 ページ）のプログラムと同じ動作をします。両者の違いは、図 29.3 では継承を用いていたのに対して図 29.5 ではインターフェースを用いている点です。

なぜインターフェースが必要なのでしょうか。スレッド化したいと考えているクラスが既に別のクラスを継承している場合には、多重継承禁止のルール（23.2 節（404 ページ）参照）から Thread クラスを継承できないからです。この章でアクターの動きをスレッド化しますが、アクタークラスはすでに JLabel クラスのサブクラスですから前節の方法が利用できないのです。

そのことを踏まえ、図 29.5 のプログラムを見ていきます。main メソッドの冒頭の

```
MyRunnable r1 = new MyRunnable("WAHAHA", 500); // Runnable オブジェ
MyRunnable r2 = new MyRunnable("WOOO", 1700);  // クトを生成

Thread t1 = new Thread(r1);              // Runnnale オブジェクトで
Thread t2 = new Thread(r2);              // スレッドを生成
```

は、前節の

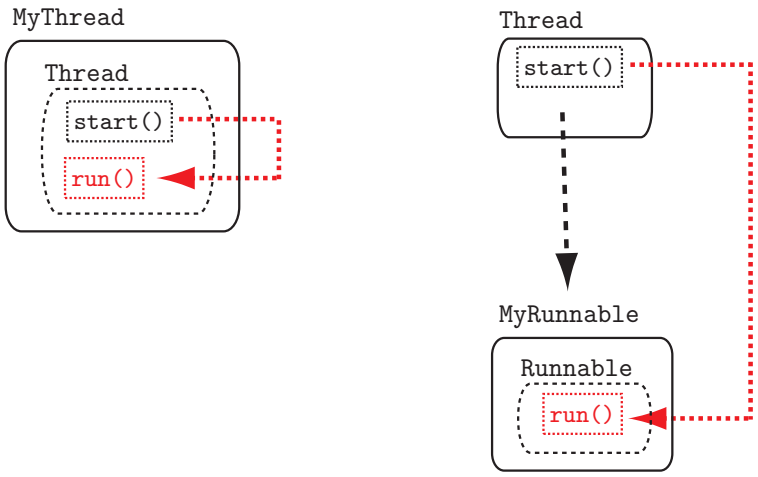
```
MyThread t1 = new MyThread("WAHAHA", 500);  // 0.5 秒の停止
MyThread t2 = new MyThread("WOOO", 1700);   // 1.7 秒の停止
```

に相当します。MyRunnable クラスの定義は、これも前節の MyThread クラスと同じ内容です。

このプログラムは前節と全く同様に実行されます。

```
import java.lang.Thread.*;

class Test {
    public static void main(String[] args) {
        MyRunnable r1 = new MyRunnable("WAHAHA", 500); // Runnable オブジェ
        MyRunnable r2 = new MyRunnable("WOOO", 1700);  // クトを生成

        Thread t1 = new Thread(r1);              // Runnnale オブジェクトで
        Thread t2 = new Thread(r2);              // スレッドを生成

        t1.start();
        t2.start();

        for (int i = 0; i < 5; i++) {
            System.out.println("WEEEE");
            Test.sleep(1000);
        }

        r1.flag = false;
        r2.flag = false;
    }

    static void sleep(int time) {
        try { Thread.sleep(time); }
        catch (Exception e){ System.exit(1); }
    }
}

class MyRunnable implements Runnable {           // 実装クラス
    String cry;
    int duration;
    boolean flag = true;

    MyRunnable(String cry, int duration) {
        this.cry = cry;
        this.duration = duration;
    }

    @Override
    public void run() {
        while (flag) {
            System.out.println(cry);
            Test.sleep(duration);
        }
    }
}
```

図 29.5: スレッドを用いるサンプルプログラム



(a) その 2 の場合　　(b) その 3 の場合

図 29.6: 簡単な例題（その 2）と（その 3）のオブジェクトの構造の違い：(a) その 2 の場合、`Thread` のサブクラスとして `MyThread` を定義しています。(b)　その 3 の場合、`Runnable` を `MyRunnable` で実装します。メソッド `run()` は (a)、(b) 共に上書きており、`start()` から呼び出されます。

29.4　プログラムの構造

図 27.8（506 ページ）はこの章のプログラムの構造です。

26 章から GUI へ移行したせいで全体に複雑になっていますが、この章での構造の変更は右下の赤い部分のみです。

29.5　ゲームクラスの変更

図 29.8（566 ページ）が、この章のゲームクラスのプログラムです。

コンストラクタ末尾には以下を追加します。

```
        for (Actor a : actors) {                // 末尾にこの for 文を追加
            if (!(a instanceof Player)) {       // Player 以外について
                (new Thread(a)).start();        // スレッドをスタート
            }
        }
```

これは、プレイヤー以外のアクター（つまり、モンスター、レッドモンスター）を子スレッドとしてスタートさせる処理です。これ以降、モンスター、レッドモンスターの動作には main スレッドはほとんど関知しません。

プレイヤーはこれまで同様、main スレッドで動作させますから、turn() を以下のように変更します。

```
    void turn() {
        actors.get(PLAYER_ID).action();      // Player のみ action() を実行
        print();
    }
```

前章までは、このメソッドでモンスター、レッドモンスターの action() も呼び出していましたが、それを除外しました。モンスター、レッドモンスターの action() は子スレッド内で勝手に実行されます。

クラスメソッド sleep() はスレッドを 3〜5 秒間停止させるメソッドです。実際の停止は Thread.sleep() が行なっています。停止時間は以下：

```
        int duration = (3+Game.random.nextInt(3))*1000;
```

で決定しています。

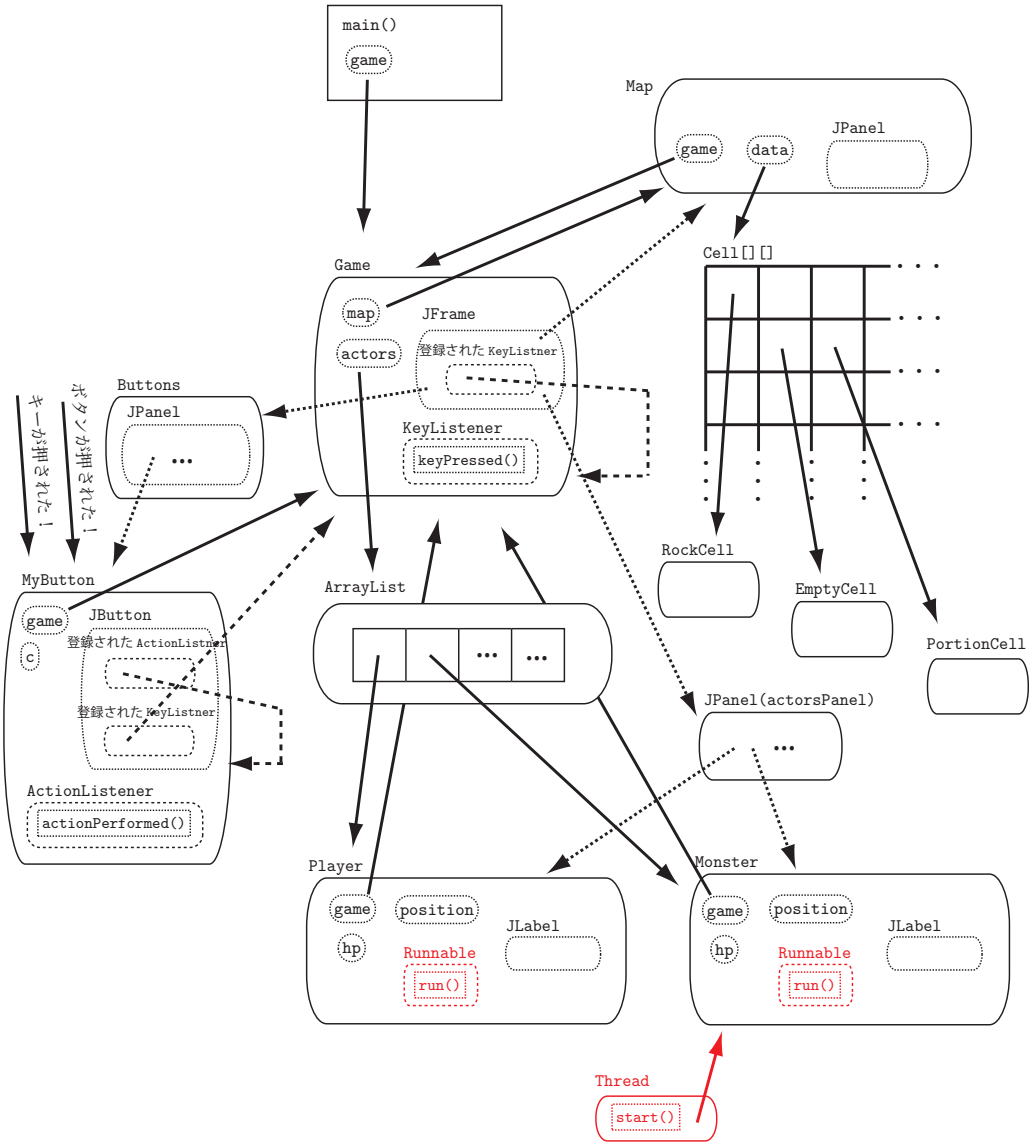


図 29.7: マルチスレッドで動作するプログラムの全体構造：アクターオブジェクトに Runnable クラスの run() メソッドを追加します。ただし、実際にはモンスターオブジェクト、レッドモンスターオブジェクトのみをスレッドオブジェクトから参照し、その run() を使用します。

```
class Game extends JFrame implements KeyListener {
    public static void main(String args[]) { ** 前章から変更なし ** }

    ** インスタンス変数などは前章から変更なし **

    Game() {
        ** 前半部は前章から変更なし **

        for (Actor a : actors) {                // 末尾にこの for 文を追加
            if (!(a instanceof Player)) {       // Player 以外について
                (new Thread(a)).start();        // スレッドをスタート
            }
        }
    }

    void print() { ** 前章から変更なし ** }

    void turn() {
        actors.get(PLAYER_ID).action();      // Player のみ action() を実行
        print();
    }

    ** 中略 **

    static void sleep() {                    // 3〜5 秒の一時停止を行う
        int duration = (3+Game.random.nextInt(3))*1000;
        try { Thread.sleep(duration); }
        catch (Exception e){ System.exit(1); }
    }
}
```

図 29.8: マルチスレッドを利用するダンジョンゲームのプログラム（ゲーム）

```
abstract class Actor extends JLabel implements Runnable { // Runnable の追加

    ** 以下の 1 行の追加を除き、前章から変更なし **

    @Override
    public void run(){}
}
```

図 29.9: マルチスレッドを利用するダンジョンゲームのプログラム（アクター）

```
class Monster extends Actor {

    ** 以下の追加を除き、前章から変更なし **

    @Override
    public void run() {        // 子スレッドで実行するメソッド
        while (hp > 0) {        // hp が正である限り
            action();            // 1 回の動作
            game.print();        // 表示の更新
            Game.sleep();        // 一時停止
        }
    }
}
```

図 29.10: マルチスレッドを利用するダンジョンゲームのプログラム（モンスター）

29.6　アクタークラスの変更

図 29.9（567 ページ）が、この章のアクタークラスのプログラムです。

変更箇所は 2 点です。ひとつは、`Runnable` インターフェースを実装する宣言を冒頭に加え、`public void run()` を何もしないメソッドとして定義します。これで、アクターとそのサブクラスが曲がりなりにもスレッドで実行できるようになります。

29.7　モンスタークラスの変更

図 29.10（567 ページ）が、この章のモンスタークラスのプログラムです。変更箇所は `run()` メソッドの追加のみです。以下：

```
        while (hp > 0) {        // hp が正である限り
```

の通り、hp が正である限り、モンスターは動き続けます。hp が 0 以下になったならば `run()` メソッドは終了し、それ以降、そのモンスターが行動を行うことはありません。

上記以外のプログラムの変更はありません。レッドモンスターはモンスターのサブクラスですから、上記のモンスタークラスのプログラムの変更はそのままレッドモンスターにも適用されます。

29.8　ゲームの実行について

ゲームを作った本人が言うのも変ですが、上のような僅かな変更だけでゲームがリアルタイムシミュレーションゲームに生まれ変わるのは驚きです。

この章のプログラムについて問題があるとすれば、モンスターの一時停止時間をどう適正に決めるかが悩ましいところです。

ここでは 3〜5 秒の乱数としましたが、乱数を用いないと全モンスターが同時に動いて同時に一時停止するため、不自然になるためです。

停止時間が短いとプレイヤーの判断がモンスターの行動に追いつかず、瞬殺されます。長すぎるとモンスターが止まっている間に逃げ切る作戦、またはモンスターが止まっている時間に連続攻撃を仕掛けて全モンスターを倒すことができます。たぶん、ゲーム的にちょうど良い停止時間は 3〜5 秒よりももう少し短いところにあるように思えます。

オブジェクト指向に関係ない与太話でした。

29.9　付録：この章の全プログラムリスト

```
//ゲームクラスのプログラム
import java.io.*;
import java.util.Random;
import java.util.ArrayList;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class Game extends JFrame implements KeyListener {
    public static void main(String args[]) {
        Game game = new Game();
    }

    static Random random = new Random();

    Map map;
    ArrayList<Actor> actors;
    final static int PLAYER_ID = 0;
    final static int NUM_OF_MONSTERS = 4;

    Game() {
        super("Dungeon Game");

        map = new Map(this);
        actors = new ArrayList<Actor>();
        actors.add(new Player(this, "Taro"));
        for (int i = 0; i < NUM_OF_MONSTERS; i++) {
            if (random.nextInt(2) == 0) {
                actors.add(new Monster(this));
            }
            else {
                actors.add(new RedMonster(this));
            }
        }

        setBounds(100, 100, Cell.SIZE * Map.XSIZE + 5 + 200, Cell.SIZE * Map.YSIZE + 25 + 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        add(map, BorderLayout.WEST);

        add(new MyButtons(this), BorderLayout.EAST);

        JPanel actorsPanel = new JPanel();
        actorsPanel.setLayout(new GridLayout(10, 1));
        for (Actor a : actors) { actorsPanel.add(a); }
        add(actorsPanel, BorderLayout.SOUTH);

        setVisible(true);

        for (Actor a : actors) {                // 末尾にこの for 文を追加
            if (!(a instanceof Player)) {       // Player 以外について
                (new Thread(a)).start();        // スレッドをスタート
            }
        }
    }

    void print() { repaint(); }
```

```
1
2       void turn() {
3           actors.get(PLAYER_ID).action();      // Player のみ action() を実行
4           print();
5       }
6
7       static char keyChar;
8       static char getKeyChar() { return keyChar; }
9       static void setKeyChar(char c) { keyChar = c; }
10
11      @Override
12      public void keyPressed(KeyEvent e) {
13          setKeyChar(e.getKeyChar());
14          turn();
15      }
16
17      @Override
18      public void keyReleased(KeyEvent e){}
19
20      @Override
21      public void keyTyped(KeyEvent e){}
22
23      static void sleep() {                    // 3〜5 秒の一時停止を行う
24          int duration = (3+Game.random.nextInt(3))*1000;
25          try { Thread.sleep(duration); }
26          catch (Exception e){ System.exit(1); }
27      }
28  }
```

```
//ボタンクラスのプログラム
class MyButtons extends JPanel {
    MyButtons(Game game) {
        setPreferredSize(new Dimension(150, 150));
        setBackground(Color.BLUE);
        setLayout(new GridLayout(3,3));
        add(new MyButton(game,'q'));
        add(new MyButton(game,'w'));
        add(new MyButton(game,'e'));
        add(new MyButton(game,'a'));
        add(new MyButton(game,'s'));
        add(new MyButton(game,'d'));
        add(new MyButton(game,'z'));
        add(new MyButton(game,'x'));
        add(new MyButton(game,'c'));
    }
}

class MyButton extends JButton implements ActionListener {
    Game game;
    char c;
    MyButton(Game game, char c) {
        super(String.valueOf(c));
        this.game = game;
        this.c = c;
        addActionListener(this);
        addKeyListener(game);
    }

    @Override
    public void actionPerformed(ActionEvent e){
        game.setKeyChar(c);
        game.turn();
    }
}
```

```
//マップクラスのプログラム
class Map extends JPanel {
    static final int XSIZE = 13;
    static final int YSIZE = 11;

    Game game;
    Cell data[][] = new Cell[XSIZE][YSIZE];

    Map(Game game) {
        this.game = game;
        for (int x = 0; x < XSIZE; x++) {
            for (int y = 0; y < YSIZE; y++) {
                if ((x == 0)||(x == XSIZE-1)||
                    (y == 0)||(y == YSIZE-1)||
                    ((Math.abs(x-XSIZE/2) <= 1)&&
                     (Math.abs(y-YSIZE/2) <= 1))) {
                    data[x][y] = new RockCell();
                }
                else {
                    switch (Game.random.nextInt(5)) {
                    case 0: data[x][y] = new PotionCell();
                             break;
                    case 1: data[x][y] = new PoisonCell();
                             break;
                    default: data[x][y] = new EmptyCell();
                    }
                }
            }
        }
        data[XSIZE-2][YSIZE-2] = new GoalCell();

        setPreferredSize(new Dimension(Cell.SIZE*XSIZE, Cell.SIZE*YSIZE));
    }

    Cell getAt(Int2 position) { return data[position.x][position.y]; }

    void setAt(Int2 position, Cell cell) {
        data[position.x][position.y] = cell;
    }

    @Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        for (int y = 0; y < YSIZE; y++) {
            for (int x = 0; x < XSIZE; x++) {
                Int2 position = new Int2(x, y);
                getAt(position).paintAt(g, position);
                for (Actor a : game.actors) {
                    a.paintAt(g, position);
                }
            }
        }
    }

    Int2 getRandomCellPosition() {
        while (true) {
            Int2 position = new Int2(Game.random.nextInt(XSIZE),
                                     Game.random.nextInt(YSIZE));
```

```

            if (!getAt(position).isRockCell()) { return position; }
        }
    }
}
```

```
//アクタークラスのプログラム
abstract class Actor extends JLabel implements Runnable {
    Game game;
    Int2 position;
    int hp;

    Actor(Game game, Int2 position, int hp) {
        super(" ");
        this.game = game;
        this.position = position;
        this.hp = hp;
    }

    Actor(Game game, int hp) {
        super(" ");
        this.game = game;
        this.hp = hp;
        while_loop: while (true) {
            position = game.map.getRandomCellPosition();
            for (Actor a : game.actors) {
                if (position.equal(a.position)) {
                    continue while_loop;
                }
            }
            return;
        }
    }

    @Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        setText(toString());
    }

    void paintAt(Graphics g, Int2 position) {
        if (this.position.equal(position)) {
            g.drawImage(getImage(),
                        position.x*Cell.SIZE + Cell.SIZE*2/24,
                        position.y*Cell.SIZE + Cell.SIZE*2/24,
                        Cell.SIZE*20/24,
                        Cell.SIZE*20/24,
                        game.map);
        }
    }

    abstract Image getImage();

    boolean moveTo(Int2 direction) {
        Int2 target = position.add(direction);

        Cell cell = game.map.getAt(target);
        if (cell.isRockCell()) { return false; }

        for (Actor a : game.actors) {
            if (target.equal(a.position)) { return false; }
        }

        Cell newCell = cell.affectOn(this);
```

```
        game.map.setAt(target, newCell);
        position = target;
        return true;
    }

    abstract void action();

    void attackTo(Int2 direction) {
        Int2 target = position.add(direction);
        for (Actor a : game.actors) {
            if (target.equal(a.position)) {
                a.takeDamageFrom(this);
                return;
            }
        }
    }

    abstract void takeDamageFrom(Actor a);
    void giveDamageTo(Player p) {}
    void giveDamageTo(Monster m) {}
    void giveDamageTo(RedMonster r) {}

    @Override
    public void run(){}
}
```

```
//プレイヤークラスのプログラム
class Player extends Actor {
    static final int INIT_HP = 100;
    String name;

    Player(Game game, String name) {
        super(game, new Int2(1, 1), INIT_HP);
        this.name = name;
    }

    Player(Game game) { this(game, "Player"); }

    public String toString() { return name+" : "+position+" "+hp; }

    @Override
    void takeDamageFrom(Actor a) { a.giveDamageTo(this); }

    @Override
    void giveDamageTo(Monster m) { m.hp -= 20; }

    @Override
    void giveDamageTo(RedMonster r) { r.hp -= 15; }

    @Override
    void action() {
        Int2 direction;
        switch (Game.getKeyChar()) {
            case 'z': direction = new Int2(-1, +1); break;
            case 'x': direction = new Int2( 0, +1); break;
            case 'c': direction = new Int2(+1, +1); break;
            case 'a': direction = new Int2(-1,  0); break;
            case 'd': direction = new Int2(+1,  0); break;
            case 'q': direction = new Int2(-1, -1); break;
            case 'w': direction = new Int2( 0, -1); break;
            case 'e': direction = new Int2(+1, -1); break;
            default: direction = new Int2(0, 0);
        }

        if (!moveTo(direction)) { attackTo(direction); }
    }

    static final Image image1 = Toolkit.getDefaultToolkit().getImage("Player1.png");
    static final Image image2 = Toolkit.getDefaultToolkit().getImage("Player2.png");

    @Override
    Image getImage() {
        if (hp > INIT_HP/2) { return image1; }
        else { return image2; }
    }
}
```

```
//モンスタークラスのプログラム
class Monster extends Actor {
    static final int INIT_HP = 150;

    Monster(Game game) { super(game, INIT_HP); }

    @Override
    public String toString() { return "Monster : "+position+" "+hp; }

    @Override
    void takeDamageFrom(Actor a) { a.giveDamageTo(this); }

    @Override
    void giveDamageTo(Player p) { p.hp -= 30; }

    @Override
    void giveDamageTo(RedMonster r) { r.hp -= 5; }

    @Override
    void action() {
        Actor player = game.actors.get(game.PLAYER_ID);
        Int2 direction = player.position.sub(position).signum();

        if (!moveTo(direction)) {
            attackTo(direction);
        }
    }

    static final Image image = Toolkit.getDefaultToolkit().getImage("Monster.png");

    @Override
    Image getImage() { return image; }

    @Override
    public void run() {       // 子スレッドで実行するメソッド
        while (hp > 0) {       // hp が正である限り
            action();           // 1 回の動作
            game.print();       // 表示の更新
            Game.sleep();       // 一時停止
        }
    }
}
```

```
//レッドモンスタークラスのプログラム
class RedMonster extends Monster {
    RedMonster(Game game) { super(game); }

    @Override
    public String toString() { return "Red" + super.toString(); }

    @Override
    void takeDamageFrom(Actor a) { a.giveDamageTo(this); }

    @Override
    void giveDamageTo(Player p) { p.hp -= 60; }

    @Override
    void giveDamageTo(Monster m) { m.hp -= 7; }

    static final Image image = Toolkit.getDefaultToolkit().getImage("RedMonster.png");

    @Override
    Image getImage() { return image; }
}
```

```
//2 次元ベクトルクラスのプログラム
class Int2 {
    int x,y;

    Int2(int x, int y) { this.x = x; this.y = y; }
    Int2 add(Int2 a) { return new Int2(x+a.x,y+a.y); }
    Int2 sub(Int2 a) { return new Int2(x-a.x,y-a.y); }
    Int2 signum() { return new Int2(Integer.signum(x),Integer.signum(y)); }
    boolean equal(Int2 a) { return (x == a.x)&&(y == a.y); }

    public String toString(){ return "("+x+","+y+")"; }
}
```

```
//セルクラスのプログラム
abstract class Cell {
    static final int SIZE = 48;

    boolean isRockCell() { return false; }

    abstract Cell affectOn(Actor a);

    abstract void paintAt(Graphics g, Int2 position);

    void drawCell(Graphics g, Int2 position) {
        g.drawRect(position.x*SIZE + SIZE/24,
                   position.y*SIZE + SIZE/24,
                   SIZE*22/24,
                   SIZE*22/24);
    }

    void fillCell(Graphics g, Int2 position) {
        g.fillRect(position.x*SIZE + SIZE/24,
                   position.y*SIZE + SIZE/24,
                   SIZE*22/24,
                   SIZE*22/24);
    }
}

abstract class SimpleCell extends Cell {
    @Override
    Cell affectOn(Actor a) { return this; }
}

abstract class TrapCell extends Cell {
    boolean visible = false;

    @Override
    Cell affectOn(Actor a) {
        visible = true;
        return this;
    }
}

class RockCell extends SimpleCell {
    @Override
    void paintAt(Graphics g, Int2 position) {
        g.setColor(new Color(100, 20, 20));
        fillCell(g, position);
    }

    @Override
    boolean isRockCell() { return true; }
}

class EmptyCell extends SimpleCell {
    @Override
    void paintAt(Graphics g, Int2 position) {
        g.setColor(new Color(128, 128, 128));
        drawCell(g, position);
    }
}
```

```
1
2  class PotionCell extends SimpleCell {
3      @Override
4      void paintAt(Graphics g, Int2 position) {
5          g.setColor(new Color(0, 255, 255));
6          fillCell(g, position);
7      }
8
9      @Override
10     Cell affectOn(Actor a) {
11         a.hp += 50;
12         return new EmptyCell();
13     }
14 }
15
16 class GoalCell extends SimpleCell {
17     @Override
18     void paintAt(Graphics g, Int2 position) {
19         g.setColor(new Color(255, 255, 0));
20         fillCell(g, position);
21     }
22
23     @Override
24     Cell affectOn(Actor a) {
25         if (a instanceof Player) {
26             System.out.println("YOU WIN");
27             System.exit(0);
28         }
29         return this;
30     }
31 }
32
33 class PoisonCell extends TrapCell {
34     @Override
35     void paintAt(Graphics g, Int2 position) {
36         if (visible) {
37             g.setColor(new Color(0, 128, 0));
38             fillCell(g, position);
39         } else {
40             g.setColor(new Color(128, 128, 128));
41             drawCell(g, position);
42         }
43     }
44
45     @Override
46     Cell affectOn(Actor a) {
47         a.hp -= 10;
48         return super.affectOn(a);
49     }
50 }
```

1

1

第30章　マップ兵器を作る：ラムダ式を関数の引数にする

プレイヤーが魔術的な兵器を手に入れたとします。

この兵器は、プレイヤーとモンスターの間の相対位置 (dx, dy) に関する条件で攻撃できるモンスターが決まるもので、モンスターの HP を半減させます。条件を満たすモンスターが複数居る場合には、その全てに攻撃が可能です。このような複数の敵を同時に攻撃できる兵器をマップ兵器[1]と呼びます。プレイヤーが使用できるマップ兵器には以下の 3 種類があるものとします。

水平・垂直攻撃 ... モンスターがプレイヤーから見て、正確に水平方向または垂直方向の位置に居るとき（つまり、$dx = 0$ または $dy = 0$ のとき）に、モンスターまでの距離に関係なく HP を半減させる。使用回数に制限はありません[2]。

斜め 45 度攻撃 ... モンスターがプレイヤーから見て、正確に斜め 45 度の方向に居るとき（つまり、$dx = dy$ または $dx = -dy$ のとき）に、モンスターまでの距離に関係なく HP を半減させる。使用回数に制限はありません。

全方位攻撃 ... マップ上の全てのモンスター（つまり、任意の (dx, dy) のとき）の HP を半減させる。ただしプレイヤーの HP は 1/10 に激減する。

さて、この兵器を実装するゲームプログラムでは、モンスターの相対位置 (dx, dy) が与えられたときに、そのモンスターを攻撃するか否かを判定する処理が必要です。Java ではそれを非常にコンパクトに記述できる機能 **ラムダ式**（lambda expression）が考案されています。これを学ぶのがこの章の真の目的です。ラムダ式はもともと関数型プログラミングのアイディア[3]であり、最近、特に有用性が認識されるようになってきています。最近のプログラミング言語の多くがこの種の機能を提供しており、今後、次世代のプログラミング技法として広く利用される可能性があります。

```
class Test {
    public static void main(String[] args) {
        int [] array = {1,2,3,4,5};

        Function mf2 = new MyFunction2();      // 関数オブジェクトの生成
        Function mf3 = new MyFunction3();

        mapCalcPrint(mf2, array);                  // マップ処理
        mapCalcPrint(mf3, array);
    }

    static void mapCalcPrint(Function func, int[] a) {
        for (int x : a) { System.out.print(func.f(x) + " "); }
        System.out.println();
    }
}

interface Function { int f(int x); }       // 1 引数関数を持つインターフェース

class MyFunction2 implements Function {
    @Override
    public int f(int x) { return 2*x; }    // 1 引数関数を実装
}

class MyFunction3 implements Function {
    @Override
    public int f(int x) { return 3*x; }    // 1 引数関数を実装
}
```

図 30.1: ラムダ式を用いない配列処理のサンプルプログラム

30.1　ラムダ式を用いない簡単な例

図 30.1（585 ページ）は、配列要素について計算と表示を行うプログラムです。

まず、演算を行う関数の外形（1 引数の関数 `int f(int)`）を以下のインターフェース `Function` で規定しておきます。

```
interface Function { int f(int x); }       // 1 引数関数を持つインターフェース
```

そして、このインターフェースのメソッド（関数） `int f(int)` をそれぞれ定義することとします。そして関数を表すオブジェクトをそれぞれ生成します。

[1]マップ兵器の正式英語名は “Mass Amplitude Preemptive-strike Weapon” 、略して MAP weapon となるらしいです。

[2]MP（magic point）を導入して、使用に制限を掛けるルールはよくありますが、煩雑なので採用しませんでした。

[3]関数型の分野にラムダ計算（lambda calculus、λ calculus）と呼ばれる計算システムがあります。この計算システムを素朴に紹介すると、たとえば x の二乗を計算する関数を $\lambda x.(x \times x)$ などと表現し、実際に 3^2 を計算するときには、3 を関数に適用し、 $(\lambda x.(x \times x))3 = 3 \times 3 = 9$ と計算します。「ラムダ」はこれに由来します。

```
        Function mf2 = new MyFunction2();       // 関数オブジェクトの生成
        Function mf3 = new MyFunction3();
```

これらはオブジェクトですが、その内容は関数 `int f(int)` が定義されているというだけのものです。次に、これらオブジェクトを引数としてメソッド `mapCalcPrint()` を呼び出します。

```
        mapCalcPrint(mf2, array);                       // マップ処理
        mapCalcPrint(mf3, array);
```

`mapCalcPrint()` では、

```
                        func.f(x)
```

を実行します。なお、

```
        for (int x : a)
```

は配列を対象とする拡張 `for` 文です。既に図 20.1.3（325 ページ）で解説した内容と同様のものです。

結果として、以下の実行結果を得ます。

```
2 4 6 8 10
3 6 9 12 15
```

```
class Test {
    public static void main(String[] args) {
        int [] array = {1,2,3,4,5};

        mapCalcPrint(y -> 2*y, array);
        mapCalcPrint(y -> 3*y, array);
    }

    static void mapCalcPrint(Function func, int[] a) {
        for (int x : a) { System.out.print(func.f(x) + " "); }
        System.out.println();
    }
}

interface Function { int f(int a); }
```

図 30.2: ラムダ式を用いる配列処理のサンプルプログラム

30.2　ラムダ式を用いる簡単な例

図 30.2（587 ページ）は前節の図 30.1（585 ページ）と同じ処理をラムダ式を用いて記述したプログラムです。図 30.2 は図 30.1 に比べ、格段にコンパクトになっています。というのも図 30.1 の以下の部分：

```
class MyFunction2 implements Function {
    @Override
    public int f(int x) { return 2*x; }    // 1 引数関数を実装
}

        Function mf2 = new MyFunction2();      // 関数オブジェクトの生成

        mapCalcPrint(mf2, array);                  // マップ処理
```

が図 30.2 では以下の 1 行に圧縮されているためです。

```
        mapCalcPrint(y -> 2*y, array);
```

その理由は以下の通りです。Java コンパイラは上の 1 行を以下のように解析します。

1. まず、式 `y -> 2*y` に演算子 `->` が含まれることからは、Java は `y -> 2*y` はラムダ記法であり、あるメソッドの略記法[4]であると判断します。`->` の左辺はメソッドの引数、右辺は戻

[4]本質を変えず、プログラムの書き易さのために導入されたルールを構文的糖衣（syntax sugaring）と呼びます。砂糖をまぶして食べ易くしている訳です。プログラミング言語ではこの種のルールがしばしば利用されます。

り値と約束されています。よって、 y -> 2*y を旧来のメソッドで表現すれば、以下のような形状です。ここに ?1、...、?3 に入るプログラム辺はまだ不明です。

```
?1 ?2(?3 y) { return 2*y; }
```

2. 次に、mapCalcPrint() のヘッダ部（宣言部）

```
void mapCalcPrint(Function func, int[] a)
```

ですから、y -> 2*y は Function インターフェースを実装するクラスの抽象メソッドに相当します。その Function インターフェースにはメソッドがひとつしか宣言されていません（ここが重要です！）から、y -> 2*y はその唯一のメソッドの宣言 int f(int x) とマッチせねばなりません。

3. となれば、?1 = int、?2 = f、?3 = int のはずです。結果として、y -> 2*y から Function インターフェースを実装する以下のクラスをコンパイラが自動的に作り出すことができます。ただし、_F1111_ はコンパイラが内部処理用に適当に作り出したクラス名と仮定します。

```
class _F1111_ implements Function {
    @Override
    public int f(int x) { return 2*x; }    // 1 引数関数を実装
}
```

4. 最後に

```
mapCalcPrint(y -> 2*y, array);
```

を以下に置き換えればプログラムの変換は終了です。ただし、_f1111_ はコンパイラが内部処理用に適当に作り出した変数名と仮定します。

```
Function _f1111_ = new _F1111_();
mapCalcPrint(_f1111_, array);
```

上のように、与えられたプログラムの整合性を基に、コンパイラがプログラムを自動的に補完する処理を**型推論**（type inference）と呼んでいます。最近の多くのプログラミング言語処理系では多かれ少なかれ型推論を行なっており、プログラム作成の省力化、プログラム内の矛盾のチェックに役立てています。

補足：より複雑な計算式　ラムダ式で複雑な計算を行いたい場合には、右辺にブロック `{ }` を用い、Java のプログラムを記述することもできます。たとえば入力値 a の階乗 $a!$ を計算したい場合、ラムダ式を以下のように記述可能です。

```
a->{
        int r = 1;
        for (int i = 1; i <= a; i++) { r *= i; }
        return r;
    }
```

表 30.1: Java が提供する関数型インターフェース

種類	インターフェース名	メソッド
コンシューマー型 （戻り値がない）	`Consumer<T>` `IntConsumer`	`void accept(T t)` `void accept(int value)`
述語型 (真偽値を返す)	`Predicate<T>` `IntPredicate`	`boolean test(T t)` `boolean test(int value)`
オブジェクト関数型 （オブジェクトを返す）	`Function<T,R>` `IntFunction<R>`	`R apply(T t)` `R apply(int value)`
`Int` 関数型 （`int` を返す）	`ToIntFunction<T>` `DoubleToIntFunction<R>`	`int applyAsInt(T t)` `int applyAsInt(double value)`
単項演算子型（引数と戻り値が同じ型）	`UnaryOperator<T>` `IntUnaryOperator`	`T apply(T t)` `int applyAsInt(int value)`
二項演算子型（ふたつの引数と戻り値が同じ型）	`BinaryOperator<T>` `IntBinaryOperator`	`T apply(T t, T u)` `int applyAsInt(int left, int right)`

ただし、`T`、`R` はいずれもオブジェクト型

30.3　標準の関数型インターフェースを用いる例

前節のプログラムの型推論がうまく行ったのは、インターフェースにメソッドがひとつしか宣言されていなかったためです。このような、メソッドをひとつしか持たないインターフェースを特に**関数型インターフェース**（functional interface）と呼んでいます。Java のラムダ式は関数型インターフェースが前提となってうまく機能します。

よく利用される関数型インターフェースには一定のパターン — たとえばひとつの整数を引数として整数を戻り値とする場合、または、二つの整数を引数として真偽値を戻り値とする場合など — がありますから、それらを事前に定義しておくとさらに便利です。実際、Java ではそれらのパターン 43 種類をパッケージ `java.util.function.*` で提供しています。代表的なパターンを抜粋して表 30.1 にまとめました。全てのパターンについては以下を参照してください。

JavaPlatform, Standard Edition 8
パッケージ java.util.function
https://docs.oracle.com/javase/jp/8/docs/api/java/util/function/package-summary.html

Java 関数型インターフェース
http://www.ne.jp/asahi/hishidama/home/tech/java/functionalinterface.html

前節のプログラムの

```
import java.util.function.*;

class Test {
    public static void main(String[] args) {
        int [] array = {1,2,3,4,5};

        mapCalcPrint(y->2*y, array);
        mapCalcPrint(y->3*y, array);
    }

    static void mapCalcPrint(IntUnaryOperator func, int[] a) {
        for (int x : a) { System.out.print(func.applyAsInt(x) + " "); }
        System.out.println();
    }
}
```

図 30.3: システム標準の関数型インターフェースとラムダ式を用いる配列処理のサンプルプログラム

```
interface Function { int f(int a); }
```

にマッチするシステム標準定義の関数型インターフェースは以下です。

```
interface  IntUnaryOperation { int applAsint(int a); }
```

そこで、このインターフェースを用いて前節のプログラムを書き換えると、図 30.3（591 ページ）の通りです。

```
interface MapFunction {
    boolean test(int x, int y);
}
```

図 30.4: ラムダ式を利用するダンジョンゲームのプログラム（関数型インターフェース）

30.4　マップ兵器のラムダ式

プレイヤーの 3 種類のマップ兵器は、相対位置 (dx, dy) から真偽値（攻撃位置にマッチすれば真、さもなくば偽）を求めるものです。

これをラムダ式で表すと以下の通りです。ここに `(dx, dy)` は 2 引数を表します。

水平・垂直攻撃 ... `(dx, dy) -> ((dx == 0) || (dy == 0))`

斜め 45 度攻撃 ... `(dx, dy) -> ((dx == dy) || (dx == -dy))`

全方位攻撃 ... `(dx, dy) -> true`

以下、これをダンジョンゲームに組み込みます。ただし、前章のマルチスレッド実行はゲームとして扱いにくい[5]ため、前々章のプログラムを基に改造を行います。実際には前章のプログラムを基に改造しても全く同じ改造が可能です。

30.5　関数型インターフェースの導入

まず、前節のラムダ式に合う関数型インターフェースを図 30.4（592 ページ）のように新規導入します。

30.6　アクタークラスの変更

前節のインターフェースを使ってマップ兵器での攻撃を行うインスタンスメソッド `mapAttack()` を図 30.5（593 ページ）のようにアクタークラスに導入します。このメソッドでは、自分以外のアクター：

```
            if (a != this) {            // 自分以外ならば
```

について、関数型インターフェースを実装するオブジェクト `mf` を用いて条件判定：

[5]ゲーム操作に焦ってしまい、機能の検証をゆっくりとできそうにありません。

```
abstract class Actor extends JLabel {
    ** 以下の追加を除いて、前章から変更なし **

    void mapAttack(MapFunction mf) {  // マップ兵器での攻撃
        for (Actor a : game.actors) {  // 各アクターについて
            if (a != this) {            // 自分以外ならば
                Int2 relpos = a.position.sub(this.position);//相対位置を計算
                if (mf.test(relpos.x, relpos.y)) { // 条件がマッチするならば
                    a.hp /= 2;            // hp を半減させる
                }
            }
        }
    }
}
```

図 30.5: ラムダ式を利用するダンジョンゲームのプログラム（アクター）

```
                if (mf.test(relpos.x, relpos.y)) { // 条件がマッチするならば
```

が真ならば、hp を半減させます。

ここではプレイヤーの攻撃を考えますから、このメソッドは本来プレイヤークラスに実装すべきですが、次章で導入する魔道士の攻撃にもこれを使おうと考え、アクタークラスに実装しました。

30.7　プレイヤークラスの変更

図 30.6（594 ページ）がプレイヤークラスの変更です。新たに入力文字 ‘1’、‘2’、‘3’ に対する処理を追加しました。

以下が、マップ攻撃の部分です。ラムダ式で攻撃できる条件を指定しています。

```
                mapAttack((dx, dy) -> ((dx == 0) || ( dy == 0)));
                                        // 引数のラムダ式の条件でマップ攻撃
```

攻撃終了後は移動もせず、直接物理攻撃も行わずメソッドを終了します。

30.8　モンスタークラスの変更

図 30.7（595 ページ）がモンスタークラスの変更です。この章の目的とは無関係なのですが、ゲーム的にそろそろ入れておいた方がよいと判断し、hp が 0 以下になったならば活動を完全停止するように 1 行を加えました。

```
class Player extends Actor {
    ** 以下を除いて、前章から変更なし **

    @Override
    void action() {
        Int2 direction;
        switch (Game.getKeyChar()) {

            ** 中略 **

            case '1':      // 1 がキー入力された（1 のボタンが押された）ならば
                mapAttack((dx, dy) -> ((dx == 0) || ( dy == 0)));
                                        // 引数のラムダ式の条件でマップ攻撃
                return;              // 移動も直接物理攻撃もしないでリターン

            case '2':
                mapAttack((dx, dy) -> ((dx == dy) || ( dx == -dy)));
                return;

            case '3':
                mapAttack((dx,dy) -> true);
                hp /= 10;
                return;

            default: direction = new Int2(0, 0);
        }

        if (!moveTo(direction)) { attackTo(direction); }
    }
}
```

図 30.6: ラムダ式を利用するダンジョンゲームのプログラム（プレイヤー）

30.9　ボタンクラスの変更

図 30.8（595 ページ）が、ボタンを配置する MyButtons クラスの変更です。この JPanel に配置する GUI 部品の数を増やすために

```
        setLayout(new GridLayout(4,3));    // 1 行増やす
```

3×3 を 4×3 へ変更し、入力文字を '1'、'2'、'3' に対応するボタンを追加配置するだけです。

30.10　プログラムの実行例

図 30.9（596 ページ）は上の改造後のプログラムの実行の例です。マップ兵器は正しく機能しています。

```
class Monster extends Actor {
    ** 以下を除いて、前章から変更なし **

    @Override
    void action() {
        if (hp <= 0) return;   // hp が 0 以下になったならば活動停止

        ** 後略 **
    }
}
```

図 30.7: ラムダ式を利用するダンジョンゲームのプログラム（モンスター）

```
class MyButtons extends JPanel {
    MyButtons(Game game) {
        setPreferredSize(new Dimension(150, 150));
        setBackground(Color.BLUE);
        setLayout(new GridLayout(4,3));    // 1 行増やす

        ** 中略 **

        add(new MyButton(game,'1'));       // 4 行目に追加配置
        add(new MyButton(game,'2'));       // 4 行目に追加配置
        add(new MyButton(game,'3'));       // 4 行目に追加配置
    }
}
```

図 30.8: ラムダ式を利用するダンジョンゲームのプログラム（ボタン）

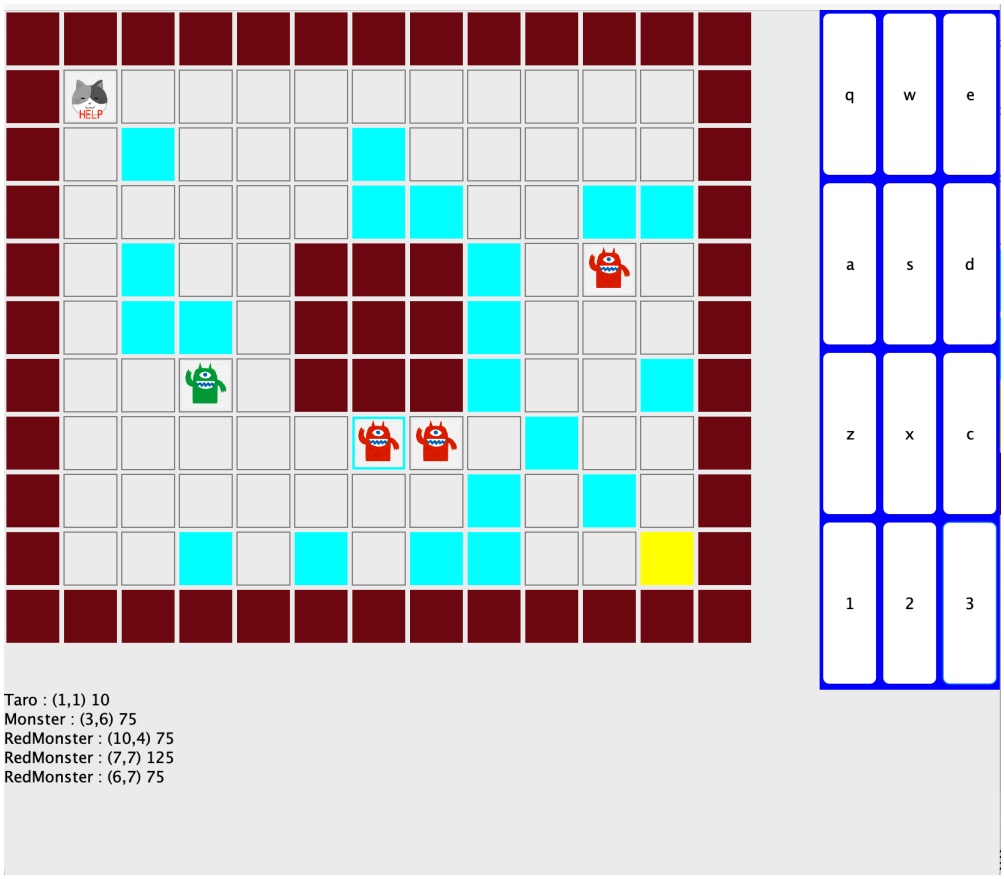


図 30.9: この章のプログラムの実行の例：ボタンが 3 個増えています。この画像はゲーム開始後すぐに ‘3’ のマップ兵器で攻撃した直後です（モンスターの HP が 75 に半減し、プレイヤーの HP が 1/10 になっています）

30.11　補足：関数的プログラミング

30.11.1　第 1 級関数

ほとんど全てのプログラミング言語では整数型（int 型）データを自由に取り扱うことができます。ここでいう「自由に取り扱う」とは、たとえば、値をプログラム中で明示できる（たとえばプログラム中の “234” は二百三十四を表すなど）、変数に代入できる、関数の引数に設定できる、関数の戻り値に設定できる、値を実行時に生成できるなどの性質を持つことを言います。そしてこれらの性質を持つデータを**第 1 級データ**（first class data）と呼びます。

さて、この節で問題にしたいのは関数についてです。背景にあるのは、整数などと同様に関数も処理の対象として自由自在に操作できるべきという関数型プログラミングのアイディアです。それが可能であるとき、そのような関数を**第 1 級関数**（first class function）と呼びます[6]。第 1 級関数では、変数に関数を代入できる、関数の引数に関数を設定できる、関数の戻り値に関数を設定できる、関数を実行時に生成できるなどの性質を持つことを言います。最近では第 1 級関数を扱うことのできるプログラミング言語が増えてきており、関数的プログラミングは今後のトレンドとなっていく可能性があります。

1990 年代半ばに開発された Java は残念ながら第 1 級関数を扱うことができません。しかし、2014 年にこの章で解説したラムダ式が言語仕様に取り入れられました[7]。Java のラムダ式は実際にはインターフェースを実装するクラスのオブジェクトであるため完全な第 1 級関数ではなく、多少の紛（まが）い物感はあるにせよ、関数型プログラミング風のプログラミングをサポートできるようになった訳です。

以下、第 1 級関数としてのラムダ式を眺めてみます。

まず、関数をプログラム中に明示することはこの章で述べました。以下がそれでした。

```
(dx,dy) -> ((dx == 0)||(dy == 0))
```

ただし、この式だけでは dx、dy のデータ型が不明ですから、これでは不完全な記述です。データ型は別途、関数型インターフェースと整合されて決定されることは解説した通りです。

次に、関数を変数に代入することも可能です。たとえば図 30.2（587 ページ）のプログラムにおいて main メソッドに以下の代入文を置くことができます。

[6] 第 1 級データと第 1 級関数を総称して**第 1 級オブジェクト**とも呼びます。ここでいうオブジェクトはオブジェクト指向のオブジェクトとは別ものです。

[7] 2014 年にはその他にも様々な機能が取り入れられています。余裕のある人は「java ラムダ式」を糸口にして検索してみてください。

```
Function f = a->4*a;
```

関数を関数の引数にすることは図 30.2 などで見た通りです。ただし、実際に引数になっているのはオブジェクトです。関数を関数の戻り値にすることも可能です。ただし実際に戻されるのはオブジェクトです。

関数を実行時に新たに生成することも可能です。ただし、実際に生成されるのはオブジェクトです。関数を実行時に加工することはできません。たとえば二つの関数の合成関数を実行時に作ることなどはできません。Java では全ての関数はコンパイル時にコンパイルされて内容が確定している必要があるからです。

30.11.2　高階関数

プログラミングで扱うデータや関数（メソッド）に以下のようにレベルを設定してみます。

レベル 0 ... 数値や文字列、リストなどの、通常、私たちがデータと認識している基本データ型や構造型データ

レベル 1 ... レベル 0 のデータを引数や戻り値とし、それらデータを加工する関数。通常の関数

レベル 2 ... レベル 1 以下のデータ（データ、関数）を引数や戻り値とし、それらデータを加工する関数。

レベル 3 ... レベル 2 以下の... 以下同様

一般にレベル 2 以上の関数を**高階関数**（higher-order function）と呼びます。

この章で用いたアクタークラスのメソッド `mapAttack()` は、マップ兵器の攻撃パターン（これは二つの整数を引数とし、真偽値を返す関数）を引数とする高階関数です。攻撃パターンを入れ替えるだけで様々なマップ攻撃が実現できる非常に便利な仕組みと言えます。なお、関数とデータ列を引数として与え、列の各要素に順次、関数を適用していくことができる高階関数はよく知られており、一般に**マップ関数**（map function）と呼ばれています（Wikipedia「高階関数」参照）。

前項にも述べたように、関数を関数の引数にできる仕組みは現在のプログラミング言語の必須機能となっています。C/C++の**関数ポインタ**（function pointer）もそれに近い機能です。

30.12　付録：この章の全プログラムリスト

```
//ゲームクラスのプログラム
import java.io.*;
import java.util.Random;
import java.util.ArrayList;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class Game extends JFrame implements KeyListener {
    public static void main(String args[]) {
        Game game = new Game();
    }

    static Random random = new Random();

    Map map;
    ArrayList<Actor> actors;
    final static int PLAYER_ID = 0;
    final static int NUM_OF_MONSTERS = 4;

    Game() {
        super("Dungeon Game");

        map = new Map(this);
        actors = new ArrayList<Actor>();
        actors.add(new Player(this, "Taro"));
        for (int i = 0; i < NUM_OF_MONSTERS; i++) {
            if (random.nextInt(2) == 0) {
                actors.add(new Monster(this));
            }
            else {
                actors.add(new RedMonster(this));
            }
        }

        setBounds(100, 100,
                  Cell.SIZE * Map.XSIZE + 5 + 200,
                  Cell.SIZE * Map.YSIZE + 25 + 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        add(map, BorderLayout.WEST);

        add(new MyButtons(this), BorderLayout.EAST);

        JPanel actorsPanel = new JPanel();
        actorsPanel.setLayout(new GridLayout(10, 1));
        for (Actor a : actors) { actorsPanel.add(a); }
        add(actorsPanel, BorderLayout.SOUTH);

        setVisible(true);
    }

    void print() { repaint(); }

    void turn() {
        for (Actor a : actors) { a.action(); }
        print();
```

```
    }

    static char keyChar;
    static char getKeyChar() { return keyChar; }
    static void setKeyChar(char c) { keyChar = c; }

    @Override
    public void keyPressed(KeyEvent e) {
        setKeyChar(e.getKeyChar());
        turn();
    }

    @Override
    public void keyReleased(KeyEvent e){}

    @Override
    public void keyTyped(KeyEvent e){}
}
```

```
//ボタンクラスのプログラム
class MyButtons extends JPanel {
    MyButtons(Game game) {
        setPreferredSize(new Dimension(150, 150));
        setBackground(Color.BLUE);
        setLayout(new GridLayout(4,3));    // 1 行増やす

        add(new MyButton(game,'q'));
        add(new MyButton(game,'w'));
        add(new MyButton(game,'e'));
        add(new MyButton(game,'a'));
        add(new MyButton(game,'s'));
        add(new MyButton(game,'d'));
        add(new MyButton(game,'z'));
        add(new MyButton(game,'x'));
        add(new MyButton(game,'c'));
        add(new MyButton(game,'1'));       // 4 行目に追加配置
        add(new MyButton(game,'2'));       // 4 行目に追加配置
        add(new MyButton(game,'3'));       // 4 行目に追加配置
    }
}

class MyButton extends JButton implements ActionListener {
    Game game;
    char c;
    MyButton(Game game, char c) {
        super(String.valueOf(c));
        this.game = game;
        this.c = c;
        addActionListener(this);
        addKeyListener(game);
    }

    @Override
    public void actionPerformed(ActionEvent e){
        game.setKeyChar(c);
        game.turn();
    }
}
```

```
//マップクラスのプログラム
class Map extends JPanel {
    static final int XSIZE = 13;
    static final int YSIZE = 11;

    Game game;
    Cell data[][] = new Cell[XSIZE][YSIZE];

    Map(Game game) {
        this.game = game;
        for (int x = 0; x < XSIZE; x++) {
            for (int y = 0; y < YSIZE; y++) {
                if ((x == 0)||(x == XSIZE-1)||
                    (y == 0)||(y == YSIZE-1)||
                    ((Math.abs(x-XSIZE/2) <= 1)&&
                     (Math.abs(y-YSIZE/2) <= 1))) {
                    data[x][y] = new RockCell();
                }
                else {
                    switch (Game.random.nextInt(5)) {
                    case 0: data[x][y] = new PotionCell();
                            break;
                    case 1: data[x][y] = new PoisonCell();
                            break;
                    default: data[x][y] = new EmptyCell();
                    }
                }
            }
        }
        data[XSIZE-2][YSIZE-2] = new GoalCell();

        setPreferredSize(new Dimension(Cell.SIZE*XSIZE, Cell.SIZE*YSIZE));
    }

    Cell getAt(Int2 position) { return data[position.x][position.y]; }

    void setAt(Int2 position, Cell cell) {
        data[position.x][position.y] = cell;
    }

    @Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        for (int y = 0; y < YSIZE; y++) {
            for (int x = 0; x < XSIZE; x++) {
                Int2 position = new Int2(x, y);
                getAt(position).paintAt(g, position);
                for (Actor a : game.actors) {
                    a.paintAt(g, position);
                }
            }
        }
    }

    Int2 getRandomCellPosition() {
        while (true) {
            Int2 position = new Int2(Game.random.nextInt(XSIZE),
                                     Game.random.nextInt(YSIZE));
```

```

            if (!getAt(position).isRockCell()) { return position; }
        }
    }
}
```

```
//アクタークラスのプログラム
abstract class Actor extends JLabel {
    Game game;
    Int2 position;
    int hp;

    Actor(Game game, Int2 position, int hp) {
        super(" ");
        this.game = game;
        this.position = position;
        this.hp = hp;
    }

    Actor(Game game, int hp) {
        super(" ");
        this.game = game;
        this.hp = hp;
        while_loop: while (true) {
            position = game.map.getRandomCellPosition();
            for (Actor a : game.actors) {
                if (position.equal(a.position)) {
                    continue while_loop;
                }
            }
            return;
        }
    }

    @Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        setText(toString());
    }

    void paintAt(Graphics g, Int2 position) {
        if (this.position.equal(position)) {
            g.drawImage(getImage(),
                        position.x*Cell.SIZE + Cell.SIZE*2/24,
                        position.y*Cell.SIZE + Cell.SIZE*2/24,
                        Cell.SIZE*20/24,
                        Cell.SIZE*20/24,
                        game.map);
        }
    }

    abstract Image getImage();

    boolean moveTo(Int2 direction) {
        Int2 target = position.add(direction);

        Cell cell = game.map.getAt(target);
        if (cell.isRockCell()) { return false; }

        for (Actor a : game.actors) {
            if (target.equal(a.position)) { return false; }
        }

        Cell newCell = cell.affectOn(this);
```

```
        game.map.setAt(target, newCell);
        position = target;
        return true;
    }

    abstract void action();

    void attackTo(Int2 direction) {
        Int2 target = position.add(direction);
        for (Actor a : game.actors) {
            if (target.equal(a.position)) {
                a.takeDamageFrom(this);
                return;
            }
        }
    }

    abstract void takeDamageFrom(Actor a);
    void giveDamageTo(Player p) {}
    void giveDamageTo(Monster m) {}
    void giveDamageTo(RedMonster r) {}

    void mapAttack(MapFunction mf) {  // マップ兵器での攻撃
        for (Actor a : game.actors) {  // 各アクターについて
            if (a != this) {            // 自分以外ならば
                Int2 relpos = a.position.sub(this.position);//相対位置を計算
                if (mf.test(relpos.x, relpos.y)) { // 条件がマッチするならば
                    a.hp /= 2;            // hp を半減させる
                }
            }
        }
    }
}

interface MapFunction {
    boolean test(int x, int y);
}
```

```
//プレイヤークラスのプログラム
class Player extends Actor {
    static final int INIT_HP = 100;
    String name;

    Player(Game game, String name) {
        super(game, new Int2(1, 1), INIT_HP);
        this.name = name;
    }

    Player(Game game) { this(game, "Player"); }

    public String toString() { return name+" : "+position+" "+hp; }

    @Override
    void takeDamageFrom(Actor a) { a.giveDamageTo(this); }

    @Override
    void giveDamageTo(Monster m) { m.hp -= 20; }

    @Override
    void giveDamageTo(RedMonster r) { r.hp -= 15; }

    @Override
    void action() {
        Int2 direction;
        switch (Game.getKeyChar()) {
            case 'z': direction = new Int2(-1, +1); break;
            case 'x': direction = new Int2( 0, +1); break;
            case 'c': direction = new Int2(+1, +1); break;
            case 'a': direction = new Int2(-1,  0); break;
            case 'd': direction = new Int2(+1,  0); break;
            case 'q': direction = new Int2(-1, -1); break;
            case 'w': direction = new Int2( 0, -1); break;
            case 'e': direction = new Int2(+1, -1); break;

            case '1':      // 1 がキー入力された（1 のボタンが押された）ならば
                mapAttack((dx, dy) -> ((dx == 0) || ( dy == 0)));
                                                // 引数のラムダ式の条件でマップ攻撃
                return;              // 移動も直接物理攻撃もしないでリターン

            case '2':
                mapAttack((dx, dy) -> ((dx == dy) || ( dx == -dy)));
                return;

            case '3':
                mapAttack((dx,dy) -> true);
                hp /= 10;
                return;

            default: direction = new Int2(0, 0);
        }

        if (!moveTo(direction)) { attackTo(direction); }
    }

    static final Image image1 = Toolkit.getDefaultToolkit().getImage("Player1.png");
    static final Image image2 = Toolkit.getDefaultToolkit().getImage("Player2.png");
```

```

    @Override
    Image getImage() {
        if (hp > INIT_HP/2) { return image1; }
        else { return image2; }
    }
}
```

```
//モンスタークラスのプログラム
class Monster extends Actor {
    static final int INIT_HP = 150;

    Monster(Game game) { super(game, INIT_HP); }

    @Override
    public String toString() { return "Monster : "+position+" "+hp; }

    @Override
    void takeDamageFrom(Actor a) { a.giveDamageTo(this); }

    @Override
    void giveDamageTo(Player p) { p.hp -= 30; }

    @Override
    void giveDamageTo(RedMonster r) { r.hp -= 5; }

    @Override
    void action() {
        if (hp <= 0) return;   // hp が 0 以下になったならば活動停止

        Actor player = game.actors.get(game.PLAYER_ID);
        Int2 direction = player.position.sub(position).signum();

        if (!moveTo(direction)) {
            attackTo(direction);
        }
    }
}
```

```
//レッドモンスタークラスのプログラム
class RedMonster extends Monster {
    RedMonster(Game game) { super(game); }

    @Override
    public String toString() { return "Red" + super.toString(); }

    @Override
    void takeDamageFrom(Actor a) { a.giveDamageTo(this); }

    @Override
    void giveDamageTo(Player p) { p.hp -= 60; }

    @Override
    void giveDamageTo(Monster m) { m.hp -= 7; }

    static final Image image = Toolkit.getDefaultToolkit().getImage("RedMonster.png");

    @Override
    Image getImage() { return image; }
}
```

```
//2 次元ベクトルクラスのプログラム
class Int2 {
    int x,y;

    Int2(int x, int y) { this.x = x; this.y = y; }
    Int2 add(Int2 a) { return new Int2(x+a.x,y+a.y); }
    Int2 sub(Int2 a) { return new Int2(x-a.x,y-a.y); }
    Int2 signum() { return new Int2(Integer.signum(x),Integer.signum(y)); }
    boolean equal(Int2 a) { return (x == a.x)&&(y == a.y); }

    public String toString(){ return "("+x+","+y+")"; }
}
```

```
//セルクラスのプログラム
abstract class Cell {
    static final int SIZE = 48;

    boolean isRockCell() { return false; }

    abstract Cell affectOn(Actor a);

    abstract void paintAt(Graphics g, Int2 position);

    void drawCell(Graphics g, Int2 position) {
        g.drawRect(position.x*SIZE + SIZE/24,
                   position.y*SIZE + SIZE/24,
                   SIZE*22/24,
                   SIZE*22/24);
    }

    void fillCell(Graphics g, Int2 position) {
        g.fillRect(position.x*SIZE + SIZE/24,
                   position.y*SIZE + SIZE/24,
                   SIZE*22/24,
                   SIZE*22/24);
    }
}

abstract class SimpleCell extends Cell {
    @Override
    Cell affectOn(Actor a) { return this; }
}

abstract class TrapCell extends Cell {
    boolean visible = false;

    @Override
    Cell affectOn(Actor a) {
        visible = true;
        return this;
    }
}

class RockCell extends SimpleCell {
    @Override
    void paintAt(Graphics g, Int2 position) {
        g.setColor(new Color(100, 20, 20));
        fillCell(g, position);
    }

    @Override
    boolean isRockCell() { return true; }
}

class EmptyCell extends SimpleCell {
    @Override
    void paintAt(Graphics g, Int2 position) {
        g.setColor(new Color(128, 128, 128));
        drawCell(g, position);
    }
}
```

```
1
2  class PotionCell extends SimpleCell {
3      @Override
4      void paintAt(Graphics g, Int2 position) {
5          g.setColor(new Color(0, 255, 255));
6          fillCell(g, position);
7      }
8
9      @Override
10     Cell affectOn(Actor a) {
11         a.hp += 50;
12         return new EmptyCell();
13     }
14 }
15
16 class GoalCell extends SimpleCell {
17     @Override
18     void paintAt(Graphics g, Int2 position) {
19         g.setColor(new Color(255, 255, 0));
20         fillCell(g, position);
21     }
22
23     @Override
24     Cell affectOn(Actor a) {
25         if (a instanceof Player) {
26             System.out.println("YOU WIN");
27             System.exit(0);
28         }
29         return this;
30     }
31 }
32
33 class PoisonCell extends TrapCell {
34     @Override
35     void paintAt(Graphics g, Int2 position) {
36         if (visible) {
37             g.setColor(new Color(0, 128, 0));
38             fillCell(g, position);
39         } else {
40             g.setColor(new Color(128, 128, 128));
41             drawCell(g, position);
42         }
43     }
44
45     @Override
46     Cell affectOn(Actor a) {
47         a.hp -= 10;
48         return super.affectOn(a);
49     }
50 }
```

1

1

第31章　ウィザードもマップ兵器を操る

この章は前章の続きです。ラムダ式でマップ兵器が簡単に記述できることが分かりましたから、この章ではマップ兵器を操る敵側のキャラクター、ウィザード[1]（Wizard）をプログラム化します。

31.1　ウィザードクラス

図 31.1（616 ページ）は、ウィザードクラスのプログラムです。

まず、たったこれだけの記述で新しい種類のキャラクタをゲームに追加できる点に注目してください。それが可能な理由のひとつは、アクタークラスを継承しており、重要な処理はアクタークラスのプログラムから自動的に借用できること、もうひとつは多相性によってウィザードとして必要なメソッドのみを上書き追加すれば、必要な時点で必要なメソッドが自動的に呼び出されることです。

以下が必要なことはダブルディスパッチ法（図 24（410 ページ））を復習してください。

```
    @Override
    void takeDamageFrom(Actor a) { a.giveDamageTo(this); }
                                                    // ダブルディスパッチ法を参照
```

関連して、アクタークラス、プレイヤークラスについてもダブルディスパッチ法に関する追加があります。

以下の `action()` がウィザードの行動です。

```
    @Override
    void action() {
        if (hp <= 0) return;

        mapAttack((x,y)->((x == 0) || ( y == 0)));       // 水平・垂直攻撃
    }
```

[1]クラス名をどう決めるか悩みました。魔道士が一番しっくりくるのですが、プログラム化する上では英語名から派生したカタカナ名がよいと考え、ウィザード、ウィッチ、メイジ、トロール、オーク、エルフなどを検討し、ウィザードを用いることにしました。ウィザードには正義の魔法使いという意味があり、敵側には相応しくないとも考えましたが、ここではダークサイドに落ちた魔法使い（たとえば「指輪物語」のサルマン）という裏設定で進めることとします。

```
class Wizard extends Actor {                          // ウィザードクラス
    static final int INIT_HP = 100;                   // 初期 HP

    Wizard(Game game) { super(game, INIT_HP); }       // コンストラクタ

    @Override
    public String toString() { return "Wizard : "+position+" "+hp; }

    @Override
    void takeDamageFrom(Actor a) { a.giveDamageTo(this); }
                                                      // ダブルディスパッチ法を参照

    @Override
    void action() {
        if (hp <= 0) return;

        mapAttack((x,y)->((x == 0) || ( y == 0)));       // 水平・垂直攻撃
    }

    static final Image image
                   = Toolkit.getDefaultToolkit().getImage("Wizard.png");

    @Override
    Image getImage() { return image; }
}
```

図 31.1: ウィザードを追加したダンジョンゲームのプログラム（ウィザード）

このゲームではウィザードは移動しない（moveTo() がない）設定です。初期配置位置にそのまま留まり、毎回、マップ攻撃を仕掛けてきます。対象は自分以外の全てのアクターです。ラムダ式 (x,y)->((x == 0) || (y == 0)) がこのウィザードのマップ攻撃のパターンです。

ウィザードの画像は以下を用います[2]。

[2]やはり「かわいいフリー素材集：いらすとや」、https://www.irasutoya.com からダウンロードしました。

```
class Game extends JFrame implements KeyListener {
    ** 以下のコンストラクタを除いて、前章から変更なし **
    Game() {
        ** 前略 **

        for (int i = 0; i < NUM_OF_MONSTERS; i++) {
            if (random.nextInt(2) == 0) {
                actors.add(new Wizard(this));  // Monster の代わりに Wizard
            }
            else {
                actors.add(new RedMonster(this));
            }
        }

        ** 後略 **
    }
}
```

図 31.2: ウィザードを追加したダンジョンゲームのプログラム（ゲーム）

```
abstract class Actor extends JLabel {
    ** 以下の 1 行を追加する **

    void giveDamageTo(Wizard w) {} // ウィザードへの直接物理攻撃のメソッド
}
```

図 31.3: ウィザードを追加したダンジョンゲームのプログラム（アクター）

31.2　ゲームクラスの変更

この章ではゲームクラスのプログラムを一箇所だけ変更します。図 31.2（617 ページ）のコメントがある行です。Monster オブジェクトの代わりに Wizard オブジェクトを登場させることとしました。

31.3　アクタークラスの変更

アクタークラスもダブルディスパッチ法に関係する一箇所だけを変更します。図 31.3（617 ページ）の 1 行です。

31.4　プレイヤークラスの変更

プレイヤークラスも、ダブルディスパッチ法によるプレイヤーからウィザードへの直接物理攻撃に関する 1 行（図 31.4（618 ページ）参照）を追加するだけです。なお、ウィザードは物理攻撃に

```
class Player extends Actor {
    ** 以下の 2 行を追加する **

    @Override
    void giveDamageTo(Wizard w) { w.hp -= 1000; } // ウィザードは瞬殺される
}
```

図 31.4: ウィザードを追加したダンジョンゲームのプログラム（プレイヤー）

非常に弱いと考え、プレイヤーからの攻撃で HP が 1000 減ります（つまり、一撃で戦闘終了です）。

このように見てくると、ウィザードの追加によって、ダブルディスパッチ法のプログラム部分に系統的な追加記述が必要になることが分かります。特筆すべきことは、ダブルディスパッチ法の各プログラム部分は独立性が高いため、それらの追加が単純なメソッドの追加で済み、既存のメソッド内の煩雑な変更（たとえば if 文の中の条件式を変更する、if 文を if-else 文へ変更するなど）は必要ないことです。よって（仕組みを分かってしまえば）プログラムの変更は容易です。

31.5　プログラムの実行例

図 31.5（619 ページ）は上の改造後のプログラムの実行の例です。ウィザードの攻撃は正しく機能しているようです。

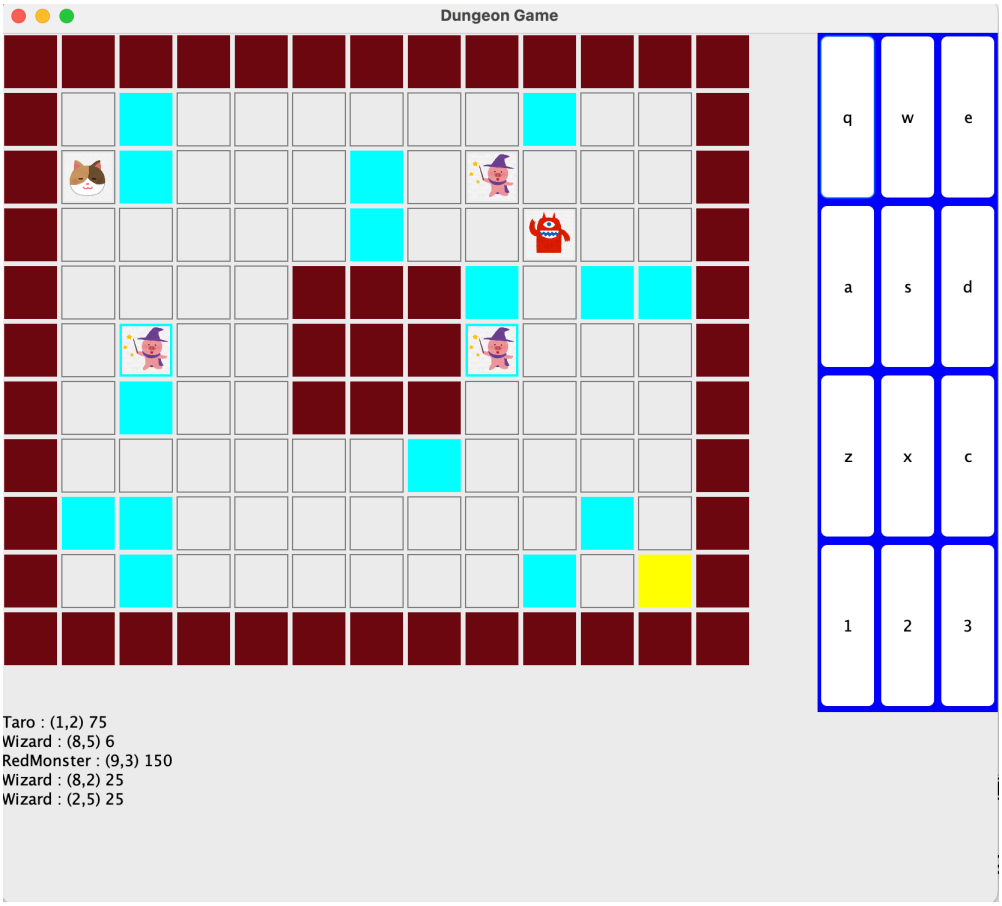


図 31.5: この章のプログラムの実行の例：3 体のウィザードどうしが水平・垂直の位置関係にあるため、互いに攻撃し合った結果、ウィザードの HP は激減しています。プレイヤーも (8,2) の位置のウィザードから水平遠距離攻撃されて HP が半減しています。

31.6　付録：この章の全プログラムリスト

```
//ゲームクラスのプログラム
import java.io.*;
import java.util.Random;
import java.util.ArrayList;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class Game extends JFrame implements KeyListener {
    public static void main(String args[]) {
        Game game = new Game();
    }

    static Random random = new Random();

    Map map;
    ArrayList<Actor> actors;
    final static int PLAYER_ID = 0;
    final static int NUM_OF_MONSTERS = 4;

    Game() {
        super("Dungeon Game");

        map = new Map(this);
        actors = new ArrayList<Actor>();
        actors.add(new Player(this, "Taro"));
        for (int i = 0; i < NUM_OF_MONSTERS; i++) {
            if (random.nextInt(2) == 0) {
                actors.add(new Wizard(this));  // Monster の代わりに Wizard
            }
            else {
                actors.add(new RedMonster(this));
            }
        }

        setBounds(100, 100, Cell.SIZE * Map.XSIZE + 5 + 200, Cell.SIZE * Map.YSIZE + 25 + 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        add(map, BorderLayout.WEST);

        add(new MyButtons(this), BorderLayout.EAST);

        JPanel actorsPanel = new JPanel();
        actorsPanel.setLayout(new GridLayout(10, 1));
        for (Actor a : actors) { actorsPanel.add(a); }
        add(actorsPanel, BorderLayout.SOUTH);

        setVisible(true);
    }

    void print() { repaint(); }

    void turn() {
        for (Actor a : actors) { a.action(); }
        print();
    }

```

```
    static char keyChar;
    static char getKeyChar() { return keyChar; }
    static void setKeyChar(char c) { keyChar = c; }

    @Override
    public void keyPressed(KeyEvent e) {
        setKeyChar(e.getKeyChar());
        turn();
    }

    @Override
    public void keyReleased(KeyEvent e){}

    @Override
    public void keyTyped(KeyEvent e){}
}
```

```
//ボタンクラスのプログラム
class MyButtons extends JPanel {
    MyButtons(Game game) {
        setPreferredSize(new Dimension(150, 150));
        setBackground(Color.BLUE);
        setLayout(new GridLayout(4,3));
        add(new MyButton(game,'q'));
        add(new MyButton(game,'w'));
        add(new MyButton(game,'e'));
        add(new MyButton(game,'a'));
        add(new MyButton(game,'s'));
        add(new MyButton(game,'d'));
        add(new MyButton(game,'z'));
        add(new MyButton(game,'x'));
        add(new MyButton(game,'c'));
        add(new MyButton(game,'1'));
        add(new MyButton(game,'2'));
        add(new MyButton(game,'3'));
    }
}

class MyButton extends JButton implements ActionListener {
    Game game;
    char c;
    MyButton(Game game, char c) {
        super(String.valueOf(c));
        this.game = game;
        this.c = c;
        addActionListener(this);
        addKeyListener(game);
    }

    @Override
    public void actionPerformed(ActionEvent e){
        game.setKeyChar(c);
        game.turn();
    }
}
```

```
//マップクラスのプログラム
class Map extends JPanel {
    static final int XSIZE = 13;
    static final int YSIZE = 11;

    Game game;
    Cell data[][] = new Cell[XSIZE][YSIZE];

    Map(Game game) {
        this.game = game;
        for (int x = 0; x < XSIZE; x++) {
            for (int y = 0; y < YSIZE; y++) {
                if ((x == 0)||(x == XSIZE-1)||
                    (y == 0)||(y == YSIZE-1)||
                    ((Math.abs(x-XSIZE/2) <= 1)&&
                     (Math.abs(y-YSIZE/2) <= 1))) {
                    data[x][y] = new RockCell();
                }
                else {
                    switch (Game.random.nextInt(5)) {
                    case 0: data[x][y] = new PotionCell();
                             break;
                    case 1: data[x][y] = new PoisonCell();
                             break;
                    default: data[x][y] = new EmptyCell();
                    }
                }
            }
        }
        data[XSIZE-2][YSIZE-2] = new GoalCell();

        setPreferredSize(new Dimension(Cell.SIZE*XSIZE, Cell.SIZE*YSIZE));
    }

    Cell getAt(Int2 position) { return data[position.x][position.y]; }

    void setAt(Int2 position, Cell cell) {
        data[position.x][position.y] = cell;
    }

    @Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        for (int y = 0; y < YSIZE; y++) {
            for (int x = 0; x < XSIZE; x++) {
                Int2 position = new Int2(x, y);
                getAt(position).paintAt(g, position);
                for (Actor a : game.actors) {
                    a.paintAt(g, position);
                }
            }
        }
    }

    Int2 getRandomCellPosition() {
        while (true) {
            Int2 position = new Int2(Game.random.nextInt(XSIZE),
                                     Game.random.nextInt(YSIZE));
```

```

            if (!getAt(position).isRockCell()) { return position; }
        }
    }
}
```

```
//アクタークラスのプログラム
abstract class Actor extends JLabel {
    Game game;
    Int2 position;
    int hp;

    Actor(Game game, Int2 position, int hp) {
        super(" ");
        this.game = game;
        this.position = position;
        this.hp = hp;
    }

    Actor(Game game, int hp) {
        super(" ");
        this.game = game;
        this.hp = hp;
        while_loop: while (true) {
            position = game.map.getRandomCellPosition();
            for (Actor a : game.actors) {
                if (position.equal(a.position)) {
                    continue while_loop;
                }
            }
            return;
        }
    }

    @Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        setText(toString());
    }

    void paintAt(Graphics g, Int2 position) {
        if (this.position.equal(position)) {
            g.drawImage(getImage(),
                        position.x*Cell.SIZE + Cell.SIZE*2/24,
                        position.y*Cell.SIZE + Cell.SIZE*2/24,
                        Cell.SIZE*20/24,
                        Cell.SIZE*20/24,
                        game.map);
        }
    }

    abstract Image getImage();

    boolean moveTo(Int2 direction) {
        Int2 target = position.add(direction);

        Cell cell = game.map.getAt(target);
        if (cell.isRockCell()) { return false; }

        for (Actor a : game.actors) {
            if (target.equal(a.position)) { return false; }
        }

        Cell newCell = cell.affectOn(this);
```

```
        game.map.setAt(target, newCell);
        position = target;
        return true;
    }

    abstract void action();

    void attackTo(Int2 direction) {
        Int2 target = position.add(direction);
        for (Actor a : game.actors) {
            if (target.equal(a.position)) {
                a.takeDamageFrom(this);
                return;
            }
        }
    }

    abstract void takeDamageFrom(Actor a);
    void giveDamageTo(Player p) {}
    void giveDamageTo(Monster m) {}
    void giveDamageTo(RedMonster r) {}
    void giveDamageTo(Wizard w) {} // ウィザードへの直接物理攻撃のメソッド

    void mapAttack(MapFunction mf) {
        for (Actor a : game.actors) {
            if (a != this) {
                Int2 relpos = a.position.sub(this.position);
                if (mf.test(relpos.x, relpos.y)) {
                    a.hp /= 2;
                }
            }
        }
    }
}

interface MapFunction {
    boolean test(int x, int y);
}
```

```
//プレイヤークラスのプログラム
class Player extends Actor {
    static final int INIT_HP = 100;
    String name;

    Player(Game game, String name) {
        super(game, new Int2(1, 1), INIT_HP);
        this.name = name;
    }

    Player(Game game) { this(game, "Player"); }

    public String toString() { return name+" : "+position+" "+hp; }

    @Override
    void takeDamageFrom(Actor a) { a.giveDamageTo(this); }

    @Override
    void giveDamageTo(Monster m) { m.hp -= 20; }

    @Override
    void giveDamageTo(RedMonster r) { r.hp -= 15; }

    @Override
    void giveDamageTo(Wizard w) { w.hp -= 1000; } // ウィザードは瞬殺される

    @Override
    void action() {
        Int2 direction;
        switch (Game.getKeyChar()) {
            case 'z': direction = new Int2(-1, +1); break;
            case 'x': direction = new Int2( 0, +1); break;
            case 'c': direction = new Int2(+1, +1); break;
            case 'a': direction = new Int2(-1,  0); break;
            case 'd': direction = new Int2(+1,  0); break;
            case 'q': direction = new Int2(-1, -1); break;
            case 'w': direction = new Int2( 0, -1); break;
            case 'e': direction = new Int2(+1, -1); break;
            case '1':
                mapAttack((dx, dy)->((dx == 0) || ( dy == 0)));
                return;
            case '2':
                mapAttack((dx, dy)->((dx == dy) || ( dx == -dy)));
                return;
            case '3':
                mapAttack((dx,dy)->true);
                hp /= 10;
                return;
            default: direction = new Int2(0, 0);
        }

        if (!moveTo(direction)) { attackTo(direction); }
    }

    static final Image image1 = Toolkit.getDefaultToolkit().getImage("Player1.png");
    static final Image image2 = Toolkit.getDefaultToolkit().getImage("Player2.png");

    @Override
```

```
    Image getImage() {
        if (hp > INIT_HP/2) { return image1; }
        else { return image2; }
    }
}
```

```
//モンスタークラスのプログラム
class Monster extends Actor {
    static final int INIT_HP = 150;

    Monster(Game game) { super(game, INIT_HP); }

    @Override
    public String toString() { return "Monster : "+position+" "+hp; }

    @Override
    void takeDamageFrom(Actor a) { a.giveDamageTo(this); }

    @Override
    void giveDamageTo(Player p) { p.hp -= 30; }

    @Override
    void giveDamageTo(RedMonster r) { r.hp -= 5; }

    @Override
    void action() {
        if (hp <= 0) return;

        Actor player = game.actors.get(game.PLAYER_ID);
        Int2 direction = player.position.sub(position).signum();

        if (!moveTo(direction)) {
            attackTo(direction);
        }
    }

    static final Image image
                   = Toolkit.getDefaultToolkit().getImage("Monster.png");

    @Override
    Image getImage() { return image; }
}
```

```
//レッドモンスタークラスのプログラム
class RedMonster extends Monster {
    RedMonster(Game game) { super(game); }

    @Override
    public String toString() { return "Red" + super.toString(); }

    @Override
    void takeDamageFrom(Actor a) { a.giveDamageTo(this); }

    @Override
    void giveDamageTo(Player p) { p.hp -= 60; }

    @Override
    void giveDamageTo(Monster m) { m.hp -= 7; }

    static final Image image
                   = Toolkit.getDefaultToolkit().getImage("RedMonster.png");

    @Override
    Image getImage() { return image; }
}
```

```
//ウィザードクラスのプログラム
class Wizard extends Actor {                          // ウィザードクラス
    static final int INIT_HP = 100;                   // 初期 HP

    Wizard(Game game) { super(game, INIT_HP); }       // コンストラクタ

    @Override
    public String toString() { return "Wizard : "+position+" "+hp; }

    @Override
    void takeDamageFrom(Actor a) { a.giveDamageTo(this); }
                                                      // ダブルディスパッチ法を参照

    @Override
    void action() {
        if (hp <= 0) return;

        mapAttack((x,y)->((x == 0) || ( y == 0)));       // 水平・垂直攻撃
    }

    static final Image image
                   = Toolkit.getDefaultToolkit().getImage("Wizard.png");

    @Override
    Image getImage() { return image; }
}
```

```
//2 次元ベクトルクラスのプログラム
class Int2 {
    int x,y;

    Int2(int x, int y) { this.x = x; this.y = y; }
    Int2 add(Int2 a) { return new Int2(x+a.x,y+a.y); }
    Int2 sub(Int2 a) { return new Int2(x-a.x,y-a.y); }
    Int2 signum() { return new Int2(Integer.signum(x),Integer.signum(y)); }
    boolean equal(Int2 a) { return (x == a.x)&&(y == a.y); }

    public String toString(){ return "("+x+","+y+")"; }
}
```

```
//セルクラスのプログラム
abstract class Cell {
    static final int SIZE = 48;

    boolean isRockCell() { return false; }

    abstract Cell affectOn(Actor a);

    abstract void paintAt(Graphics g, Int2 position);

    void drawCell(Graphics g, Int2 position) {
        g.drawRect(position.x*SIZE + SIZE/24,
                   position.y*SIZE + SIZE/24,
                   SIZE*22/24,
                   SIZE*22/24);
    }

    void fillCell(Graphics g, Int2 position) {
        g.fillRect(position.x*SIZE + SIZE/24,
                   position.y*SIZE + SIZE/24,
                   SIZE*22/24,
                   SIZE*22/24);
    }
}

abstract class SimpleCell extends Cell {
    @Override
    Cell affectOn(Actor a) { return this; }
}

abstract class TrapCell extends Cell {
    boolean visible = false;

    @Override
    Cell affectOn(Actor a) {
        visible = true;
        return this;
    }
}

class RockCell extends SimpleCell {
    @Override
    void paintAt(Graphics g, Int2 position) {
        g.setColor(new Color(100, 20, 20));
        fillCell(g, position);
    }

    @Override
    boolean isRockCell() { return true; }
}

class EmptyCell extends SimpleCell {
    @Override
    void paintAt(Graphics g, Int2 position) {
        g.setColor(new Color(128, 128, 128));
        drawCell(g, position);
    }
}
```

```
class PotionCell extends SimpleCell {
    @Override
    void paintAt(Graphics g, Int2 position) {
        g.setColor(new Color(0, 255, 255));
        fillCell(g, position);
    }

    @Override
    Cell affectOn(Actor a) {
        a.hp += 50;
        return new EmptyCell();
    }
}

class GoalCell extends SimpleCell {
    @Override
    void paintAt(Graphics g, Int2 position) {
        g.setColor(new Color(255, 255, 0));
        fillCell(g, position);
    }

    @Override
    Cell affectOn(Actor a) {
        if (a instanceof Player) {
            System.out.println("YOU WIN");
            System.exit(0);
        }
        return this;
    }
}

class PoisonCell extends TrapCell {
    @Override
    void paintAt(Graphics g, Int2 position) {
        if (visible) {
            g.setColor(new Color(0, 128, 0));
            fillCell(g, position);
        } else {
            g.setColor(new Color(128, 128, 128));
            drawCell(g, position);
        }
    }

    @Override
    Cell affectOn(Actor a) {
        a.hp -= 10;
        return super.affectOn(a);
    }
}
```

1

1

第32章　死者は生き返らない：アクセス制御の考え方

このテキストではオブジェクト指向技術がプログラミングの生産性を上げるという観点から様々な新しいプログラミング技法を紹介してきました。それらが言わば「攻めのプログラミング」であったのに対し、この章では「守りのプログラミング」 — プログラムをより堅牢にする技術 — を紹介します。堅牢であることはバグの混入を減らし、最終的にはやはり生産性に結びつきます。

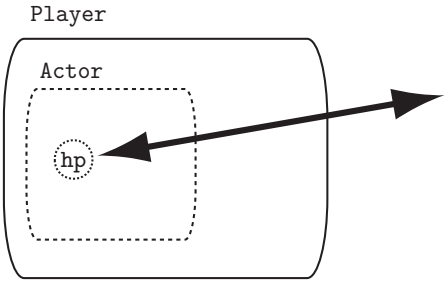
もし全てのプログラマが善良であり、如何なる事態においても、たとえばクライアント（発注元）へのプログラムの納品が明朝に迫っているにも関わらず深夜のデバッグが進んでいない事態においても、ルール違反のデータアクセスを行わない賢明さを持つならば、この章は不要です。しかし現実のプログラマは苦し紛れのルール違反をやってしまいがちですし、そもそもルール違反に気づいてさえいないかもしれません。それを未然に防ぐのが**アクセス制御**（access control） — 不用意にデータにアクセスさせない技術です。オブジェクト指向ではオブジェクトの内部を不用意に外部に晒（さら）さないことがそれです。これはオブジェクトの**カプセル化**（encapsulation、意訳して情報隠蔽）とも呼ばれます。

この章ではカプセル化の基本的な考え方をアクタークラスのインスタンス変数 `hp`、`position` へのアクセス制御を主な事例として解説します。

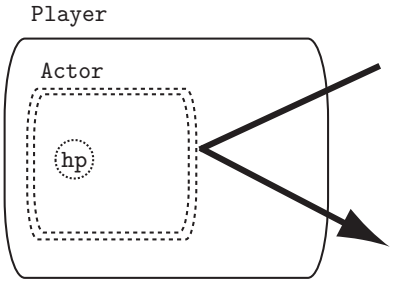
32.1　HP のアクセス制御

前章までのアクターの HP（Health Point）の扱いはかなりぞんざいでした。HP が負値になったり、逆にポーションで HP が初期値を超えて大きな値になることも黙認してきました。最も奇妙なのは、一旦、負値になった HP がその後に正値になる場合もあることです。これは一度、死んだアクターが生き返るようなものです。この節では HP へのアクセスに検閲を設け、このような野放図な状況を制御します。

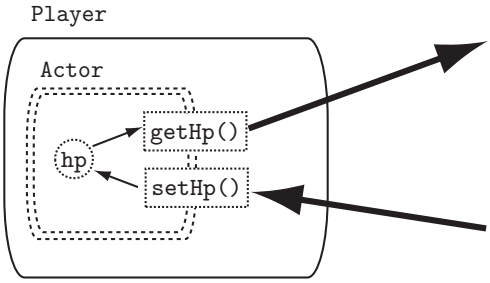
図 32.1（638 ページ）にカプセル化の 4 状態を図示しました。以下、順に解説します。



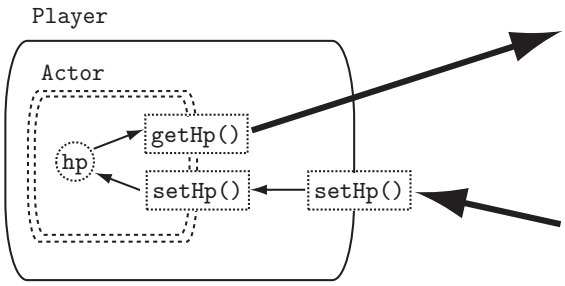
(a) 前章までの方法：外部から自由にアクセスできますが、非常に危険な状態です。



(b) この章の方法（その 1）：hp を private 修飾すると、hp を宣言しているアクタークラス以外からは hp に全くアクセスできなくなります。



(c) この章の方法（その 2）：hp へのアクセスはゲッター（getHp()）/セッター（setHp()）を用いて可能です。



(d) この章の方法（その 3）：プレイヤークラス、アクタークラスのセッター内で検閲を行い、hp への不当な値の代入を防ぎます。

図 32.1: プレイヤーの hp に外部からアクセスする方法：

32.1.1　無防備な状態

図 32.1(a) が前章までのインスタンス変数 hp へのアクセスの様子を図示したものです。プレイヤーオブジェクトの中のアクターオブジェクト部分の hp に外部からダイレクトにアクセスできます。たとえばモンスターがプレイヤーを攻撃した場合、 プレイヤーの hp を減らす必要がありますから、このアクセスはゲームを成り立たせる上で不可避です。しかし、このままでは hp がどこからでも自由に改変できることとなり、危険です。

32.1.2　private 修飾子

変数、メソッド（関数）をクラス外に公開しないための修飾子として、C++、Java では private 修飾子を利用できます。これによって外部からの野放図なアクセスを完全に防止することができます。

インスタンス変数 hp の場合、前章までのアクタークラス内での宣言：

```
int hp;
```

を以下のように変更するだけです。

```
private int hp;                          // private 化
```

これによってアクタークラス以外からはアクセスできなくなります。図 32.1 (b) はそれを図示したものです。プレイヤークラスからもアクセスできません。プレイヤークラスから見れば hp は自分の体の一部のようなものですが、private 修飾は自身の体の中もブラックボックス化してしまいます。

32.1.3　アクセス制御の種類

実は Java のアクセス制御には四種類があります。それを表 32.1 にまとめました。

表中の説明に「パッケージ」という言葉で使われてますが、これについてはテキストではほとんど触れてきませんでした。ダンジョンゲームのプログラムにおいて

表 32.1: アクセス修飾子の違い

修飾子名	制限の内容
public	すべてのクラスからアクセスできます。
protected	修飾子を宣言したクラス、宣言したクラスのサブクラス、宣言したクラスと同一のパッケージ内のクラスからアクセスできます。
修飾無し	修飾子を宣言したクラス、宣言したクラスと同一のパッケージ内のクラスからアクセスできます。
private	修飾子を宣言したクラスのみからアクセスできます。

```
import java.io.*;
import java.util.Random;
import java.util.ArrayList;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
```

と記述したものが、Java システムで提供されるパッケージの取り込みを意味しています。

パッケージはプログラマ自身が作成することも可能です。規模の大きなプログラムを開発する際に、自身が開発するクラスを役割に応じて分類・パッケージ化し、全体の見通しをよくすることができます。その際にはプログラム中に package 文を用いますが、その説明はこのテキストでは省略します。

表 32.1 の「パッケージ」は、そのようなシステム提供のパッケージおよびプログラマ自身が作成するパッケージの両方のことです。そして、表中の、たとえば「同一のパッケージ内のクラスからアクセスできます」とは、逆に言えば、「異なるパッケージからはアクセスできない」ことを意味しています。

このテキストのダンジョンゲームで開発するクラスは、パッケージ化を指定されておらず、言い換えれば、全てのクラスは「デフォルト」のパッケージに含まれる設定になっています。よって全てのクラスは同一のパッケージに含まれ、上に挙げた「異なるパッケージからはアクセスできない」ような状況はこのテキストのプログラムには該当しません。つまり、このテキストのダンジョンゲームのクラス間では表 32.1 の public、protected、修飾なしの指定の違いは生じません。

修飾子の使い方について詳細に触れる余裕はありませんから、余裕のある人は自学してください。

```
abstract class Actor extends JLabel {
    private int hp;                          // private 化

    int getHp() { return hp; }               // ゲッター

    void setHp(int hp) { this.hp = hp; }     // セッター

    ** 後略 **
}
```

図 32.2: ゲッター・セッターを用いるダンジョンゲームのプログラム（アクター、その 1）

32.1.4　ゲッター・セッター

さて、private 修飾子を指定すると、インスタンス変数 hp の取り扱いは安全になりますが、クラス外から全くアクセスできませんから、プログラムが正常動作しません。コンパイルエラーとなります。

そこで安全に外部からアクセスさせるために、変数値を取り出すための専用メソッドと変数に値を代入するための専用メソッドを用意するのが一般的です。それぞれのメソッドを特に**ゲッター**（getter）、**セッター**（setter）と呼んでいます。

図 32.2（641 ページ）は、Hp のゲッター、セッターを実装したプログラムです。図 32.1 (c) は、ゲッター、セッターを介してアクセスする様子の図示です。

ゲッターのメソッド名、引数、戻り値の型に関する厳格な規則はありません。しかし、インスタンス変数値を取り出すのですから、戻り値の型はインスタンス変数の型と同じであるべきです。Java ではゲッターのメソッド名は “get ＋ インスタンス変数名” とするのが一般的なようです。ゲッターに引数はありません。hp の場合、図 32.2 のように、

```
    int getHp() { return hp; }
```

と定義されます。戻り値はインスタンス変数値です。

セッターのメソッド名、引数、戻り値の型についても厳格な規則はありません。まず、セッターは値を代入することが目的ですから、戻り値はありません。セッターのメソッド名は “set ＋ インスタンス変数名” とするのが一般的です。引数はインスタンス変数と同じデータ型の引数 1 個です。よって hp の場合、図 32.2 のように、

```
    void setHp(int hp) { this.hp = hp; }
```

と定義されます。

前章のプログラム中で hp の値を参照する箇所は全て getHp() に置き換える必要があります。hp の値を整数値 v で更新する箇所は全て setHp(v) に置き換える必要があります。たとえば、前の章のモンスタークラスの

```
void giveDamageTo(Player p) { p.hp -= 30; }
```

は、以下のように変更します。

```
void giveDamageTo(Player p) { p.setHp(p.getHp()-30); }
```

32.1.5　セッターによる検閲

ゲッター、セッターによって hp への野放図なアクセスが制限されたように思えますが、必ずしもそうではありません。セッター setHp(v) による値の更新では v をそのまま hp へ代入しており、結局、インスタンス変数 hp にダイレクトにアクセスする場合と何ら変わりはありません。

セッターが真価を発揮するのは、セッター内で引数値を検閲する場合です。それをプログラム化しましょう。このテキストでは、以下の点でインスタンス変数 hp の値をぞんざいに扱ってきました。

1. これまでは攻撃によって hp の値が負値になることがありますが、この章では hp の最小値を 0 とします。もし攻撃によって計算上の hp が負値になる場合には更新値を 0 に強制します。
2. これまでは一旦、負値になった hp がポーションによって正値に戻ることがありましたが、これは死者がポーションで生き返ることに相当し、不自然です。この章では hp が 0 の場合には如何なる理由でも hp を変更しないことにします[1]。
3. これまでは hp の上限を決めておらず、ポーションによって hp が初期 HP を超える場合も多々ありました。これも不自然と考え、計算上の hp が初期 HP を超える場合には更新値を初期 HP に強制します。

上記 1.、2. を実装したプログラムが図 32.3（643 ページ）です。上記 3. を実装したプログラムが図 32.4（643 ページ）です。ここに図 32.4 の setHp() はアクタークラスの setHp() を上書き

[1]死者を生き返らせる術を使うときは例外ですが、そのような術はこのテキストでは扱いません。

```
abstract class Actor extends JLabel {
    private int hp;                                          // private 化

    int getHp() { return hp; }                          // ゲッター

    void setHp(int hp) {                                // セッター
        if (this.hp == 0) return;  // アクターの hp が 0 ならば、値を更新しない

        this.hp = Math.max(hp,0);    // hp と 0 の大きい方で更新
    }

    ** 後略 **
}
```

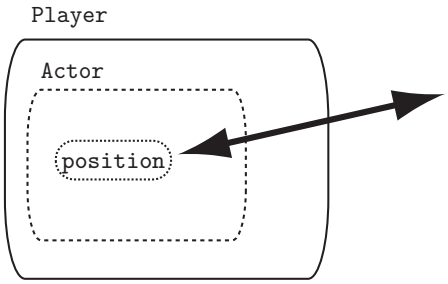
図 32.3: ゲッター・セッターを用いるダンジョンゲームのプログラム（アクター、その 2）：検閲ありの場合

```
class Player extends Actor {
    static final int INIT_HP = 100;

    @Override
    void setHp(int hp) { super.setHp(Math.min(hp,INIT_HP)); } // セッター
                    // hp と INIT_HP の小さい方をアクタークラスの setHP() へ渡す

    ** 後略 **
}
```

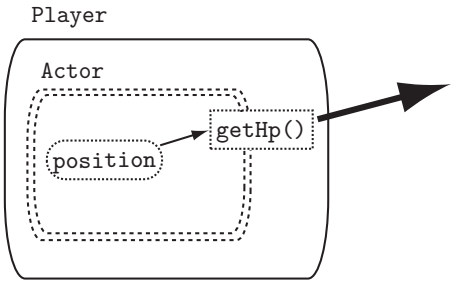
図 32.4: ゲッター・セッターを用いるダンジョンゲームのプログラム（プレイヤー）：検閲ありの場合

していることに注意してください。モンスタークラスについても図 32.4 の setHp() と同じプログラムを追加すべきです。

このプログラムの改造を図示したものが図 32.1 (d) です。



(a) 前章までの方法：外部から自由にアクセスできますが、非常に危険な状態です。



(b) この章の方法：position へのアクセスはゲッターを用いて可能です。このダンジョンゲームのプログラムではセッターは不要です。

図 32.5: プレイヤーの position に外部からアクセスする方法：

32.2　position のアクセス制御

図 32.5(a) が前章までのインスタンス変数 position へのアクセスの様子を図示したものです。外部からダイレクトにアクセスできる状態でした。

position にも private 修飾子を指定すると安全性が高まります。しかも position の値の更新はアクタークラスの moveTo() 内でのみ行われ、外部からの更新が無いため、セッターは不要です。ゲッターのみを用意します。図 32.6（645 ページ）はそのプログラムです。

ゲッター・セッターはインスタンス変数毎に必ず必要という訳ではありません。もしセッターが不要ならば、その変数への意図しない値の変更は完全に遮断できますから、プログラムはより堅牢になります。

インスタンス変数 game も同じく、ゲッターのみ必要です。

ここまでアクタークラスのインスタンス変数について述べてきましたが、他のクラスについても同様のアクセス制御が可能です。以下、各クラスについて概説します。

```
abstract class Actor extends JLabel {
    private Game game;                          // private 化
    game getGame() { return game; }             // ゲッターのみを定義

    private Int2 position;                      // private 化
    Int2 getPosition() { return position; }     // ゲッターのみを定義

    ** 後略 **
}
```

図 32.6: ゲッター・セッターを用いるダンジョンゲームのプログラム（アクター、その 3）：position、game の private 化

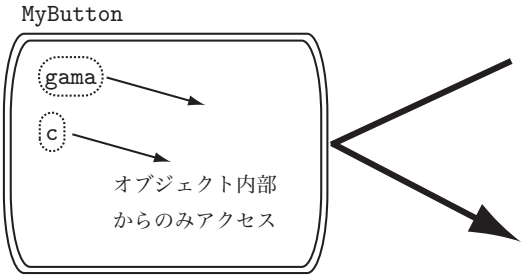


図 32.7: MyButton のインスタンス変数の取り扱い：外部からのアクセスがないため、ゲッター、セッターは不要です。

32.3　その他のクラスの変更

アクタークラス以外のクラスのインスタンス変数も同様に private 修飾できます。以下、概要のみ述べます。具体的なプログラムは付録を参照してください。

32.3.1　ゲームクラスの変更

インスタンス変数 map、actors を private 修飾します。これらにはゲッターが必要です。

クラス変数 keyChar も private 修飾します。これについては既に 27 章（497 ページ）でゲッター・セッターを作成済みです。

32.3.2　ボタンクラスの変更

MyButton クラスのインスタンス変数 game、c を private 修飾します。これらはクラス内部でのみの利用のため、ゲッター・セッターとも不要です。図 32.7（645 ページ）はそれを図示したものです。外部アクセスを完全に遮断できるため、安全です。

32.3.3　マップクラスの変更

マップクラスのインスタンス変数 data を private 修飾します。これについては、既に 14.3 節（202 ページ）においてゲッター getAt() を導入していました。ただし、このゲッターではマップ上の位置を引数で指定する方式でした。同じくセッターは 22.6 節（381 ページ）において setAt() を導入していました。これも位置を指定してマップを更新するものでした。

インスタンス変数 game は、クラス内部でのみの利用のため、ゲッター・セッターとも不要です。

32.3.4　プレイヤークラスの変更

インスタンス変数 name を private 修飾します。クラス内部でのみの利用のため、ゲッター・セッターとも不要です。

32.3.5　モンスタークラスの変更

プレイヤークラスと同様に、setHp() の上書きが必要です。

32.3.6　ウィザードクラスの変更

プレイヤークラスと同様に、setHp() の上書きが必要です。

32.3.7　Int2 クラスの変更

このクラスは汎用データ型を提供するものであって、他のクラスとは性格が異なるため、このテキストではインスタンス変数 x、y を private 修飾しないこととします。

32.3.8　TrapCell クラスの変更

インスタンス変数 visible を private 修飾します。ゲッターが必要です。

32.4　補足：C++と Java のアクセス修飾子の効果の違い

C++でも Java と同じアクセス修飾子 public、protected、private を利用できます。しかし、プログラム中の利用方法が異なります。たとえば以下の Java のプログラム：

```
class ClassA {
    private int a;
    private float b;
    private double c;

    public int getA() { return a; }
    public float getB() { return b; }
    public double getC() { return c; }
}
```

は、C++で記述すると以下の通りです。

```
class ClassA {
  private:
    int a;
    float b;
    double c;
  public:
    int getA() { return a; }
    float getB() { return b; }
    double getC() { return c; }
}
```

Java では個々のインスタンス変数、メソッド毎にアクセス修飾子を付ける必要がありますが、C++では修飾子が記載された行以降がその修飾子の効果が作用する部分となります。

また、Java の「パッケージ」に相当するクラスの取りまとめ機能が C++にはありません。結果、`public`、`protected`、`private` の意味が微妙に異なる場合がありますから、C++でプログラミングする場合には注意してください。このテキストでは省略しますが、C++、Java とも `protected` については特に正確な理解が必要です。

この章の内容は、クラスのセキュリティに関するものです。となると、ある意味で興味深いのはセキュリティの抜け穴です。「java アクセス 抜け穴」でネット検索してみてください。アクセス制御の問題点を詳細に語っているサイトに出会うはずです。面白い発見があるはずです。またこの話題の奥深さを知ることができると思います。

このテキストではこれ以上、解説する余裕はありません。是非、みなさん自身で調べてみてください。

32.5　付録：この章の全プログラムリスト

以下のプログラムリストについて、プログラム中に改造箇所のコメントを付けました。

```
//ゲームクラスのプログラム
import java.io.*;
import java.util.Random;
import java.util.ArrayList;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class Game extends JFrame implements KeyListener {
    public static void main(String args[]) {
        Game game = new Game();
    }

    static Random random = new Random();

    private Map map;                                          // private 化
    Map getMap() { return map; }                              // ゲッター

    private ArrayList<Actor> actors;                          // private 化
    ArrayList<Actor> getActors() { return actors; }           // ゲッター

    final static int PLAYER_ID = 0;
    final static int NUM_OF_MONSTERS = 4;

    Game() {
        super("Dungeon Game");

        map = new Map(this);
        actors = new ArrayList<Actor>();
        actors.add(new Player(this, "Taro"));
        for (int i = 0; i < NUM_OF_MONSTERS; i++) {
            if (random.nextInt(2) == 0) {
                actors.add(new Wizard(this));
            }
            else {
                actors.add(new RedMonster(this));
            }
        }

        setBounds(100, 100, Cell.SIZE * Map.XSIZE + 5 + 200,
                            Cell.SIZE * Map.YSIZE + 25 + 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        add(map, BorderLayout.WEST);

        add(new MyButtons(this), BorderLayout.EAST);

        JPanel actorsPanel = new JPanel();
        actorsPanel.setLayout(new GridLayout(10, 1));
        for (Actor a : actors) { actorsPanel.add(a); }
        add(actorsPanel, BorderLayout.SOUTH);

        setVisible(true);
    }

```

```
    void print() { repaint(); }

    void turn() {
        for (Actor a : actors) { a.action(); }
        print();
    }

    static private char keyChar;                                // private 化
    static char getKeyChar() { return keyChar; }                // ゲッター
    static void setKeyChar(char c) { keyChar = c; }             // セッター

    @Override
    public void keyPressed(KeyEvent e) {
        setKeyChar(e.getKeyChar());
        turn();
    }

    @Override
    public void keyReleased(KeyEvent e){}

    @Override
    public void keyTyped(KeyEvent e){}
}
```

```
//ボタンクラスのプログラム
class MyButtons extends JPanel {
    MyButtons(Game game) {
        setPreferredSize(new Dimension(150, 150));
        setBackground(Color.BLUE);
        setLayout(new GridLayout(4,3));
        add(new MyButton(game,'q'));
        add(new MyButton(game,'w'));
        add(new MyButton(game,'e'));
        add(new MyButton(game,'a'));
        add(new MyButton(game,'s'));
        add(new MyButton(game,'d'));
        add(new MyButton(game,'z'));
        add(new MyButton(game,'x'));
        add(new MyButton(game,'c'));
        add(new MyButton(game,'1'));
        add(new MyButton(game,'2'));
        add(new MyButton(game,'3'));
    }
}

class MyButton extends JButton implements ActionListener {
    private Game game;                                          // private 化
    private char c;                                             // private 化

    MyButton(Game game, char c) {
        super(String.valueOf(c));
        this.game = game;
        this.c = c;
        addActionListener(this);
        addKeyListener(game);
    }

    @Override
    public void actionPerformed(ActionEvent e){
        game.setKeyChar(c);
        game.turn();
    }
}
```

```
//マップクラスのプログラム
class Map extends JPanel {
    static final int XSIZE = 13;
    static final int YSIZE = 11;

    private Game game;                                          // private 化

    private Cell data[][] = new Cell[XSIZE][YSIZE];             // private 化

    Map(Game game) {
        this.game = game;
        for (int x = 0; x < XSIZE; x++) {
            for (int y = 0; y < YSIZE; y++) {
                if ((x == 0)||(x == XSIZE-1)||
                    (y == 0)||(y == YSIZE-1)||
                    ((Math.abs(x-XSIZE/2) <= 1)&&
                     (Math.abs(y-YSIZE/2) <= 1))) {
                    data[x][y] = new RockCell();
                }
                else {
                    switch (Game.random.nextInt(5)) {
                    case 0: data[x][y] = new PotionCell();
                            break;
                    case 1: data[x][y] = new PoisonCell();
                            break;
                    default: data[x][y] = new EmptyCell();
                    }
                }
            }
        }
        data[XSIZE-2][YSIZE-2] = new GoalCell();

        setPreferredSize(new Dimension(Cell.SIZE*XSIZE, Cell.SIZE*YSIZE));
    }

    Cell getAt(Int2 position) { return data[position.x][position.y]; }

    void setAt(Int2 position, Cell cell) {
        data[position.x][position.y] = cell;
    }

    @Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        for (int y = 0; y < YSIZE; y++) {
            for (int x = 0; x < XSIZE; x++) {
                Int2 position = new Int2(x, y);
                getAt(position).paintAt(g, position);
                for (Actor a : game.getActors()) {
                    a.paintAt(g, position);
                }
            }
        }
    }

    Int2 getRandomCellPosition() {
        while (true) {
            Int2 position = new Int2(Game.random.nextInt(XSIZE),
```

```
                                        Game.random.nextInt(YSIZE));

                if (!getAt(position).isRockCell()) { return position; }
            }
        }
    }
```

```
//アクタークラスのプログラム
abstract class Actor extends JLabel {
    private Game game;                                          // private 化
    private Int2 position;                                      // private 化
    private int hp;                                             // private 化

    game getGame() { return game; }                     // ゲッターのみを定義

    Int2 getPosition() { return position; }             // ゲッターのみを定義

    int getHp() { return hp; }                          // ゲッター

    void setHp(int hp) {                                // セッター
        if (this.hp == 0) return;    // アクターの hp が 0 ならば、値を更新しない

        this.hp = Math.max(hp,0);    // hp と 0 の大きい方で更新
    }

    Actor(Game game, Int2 position, int hp) {
        super(" ");
        this.game = game;
        this.position = position;
        this.hp = hp;
    }

    Actor(Game game, int hp) {
        super(" ");
        this.game = game;
        this.hp = hp;
        while_loop: while (true) {
            position = game.getMap().getRandomCellPosition();
            for (Actor a : game.getActors()) {
                if (position.equal(a.position)) {
                    continue while_loop;
                }
            }
            return;
        }
    }

    @Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        setText(toString());
    }

    void paintAt(Graphics g, Int2 position) {
        if (this.position.equal(position)) {
            g.drawImage(getImage(),
                        position.x*Cell.SIZE + Cell.SIZE*2/24,
                        position.y*Cell.SIZE + Cell.SIZE*2/24,
                        Cell.SIZE*20/24,
                        Cell.SIZE*20/24,
                        game.getMap());
        }
    }

    abstract Image getImage();
```

```
1
2       boolean moveTo(Int2 direction) {
3           Int2 target = position.add(direction);
4
5           Cell cell = game.getMap().getAt(target);
6           if (cell.isRockCell()) { return false; }
7
8           for (Actor a : game.getActors()) {
9               if (target.equal(a.position)) { return false; }
10          }
11
12          Cell newCell = cell.affectOn(this);
13          game.getMap().setAt(target, newCell);
14          position = target;
15          return true;
16      }
17
18      abstract void action();
19
20      void attackTo(Int2 direction) {
21          Int2 target = position.add(direction);
22          for (Actor a : game.getActors()) {
23              if (target.equal(a.position)) {
24                  a.takeDamageFrom(this);
25                  return;
26              }
27          }
28      }
29
30      abstract void takeDamageFrom(Actor a);
31      void giveDamageTo(Player p) {}
32      void giveDamageTo(Monster m) {}
33      void giveDamageTo(RedMonster r) {}
34      void giveDamageTo(Wizard w) {}
35
36      void mapAttack(MapFunction mf) {
37          for (Actor a : game.getActors()) {
38              if (a != this) {
39                  Int2 relpos = a.position.sub(this.position);
40                  if (mf.test(relpos.x, relpos.y)) {
41                      a.hp /= 2;
42                  }
43              }
44          }
45      }
46  }
47
48  interface MapFunction {
49      boolean test(int x, int y);
50  }
```

```
//プレイヤークラスのプログラム
class Player extends Actor {
    static final int INIT_HP = 100;

    void setHp(int hp) { super.setHp(Math.min(hp,INIT_HP)); }   // セッター
                    // hp と INIT_HP の小さい方をアクタークラスの setHP() へ渡す

    private String name;                                        // private 化

    Player(Game game, String name) {
        super(game, new Int2(1, 1), INIT_HP);
        this.name = name;
    }

    Player(Game game) { this(game, "Player"); }

    public String toString() { return name+" : "+getPosition()+" "+getHp(); }

    @Override
    void takeDamageFrom(Actor a) { a.giveDamageTo(this); }

    @Override
    void giveDamageTo(Monster m) { m.setHp(m.getHp()-20); }

    @Override
    void giveDamageTo(RedMonster r) { r.setHp(r.getHp()-15); }

    @Override
    void giveDamageTo(Wizard w) { w.setHp(w.getHp()-1000); }

    @Override
    void action() {
        Int2 direction;
        switch (Game.getKeyChar()) {
            case 'z': direction = new Int2(-1, +1); break;
            case 'x': direction = new Int2( 0, +1); break;
            case 'c': direction = new Int2(+1, +1); break;
            case 'a': direction = new Int2(-1,  0); break;
            case 'd': direction = new Int2(+1,  0); break;
            case 'q': direction = new Int2(-1, -1); break;
            case 'w': direction = new Int2( 0, -1); break;
            case 'e': direction = new Int2(+1, -1); break;
            case '1':
                mapAttack((dx, dy)->((dx == 0) || ( dy == 0)));
                return;
            case '2':
                mapAttack((dx, dy)->((dx == dy) || ( dx == -dy)));
                return;
            case '3':
                mapAttack((dx,dy)->true);
                setHp(getHp()/10);
                return;
            default: direction = new Int2(0, 0);
        }

        if (!moveTo(direction)) { attackTo(direction); }
    }

```

```
    static final Image image1
                    = Toolkit.getDefaultToolkit().getImage("Player1.png");
    static final Image image2
                    = Toolkit.getDefaultToolkit().getImage("Player2.png");

    @Override
    Image getImage() {
        if (getHp() > INIT_HP/2) { return image1; }
        else { return image2; }
    }
}
```

```
//モンスタークラスのプログラム
class Monster extends Actor {
    static final int INIT_HP = 150;

    void setHp(int hp) { super.setHp(Math.min(hp,INIT_HP)); }   // セッター

    Monster(Game game) { super(game, INIT_HP); }

    @Override
    public String toString() { return "Monster : "+getPosition()+" "+getHp(); }

    @Override
    void takeDamageFrom(Actor a) { a.giveDamageTo(this); }

    @Override
    void giveDamageTo(Player p) { p.setHp(p.getHp()-30); }

    @Override
    void giveDamageTo(RedMonster r) { r.setHp(r.getHp()-5); }

    @Override
    void action() {
        if (getHp() <= 0) return;

        Actor player = getGame().getActors().get(getGame().PLAYER_ID);
        Int2 direction = player.getPosition().sub(this.getPosition()).signum();

        if (!moveTo(direction)) {
            attackTo(direction);
        }
    }

    static final Image image
                   = Toolkit.getDefaultToolkit().getImage("Monster.png");

    @Override
    Image getImage() { return image; }
}
```

```
//レッドモンスタークラスのプログラム
class RedMonster extends Monster {
    RedMonster(Game game) { super(game); }

    @Override
    public String toString() { return "Red" + super.toString(); }

    @Override
    void takeDamageFrom(Actor a) { a.giveDamageTo(this); }

    @Override
    void giveDamageTo(Player p) { p.setHp(p.getHp()-60); }

    @Override
    void giveDamageTo(Monster m) { m.setHp(m.getHp()-7); }

    static final Image image
                   = Toolkit.getDefaultToolkit().getImage("RedMonster.png");

    @Override
    Image getImage() { return image; }
}
```

```
//ウィザードクラスのプログラム
class Wizard extends Actor {
    static final int INIT_HP = 100;

    void setHp(int hp) { super.setHp(Math.min(hp,INIT_HP)); }

    Wizard(Game game) { super(game, INIT_HP); }

    @Override
    public String toString() { return "Wizard : "+getPosition()+" "+getHp(); }

    @Override
    void takeDamageFrom(Actor a) { a.giveDamageTo(this); }

    @Override
    void action() {
        if (getHp() <= 0) return;

        mapAttack((x,y)->((x == 0) || ( y == 0)));
    }

    static final Image image
                   = Toolkit.getDefaultToolkit().getImage("Wizard.png");

    @Override
    Image getImage() { return image; }
}
```

```
//2 次元ベクトルクラスのプログラム
class Int2 {
    int x, y;

    Int2(int x, int y) { this.x = x; this.y = y; }
    Int2 add(Int2 a) { return new Int2(x + a.x, y + a.y); }
    Int2 sub(Int2 a) { return new Int2(x - a.x, y - a.y); }
    Int2 signum() { return new Int2(Integer.signum(x), Integer.signum(y)); }
    boolean equal(Int2 a) { return (x == a.x) && (y == a.y); }

    public String toString() { return "(" + x + "," + y + ")"; }
}
```

```
//セルクラスのプログラム
abstract class Cell {
    static final int SIZE = 48;

    boolean isRockCell() { return false; }

    abstract Cell affectOn(Actor a);

    abstract void paintAt(Graphics g, Int2 position);

    void drawCell(Graphics g, Int2 position) {
        g.drawRect(position.x*SIZE + SIZE/24,
                   position.y*SIZE + SIZE/24,
                   SIZE*22/24,
                   SIZE*22/24);
    }

    void fillCell(Graphics g, Int2 position) {
        g.fillRect(position.x*SIZE + SIZE/24,
                   position.y*SIZE + SIZE/24,
                   SIZE*22/24,
                   SIZE*22/24);
    }
}

abstract class SimpleCell extends Cell {
    @Override
    Cell affectOn(Actor a) { return this; }
}

abstract class TrapCell extends Cell {
    private boolean visible = false;                            // private 化
    boolean getVisible() { return visible; }

    @Override
    Cell affectOn(Actor a) {
        visible = true;
        return this;
    }
}

class RockCell extends SimpleCell {
    @Override
    void paintAt(Graphics g, Int2 position) {
        g.setColor(new Color(0, 0, 0));
        fillCell(g, position);
    }

    @Override
    boolean isRockCell() { return true; }
}

class EmptyCell extends SimpleCell {
    @Override
    void paintAt(Graphics g, Int2 position) {
        g.setColor(new Color(128, 128, 128));
        drawCell(g, position);
    }
```

```
}

class PotionCell extends SimpleCell {
    @Override
    void paintAt(Graphics g, Int2 position) {
        g.setColor(new Color(0, 255, 255));
        fillCell(g, position);
    }

    @Override
    Cell affectOn(Actor a) {
        a.setHp(a.getHp()+50);
        return new EmptyCell();
    }
}

class GoalCell extends SimpleCell {
    @Override
    void paintAt(Graphics g, Int2 position) {
        g.setColor(new Color(255, 255, 0));
        fillCell(g, position);
    }

    @Override
    Cell affectOn(Actor a) {
        if (a instanceof Player) {
            System.out.println("YOU WIN");
            System.exit(0);
        }
        return this;
    }
}

class PoisonCell extends TrapCell {
    @Override
    void paintAt(Graphics g, Int2 position) {
        if (getVisible()) {
            g.setColor(new Color(0, 128, 0));
            fillCell(g, position);
        } else {
            g.setColor(new Color(128, 128, 128));
            drawCell(g, position);
        }
    }

    @Override
    Cell affectOn(Actor a) {
        a.setHp(a.getHp()-10);
        return super.affectOn(a);
    }
}
```

1

1

第33章　おわりに

この講義テキストでは、C++と Java を用いてクラスベースのオブジェクト指向プログラミングを解説しました。

オブジェクト指向の三大基本原理（2.4 節（23 ページ））は (1) 継承、(2) 多相性、(3) 情報隠蔽ですが、このテキストでは (1) と (2) を中心にしました。初学者向けのオブジェクト指向の解説では (1) と (3) を中心に解説し、(2) が十分に解説しきれないケースが見受けられます。しかしそれではオブジェクト指向の醍醐味、たとえば AWT/Swing を使って GUI プログラムを開発する仕組みやマルチスレッド実行などが雰囲気でしか理解できないことになってしまいます。そうならないように、受講生には少々大変だったと思いますが、(2) を真正面から取り上げました。ダブルディスパッチ法などは難解だったかもしれません。

講義テキストの第 1 章に「講義の途中では、問題提起の意図もあって、敢えて読みにくいエレガントでないプログラムを提示する箇所があります。しかし最終的には全て正しいプログラムの書き方に修正していきます」と書きました。その意味での伏線回収はほぼできているはずですが、もし回収できていない未解決部分があれば連絡してください。

最後に、この講義テキストを書くにあたって講義担当者が参考にしたサイトの一部を紹介します。各サイトの著者さまに深く感謝いたします。いずれもこの授業で触れることができなかった事項などが載っている有用なサイトです。参考にしてください。

東京電機大学の坂本直志先生の「EC 科データ構造とアルゴリズム I」の講義資料

`http://edu.net.c.dendai.ac.jp/ad1/2014/`

琉球大学の山田孝治 ?[1]先生の講義資料（科目名不明）

`https://ie.u-ryukyu.ac.jp/~e085739/java.html`

[1]調べた限りでは、琉球大のソフトウェア関係で山田先生といえば山田孝治先生しか居ないように思うのですが、間違っていたら申し訳ありません　＞ 山田先生。

33.1　AWT/Swing をもっと勉強したいとき

赤坂玲音さんの「Java アプリケーション作成講座」

http://www.wisdomsoft.jp/558.html

33.2　ゲーム開発

John Horton さんの「Beginning C++ Game Programming」（少し古い版が無料公開されています）

https://pisaucer.github.io/book-c-plus-plus/Beginning_Cpp_Through_Game_Programming.pdf

Learn Object Oriented Programming By Creating a Game