Your Brain on Java—A Learner's Guide

第2版

Head First Java™

頭とからだで覚えるJavaの基本

Learn how threads can change your life

Make Java concepts stick to your brain

Avoid embarassing OO mistakes

Fool around in the Java Library

Bend your mind around 42 Java puzzles

Make attractive and useful GUIs

Kathy Sierra, Bert Bates 著
島田 秋雄, 高坂 一城, 神戸 博之 監修
夏目 大 訳

O'REILLY®
オライリー・ジャパン

tools@maruhouse.org

Head First Java™

第2版

頭とからだで覚えるJavaの基本

Kathy Sierra、Bert Bates　著
島田 秋雄、高坂 一城、神戸 博之　監修
夏目 大　訳

O'REILLY®
オライリー・ジャパン

This PDF is prepared for tools@maruhouse.org, personal use only.

本文中の製品名は、一般に各社の登録商標、商標、または商品名です。
本文中では™、®、©マークは省略しています。

This PDF is prepared for tools@maruhouse.org, personal use only.

Head First Java™

Second Edition

Kathy Sierra
Bert Bates

O'REILLY®

Beijing · Cambridge · Farnham · Köln · Sebastopol · Tokyo

This PDF is prepared for tools@maruhouse.org, personal use only.

© 2004, 2006 O'Reilly Japan, Inc. Authorized translation of the Engish edition © 2003, 2005 O'Reilly Media, Inc. This translation is published and sold by permission of O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

本書は、株式会社オライリー・ジャパンがO'Reilly Media, Inc.の許諾に基づき翻訳したものです。日本語版についての権利は、株式会社オライリー・ジャパンが保有します。

日本語版の内容について、株式会社オライリー・ジャパンは、最大限の努力をもって正確を期していますが、本書の内容に基づく運用結果についての責任を負いかねますので、ご了承ください。

This PDF is prepared for tools@maruhouse.org, personal use only.

いつまでも読者の脳に刺激を与える本……

そうなることを祈って

This PDF is prepared for tools@maruhouse.org, personal use only.

著者紹介

Kathy Sierra

かつてはゲーム・デザイナーだった彼女ですが(Virgin、MGM、Amblin'などでゲームデザインをしていました)、その当時から教育理論に強い関心を持っていました。本書のような教育スタイルは、彼女がUCLAエクステンションのエンターテインメントスタディプログラムでニューメディアオーサリングを教えていた頃にはほぼ確立されていたものです。最近では、Sun Microsystemsで、マスタートレーナーとして、最新のJavaテクノロジーをどのように教えればよいかを社内のインストラクターに指導するという仕事をしています。共著者であるBert Batesとともに、本書と同様の教育法を用いて、これまでに数百人にJavaを教えてきました。彼女の「生徒」の中にはインストラクター、デベロッパーだけでなく、「プログラムに関してはまったくの素人」という人も含まれています。Kathyは、世界でも最大級のJavaコミュニティWebサイト、javaranch.comの創始者、そして"Creating Passionate Users"というブログの創始者でもあります。

本書の他には、『Head First Servlets & JSP』、『Head First EJB』、『Head First Design Patterns』(『Head Firstデザインパターン ー 頭とからだで覚えるデザインパターンの基本』、佐藤直生監訳、木下哲也、福龍興業訳、オライリー・ジャパン)などの共著者にもなっています。

時間の空いた時には、最近飼い始めたアイスランディックホース(ポニーの一種)と遊んだり、スキーやランニング(足は非常に速いです)を楽しんだりしています。

メールアドレス：kathy@wickedlysmart.com

Bert Bates

Bertは元来、ソフトウェアデベロッパ、そしてアーキテクトなのですが、10年にわたりAI(人工知能)関連の業務を担当したことがきっかけとなって、教育理論やTBT(Technology-Based Training)に関心を持つようになりました。以降は、ずっと顧客にプログラミングを教えるということをしてきたのですが、最近では、Sunのいくつかの Java資格試験の作成メンバーにもなっています。

ソフトウェア関連の仕事をするようになって最初の10年は、Radio New Zealand、Weather Channel、A&E(Arts & Entertainment Network)といった放送業界の顧客の支援のため、世界中を旅する生活をしていました。これまで関わったプロジェクトで本人が最も気に入っているのは、Union Pacific Railroadの路線の完全シミュレーションシステムを作る、というものです。

Bertは救いようのないくらいの囲碁中毒で、長々と囲碁プログラムの開発に熱中しています。音楽が好きでギターがかなりうまく、最近ではバンジョーも始めています。スキーやランニングも好きで、「Andi」と名付けたアイスランディックホースの調教(逆に色々と教わることも多い)にも取り組んでいます。

Kathyとの共著に加え、新たに何冊かの本を執筆中です(最新情報についてはブログ"Creating Passionate Users"を参照)。

"IGS GO Server"というサイトにも時々出没しています(ハンドルネーム"jackStraw")。

メールアドレス：terrapin@wickedlysmart.com

Kathy、Bertともに、いただいたメールにはできる限り返事をするようにしていますが、メールの量によって、あるいは旅行や出張のスケジュールなどによってはそれが困難になる場合があります。テクニカルな情報を得たいというのであれば、javaranch.comのビギナー向けフォーラム(メンバーどうしのやりとりが非常に活発です)にアクセスするのが最も早いでしょう。

This PDF is prepared for tools@maruhouse.org, personal use only.

目次

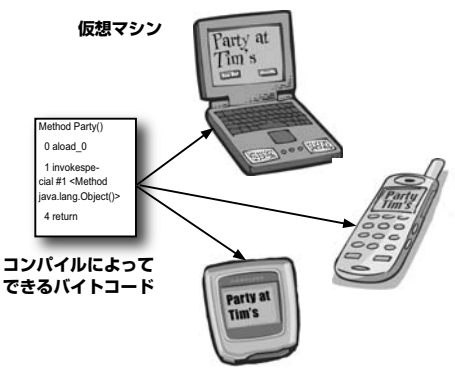
序章　この本の使い方 xxiii

Javaを学ぶのは「脳」です。Javaに限らず、人間が何かを学ぶためには、脳を退屈させないことが重要です。脳は目の前に起きていることを重要でないとみなすと退屈し、もっと「重要なこと」を探し始めます。脳の構造は人間が野生に暮らしていた時のままなので、通常、肉食獣が近づいてくる、裸でスノーボードをするといった出来事以外、重要とは見なされません。Javaを学ぶには、脳に「Javaを学ばなければ命に関わる」と思わせる必要があります。

この本が対象としている読者	xxiv
本当に真面目なJavaプログラムの本なのか？	xxv
単に読むだけではなく「学習」できる本	xxvi
自分の思考について考える	xxvii
この本で著者が工夫したこと	xxviii
この本を読むときに気を付けること	xxix
用意するもの	xxx
その他の注意事項	xxxi
テクニカルエディタの紹介	xxxii
謝辞	xxxiii

1章　Javaの世界に飛び込もう ——ともかくまずは始めてみる 1

Javaは独自の特徴を持つプログラミング言語です。そのため、発表以来（発表当時のバージョンは1.02）、数多くのプログラマを惹きつけています。Javaには、文法がわかりやすい、オブジェクト指向である、メモリ管理が容易といった特徴もありますが、何と言っても重要なのは、その「移植性」です。一度書いてコンパイルしたコードがどこでも実行できるというのはJavaの最大の特徴と言っていいでしょう。1章ではこうしたJavaの特徴についておおまかに学びます。

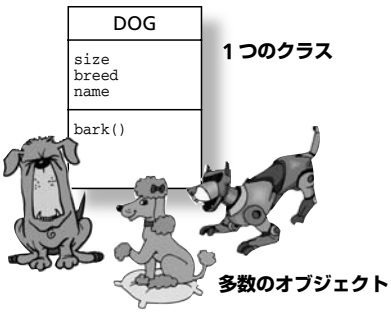


Javaプログラムのできるまで	2
Javaプログラム作成の手順	3
Javaの歴史（概略）	4
Javaプログラム（ソースファイルの構造）	7
クラスの構造	8
mainメソッドを持つクラスの作成	9
mainメソッドの「中身」	10
分岐	13
本格的なJavaアプリケーション	14
PhraseOMaticプログラムの仕組み	17
特別座談会（コンパイラとJVM）	18
エクササイズとパズル	20

This PDF is prepared for tools@maruhouse.org, personal use only.

2章　クラスとオブジェクト ──オブジェクト指向の国へ 27

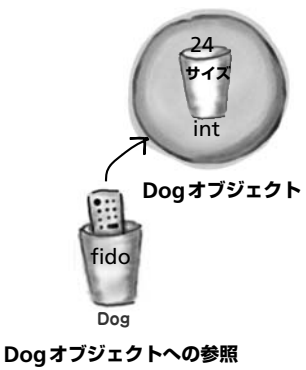
2章では、オブジェクト（クラス）について解説します。1章で紹介するJavaプログラムの例は、すべてmain()メソッドのみから構成されるものですが、これは正確には「オブジェクト指向」のプログラムとは言えません。オブジェクト指向でない言語のことを「手続き指向言語」と呼びますが、2章では、どのようにすれば手続き指向的ではない、オブジェクト指向的なプログラムが書けるのか、どうすればオブジェクトを作れるのかを具体的に学ぶことになります。Javaでのオブジェクト指向プログラミングの楽しさを知ってもらえればと思います。その他、この章では「クラス」、「オブジェクト」という言葉の違い、オブジェクト指向のメリットなどについても触れます。



対決！　オブジェクト指向対手続き指向	28
クラスを作る際に考えるべきこと	34
クラスとオブジェクトの違い	35
クラスの作成と使用	36
Movieクラスの作成とテスト	37
mainから外へ飛び出す	38
アプリケーションの実行	40
エクササイズとパズル	42

3章　プリミティブと参照 ──変数の基礎知識 49

変数にはプリミティブ型、参照型という2つの種類があります。また、参照型の変数には、配列型、文字列（String）型、クラス型などの種類があります。こうした変数を使用すれば、単なる数値だけを使用するよりも複雑なプログラムが作れます。この章では、Javaにおける変数がどのようなもので、どのように使うのか、といったことを解説します。また、「ガーベージコレクション」についても詳しく説明します。

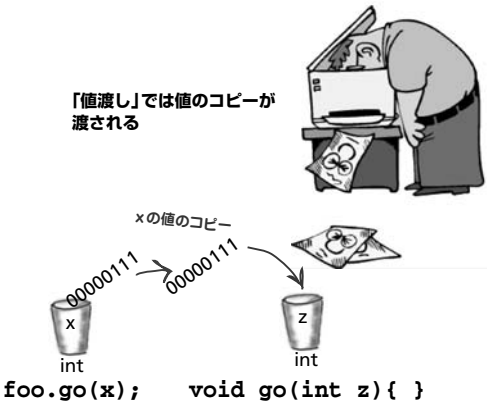


変数の宣言	50
「ダブルモカをください。intサイズで」	51
変数に値を代入する際の注意	52
名前付けのルール	53
オブジェクトを操作する	54
オブジェクトへの参照も変数の値である	55
Javaの真実（ゲスト：オブジェクトリファレンス）	56
オブジェクト、参照、ガーベージコレクション	57
オブジェクトの生と死	58
配列はコップを並べたトレイのようなもの……	59
Dog型の配列を作る	60
Dogオブジェクトの操作	61
エクササイズとパズル	63

This PDF is prepared for tools@maruhouse.org, personal use only.

4章　メソッドとインスタンス変数
―オブジェクトの「ふるまい」……71

ステートは、メソッドに影響を与え、メソッドはステートに影響を与えます。オブジェクトにはインスタンス変数（ステート）とメソッド（機能）があります。4章では、インスタンス変数とメソッドの関係がどのようになっているかということを学びます。インスタンス変数の値は、同じクラスのオブジェクトでも個々に変わります。そして、インスタンス変数の値は、メソッドの機能に影響を与えるのです。メソッドがインスタンス変数の値を利用して機能するとも言えます。たとえば、weight変数の値が「5」増えるごとに、makeNoise()メソッドが出す音が少しずつ大きくなる、というようなことが起きるのです。

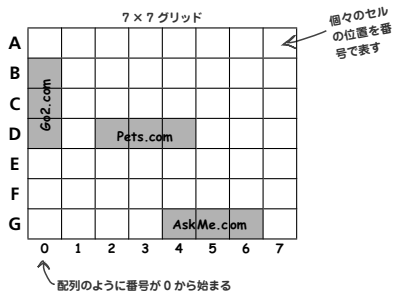


クラスにはインスタンス変数とメソッドがある　72
メソッドの機能に影響するインスタンス変数（ステート）　73
メソッドに値を渡す　74
メソッドは値を戻す　75
メソッドには、複数のパラメータを宣言することもできる　76
引数として指定された値の「渡され方」　77
パラメータ、戻り値の有効な使い方　79
カプセル化　80
データの保護　81
Javaの真実（ゲスト：オブジェクト）　81
Gooddogクラスをカプセル化する　82
配列でのオブジェクトの扱い　83
インスタンス変数の宣言と初期化　84
インスタンス変数とローカル変数の違い　85
変数の比較　86
エクササイズとパズル　88

5章　本格的なプログラムの作成
―本格的なプログラム（メソッド）を書く……95

この章では、実用に耐えうる本格的なプログラムを作成するのに必要ないくつかのことがらについて解説します。実用的なプログラムを書くには、演算子、ループなどを使いこなせるようになる必要があります。ループはたとえば、乱数の生成や、Stringからintへのデータ変換などに使えます。この章では、ゲーム（バトルシップゲーム）のプログラムを例にとり、本格的なプログラムをゼロから作り、テストするという作業がどのようなものになるのか、ということについても学びます。

バトルシップゲーム



バトルシップゲームプログラムを作る　96
プログラムの設計　97
簡易版ゲームプログラム　98
クラスの作成　99
テストコードの作成　101
SimpleDotComクラスのテストコードを作成する　102
SimpleDotComクラスのテストコード　103
checkYourself()メソッド　104
新しいテクニック　105
SimpleDotComクラスのコードとそのテストコードの最終形　106

This PDF is prepared for tools@maruhouse.org, personal use only.

SimpleDotComGameクラス（のmain()メソッド）の仮コード 108
SimpleDotComGameクラスのmain()メソッド 110
random()とgetUserInput() 111
GameHelperクラス 112
forループの基礎知識 114
ループの処理 115
拡張forループ 116
Stringをintに変換する 117
プリミティブ型のキャスト 117
エクササイズとパズル 118

6章　Java APIの基礎 ―Javaライブラリを使用する……125

Javaの開発環境にはあらかじめ数多くのクラスが用意されています。そうしたクラスの集まりをJavaライブラリ（Java API）と呼びます。作りたいプログラムに合うクラスをJavaライブラリで見つけることができれば、自分で同じようなクラスを改めて作る必要はなくなります。その分の時間や労力を別のことに振り向けることができるのです。Java APIを使用すれば、それぞれのプログラムの、他のプログラムにない特徴的な部分だけを書けばよくなります。通常は、プログラムのかなりの部分が、Java APIの中でも特に重要な「コアライブラリ」に用意された大量のクラスを「ブロック」のように使うだけでできあがります。

「java.utilパッケージにArrayListというクラスがあることがわかったのはよかったと思います。でも、自分の力でこのようなクラスを見つけるにはどうしたらいいのですか？」

― Julia（31才）、職業ハンドモデル

簡易版バトルシップゲームプログラムのバグ 126
バグの原因は？ 127
バグの修正方法 128
方法1はあまりに複雑 129
Java APIのクラスを使う 132
ArrayListクラスの使い方 133
Javaの真実（ゲスト：ArrayListクラス） 135
ArrayListと配列の比較 136
ArrayListを使用したバグの修正 138
ArrayListを使って修正したコード 139
「完全版」バトルシップゲームプログラムの作成 140
簡易版から完全版への変更ポイント 141
DotComBustクラスの処理 142
DotComBustクラスの仮コード 144
DotComBustクラスのリアルコード 146
DotComクラスのリアルコード 150
高度な条件判定式 151
GameHelperクラス 152
ライブラリ（Java API）の使用 154
APIの使い方 158
エクササイズとパズル 161

This PDF is prepared for tools@maruhouse.org, personal use only.

7章　継承とポリモーフィズム ──オブジェクト指向をより深く知る 165

プログラムを作る際には、将来のことを考えて設計をしなければなりません。たとえば、自分以外の人が簡単に改造できるようにしておくということは重要です。また、「もうすぐ完成」という段階になっての突然の仕様変更などにも柔軟に対応できるようにすべきでしょう。この章では、そのために役立つ、オブジェクト指向言語の「ポリモーフィズム」、「継承」といったことがらについて解説します。ポリモーフィズムと継承について十分に理解すれば、より柔軟性のある、改造の容易なプログラム（クラス）を設計することができるようになります。

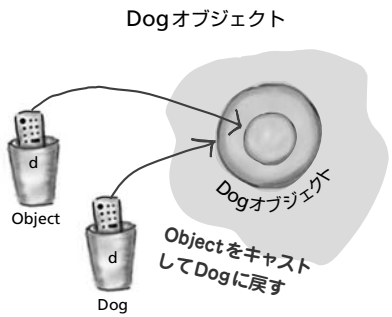
2章の復習 166
継承について理解する 168
動物シミュレーションプログラムを構成する 170
重複を防ぐのに役立つ継承 171
動物の「食べ方」はみな同じか 172
継承関係を再検討する 173
呼び出されているメソッドは？ 175
継承関係を図に表す 176
IS-A関係とHAS-A関係 177
IS-A関係に関する注意事項 178
クラスの継承関係が正しいことを確認する方法 179
サブクラスが継承できるもの、できないもの 180
継承の適切な使用 181
継承は何のために行う？ 182
オーバーライドに関するルール 190
メソッドのオーバーロード 191
エクササイズとパズル 192

8章　インターフェースと抽象クラス ──ポリモーフィズムの高度な利用方法 197

継承とポリモーフィズムがどのようなものかをただ学んだだけではあまり意味がありません。この章では、その継承やポリモーフィズムをより有効に利用するために必要な、「インターフェース」というものについて学びます。インターフェースをうまく利用すれば、単にクラスを継承しただけの場合より、さらに柔軟性、拡張性の高いプログラムが書けます。インターフェースは、「抽象クラス」と呼ばれる種類のクラスに似ています。抽象クラスというのは、インスタンス化ができないクラスのことです。この章を読めば、インターフェースのメリットがよくわかるはずです。

前章の例のおさらい 198
Animalオブジェクトはどんな動物を表す？ 200
抽象クラスのインスタンスは作成できない 201
抽象クラスと具象クラス 202
抽象メソッド 203
抽象メソッドの本体はサブクラスで必ず作る 204
ポリモーフィズムの応用例 206

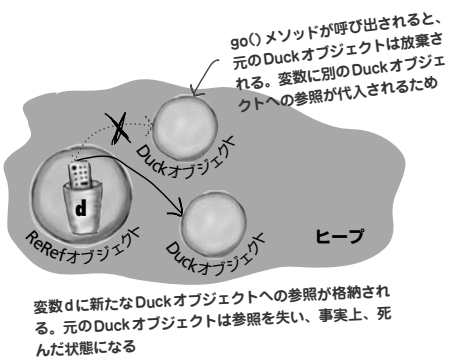
This PDF is prepared for tools@maruhouse.org, personal use only.



クラスの異なるサブジェクトを混在させる 207
Animalのサブクラス以外のオブジェクトを格納するにはどうするか 208
Objectクラスのメソッド 209
Objectクラスを使用する際の注意 211
DogオブジェクトがDogクラスのオブジェクトではなくなるとは？ 212
元のクラスのメソッドは使えない 213
すべてのクラスはObjectクラスの特性、機能を受け継ぐ 214
「ポリモーフィズム」は元来「多くの形」を意味する 215
利用できるメソッドを変更する 218
ペットショッププログラムのための修正方法 219
インターフェース 224
Petインターフェースの作成とインプリメント 225
スーパークラスバージョンのメソッドを使用する 228
エクササイズとパズル 230

9章　コンストラクタとガーベージコレクション——オブジェクトの生と死 235

オブジェクトには、「生まれて、やがて死ぬ」というライフサイクルがあります。そのライフサイクルをどのようなものにするか決めるのはプログラマの仕事です。あるオブジェクトがいつ作られ、いつ削除されるか、ということをプログラマが決めるということです。ただし、オブジェクトを削除するためのコードをプログラマが直接書くわけではありません。オブジェクトを削除するのは、非情なガーベージコレクタです。この章では、オブジェクトをどのように作るか、作ったオブジェクトはどこに置かれるか、どうすればオブジェクトを効率的に消滅させることができるか、といったことについて解説します。具体的には、ヒープ、スタック、スコープ、コンストラクタ、スーパーコンストラクタなどについて詳しく触れることになります。



オブジェクトはどこに存在するか 236
メソッドはスタックに格納される 237
ローカル変数が参照型のものである場合は？ 238
インスタンス変数の格納場所 239
オブジェクト誕生の神秘 240
コンストラクタのコード 242
オブジェクトのインスタンス変数の設定 243
コンストラクタを使用してオブジェクトのインスタンス変数の初期設定をする 244
複数のコンストラクタを作る 245
引数のないデフォルトコンストラクタはどのような場合でも必ず作られるのか 246
コンストラクタの復習 249
スーパークラスのコンストラクタとサブクラスのコンストラクタは同時に実行されるのか 250

This PDF is prepared for tools@maruhouse.org, personal use only.

オブジェクトのコンストラクタにおける
　スーパークラスのコンストラクタの役割 251
Hippoオブジェクトを作るには、Animalオブジェクト、
　Objectオブジェクトとしての部分も必要 252
スーパークラスのコンストラクタを意図的に呼び出す場合は？ 253
親より先に子供は産まれるか？ 254
スーパークラスのコンストラクタに引数がある場合 255
コンストラクタの中でコンストラクタを呼び出すには？ 256
オブジェクトの「寿命」はどのくらい？ 258
参照型変数の場合 260
特別座談会（インスタンス変数とローカル変数） 264
エクササイズとパズル 266

10章　数値の扱いとスタティック変数、メソッド ―数値の扱い……273

この章ではJavaにおける数値の扱いについて解説します。「数値の扱い」には、絶対値を求める、丸め（概算）を行う、2つの数値の大小を比較する、など様々なことがらが含まれますが、Java APIにはそうしたことを容易に行えるメソッドが用意されています。小数点以下の桁数を正確に定めて出力することなども必要になるかもしれませんし、桁数の大きい数値には、読みやすいよう、途中にカンマを入れて出力することなども必要になるでしょう。日付の扱いも問題です。日付と一口に言ってもその表示方法はさまざまだからです。その他、String（文字列）から数値、あるいは数値からStringへの変換についても知っておく必要があるでしょう。この章ではJavaにおいて変数やメソッドがスタティックであるとはどういうことであるか、ということについても解説します。

スタティック変数は、あるクラスのすべてのインスタンスによって共有される

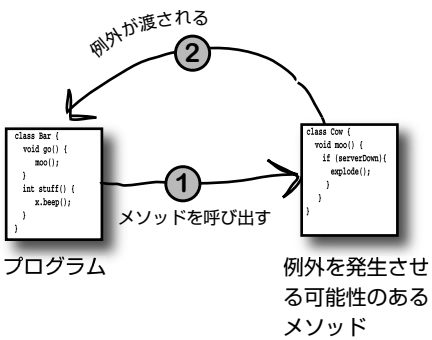
Mathクラスのメソッド：「グローバル的」なメソッド 274
通常のメソッドとスタティックメソッドの違い 275
スタティックメソッドを持つクラスを作る 276
スタティックメソッドではスタティックでない変数
　（インスタンス変数）は利用できない 277
スタティックメソッドの中で通常のメソッドを呼び出すことは
　できない 278
スタティック変数：
　クラスのすべてのインスタンスで値が同じになる変数 279
スタティック変数の初期化 281
static finalと宣言した変数は定数のようになる 282
finalは必ずしもstaticと同時に使用しなくてよい 283
Mathクラスのメソッドの概要 286
プリミティブのラッピング 287
プリミティブとオブジェクトの境界線を曖昧にする機能 289
さまざまなケースで行われる自動ラッピング、アンラッピング 290
ラッパに用意されているユーティリティメソッド 292
数値を数字に変換する 293
数値のフォーマッティング 294
フォーマッティングについての指示の仕方 295

This PDF is prepared for tools@maruhouse.org, personal use only.

"%"は「この部分にフォーマッティング対象の値が入る」という
　ことを意味する　296
フォーマット文字列には独自の文法がある　297
フォーマッティング指示のコード　298
必須なのは型の指定だけ　299
フォーマッティング対象を複数指定する　300
日付のフォーマッティング　301
日付の活用　302
時間を進める、さかのぼる　303
Calenderを継承するオブジェクトの取得　304
Calenderオブジェクトを使う　305
Calender APIの主要機能　306
スタティックインポート　307
特別座談会（インスタンス変数とスタティック変数）　308
エクササイズとパズル　310

11章　例外処理
―危険をともなうプログラムの作成 315

プログラムにはトラブルはつきものです。原因は、必要とされているファイルが実は存在しない、サーバがダウンしている……などさまざまです。プログラマがいかに優秀でも、あらゆる物事を完全にコントロールすることはできません。トラブルの起きそうな処理を行うプログラムを書く時には、あらかじめそのトラブルに対処するコードを組み込んでおく必要があります。ただ、問題は、トラブルが起きそうかどうかをどのように判断するか、ということです。トラブルに対処するコードをどこに入れるかということも知っておかねばなりません。この章では、JavaSound APIを利用した音楽演奏プログラムを例に、そうしたことを解説していきます。



音楽演奏プログラムを作る　316
基礎的なことがらについて学ぶ　317
まず必要なのはSequencerオブジェクト　318
正しく動作しない恐れがある
　（他人が書いた）メソッドを呼び出す　319
Javaのメソッドでは、実行中に何か例外が起きたことを
　呼び出し側のコードに知らせるために「例外」を発生
　させる　320
try/catchブロック　321
例外はすべてException型を継承するクラスのオブジェクト　322
例外を発生させるメソッドを作る　323
try/catchブロックでの処理の流れ　326
finallyブロック：例外が発生した場合も
　しなかった場合も行うべき処理　327
1つのメソッドが複数種の例外を発生させることもある　329
例外とポリモーフィズム　330

This PDF is prepared for tools@maruhouse.org, personal use only.

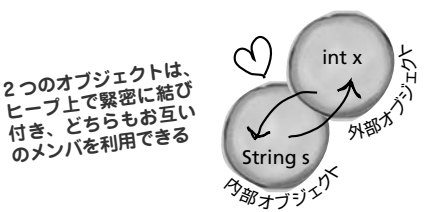
catchブロックを複数作る場合は、
　対応する例外クラスが下位のものから順に並べる　332
上位の例外クラスに対応するcatchブロックを
　先にすることはできない　333
例外処理をしたくない時は……「回避！」　335
例外処理の回避は実は「先送り」　336
例外についてのまとめ　337
音楽演奏プログラム　338
プログラムキッチン（音楽演奏プログラム）　339
音楽演奏プログラムの基本的な作り方　340
簡単な音楽演奏プログラム　342
MidiEventオブジェクト（MIDIデータ）の作成　343
ShortMessageオブジェクトの内容　344
setMessage()メソッド　345
楽器と音程を変更できるプログラム　346
12章以降で作成するプログラムの概要　347
エクササイズとパズル　348

12章　GUIの基礎 ―グラフィカルなインターフェース 353

GUI（Graphical User Interface）は今や、プログラム（アプリケーション）にとって不可欠なものと言えるでしょう。たとえ、作るのがWebブラウザによって利用されるサーバサイドのプログラムであったとしても、その点に変わりはありません。12、13章では、このGUIについて、そしてイベント処理、内部クラス、といったJava言語にとって重要な概念について解説していきます。12章ではまず、画面上にボタンを作る方法を学ぶことになります。その他、画面上に絵を描く方法、JPEG画像を表示する方法、アニメーションを作る方法なども学んでいきます。

```
class MyOuterClass {

    class MyInnerClass {
      void go() {
      }
    }

}
```

外部オブジェクトと内部オブジェクトは緊密に結び付く



まずウィンドウを作る　354
ボタンを備えたウィンドウ（フレーム）を作る　355
前ページのプログラムの場合、
　ボタンをクリックしても何も起きない……　356
イベント処理　357
イベントリスナーとイベントソースのコミュニケーション　359
ActionEventの扱い　360
リスナー、ソース、イベント　361
再びGUIの話……　363
描画パネルをフレームに組み込む　364
paintComponent()メソッドの例　365
Graphics2DをGraphicsの代わりに使用する　366
描画する円の色をグラデーションにするためのコード　367
イベントと図形の描画　369
フレームに複数のウィジェットを組み込む　370
ボタンのクリックによって円の色を変化させる　371

This PDF is prepared for tools@maruhouse.org, personal use only.

ボタンを 2 つにする 372
内部クラス 376
内部オブジェクトを作る 378
Javaの真実（ゲスト：内部オブジェクト） 380
内部クラスをアニメーションに利用する 382
アニメーションプログラムのコード 384
プログラムキッチン（ミュージックビデオプログラム） 386
GUI部品以外で発生するイベントに対応するリスナーの作成 387
MidiEventオブジェクトの作成 388
ユーティリティメソッドmakeEvent()の使用例 389
第 2 段階：Controllerイベントのリスナーの作成、登録 390
第 3 段階：音楽に合わせて図形を描画する 391
エクササイズとパズル 394

13章　Swingの基礎知識 ——Swingを利用する 399

Swingはさほど難しいものではありません。ウィジェットが画面上のどこに表示されるかを気にしなければ、の話ですが……。Swingを利用するコード自体は比較的簡単に書けるのですが、プログラムをコンパイルして実行してみると「あれ、なぜこれがこんなところに……」ということが起きやすいのです。実は、コードが簡単に書けるのも、ウィジェットが思いがけない位置に表示されるのも、その理由は「レイアウトマネージャ」というオブジェクトにあります。レイアウトマネージャは、少し手間をかければ思いどおりにコントロールすることができます。この章では、レイアウトマネージャをはじめ、Swingに関する基本的なことがらについて解説します。

east、west領域のコンポーネントは、横の長さを必要に応じてのばせる

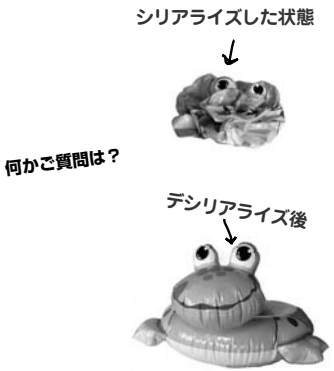
north、south領域のコンポーネントは、縦の長さを必要に応じてのばせる

Swingコンポーネント 400
レイアウトマネージャ 401
レイアウトマネージャの機能 402
レイアウトマネージャの分類 403
さまざまなSwingコンポーネント 413
プログラム・キッチン（BeatBoxプログラム） 418
エクササイズとパズル 424

14章　シリアライゼーションとファイルI/O ——オブジェクトの保存 429

オブジェクトは、ファイルに保存することができます。オブジェクトは大きく「機能」と「状態（ステート）」という要素に分けられるのですが、オブジェクトの機能は、そのオブジェクトが属するクラスのコードによって決まります。一方、「状態」はオブジェクトごとに異なります。この「状態」、つまりインスタンス変数の値さえ保存すれば、いつでも同じオブジェクトを再現できます。とはいえ、インスタンス変数の値を逐一確認し、それをファイルに保存するコードを書くのは大変です。そこで、Javaには、そうしたことを自動的に（オブジェクト指向言語らしく）行う「シリアライゼーション」という機能が用意されています。

This PDF is prepared for tools@maruhouse.org, personal use only.



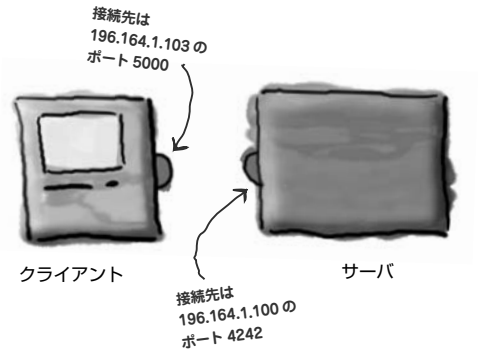
ドラムパターンを保存する 430
オブジェクトの保存方法 431
オブジェクトをシリアライズしてファイルに保存する 432
データの通り道となる「ストリーム」 433
シリアライゼーションとは 434
インスタンス変数の値はどのようなかたちで保存されるか 435
クラスにシリアライズの機能を持たせるには
　Serializableインターフェースをインプリメントする 437
インスタンス変数をシリアライズの対象から外すには
　「transient」宣言する 439
デシリアライゼーション：保存したオブジェクトの再現 441
デシリアライズはどのように行われるか 442
ゲームキャラクターオブジェクトのシリアライズと
　デシリアライズ 444
オブジェクトのデータを
　プレーンテキストファイルに書き込む 447
暗記カードプログラム 448
QuizCardBuilderクラスの概要 449
java.io.Fileクラス 452
バッファを使うメリット 453
テキストファイルからデータを読み込む 454
QuizCardPlayerクラスの概要 455
StringTokenizerの使用方法 458
クラスのバージョン 460
バージョンID 461
プログラム・キッチン
　（BeatBoxプログラムにドラムパターンを保存する） 462
エクササイズとパズル 466

15章　ネットワークとスレッド
——ネットワーク接続のためのプログラムを書く 471

Javaではネットワーク接続のためのプログラムも簡単に書くことができます。複雑な、いわゆる「低水準の」処理は、java.netライブラリのクラスによってすべて自動的に行われるからです。その他、ネットワーク経由でのデータ送受信を単なるI/Oのように扱うことができる、というのもJavaの大きな特徴でしょう。ただ、同一マシン上のI/Oとは使用する接続ストリームが若干異なるだけです。この章では、ネットワーク経由でのデータ送受信、そして、複数の処理を同時に行うために必要な「スレッド」という概念について解説します。サンプルプログラムとして、ドラムパターンをサーバとの間でやりとりできる一種の「マルチスレッドチャットクライアント」も紹介します。

チャットクライアントとチャットサーバ 472
チャットプログラムの概要 473
接続、送信、受信 474
ネットワーク（ソケット）接続を確立する 475
TCPポートもアドレスの一部 476

This PDF is prepared for tools@maruhouse.org, personal use only.

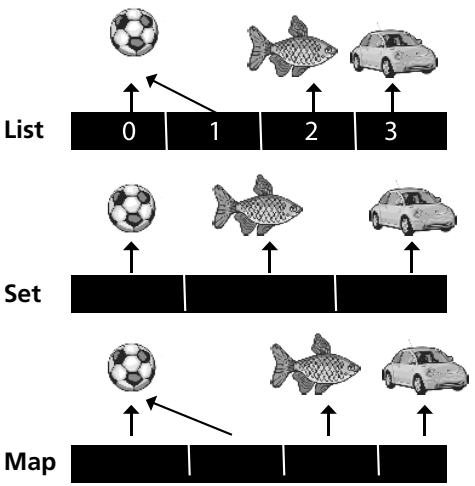


ソケット接続経由でのデータの読み取りにも
　BufferdReaderが使用できる　478
ソケット接続経由でのデータ書き込みには
　PrintWriterを使用する　479
簡単なクライアントプログラム　480
クライアントプログラムのコード　481
サーバプログラムの作成　483
サーバプログラムのコード　484
簡単なチャットクライアントの作成　486
スレッドを表すThreadクラス　490
「マルチスレッド」の意味　491
スレッドを起動するためのコード　492
スレッドには「ジョブが必要」　493
Runnableインターフェースをインプリメントする　494
スレッドの状態の変化　495
通常は実行可能／実行中という
　2つの状態の間で切り替えが行われる　496
スレッドスケジューラ　497
スレッドをスリープさせる　501
Sleepメソッドを使用すれば
　スケジューラをある程度コントロールできる　502
ユーザースレッドを2つ作る　503
スレッドの問題点「競合」　504
Dr.Steveショー「離婚の危機？　このカップルは修復可能か」　505
複数のスレッドによるオブジェクトの共有　506
「RyanとMonica」プログラム　507
オブジェクトのアクセスには「ロック」をかけることができる　509
makeWithdrawl()メソッドに
　同時にアクセスできるスレッドを1つに限定する　510
すべてのオブジェクトが備える「ロック」　511
「更新が無効になる」という問題　512
コードを実行してみる……　513
incremant()メソッドをsynchronized宣言する　514
synchronized宣言の問題点　516
改良型チャットクライアント　518
チャットサーバのコード　520
プログラム・キッチン
　（BeatBoxプログラムの完成バージョン）　523
エクササイズとパズル　524

This PDF is prepared for tools@maruhouse.org, personal use only.

16章　コレクションとジェネリック型——データ構造 529

Javaはソートが簡単にできる言語です。そのためのツールがあらかじめ用意されているので、プログラマが自らソートのためのコードを書く必要はありません。Javaのコレクションフレームワークに用意されたクラスを使用すれば複数のデータをまとめ、操作するためのコードはほぼどのようなものでも書けると言っていいでしょう。データのリストを作ることも、リストに新たな要素を追加することも、リストの中から名前を手掛かりに必要なデータを見つけ出すことも簡単にできます。データの重複を自動的に排除する機能を持たせることなども可能です。社員名簿を業績順にソートするといったことも容易です。



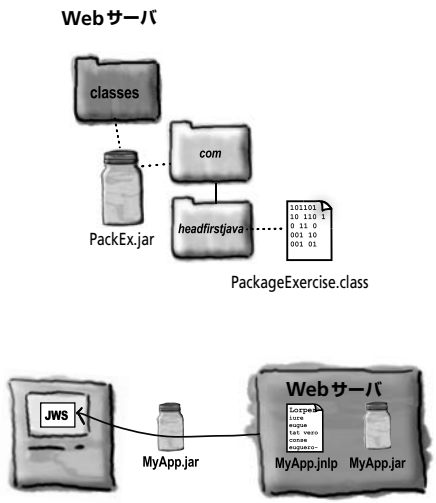
ジュークボックスで人気のある曲を調べる 530
Arraylistに曲名を保持するコード（ソート機能なし） 531
ArrayListクラスにはソートメソッドはない 532
ArrayListだけがコレクション（複数のデータをひとまとめにするクラス）ではない 533
TreeSetを使用してもよいが……
Collections.sort()メソッドを使用することもできる 534
Collections.sort()を使用してコードを改良する 535
StringではなくSongオブジェクトをソートする 536
Songクラスを利用してコードを改造する 537
コンパイルできない！ 538
sort()メソッドの宣言 539
コレクションを「タイプセーフ」にするジェネリック型 540
ジェネリック型 541
ジェネリッククラスを使用する 542
ArrayListの型パラメータを使用する 543
ジェネリックメソッドを使用する 544
sort()メソッド 547
この場合の"extends"は「"extends"あるいは"implements"」という意味 548
これで問題が明らかに…… 549
comparableインターフェースをインプリメントしたSongクラス 550
リストのソートは確かにできたが…… 551
Comparatorを使用する 552
Comparatorを使用してコードを改造する 553
ソートは問題なくできるが、要素の重複が 556
必要なのはListではなくSet 557
Collection API（一部） 558
HashSetを使用する 559
オブジェクトを「同等」と見なす条件は？ 560
HashSetの重複確認：hashcode()とequals() 561

This PDF is prepared for tools@maruhouse.org, personal use only.

hashCode()とequals()をオーバーライドしたSongクラス 562
ソートした要素の順序が狂わないためにはTreeSetを使う 564
TreeSetについての注意点 565
TreeSetの要素は「比較のできる」ものにする 566
List、Setの次はMap…… 567
再びジェネリック型について 568
ポリモーフィズムとジェネリック型 569
ArrayList<Dog>ではどうか…… 570
これが許容されたとしたら…… 571
ワイルドカードを使えば…… 574
別の書き方…… 575
エクササイズ&パズル 576

17章　プログラムの提供 ―プログラムが完成したら…… 581

プログラムが完成したら……。何度も修正をし、さまざまな苦労を重ねて(途中、「もう、やめたい」などと言ったりしながら)素晴らしいプログラムを完成させても、人に使ってもらえなければ意味がありません。しかし、どうすれば使ってもらえるのでしょうか。17章と18章では、プログラムの「パッケージング」などについて触れる他、プログラムの提供方法の種類(ローカル、リモート、その中間、など)などにも触れていきます。具体的には、実行可能JARファイル、Java Web Start、RMI、Java Web Startなどについて解説することになります。一見、難しそうですが、実際にはそれほどでもないので安心してください。

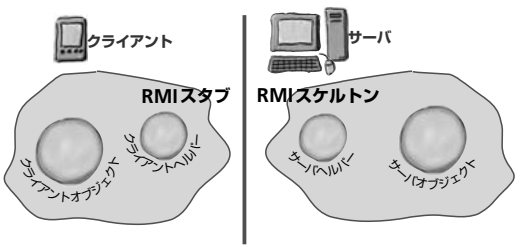


Javaプログラムの提供 582
プログラムができた。その後は…… 583
ソースファイルとクラスファイルを分けて保存する 584
プログラムに必要なファイルをJARファイルにまとめる 585
JARファイルの実行 586
クラスをパッケージにまとめる 587
パッケージ名の重複を避ける 588
クラスをパッケージに所属させる 589
パッケージに属するクラスのコンパイル、実行 590
-dオプションの優れた機能 591
パッケージに属するクラスを
　実行可能JARファイルにまとめる 592
JARファイルの内容確認、内容の抽出 593
Java Web Start 597
.jnlpファイル 599
JWSを利用するプログラムを提供する手順 600
エクササイズとパズル 601

This PDF is prepared for tools@maruhouse.org, personal use only.

18章　RMIによるプログラムの提供 ―分散コンピューティング……607

プログラムの構成要素は一か所にまとまっていなくてはならないわけではありません。すべてが同じコンピュータ上の同じヒープにあり、使用するJVMも1つで済めば、これほど楽なことはありません。しかし、それがいつでも可能とは限らないのです。また、そうすべきでないこともあります。たとえば、プログラムは大変な処理能力を必要とするものなのに、ユーザーが携帯用の小型デバイスしか持っていないとしたらどうでしょうか。セキュリティ上の理由から、データベースアクセスをサーバ上のコードにのみ許可したい、ということもあるでしょう。この章では、主に、RMI（Remote Method Invocation）という技術について解説します。その他、サーブレットや、EJB（Enterprise Java Beans）、Jiniといった技術にも簡単に触れます。

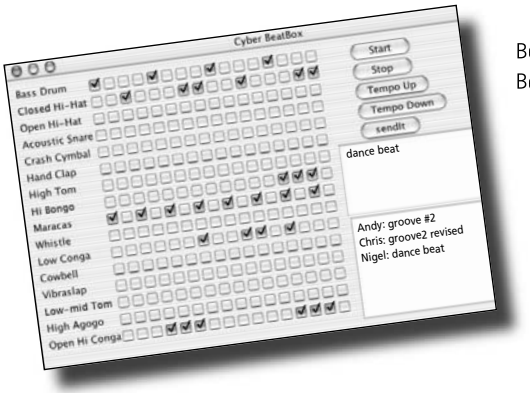


メソッドの呼び出しは
　通常、同一のヒープ上のオブジェクト間で行われる　608
他のマシン上のオブジェクトのメソッドを呼び出すには？　609
小型デバイス上のオブジェクトAからパソコン上の
　オブジェクトBのメソッドを呼び出すには？　610
RMIを設計してみる　611
ヘルパーの役割　612
RMIのクライアントヘルパー、サーバヘルパー　614
クライアントがスタブを取得するには　620
スタブのクラスファイルをクライアントに提供するには　621
クライアントマシン、サーバマシンに必要なクラスファイル　622
RMIは実際にどのように応用されているのか？　624
サーブレット　625
サーブレットの作成・実行　626
簡単なサーブレットの例　627
PhraseOMaticプログラムを利用した
　サーブレットを作ってみる　629
サーブレットで利用しやすいよう改良した
　PhraseOMaticクラスのコード　630
EJB（Enterprise JavaBeans）：大規模システムのためのRMI　631
Jiniとは　632
サービスの自動検索機能　633
ルックアップサービスの自動修復機能　635
ユニバーサルサービスブラウザ　636
ユニバーサルサービスブラウザの動き　637

This PDF is prepared for tools@maruhouse.org, personal use only.

付録A　BeatBoxプログラム完全版 649

BeatBoxプログラム完成版。付録Aには、本書の後半で紹介したBeatBoxプログラム（ドラムマシンプログラム）の完成版のコードを載せてあります。このプログラムがあればロックスターも夢ではありません。



BeatBoxプログラム完成版（クライアント）のコード　650
BeatBoxサーバプログラム完成版　657

付録B　本文で触れなかった重要事項ベスト10 659

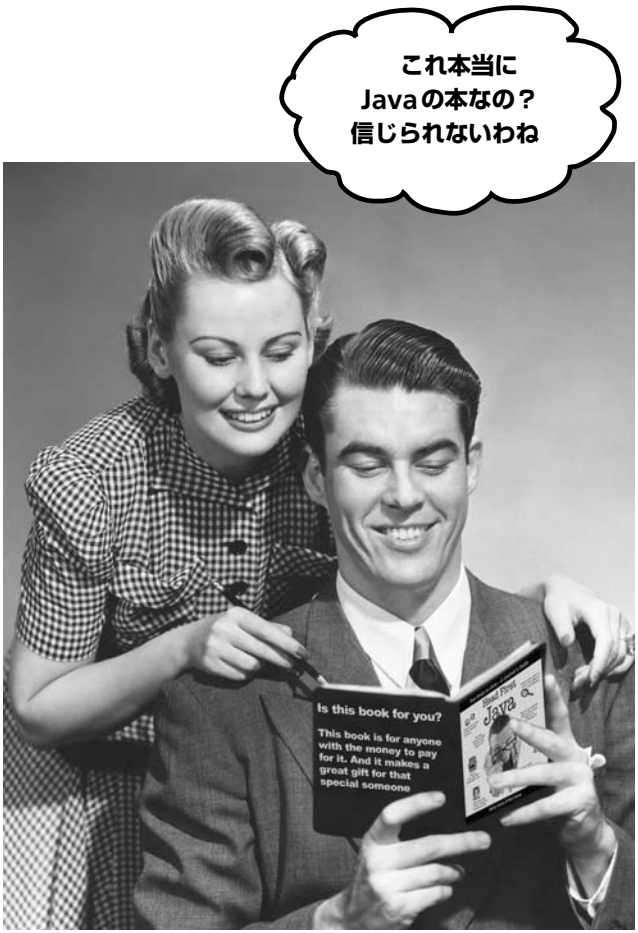
本文で触れなかった重要事項ベスト10。本文で学んだことだけではまだ一人前のJavaプログラマとは言えません。付録Bでは、本文では触れなかった重要なことがらについて解説します。これで、この本での勉強は本当に終わりです。

索引 677

訳者あとがき 682

This PDF is prepared for tools@maruhouse.org, personal use only.

序章　この本の読み方



はじめに、この本はどんな本なのか、どのように読めば
いいのかを明らかにしておきます。

This PDF is prepared for tools@maruhouse.org, personal use only.

この本が対象としている読者

この本は、次のような人を対象に書かれています。

① プログラミングをしてみたいと思ったことがある。

② Javaを学びたい。

③ 楽しく勉強をしたい。

本書はいわゆる「リファレンスブック」ではなく、Javaの「百科事典」でもありません。あくまでJavaを学習するために作られた本です。

この本を読むべきでない人

次のような人はこの本を読むべきではありません。

① HTMLさえできれば十分と思っていて、スクリプト言語でさえ使ってみたいとは思わない。
（プログラミングに対して少しでも興味があればいいですが、HTMLのタグにしか興味がないという人にはこの本は向きません）

② C++専門のプログラマで、Javaを本気で学ぶ気はない。

③ 保守的で新しいもの、変わったものが好きではない。技術書に右のようなアヒルの写真などが載っていると「ふざけるな」と思う。

［この本はその他、Javaを勉強したいと思っている人へのプレゼントとしても最適だと思います。そういう友人、知人のいる人は検討してみてください］

This PDF is prepared for tools@maruhouse.org, personal use only.

本当に真面目なJavaプログラミングの本なのか？

どうしてこんなにおかしな絵や写真が入っているんだ？
こんなので本当に勉強できるのか？
なぜピザがJavaに関係あるんだ？
……。

こんな風に思う人がいるかもしれません……

でも、この本がこうなっているのには意味があります……。

人間の脳は常に「目新しいもの」を求めています。いつも、変わったもの、普通でないものを探しているのです。人間が生き延びるため、生来、そのように作られているからです。

現代の社会で人間がトラのエサになってしまう、というようなことはめったにありませんが、脳の仕組みは大昔とまったく変わっておらず、トラの出現を今でも警戒しています。ただ、誰もそれを意識していないだけです。

逆に、何も変わったところのない、決まりきったことばかりが連続していると脳はどうなるでしょうか。その場合、脳は、目の前で起きていることを「重要」とはみなさず、記憶しようとはしません。脳は退屈なことは覚えないのです。脳には重要なこととそうでないことを振り分けるフィルタのようなものがあって、当たり前のできごとはそのフィルタを通過できないようになっているのです。

脳が重要とみなすのは、山へハイキングに出掛けた時、突然にトラが襲いかかってきた、というようなできごとです。

そんな時、脳の神経細胞や化学物質は活性化し、感情はたかぶります。

そして脳はこう判断するのです。

これは重要だから絶対に忘れるな！

しかし、家や図書館といった、トラも出てこない、安全で暖かい場所でテスト勉強などをしていたとしたらどうでしょうか。あるいは、上司に1週間～10日間かけてよいから読めと言われた難しい技術文書に目を通していたとしたら？

こんな場合、脳は目の前のできごとをまったく重要とはみなしません。これは脳の持ち主への裏切りのようですが、あくまで持ち主の生存のためです。勉強したこと、読んだことは、トラに襲われたり、火が迫ってきたことのようには記憶されないのです。

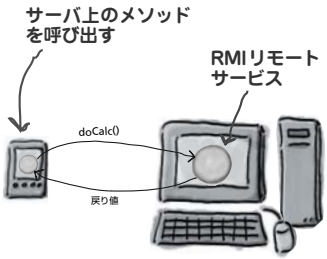
我が身を危険にさらすことなく、感情をたかぶらせることなく、脳に何かを覚えさせるというのは簡単なことではありません。どれほど「勉強したい、覚えたい」と望んだとしても、退屈した脳は何も覚えてくれません。

This PDF is prepared for tools@maruhouse.org, personal use only.

単に読むだけでなく「学習」できる本

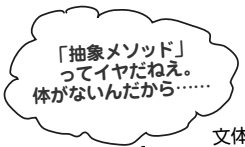
この本では、読者がただ読むのではなく、Javaを学習し、十分な技術を身に付けることができるよう数々の工夫をしています。最も気を配ったのは、「一度学習したことを忘れないようにする」ということです。単に事実を羅列しているわけではありません。文字以外に絵や写真などを数多く盛り込んだのもそうした工夫の一つです。これは、認知科学、神経生物学、教育心理学の最新の研究成果を踏まえたものです。簡単に言えば、脳を退屈させないようにしているわけです。

この本の主な特徴を以下にまとめておきましょう。



ビジュアルの多用

視覚情報は、文字よりもはるかに記憶されやすいので、学習効果を高めるのに役立ちます（最大89%学習効果が向上するというデータもある）。もちろん、内容をわかりやすくできる、というメリットもあります。本書では、絵や図、写真などを本文の中、あるいはすぐそばに載せているので、ページの下や本文と別のページにまとめて載せてあるよりもはるかに読みやすいはずです。



読者一人一人に語りかけるように書かれている

最近の研究では、読者に語りかけるような文章を読んだ後、形式張った文章を読んだ後でそれぞれ内容の理解度をテストすると、最大で40%もの違いが出るという結果も得られています。本書は、できる限り平易な言葉を使い、説明するというよりは気軽に話をするという文体で書きました。あまり肩肘を張らずに楽な気持ちで読んでください。真面目な講義を聞くよりも、楽しいパーティでの会話の方が一生懸命になれるという人は多いでしょう。

```
public abstract void eat ();
```

「本体」のコードはない
最後にはセミコロンを付ける

考えながら読むことができる

自らの頭を使って考え、脳細胞を活性化させなければ、学習効果はなかなかあがりません。受け身の姿勢でただ読んでいても、やる気も集中力も持続しないでしょう。本書は、次に学ぶ内容を読者に推測するよう促したり、練習問題を数多く盛り込むなど、自ら考え、答えを出す、という姿勢で学習ができるように作ってあります。

読み手の注意を引き、注意をそらさない

「どうしても勉強のためにこの本を読まねばならない。しかし、1ページも読まないうちにどうしても眠ってしまう……」という体験は誰にでもあるでしょう。すでに述べたとおり、人間の脳は、普通でないもの、新奇なもの、面白いもの、目立つもの、予想外のものなどに注意を向けるようにできています。これは、専門的で難しいことがらを学ぶことが必ずしも退屈とは限らない、ということを意味します。もし退屈でなければ、脳はたとえ難しいことでもすぐに理解してしまうでしょう。

感情に訴えるように書かれている

人間が何かを記憶するかどうかは、かなりの部分、感情の動きに左右されます。心の動いたこと、何かを感じたことは記憶しやすいのです。心が動くのは、何も「涙と感動のストーリー」のようなもの読んだ時だけとは限りません。何かに驚いたり、興味を持ったり、楽しいと思ったりすることも含まれます。変なものを見て「何だこれは？」と思ったり、練習問題が解けて「やったぞ！」と思ったりした時、誰もが難しいと思うようなことをすっきりと理解できた時などにも心は動きます。プログラムの得意な友人にもわからないようなことを理解できた……などと思った時にも、大いに心は動くでしょう。

This PDF is prepared for tools@maruhouse.org, personal use only.

自分の思考について考える

自分は物事をどのように考えるのか、また、物事をどのように学ぶのか、ということを知れば、何かを学習する時に役立ちます。自分の思考についてよく知っていれば、同じことをより早く、深く学べるのです。

自分の思考について知ることを「メタ認知」と呼びますが、学校でメタ認知について学ぶ人はほとんどいません。勉強をしろとは言われても、勉強の仕方を教わることはまずないのです。

この本を手に取った人はおそらくJavaを本気で勉強したい人でしょう。しかし、勉強に長い時間をかけたいという人はいないはずです。

この本（あるいは勉強のために読むすべての本）を最大限に活かすには、自分の脳に注目する必要があります。どうすれば脳は学ぼうとするかを知らねばならないということです。

脳は「生存、安全に大きな影響を与える」と判断したものに対しては注意を向け、それについて学ぼうとします。つまり、脳にそう判断させる方法を知っていれば、効率的に学習ができるのです。学習する内容を不意に現れたトラのようなもの、戦場にいる敵のようなものと思わせることができればいいのです。

どうすれば脳はJavaに注意を向けるのか？

方法は2つあります。1つは退屈で時間のかかる方法、もう1つは効率的で時間のかからない方法です。「退屈で時間のかかる方法」とは、「同じことをひたすら反復する」というものです。どれほどつまらなく、関心のないことでも、何度も繰り返し学べば覚えられる、というのは誰しも経験していることでしょう。十分に繰り返せば、脳は「これは重要とは思えないのだが、これだけ繰り返しているのだからおそらく重要なのだろう」と判断するのです。

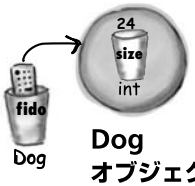
「効率的で時間のかからない方法」とは、「脳のはたらきを活性化する」というものです。特に、異なった種類のはたらきを同時に活性化すると効果が高くなります。前のページで触れた「ビジュアルの多用」や、「語りかけるような文体」は、脳のはたらきの活性化には大いに役立ちます。文章の中に、その内容を説明するような絵や写真を入れておくと、絵や写真と文章との関係を知ろうとして脳が活発にはたらく、という研究結果もいくつか報告されています（絵や写真を関連する文章から離してしまうと効果が薄れる）。脳が活発にはたらくというのは、それだけ注意が向けられるということでもあります。注意が向けられれば、当然、理解が深まり、よく記憶されることになります。

「語りかけるような文体」が効果的なのは、自分が会話の中に入っていると認識すると、注意を向けるという性質が脳にあるためです。最後まで会話に遅れずについていこうとするため、注意が持続するのです。面白いのは、会話が人間対人間のものである必要はない、という点です。人間対本であっても、会話になっていればいいのです。逆に、文章が形式張った、無味乾燥なものであれば、大勢に混じって椅子に座り、受け身の姿勢で講義を受けているのと同じような状態になります。こういう時、脳は眠ってしまうだけです。

しかし、もちろん、ビジュアルを多用したり、語りかけるような文体で書くということだけが脳を活性化するのに役立つわけではありません。

This PDF is prepared for tools@maruhouse.org, personal use only.

この本で著者が工夫したこと



すでに述べたとおり、この本では絵や写真を多用しています。これは、人間の脳が、文字よりも絵や写真などをよく認識できるように作られているからです。絵や写真から得られる情報は、文字から得られる情報の 1000 倍になると言っても決して大げさではありません。絵や写真を使う時は、関係する文章の中に埋め込むようなかたちで入れました。その方が、文章から離れた場所に載せるより効果的だからです。

また、同じことを繰り返し説明する、という工夫もしました。ただ文章で繰り返すのではなく、文章で書いたら次は絵や写真、グラフ、表を使って説明する、といった具合です。脳のさまざまな領域に学んだ内容が記憶されるということを狙っているのです。

絵や写真、その他、さまざまな工夫を盛り込むにあたっては、読者の予想を裏切るものになるよう心がけました。脳には、予想していなかった新奇なものに注意を向けるという特性があるからです。また、ちょっとしたユーモアを交えるなど、常に少しでも感情に訴えるように、ということも考えました。脳は感情を動かしたものには関心を持つからです。

すでに述べたとおり、文章は読者個人に話しかけるような文体で書きました。受け身の姿勢で講義を受けるより、会話をしている時の方が、脳は集中できるからです。たとえ、本の中の文章であっても、脳が「会話に参加している」と感じれば、学習効果は高まります。

50 を超える数の練習問題も学習効果を高めるのに役立つでしょう。脳は、ただ読んだだけのことより、自分でやってみたことの方をよく記憶するからです。練習問題のレベルは、「難しいがまったく手が出ないということはない」というくらいにしてあります。そのくらいのレベルの問題を「楽しい」と感じる人が多いからです。

同じことを複数のアプローチで学べるように、という配慮もしています。基礎から順を追って学習する方がいい、という人もいれば、ともかくまずだいたいの要点をつかみたいという人、コード例が見られればそれでいいという人もいるからです。好みのアプローチはそれぞれに異なっていても、本書なら皆満足できるはずです。

左脳と右脳の両方を使えるような工夫もしました。脳を使えば使うほど、学習内容がよく記憶されるからです。また、左脳を使った後に右脳を使う、というようなことができれば、集中力が長く持続できます。一方の脳を使っている間、もう一方の脳は休むことができるからです。集中力が持続できれば、当然、その分学習効果も上がります。

練習問題は、学んだことを違った角度から見られるようなものにしています。その方が脳が活性化され、学習内容をより深く理解できるからです。

学習にはもちろん努力が必要ですが、本書では無駄な努力をしなくて済むよう工夫しています。学習内容との関係がよくわからない難解な例をあげたり、無意味に専門用語をちりばめた説明をしたり、あまりに当たり前のことを書いて無意味に説明文を長くしたり、といったことはしていません。

本書は「80/20 アプローチ」を採用した本です。今後、Java のエキスパートになりたい、という人は他にも何冊かの本を読まねばならないでしょう。Java についてのあらゆることが書かれているわけではないからです。本書に書かれているのは、Java プログラマがプログラミングの実作業を行う際に知っていなければならないということがらばかりです。

This PDF is prepared for tools@maruhouse.org, personal use only.

この本を読む時に気を付けること

著者はこの本が良いものになるようできる限り努力したつもりです。しかし、この本が実際に成果をあげるためには読者の皆さんの協力も必要です。ここでは、どのような点に気を付けて読めば、学習効果を高めることができるかを簡単にまとめておきます。もちろん、ここに書いたことがすべてではありません。どのような読み方が合うかは人によって違うからです。色々と試して自分に合った読み方を見つけてください。

ここから下を切り取って、冷蔵庫にでも貼っておくといいでしょう

① ゆっくり読む。書いてあることを覚えようとするのではなく、深く理解しようとして読む。

ただ読み流すのではなく、時々立ち止まって考えるようにしてください。練習問題なども、すぐに答えを見ず、よく考えてから自分で答えを出してください。目の前にいる人に何か質問をされてそれに答えるようなつもりで考えるといいでしょう。そのように脳を使って深く考えれば考えるほど、内容をよく覚えることができ、学習効果が高まるものです。

② 練習問題には「手」を使って取り組む。

問題の答えは本の中に書いてあります。頭の中で答えを考えて、それが書いてあるものと合っているかどうか確かめるだけでもいいような気がします。しかし、やはり自分で鉛筆を持ち、手を動かして練習問題に取り組んだ方がいいでしょう。体を動かした場合の方が学習の効果が高まるということは、さまざまな研究でも明らかになっていることです。

③ Q&Aのコーナー（「素朴な疑問に答えます」というタイトルになっている）もじっくり読む。

本書にはQ&Aのコーナーが数多くあります。これは単に本文を補足するというよりも、むしろ本文の一部であると考えた方がいいでしょう。時には答えより質問の方が重要であるということもあるので注意してください。

④ 一か所にじっと座って読み続けるということをしない。

途中で立ち上がったり、ストレッチをしたり、座る椅子や読む部屋を変えたりするのもいいでしょう。それによって脳が少しリフレッシュされ、集中力が長く持続します。

⑤ 本書を読んだ後は寝るまで他に何も（少なくとも難しい本は）読まないようにする。

学習は本を閉じたらそれで終わりというものではありません。脳は、学習したことを短期記憶から長期記憶に転送し、確実に自分のものにするために少し時間を必要とするのです。この時間に新たに別のことを学習すると、前に学習したことが（少なくとも一部）消去されてしまう恐れがあります。

⑥ 水をたくさん飲む。

脳がはたらくには水分が必要です。脱水状態になると認知能力は衰えます（脱水状態になってから、のどが渇いたと感じるまでには少し時間がかかります）。

⑦ 大きな声を出す。

声を出して話す場合と、本を読む場合とでは、脳の中の違う部分を使うことになります。この本で読んだことを深く理解し、十分に記憶に留めるためには、読んだ内容を後で声に出して言ってみるといいでしょう。さらにいいのは、読んだ内容を（できるだけ大きな声で）他の人に説明してみることです。そうすれば、学習効果が高まるだけでなく、まだ十分に理解できないところなども明らかになります。

⑧ 脳が疲れたら休む。

学習中は脳に過度の負担をかけ過ぎないように注意しましょう。内容を表面的にしか理解できなくなったり、読んだばかりのことを忘れるようになったら脳が疲れてきた証拠です。あまり根を詰めても学習効果はあがらず、かえって時間がかかることになる可能性もあります。

⑨ 自分なりの感想を持つ。

人間の脳は何かを感じた時に物事をよく記憶するようにできています。この本を読む時はできるだけ感情移入をし、世界に入り込むようにしてください。そうすれば何かを感じる機会が多くなると思います。写真や絵などに自分なりの注釈を入れるというのもいい方法です。くだらないジョークがあれば、それに文句を言うだけでも、何も感じないよりはいいでしょう。

⑩ 自分でコードを入力し、実行してみる。

本の中には数多くのプログラム例が紹介されています。すべて自分でコードを入力し、実行してみてください。また、変更や改良を加えてみたり、いくつかに分解してみたりするのも勉強になるでしょう（特に、分解すると、プログラムの各部分の動きがよくわかる）。

This PDF is prepared for tools@maruhouse.org, personal use only.

用意するもの

この本を読むには、コンピュータと、Javaプログラムを作り、動かすための環境が必要になります。

「Javaプログラムを作り、動かすための環境」というのは、ここではとりあえず、テキストエディタと、後述のJava 2 SDK(Software Development Kit)を指します。その他、IDE(Integrated Development Environment)と呼ばれる開発ツールなどもありますが、今のところは導入を考えなくていいでしょう。IDEの導入は、本書での学習が終わってから(少なくとも16章までの学習が終わってから)考えても遅くありません。IDEを使うと、プログラミングの作業のかなりの部分が簡略化され、確かに楽なのですが、そのためにJavaについて本来学ぶべきことがらの多くを学べなくなってしまう恐れがあります。

Java 2 SDKのセットアップ

- 本書を読むためには、バージョン1.5以降のJava 2 Standard Edition SDK(Software Development Kit)を入手する必要があります。Linux版、Windows版、Solaris版はjava.sun.com(Sunが運営するJavaデベロッパのためのWebサイト)から無料で入手できます。ダウンロードページには、メインページから2クリック程度で到達できるはずです。その時点での最新バージョン(ただしベータ版でないもの)を入手するようにしてください。SDKには、Javaプログラムをコンパイル、実行するのに必要なものがすべて用意されています。Mac OS Xを使用している読者は、Java SDKがはじめから用意されているので、ダウンロードなどの作業は必要ありません。OS Xより前のバージョンのMac OSを使用していて、Javaのバージョンが古いという場合も、本書のコードの95%は正しく動作するはずです。
  注意:本書の内容はJava 1.5対応になっていますが、Sunは、おそらくマーケティング上の理由により、リリース直前にJava 1.5を"Java 5.0"という名前にしてしまいました。ただ、"1.5"も開発キットのバージョン番号としては残っています。従って、Java 1.5、Java 5、Java 5.0、Tiger(Java 5.0のコードネーム)はすべて同じものと考えていいでしょう。Java 3.0や4.0は存在せず、Java 1.4から5.0に番号が飛んでいることになります。Java 5.0ではなく、Java 1.5と呼んでいる人も珍しくはありません。このあたりのことに関しては、あまり深く考えなくてかまいません(面白いのは、Java 5.0とMac OS X 10.4のコードネームがどちらも同じ"Tiger"だということです。Mac上でJava 5.0を機能させるには、Mac OS X 10.4が必要になりますが、これはつまり、「TigerにはTiger」が必要ということです)。

- SDKにはAPIのドキュメントが含まれていないので、これもjava.sun.comからSDKとは別に入手する必要があります。ダウンロードせずにオンラインでアクセスすることもできますが、やはりダウンロードしてしまった方が面倒は少ないでしょう。

- テキストエディタはほとんどどのようなものでも使用できます。vi、emacs、picoなどの他、メモ帳、ワードパッド、TextEditなどOSに付属しているGUIを備えたテキストエディタも使用可能です。ただ、テキストエディタの中には、ファイル名の末尾に「.txt」という拡張子を自動的に付加してしまうものもあるのでその点には注意が必要です。

- SDKをダウンロード、インストール(その手順はSDKのバージョン、使用するOSなどによって異なる)したら、環境変数PATHの値として、Javaディレクトリ(具体的にどのディレクトリがJavaディレクトリになるかはSDKのバージョンによって異なる)の下の/binディレクトリを指定する必要があります。たとえば、Javaディレクトリが"j2sdk1.4.1"という名前のディレクトリであれば、j2sdk1.4.1/binディレクトリを指定するわけです。/binディレクトリには、SDKに用意されている各種ソフトウェアの実行ファイルが納められます。PATH変数の値の指定が適切でなければ、以下のようなコマンドは正しく実行できません。

  ```
  % javac
  ```

  これはJavaコンパイラを使用するためのコマンドですが、PATH変数を適切に設定しなければ、システムはコンパイラの実行ファイルを見つけることができないのです。

本書のコードの大半はwickedlysmart.comで入手できます。

This PDF is prepared for tools@maruhouse.org, personal use only.

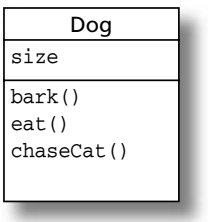
その他の注意事項

この本は順を追って読みながら学習を進めていくという本なので、辞書などのように参考資料として使用するための本とは違います。したがって、「何でも載っている」というわけにはいきません。この本で取りあげるとかえって学習の妨げになる恐れのあるような事項についてはあえて割愛していることもあります。また、注意して欲しいのは、この本を初めて読む場合には、最初のページから順に飛ばさずに読むべき、ということです。前のページで学んだことを踏まえて次のことがらを説明する、という構成になっているからです。

UMLは使っていない

この本では、便宜上、UMLの図に似たものは使っていますが「本物」のUMLは使いません。UMLには一部、Javaには合わない部分があるためです。この本で使っているのは、Javaに合わない部分を排除した、いわば「簡略版UML」とでもいうべきものです。この本でJavaとUMLを同時に学ぶようなことは必要ないので、安心してください。

UMLのようだが、
実は簡略版



プログラムのパッケージング、提供などに関することは後で学ぶ

この本の目的はまず、Javaの基本を学び、プログラムを書けるようになることです。書き終わったプログラムをパッケージングし、ユーザに提供するといったことについては、初めの段階では取りあえず知る必要がないので、かなり後になってから学ぶようになっています（17章で学びます）。後に回してはいるものの、かなり詳しい説明をしているので、本を最後まで読めばパッケージングや提供についても十分な知識が付くはずです。

各章の最後には練習問題（とその解答）が用意されている

各章の最後には、さまざまな練習問題を用意しています。この本では、練習問題を2種類に分けています。1つはスニーカーのマークのついた問題です。これは、その章で学んだことの理解度を試すための問題なので必ず取り組んでみてください。もう1つは、パズルのピースのマークがついた問題です。これはいわば頭の体操のための問題なので、興味と意欲のある人だけが取り組めばいいでしょう。中にはかなり難しいものも含まれています。後者の問題に関しては、解けなくてもそれほどがっかりすることはありません。

「自分で考えてみよう」コーナーの問題にはすべて自分で答えを出す練習をしてみよう

自分で考えてみよう

スニーカーのマークの練習問題はその章で学んだことの理解度を試すためのもの。すべて飛ばさずに取り組むべき

エクササイズ

パズルのピースのマークがついた問題は、興味と意欲のある人だけ取り組めばよい。解けなくても特に気にする必要なし

「自分で考えてみよう」のコーナーには解答がない

章末の練習問題の他にも、この本の中には「自分で考えてみよう」と題された問題が数多く用意されています。ただ、この問題には解答がありません。中には、本の中に解答が書かれていないだけでなく、「唯一の正解」というものがない問題も存在します。自分なりの正解にたどり着けるよう頑張ってみてください。

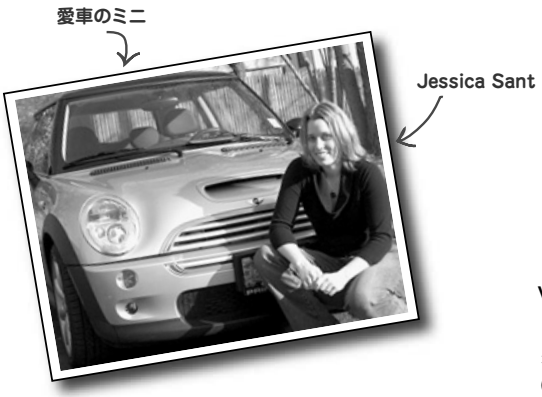
コードサンプルはできる限り簡単なものに

200行もあるような長いコード例を示されて、実は本当に理解しなければならないコードはたったの2行だけだった、というようなことがあると非常にストレスがたまります。この本では、理解すべきポイントがすぐに明確にわかるよう、コード例はなるべくシンプルなものにしています。ただ、それは逆に言えば、そのまま即、実践に使えるような完成度の高いものにはなっていないということでもあります。完成度を上げることは、本書を読み終わってからの読者の皆さんの課題としてください。本の中のコードサンプルはあくまで学習のためのもので、実用のためのものではないのです。

This PDF is prepared for tools@maruhouse.org, personal use only.

テクニカルエディタの紹介

本の製作に関わった人たちを紹介する場合、著者は本来、「この本に良いところがあれば彼らのおかげ、悪いところがあれば著者のせい」などと言うのでしょうが、それを真に受ける読者はいないでしょう。本書に何か技術的な問題があれば、それはおそらく、下で紹介する2人のテクニカルエディタの責任です。



Jessica Sant

Hewlett-Packard勤務（自己修復サービスチームに所属）。Villanova大学でコンピュータ工学を学び、SCJP(Sun Certified Programmer for the Java 2 Platform) 1.4 とSCWCD（Sun Certified Web Component Developer）の資格を取得。あと数ヶ月でDrexel大学にてソフトウェア工学修士号を取得予定（すごい！）。

休日は勉強にあてるか、愛車ミニクーパーSで外出をする。編み物にも凝っていて、ネコと毛糸を取り合う姿を時々見ることができる（毛糸の帽子なんかが欲しい人は頼んでみては？）。元々はUtah州Salt Lake City出身（と聞くとすぐ「モルモン教徒？」と思うが、彼女はそうではないらしい）だが、今はPhiladelphiaで夫のMendra氏と2匹のネコ（名前はChaiとSake）とともに暮らしている。

その他、javaranch.comというWebサイトのいくつかの技術フォーラムの運営にも携わっている。

Valentin Crettaz

スイスのLausanne工科大学（EPFL）でコンピュータサイエンス修士号を取得。SRI International（California州Menlo Park）のソフトウェアエンジニアとなり、EPFLのソフトウェア工学研究所の首席エンジニアともなった。

ソフトウェアアーキテクチャソリューションの開発を専門とする企業、Condris Technologiesの共同設立者、CTOでもある。

アスペクト指向の技術、デザイン、Webサービス、ソフトウェアアーキテクチャに強い関心を寄せている。現在は妻と二人暮らしで、趣味はガーデニング、読書、スポーツを少々。javaranch.comサイトでは、SCBCD、SCDJWSのフォーラムの運営に携わっている。彼自身は、SCPJ（Sun Certified Programmer for the Java 2 Platform）、SCDJ（Sun Certified Developer for the Java 2 Platform）、SCBCD（Sun Certified Business Component Developer）、SCWCD（Sun Certified Web Component Developer）、SCDJWS（Sun Certified Developer for Java Web Services）などの資格を持つ。Whizlabs SCBCD Exam Simulatorの作者の一人でもある。
（彼のネクタイ姿は貴重）

This PDF is prepared for tools@maruhouse.org, personal use only.

謝辞

O'Reilly のスタッフ

まず、誰より感謝しなくてはならないのは、Mike Loukides氏です。そもそもこの本の企画の可能性を認め、かたちにした（そしてシリーズ化まで実現した）のは彼だからです。この『Head First Java 第 2 版』でHead Firstシリーズも 5 冊目ですが、彼はすべてに関わっています。もちろん、Tim O'Reilly氏の功績についても触れないわけにはいきません。O'Reilly氏は、会社としてはまったく前例のない種類の本の出版を快く許可してくれました。Kyle Hart氏にも非常に感謝しています。Hart氏は、どうすれば本書が世に受け入れられるものになるか、ということについて素晴らしいアイデアを出してくれました。また、本書のシリーズ化に尽力してくれたのもHart氏です。最後に、本書の内容に完璧にマッチした素晴らしいカバー（「頭」が強調されている）をデザインしてくれた、Edie Freedman氏にも感謝の言葉を述べておきます。

レビューをお願いした人

Javaranchテックレビューチームのディレクター、Johannes de Jong氏には大変お世話になりました。Head Firstシリーズの本はこれで 5 冊目ですが、1 冊目からずっと関わっていただけて本当に嬉しく思っています。Jeff Cumps氏は、これで 3 冊目のHead Firstシリーズですが、自身の専門分野に関しては情け容赦のないコメントをくれ、著者は不勉強を思い知らされました。

Corey McGlone氏もすごい人です。javaranchサイトでいつも最も明解な説明をしているのが彼です。本書でも彼の見事な説明をいくつか拝借させてもらいました。Jason Menard氏にもこまごまとしたポイントに留まらない貴重なアドバイスをいくつももらいました。Thomas Paul氏のフィードバックにも助けられました。他の誰もが気付かないJavaの微妙な問題点を指摘してくれたのも彼でした。Javaの知識に加え、文章についても一家言あるJane Griscti氏も、第 2 版の作成に関して、javaranchの長年のメンバーであるBarry Gaunt氏とともに力になってくれました。

Marilyn de Queiroz氏は、第 1 版、第 2 版ともに、大いに助けになってくれました。第 1 版に協力してくれたChris Jones、John Nyquist、James Cubeta、Terri Cubeta、Ira Becker氏にも感謝しています。

Head Firstシリーズ開始当初から協力してくれている人としては、他にAngelo Celeste、Mikalai Zaikin、Thomas Duff（twduff.com）氏などがいます。彼らにも感謝しなくてはいけません。それから、我々の素晴らしいエージェント、StudioBのDavid Rogelbergにも感謝しなくてはいけないでしょう（そういえば、Head Firstシリーズの映画化権の話はどうなったのでしょうか）。

お世話になった人たち

This PDF is prepared for tools@maruhouse.org, personal use only.

その他にも感謝しなくてはならない人はたくさんいます*。

第 1 版作成にあたり技術点な面でサポートしていただいた人（順不同）

Emiko Hori 氏、Michael Taupitz 氏、Mike Gallihugh 氏、Manish Hatwalne 氏、James Chegwidden 氏、Shweta Mathur 氏、Mohamed Mazahim 氏、John Paverd 氏、Joseph Bih 氏、Skulrat Patanavanich 氏、Sunil Palicha 氏、Suddhasatwa Ghosh 氏、Ramki Srinivasan 氏、Alfred Raouf氏、Angelo Celeste氏、Mikalai Zaikin氏、John Zoetebier氏、Jim Pleger氏、Barry Gaunt氏、Mark Dielen氏。

第 1 版パズル作成チーム（順不同）

Dirk Schreckmann氏、Mary "JavaCross Champion" Leners氏、Rodney J. Woodruff氏、Gavin Bong氏、Jason Menard氏。

その他（順不同）

Paul Wheaton氏：javaranchスタッフ。数千人ものJava学習者をサポートした実績を持つ。
Solveig Haugland氏：J2EEのエキスパート。"Dating Design Patterns" の著者。
Dori Smith氏、Tom Negrino氏（backupbrain.com）：自身も多数の著者があることから、技術書の書き方について指導してくれた。
Eric Freeman氏、Beth Freeman氏（Head First シリーズのパートナーであり、"Head First Design Patterns" の共著者（共犯者）でもある）：第 2 版の作業が期日どおりに終わるよう、Bawls（米国のソフトドリンク。カフェインを多く含む）を差し入れてくれた。
Sherry Dorris：困った時に頼りになる存在。

Head First シリーズの（勇気ある）発表当初からの読者の方々（順不同）

Joe Litton 氏、Ross P. Goldberg 氏、Dominic Da Silva 氏、honestpuck 氏、Danny Bromberg 氏、Stephen Lepp 氏、Elton Hughes 氏、Eric Christensen 氏、Vulinh Nguyen 氏、Mark Rau 氏、Abdulhaf氏、Nathan Oliphant氏、Michael Bradly氏、Alex Darrow氏、Michael Fischer氏、Sarah Nottingham氏、Tim Allen氏、Bob Thomas氏、Mike Bibby氏（読者第 1 号）。

* 謝辞にたくさんの名前を載せるのには、名前が載った人は最低 1 冊は本を買ってくれる、という理由もあります。うまくすれば、載った人の家族、親戚や知人、友人まで本を買ってくれることがあります。読者の中に「次の本の謝辞には名前を載せて欲しい」という方がいれば連絡をください。特に家族の多い方は歓迎です。

This PDF is prepared for tools@maruhouse.org, personal use only.

1章　Javaの世界に飛び込もう

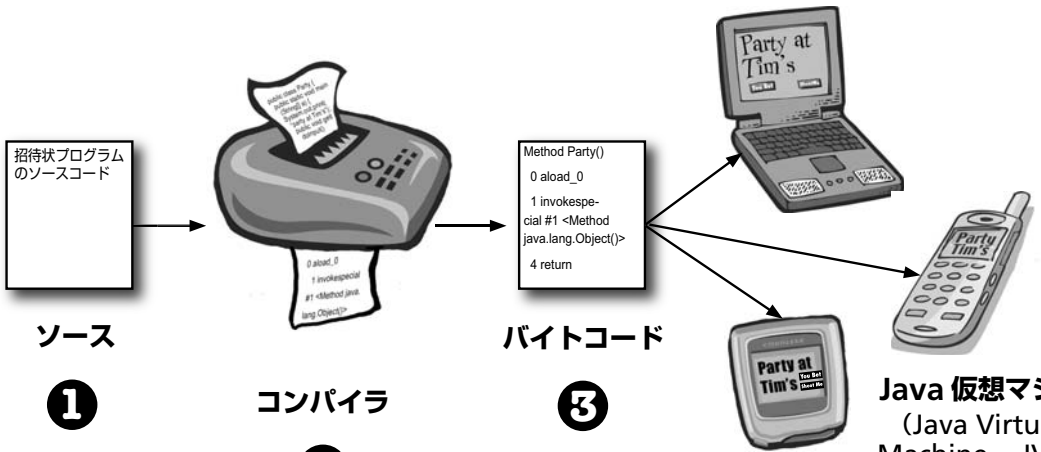
とにかくまずは始めてみる

Javaは独自の特徴を持つプログラミング言語です。そのため、発表以来（発表当時のバージョンは 1.02）、数多くのプログラマを惹きつけています。Javaには、文法がわかりやすい、メモリ管理が容易といった特徴もありますが、何と言っても重要なのは、その「移植性」です。Javaの移植性は “Write Once, Run Anywhere.” というキャッチフレーズによってよく知られています。当初は欠点も多く、急速に増えたJavaの支持者はその欠点との闘いを余儀なくされました（特にプログラムの実行速度の低さが問題になりました）が、今ではそれも昔の話です。これからJavaを始める人は恵まれていると言えるでしょう。先人たちが道を踏み固め、ならしてくれたおかげで、以前より少ない労力で、より優れた言語に成長したJavaを使えるからです。また、本書のような解説書が手に入るということも過去には考えられなかったことでしょう。

This PDF is prepared for tools@maruhouse.org, personal use only.

Javaプログラムのできるまで

Javaのプログラムは、簡単にまとめれば、以下の図のように作られ、実行されることになります(ここでは、パーティ招待状を表示するプログラムを例にとっています)。



ソース

❶

Javaプログラムははじめ、「ソースコード」という形式で作られます。

コンパイラ

❷

ソースコードは「コンパイラ」と呼ばれるソフトウェアによって翻訳(この翻訳を「コンパイル」という)されます。その際には、ソースコードに誤りがないかのチェックも行われます。誤りが見つかった場合は、コンパイルは中断されます。誤りが修正されない限り、コンパイルは完了しません。

バイトコード

❸

コンパイラがソースコードを翻訳することによって、「バイトコード」と呼ばれる形式のプログラムができます。コンピュータは、このバイトコードの内容を解釈し、プログラムを実行します。バイトコードの内容は、プログラムを実行するコンピュータが何であろうと同じです。

Java仮想マシン (Java Virtual Machine = JVM)

❹

バイトコードを解釈、実行するためには、各コンピュータに「Java仮想マシン」と呼ばれるソフトウェアが必要です。厳密には、このJava仮想マシンがバイトコードを解釈、実行することになります(Javaプログラムは、パソコンなどのいわゆる「コンピュータ」以外に、携帯電話、PDAなどといったコンピュータを内蔵した機器でも実行できます)。

This PDF is prepared for tools@maruhouse.org, personal use only.

Javaプログラム作成の手順

Javaプログラムの作成作業の手順を、作る人間の側から見れば、下の図のようになります。

```
import java.awt.*;
import java.awt.event.*;
class Party {
  public void buildInvite(){
    Frame f =new Frame();
    Label l =new Label("Party at Tim 's");
    Button b =new Button("You bet");
    Button c =new Button("Shoot me");
    Panel p =new Panel();
    p.add(l);
} //more code...
```

ソースコードの作成

❶

まずソースコードを作り、それを「～.java」という形式のファイルに保存します（ここでは、仮にファイル名を「Party.java」とします）。ソースコードを保存したファイルを「ソースファイル」と呼びます。

```
File  Edit  Window  Help  Plead
%javac Party.java
```

ソースファイルのコンパイル

❷

ソースファイル（ここではParty.java）をコンパイルします。コンパイルのためには、javacコマンド（コンパイラのコマンド）を実行します。

```
Method Party()
  0 aload_0
  1 invokespecial #1 <Method java.lang.Object()>
  4 return
Method void buildInvite()
  0 new #2 <Class java.awt.Frame>
  3 dup
  4 invokespecial #3 <Method java.awt.Frame()>
```

クラスファイル

❸

ソースコードに誤りがなければ、コンパイルによって、バイトコードを納めたファイル（クラスファイル）が自動的に作成されます。クラスファイルの名前は「ソースファイルの名前から ".java" を除いたもの.class」という形式になります。Party.javaをコンパイルした場合、クラスファイルは「Party.class」になるわけです。

プログラムの実行

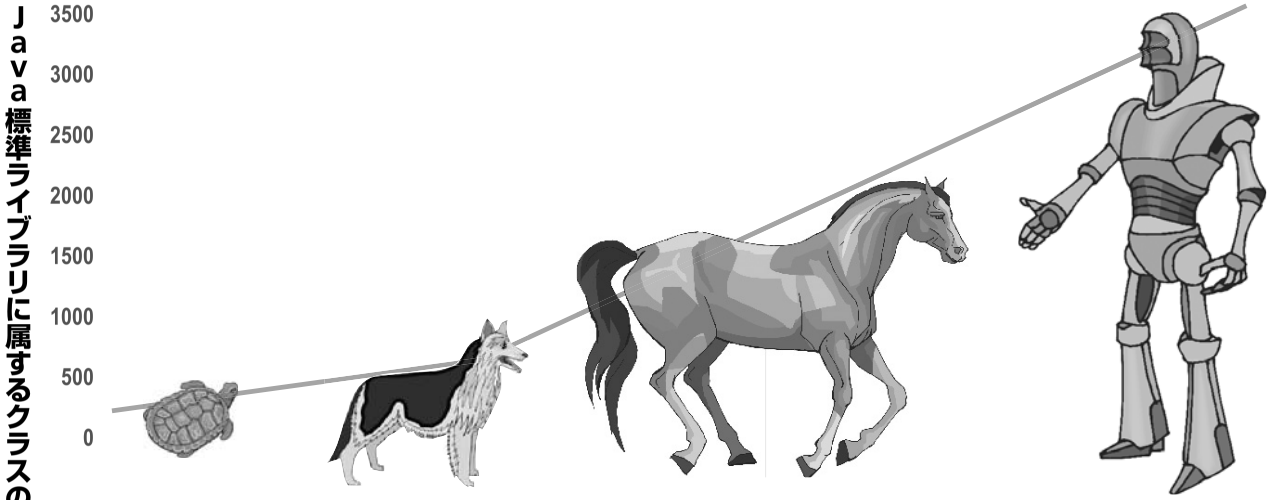
❹

クラスファイルができたら、Java仮想マシンを使ってプログラムを実行します。そのためには「java クラスファイルから ".class" を除いたもの」という形式のコマンドを使用します。クラスファイルが「Party.class」であればコマンドは「java Party」となるわけです。Java仮想マシンはクラスファイルのバイトコードの内容を解釈し、実行します。

（注意：上記の説明はおおまかなものに過ぎません。詳細な部分については実際にプログラミング作業を始めてから覚えるとして、ここではだいたいの感じをつかむようにしてください）

This PDF is prepared for tools@maruhouse.org, personal use only.

Javaの歴史（概略）



Java 1.02

クラス数：約 250
実行速度：低

親しみやすい名前とロゴ、使いやすさで注目を集める。バグ多し。用途は主にアプレット。

Java 1.1

クラス数：約 500
実行速度：やや改善

より高機能になり、さらに使いやすくなった。
1.02 の時点より広く普及。
GUI 関連のコードにも改善がみられる。

Java 2（バージョン 1.2 ～ 1.4 以降）

クラス数：約 2300
実行速度：大きく改善。他の言語とほぼ遜色なし。

さらに高機能になり、あらゆる用途に対応できるようになった。
Micro Edition（J2ME）、Standard Edition（J2SE）、Enterprise Edition（J2EE）という 3 つのタイプがある。
eビジネス、Webアプリケーション、モバイルアプリケーションにおいて主役の位置を占める言語となった。

Java5.0（バージョン 1.5 以降）

クラス数：約 3500
機能、開発効率ともに向上。

Java 5.0（Tiger）では、1000 を超えるクラスが追加されたことに加え、言語そのものにも大幅な改善が加えられた。その結果、以前よりプログラムの開発が（少なくとも理論上は）さらに容易になった。「他の言語では一般的だが Java にはなかった」という機能もいくつか追加された。

This PDF is prepared for tools@maruhouse.org, personal use only.

自分で考えてみよう

Javaプログラムを書くのは簡単

本文を読み、それぞれの行がどのような処理をするためのものか考えてみてください（答えは次のページ）。

```
int size = 27;
String name = "Fido";
Dog myDog = new Dog(name, size);
x = size - 5;
if (x < 15) myDog.bark(8);

while (x > 3) {
        myDog.play();
}

int[] numList = {2,4,6,8};
System.out.print("Hello");
System.out.print("Dog: " + name);
String num = "8";
int z = Integer.parseInt(num);

try {
        readTheFile("myFile.txt");
}
catch(FileNotFoundException ex) {
        System.out.print("File not found.");
}
```

int型の変数、sizeを宣言し、27を代入する。

Q：Java 2とJava 5.0はありますが、Java 3や4はあるのですか。また、なぜ、Java 5.0は"5.0"なのに、Java 2は"2.0"ではないのですか。

A：一言で言えば「マーケティングの都合」ということになるでしょうか。バージョン1.1から1.2に移行する際、Javaは大きく変化しました。本来は"Java 1.2"とそのまま呼ぶべきなのでしょうが、大きく変わったことが一目でわかる名前が必要だとマーケティングの担当者は判断したようです。その後のバージョン1.3、1.4はまだ"Java 2"の範疇だと考えられていたので、"Java 3"、"Java 4"などと呼ばれることはありませんでした。ところが、"Java 1.5"でJavaは再び大きく変化したため、これも"Java 1.5"ではない名前で呼ぶべきと判断されました（デベロッパも多くがそれに賛同しました）。"Java 2"の次ということで"Java 3"も名前の候補としてあがったのですが、混乱を招きやすいということで、"1.5"の"5"をとって"Java 5.0"と呼ぶことに決まりました。

オリジナルのJava（最初の公式リリース）は、バージョン1.02でした。その後、バージョン1.1までは単に"Java"と呼ばれ、バージョン1.2、1.3、1.4は"Java 2"、そしてバージョン1.5が"Java 5.0"というわけです。Java 5.0は、".0"を付けずに"Java 5"、あるいは"Tiger"とコードネームで呼ばれることもあります。次のリリースがどう呼ばれることになるのか、我々にはまったく予測も付きません……。

This PDF is prepared for tools@maruhouse.org, personal use only.

自分で考えてみよう 答え

Javaプログラムを書くのは簡単

わからないところがあってもまだ心配は要りません。
下記のコードは、本書を読み進めていけば理解できるようになるものばかりです（大半は40ページも進まない間に理解できるようになるでしょう）。Javaに似た言語を使用した経験のある人であれば、より簡単に理解できるようになるはずです。ですから心配は要りません。

`int size = 27;`	int型の変数、sizeを宣言し、27を代入する。
`String name = “Fido” ;`	"name"という名前の文字列を宣言し、値"Fido"を代入する。
`Dog myDog = new Dog(name, size);`	"myDog"という名前のDog型の変数を宣言し、"name"、"size"などの値を使用してDogオブジェクトを新規に作成する。
`x = size - 5;`	27（sizeの値）から5を引き、その結果を変数"x"に代入する。
`if (x < 15) myDog.bark(8);`	x（値「22」が代入されている）が15未満なら、Dogオブジェクトにbark()メソッドを8回実行するよう指示する。
`while (x > 3) {`	xが3より大きい間はループを継続する。
`myDog.play();`	Dogオブジェクトにplay()メソッドを実行するよう指示する（play()メソッドの機能がどのようなものであっても同じ）。
`}`	ループのコードはここまで。ループ継続中、{}の中のコードはすべて実行される。
`int[] numList = {2,4,6,8};`	intの配列"numList"を宣言し、値「2」、「4」、「6」、「8」を代入する。
`System.out.print( “Hello” );`	"Hello"と出力する（通常はコマンドラインに出力される）。
`System.out.print( “Dog: “ + name);`	コマンドラインに"Hello Fido"と出力する（nameの値が"Fido"になっている）。
`String num = “8” ;`	"num"という名前の文字列を宣言し、値"8"を代入する。
`int z = Integer.parseInt(num);`	文字列"8"を数値"8"に変換する。
`try {`	必ず成功するとは限らない処理の実行を試みる。
`readTheFile( “myFile.txt” );`	"myFile.txt"という名前のテキストファイルを読み込む（読み込みを試みる）。
`}`	試みる処理は、この}の前に記述する。複数の処理を試行することも可能。
`catch(FileNotFoundException ex) {`	試行した処理が失敗した場合に実行するコード。
`System.out.print( “File not found.” );`	試行した処理が失敗した場合は、コマンドラインに"File not found"と出力する。
`}`	{}内に記述されているコードはすべて、試行した処理が失敗した場合に実行する。

This PDF is prepared for tools@maruhouse.org, personal use only.

Javaプログラム（ソースファイル）の構造

**ソースファイルは
クラスから構成される。**

**クラスは
メソッドから構成される。**

**メソッドはステートメントから
構成される。**

ソースファイルの構成

ソースファイル（.javaという拡張子の付くファイル。「ソースコードファイル」ともいう）は、「クラス」と呼ばれる要素から構成されます。多くのプログラムは、複数のクラスから成りますが、ごく簡単なプログラムの場合はクラスが1つだけ、ということもあります。クラスの本体部分は右の図のように、{で始まり、}で終わります。

```
public class Dog {



}
```

クラス

クラスの構成

クラスは、「メソッド」と呼ばれる要素から構成されます。右の図のDogクラスには、**bark()**というメソッドが含まれています。メソッドは、属するクラスの始めの{と、終わりの}の間に置かれます。また、メソッドの本体部分も{で始まり、}で終わります。

```
public class Dog {
    void bark(){



    }
}
```

メソッド

メソッドの構成

メソッドの始めの{と、終わりの}の間には、「そのメソッドでどのような処理を行うか」の指示が書かれます。その指示は、「ステートメント」と呼ばれる単位で構成されます。

```
public class Dog {
    void bark() {
      statement1;
      statement2;
    }
}
```

ステートメント

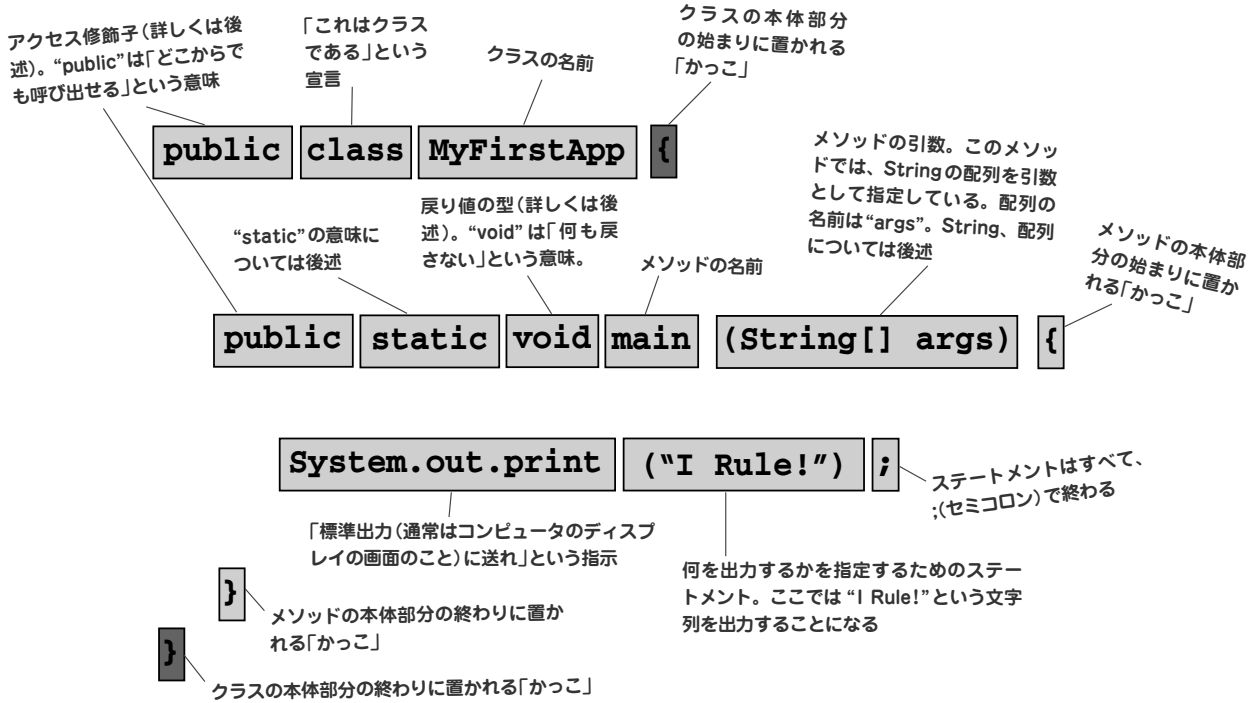
This PDF is prepared for tools@maruhouse.org, personal use only.

クラスの構造

Java仮想マシンは、プログラムの実行を開始するとまず、プログラムの中から、実行コマンドに指定されたものと同じ名前のクラス（コマンドが「java Party」であれば「Party」）を探します。そして、そのクラスの中から、以下のような、「main」という名前の特殊なメソッド（mainメソッドと呼ばれる）を探します。

```
public static void main (String [] args){
   // メインメソッドの本体
}
```

mainメソッドが見つかったら、その始めの { と最後の } の間のステートメントをすべて実行します。Javaのプログラムのうち、「アプリケーション」と呼ばれるものには、必ず、1つ以上のクラスと、1つのmainメソッドがあります（mainメソッドはクラスに1つではなく、アプリケーションに1つであるという点に注意）。



This PDF is prepared for tools@maruhouse.org, personal use only.

mainメソッドを持つクラスの作成

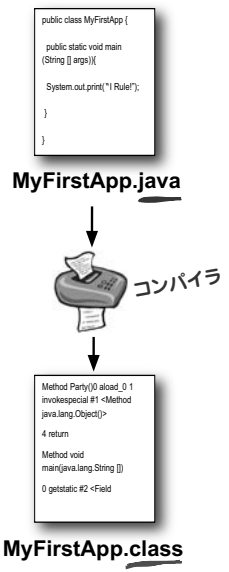
Javaプログラムの作成はすでに述べたとおり、まずソースファイル（.javaという拡張子を持つファイル）を作るところからはじまります。このファイルをコンパイルしてクラスファイル（.classという拡張子を持つファイル）を作るのです。プログラムの実行にはこのクラスファイルを使います。

プログラム実行時にJava仮想マシンは、クラスファイルと同じ名前（ただし.classはつかない）のクラスを探してそれをロード（メモリに読み込むこと）します。そして、そのクラスの中のmainメソッドを探し、中のステートメントをすべて実行します。

「クラス」というものについては、2章で詳しく解説します。ここではともかく、「実行できるJavaプログラムを作るには最低限何をしなければならないか」だけを覚えてください。まず大切なのはmainメソッドの書き方です。

mainメソッド（main()メソッドという表記もよく使用される）は、プログラム実行の起点となるメソッドです。

プログラムがいかに大きくても（いかに多くのクラスで構成されていても）、実行にはまずmainメソッドが必要である、という点は同じです。




```
public class MyFirstApp {

  public static void main (String [] args){
    System.out.println("I Rule!");
    System.out.println("The World");
  }

}
```

1 ソースファイルの作成
`MyFirstApp.java`

2 コンパイル
`javac MyFirstApp.java`

3 実行

```
File  Edit  Window  Help  Plead
%java MyFirstApp
I Rule!
The World
```


This PDF is prepared for tools@maruhouse.org, personal use only.

mainメソッドの「中身」

main メソッドに限らず、メソッドの「中身」を書く、という作業はプログラミングの中でも特に重要な位置を占めるものです。メソッドの中に書くプログラム（「コード」とも呼ばれる）は、いずれもコンピュータに何かを指示するためのものです。コンピュータへの指示の具体的な内容はJavaでも、他の言語でも基本的には違いはありません。

メソッドの中に書くコードにはおおまかに分けて次のような種類があります。

❶ 単純に「何かをせよ」と指示するもの

変数の宣言、変数への値の代入、他のメソッドの呼び出し、など

```
int x = 3;
String name = "Dirk";
x = x * 17;
System.out.print("x is "+ x);
double d = Math.random();
//コメント
```

❷ 「何かを繰り返し行え」と指示するもの

for文、while文を使ったループ

```
while (x > 12){
   x = x -1;
}
for (int x = 0; x < 10 x = x + 1) {
   System.out.print("x is now " + x);
}
```

❸ 「一定の条件が満たされた時に、ある処理を行え」と指示するもの

if文、else文を使った分岐

```
if (x == 10){
   System.out.print("x must be 10");
}else {
   System.out.print("x isn't 10");
}
if ((x < 3)&(name.equals("Dirk"))){
   System.out.println("Gently");
}
System.out.print("this line runs no matter what");
```

コードに関する注意事項

★コードを構成する各ステートメントの末尾には、;（セミコロン）を付ける。

```
x = x + 1 ;
```

★1行で終わるコメントは、二重スラッシュで始める。

```
x = 22;
// これはコメントです
```

★空白文字を入れてもコードの意味は通常変わらない。

```
x      =      3   ;
```

★変数の宣言の際は名前と型を書く（詳しくは3章を参照）。

```
int weight;
// 型:int, 名前:weight
```

★クラス、メソッドの本体部分は、{で始め、最後に}を付ける。

```
public void go(){
  //メソッドの本体
}
```


This PDF is prepared for tools@maruhouse.org, personal use only.

```
while (moreBalls ==true){
   keepJuggling();
}
```

ループ

Javaでは、while文、do-while文、for文という3つの文を使ったループを作ることができます。ループに関しては後で詳しく説明します。ここでは、while文に関してだけ簡単に触れておきましょう。

while文はさほど難しいものではありません。「一定の条件が満たされている間、同じ処理を繰り返せ」と指示するだけのものです。繰り返し行うべき処理を書く際は、{で始め、最後には}を付けます。

while文で大切なのは、条件が満たされているかどうかの確認（条件判定）です。Javaでは、その確認に、「boolean型」と呼ばれる型の変数を使用します。boolean型の変数というのは、値が必ず「true（真）」か「false（偽）」になる変数のことです。trueは、ある条件が満たされていることを、falseは条件が満たされていないことを表します。

たとえば、「器にアイスクリームが残っている間、スプーンですくい続けよ」と指示する場合は、「器にアイスクリームが残っている」という条件が満たされているかどうかの確認をすることになります。残っていることはtrue、残っていないことはfalseで表されます。

条件判定

条件判定のためには、次のような比較演算子を使用して変数の値を確認する、という方法がよく使われます。

< （小なり）
> （大なり）
== （等しい。等号が2つである点に注意）

「等しい」を意味する比較演算子は「==」のように等号を2つ重ねたものになります。等号が1つだと、変数に値を代入する際に使用する「代入演算子」になってしまいます。間違えやすいので、注意が必要です。

```
int x = 4; //変数xに4を代入
while (x > 3) {
  // xが3より大きければ
  // 以下の処理を繰り返す
  x = x + 1; //つまりこの場合永久に繰り返すことになる
}
int z = 27;
while (z == 17){
  // zが17になることはないので
  // 何の処理も行われない
}
```


This PDF is prepared for tools@maruhouse.org, personal use only.

素朴な疑問に答えます

Q: Javaのプログラムがクラスで構成されるのはなぜですか。

A: それは、Javaが「オブジェクト指向言語」と呼ばれるプログラミング言語だからです。「クラス」はオブジェクト指向言語の主要な概念の1つです。「クラス」は「オブジェクト」と呼ばれることもあります。オブジェクト指向言語には、オブジェクト指向でない言語(「手続き指向言語」などと呼ばれる)にはない様々な特徴があります。クラス(オブジェクト)については、2章で詳しく触れます。

Q: mainメソッドはすべてのクラスに必要なのですか。

A: いいえ、そんなことはありません。1つのJavaプログラムが多数(数百ということもある)のクラスから構成されるということはよくありますが、mainメソッドを必要とするのはそのうちの1つだけです。プログラムの実行は、その、mainメソッドを含むクラスから開始されます。ただし、テストの目的で仮に複数のクラスにmainメソッドを持たせるということはあり得ます。

Q: 他のプログラミング言語では、以下のようにint型の値を使った条件判定を行うことができます。Javaではできないのですか。

```
int x = 1;
while (x) {}
```

A: できません。Javaの場合、int型とboolean型に互換性がないので、条件判定に直接使用できるのはboolean型の変数のみです(int型の値を比較演算子で確認する式を条件判定に使用することは可能)。条件判定のコードはたとえば以下のように書きます。

```
boolean isHot = true;
while(isHot){}
```

while文を使ったループの例

```
public class Loopy {
  public static void main (String [] args){
    int x = 1;
    System.out.println("Before the Loop");
    while (x < 4){
      System.out.println("In the loop");
      System.out.println("Value of x is " + x);
      x = x + 1;
      }
      System.out.println("This is after the loop");
    }
}
```

上記のコードの実行結果

```
% java Loopy
Before the Loop
In the loop
Value of x is 1
In the loop
Value of x is 2
In the loop
Value of x is 3
This is after the loop
```

重要ポイント

- ステートメントの末尾には ;(セミコロン)を付ける。
- クラス、メソッドのコード、ループで繰り返し行う処理などは、{} で囲む。
- 変数は型と名前を書いて宣言する。
- 等号(=)が1つだと代入演算子になる。
- 「等しい」という意味の比較演算子は等号(=)を2つ重ねる。
- while文は、一定の条件が満たされている間、同じ処理を繰り返す、というループを作るのに使う。
- while文では、条件が満たされなければ、{} 内の処理はまったく行われない。
- 条件判定のための式は、以下の例のように () の中に書く。
  while (x == 4){}

This PDF is prepared for tools@maruhouse.org, personal use only.

分岐

if文などを使った分岐でも条件判定を行いますが、この条件判定は、Javaの場合、すでに触れたwhile文のものと基本的には同じです。ただ、while文の場合は条件が満たされている間、処理が繰り返されるのですが、if文の場合は条件が満たされている場合に所定の処理が一度行われるだけです。if文の例を以下に示します。

```
class IfTest {
  public static void main (String [] args){
    int x = 3;
    if (x == 3){
      System.out.println("x must be 3");
    }
    System.out.println("This runs no matter what");
  }
}
```

```
% java IfTest
x must be 3
This runs no matter what
```

実行結果

上記のコードの場合、"x must be 3"という文字列を出力する部分は、「xが3である」という条件が満たされた場合（x == 3という式によって値「true」が得られた場合）にのみ実行されます。また、"This runs no matter what"という文字列を出力する部分は、無条件に実行されます。つまり、xの値によって、両方の文字列が表示されるか、片方だけが表示されるかが決まるわけです。

これにelse文を付け加えると、以下のようなコードも作れます。この場合、xが3でなかった場合には"x is NOT 3"という文字列が出力されます。

```
class IfTest2 {
  public static void main (String [] args){
    int x = 2;
    if (x == 3){
      System.out.println("x must be 3");
    }else {
      System.out.println("x is NOT 3");
    }
    System.out.println("This runs no matter what");
  }
}
```

```
% java IfTest2
x is NOT 3
This runs no matter what
```

実行結果

System.out.printとSystem.out.println

読者の中には、文字列を出力するために使用するステートメントがSystem.out.printとSystem.out.printlnの2種類あることに気付いた人もいるでしょう。

両者の違いはどこにあるのでしょうか。

System.out.printlnは、表示する文字列の末尾に自動的に改行文字を挿入するのですが、System.out.printの場合はそんなことはありません。System.out.printを複数使用した場合は、すべての文字列が1つの行に固まって表示されますが、System.out.printlnを複数使用した場合は、文字列ごとに改行が行われます。

自分で考えてみよう

```
%java DooBee
DooBeeDooBeeDo
```

上記のような実行結果を得るために、以下のプログラムの_____に何を入れればよいか考えてください。

```
public class DooBee {
  public static void main (String
[] args){
    int x=1;
    while (x < _____){
      System.out._____("Doo");
      System.out._____("Bee");
    x=x+1;
    }
    if(x== _____){
      System.out.print("Do");
    }
  }
}
```


This PDF is prepared for tools@maruhouse.org, personal use only.

本格的なJavaアプリケーション

それでは、これまでに出てきたクラス、mainメソッド、ループ、int型の変数、String、条件判定などを使った、実用的なプログラム（アプリケーション）の例を見てみましょう。これは、有名な“99 Bottles of Beer”という歌の歌詞を表示するプログラムで、少し改造すれば、実際のビジネスなどにも十分に応用できるものです。ループを使って同様の歌詞が繰り返し表示されるのですが、ビールの本数が1本ずつ減るようになっています。

```
public class BeerSong {
  public static void main (String [] args){
    int beerNum = 99;
    String word = “bottles”;

    while (beerNum > 0){

      if (beerNum == 1){
        word = “bottle”; //ビールが１本の時は単数形
      }
      System.out.println(beerNum +“ ”++word + “of beer on the wall”);
      System.out.println(beerNum +“ ”++word + “of beer.”);
      System.out.println(“Take one down.”);
      System.out.println(“Pass it around.”);
      beerNum = beerNum -1;

      if (beerNum >0){
        System.out.println(beerNum +“ ”++word + “of beer on the wall”);
      }else {
        System.out.println(“No more bottles of beer on the wall”);
      }  //else文終了
    }  //whileループ終了
  }  //mainメソッド終了
}  //クラス終了
```

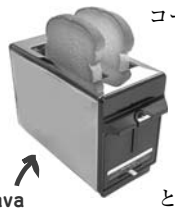
実は、このプログラムには1つ欠点があります。実際にコンパイルして実行してみるとわかりますが、出力が完全ではないのです。欠点が何かを見つけ、修正してみてください。

This PDF is prepared for tools@maruhouse.org, personal use only.

ある月曜の朝 ……

ある月曜の朝、8時30分にBobの目覚まし時計は鳴り出しました。平日はいつもこの時間に起きるのですが、週末遊び過ぎ、疲れていたBobはスヌーズボタンを押し、また寝てしまいました。Bob自身は寝てしまったのですが、彼の部屋にある機械たちは、目覚ましの音を合図に忙しく動き出しました。どの機械にもJavaプログラムが組み込まれていたからです。

目覚まし時計はまず、コーヒーメーカーに「Bobは二度寝してしまったので、コーヒーを作るのは12分遅らせるように」というメッセージを送りました*。そして、コーヒーメーカーもMotorola製のトースターに「Bobは寝ているからパンを焼くのはしばらく待て」というメッセージを送りました。



さらに目覚まし時計はBobの携帯電話に「9時になったら、遅刻しそうになっていることをBobに知らせるように」と指示を出し、Sam（Bobの犬）の首輪には、「新聞をとりにいけ。ただし、今日は散歩は中止」というSamにとっておなじみのメッセージを送りました。

数分後、目覚ましは再び鳴ったのですが、Bobの手はまたもやスヌーズボタンに。仕方なく、機械たちはまた同じ会話を交わしました。しかし、3度目にスヌーズボタンが押された時、目覚ましはもう容赦しませんでした。Samの首輪に「ジャンプして思いっきり吠えろ！」というメッセージを送ったのです。驚いてすっかり目を覚ましたBobは、「Javaが使えてよかった……」と心の底から思うのでした。

家中の機械がJavaに対応していれば、朝、目が覚めた時には、

トーストも、
コーヒーも、
新聞も、

全部用意できているというわけです。

類似技術に注意

Bobの部屋のシステムはJavaを使って作られているのですが、Java以外にも同様のことができるという謳い文句の技術はいくつかあるので注意が必要です。どの技術も、「プラグ&プレイ」、「実行環境を選ばない」といったJavaと同じようなキャッチフレーズが付けられていますが、実際にはJavaを使った場合ほどうまくはいかないのです。

ボブのような生活は本当に可能なのでしょうか。見方によっては可能であるとも、不可能であるとも言えるでしょう。すでに、Javaに対応したPDA、携帯電話、ポケットベル、指輪、スマートカードなどは存在します（特にJava対応の携帯電話はよく知られている）。しかし、Java対応のトースターや犬の首輪などはないのです。その意味ではボブのような生活はまだ不可能であると言ってもいいかもしれません。ただ、たとえ、Javaに対応していない機械であっても、コンピュータ（マイクロチップ）さえ組み込まれていれば、ノートパソコンなどからコントロールして、Java対応の機械と同じような動きをさせることはできます。それに使用されるのが、Jiniサロゲートアーキテクチャと呼ばれる技術です。この技術をうまく使えば、ボブのような「夢の生活」を手に入れることもできるのです。

* メッセージ送信にはIPマルチキャストというプロトコルを使用する。

This PDF is prepared for tools@maruhouse.org, personal use only.

"99 Bottles of Beer" のプログラムよりさらに高度なプログラムを紹介しましょう。これは、あらかじめ用意されたリストから無作為に単語を選び出し、自動的に言葉を作り出す、というプログラムです。

注意：右のソースコードでは、単語のリストの途中で何度か改行されているように見えますが、実際にエディタでコードを入力する際には、リストの途中で改行文字を入れないよう注意してください。エディタのワードラップ機能で自然に行が送られるのはかまいません。「multi-tier」などに使われているハイフンはその通りに入力してください

```
public class PhraseOMatic {
public static void main (String [] args){
```

1
```
//3 つの単語リストを作成する。単語は任意に追加してもよい
    String [] wordListOne ={"24/7","multi-
Tier","30,000 foot","B-to-B","win-win","front-
end","web-based","pervasive","smart","six-
sigma","critical-path","dynamic"};

    String [] wordListTwo ={"empowered","sticky",
"value-added","oriented","centric","distributed",
"clustered","branded","outside-the-box","positioned",
"networked","focused","leveraged","aligned",
"targeted","shared","cooperative","accelerated"};

    String [] wordListThree ={"process","tipping-
point","solution","architecture","core competency",
"strategy","mindshare","portal","space","vision",
"paradigm","mission"};
```

2
```
// 単語リストにいくつの単語が含まれているかを調べる
    int oneLength =wordListOne.length;
    int twoLength =wordListTwo.length;
    int threeLength =wordListThree.length;
```

3
```
//3 つの数を無作為に作る
    int rand1 =(int)(Math.random()*oneLength);
    int rand2 =(int)(Math.random()*twoLength);
    int rand3 =(int)(Math.random()*threeLength);
```

4
```
// 単語を組み合わせて言葉を作る
    String phrase =wordListOne [rand1] +" " +
wordListTwo [rand2] +" " +wordListThree [rand3];
```

5
```
// 作った言葉を画面に表示する
    System.out.println("What we need is a " +phrase);
    }
}
```


This PDF is prepared for tools@maruhouse.org, personal use only.

PhraseOMaticプログラムの仕組み

すでに述べたとおり、前ページの「PhraseOMatic」プログラムは、簡単に言えば、あらかじめ用意された3つのリストから無作為に1つずつ単語を選び出して自動的に言葉を作り出し、画面に表示するというプログラムです。つまり、「30,000 foot outside-the-box architecture」、「web-based leveraged paradigm」といった言葉を作り、表示するわけです。今の時点では、プログラムの各行が何を意味するのか正確にわからなくてかまいません。それは、本書を読み終わる頃にはわかるようになるはずですから、安心してください。以下にプログラムで具体的にどのようなことを行うのかを簡単にまとめます。

1. まず、Stringの配列を3つ作ります。この配列には、単語のリストが入ります。配列を作る（配列を宣言する）のは非常に簡単です。たとえば、次のようなコードを書くだけです。

```
String [] pets ={"Fido", "Zeus", "Bin"};
```

Stringの配列の場合、配列の要素となるのは文字列です。文字列はそれぞれ"（ダブルクォーテーション）で囲みます。文字列と文字列の間には、カンマを入れます。

2. 単語のリストからはそれぞれ、無作為に単語を選び出すことになります。そのためにはまず、リスト中にいくつの単語があるかを確認します。言い換えれば、これは、配列の要素の数を調べる、ということです。リスト中に単語が14個ある場合は、それに対応して0～13の範囲の数値を無作為に作ることになります（Javaでは、配列の要素に0から始まる番号を振るため。1つ目の単語には番号0、2つ目の単語には番号1、14番目の単語には番号13を割り振る）。Javaの場合、配列の要素の数は、lengthという命令を使えば簡単に知ることができます。たとえば、上記の配列「pets」であれば、要素数は次のようにして確認します。

```
int x = pets.length;
```

変数xには、値「3」が入ります。

3. リストは3つなので、無作為の数は3つ作る必要があります。Javaにはそのためのrandom()というメソッドがあらかじめ用意されています。random()は、0以上1未満の数を無作為に作り出すメソッドです。ここでは、random()によって作られた数に、リストの要素数を掛けます。この計算の結果として得られる数には多くの場合、小数点以下の端数が出ますが、int型の変数に代入することで、端数はなくなります（このことに関しては4章で詳しく説明します）。たとえば、以下のようなコードでも、xには整数が代入されます。

```
int x = (int) 24.6;
```

4. 無作為に数を作ったら、それを使ってリストから単語を選び、言葉を作ります。単語と単語の間にはスペースが必要です。文字列をつなげるには、「+」という演算子を使用します。配列の要素を取り出すには、配列名と要素の番号を指定します。

```
String s = pets [0];  // sに"Fido"という文字列が入る
s = s + " " + "is a dog";  // sに"Fido is a dog"という文字列が入る
```

5. 最後は、作った言葉を画面に表示します。

PhraseOMatic
プログラムで
作られる言葉の例

pervasive targeted process

dynamic outside-the-box tipping-point

smart distributed core competency

24/7 empowered mindshare

30,000 foot aligned vision

six-sigma net-worked portal

This PDF is prepared for tools@maruhouse.org, personal use only.

特別座談会

今夜は、コンパイラさんとJVM(Java Virtual Machine)さんにお越しいただき、「Javaにとって、コンパイラとJVMのどちらが重要なのか」というテーマで語っていただきました。

JVM：私とコンパイラさんのどちらが重要かということですが、はっきり言って愚問ですね。重要と言うか、「私がJavaである」と言い切ってもいいくらいですよ。誰がプログラムを動かすと思っているんですか。コンパイラさんはただ、クラスファイルを作るだけです。所詮はファイルですよ。せいぜい内容をプリントアウトして壁紙にしたり、火にくべたり、鳥かごに敷いたりするくらいしか役に立ちません。私が実行をしない限り、クラスファイルには何の意味もないんです。

コンパイラ：その言い方はどうかと思いますね。

JVM：それにね、コンパイラさんはユーモアのセンスのまったくない堅物です。いつもソースコードから重箱の隅をつつくようにして間違いばかり探しているんですから、イヤな人ですよ。

コンパイラ：すみません。じゃあ、うかがいますが、私がいなければ、いったいあなたは何を動かすんですか。念のために言っておきますが、Javaがそもそも私、コンパイラによってバイトコードを作り、それをあなた、JVMがインタープリタ方式で逐一解釈し、実行するという設計になっているのには立派な理由があるんですよ。もし、JVMがソースコードを逐一解釈して実行しなければならなかったとしたら、実行速度はとんでもなく遅くなっていたはずです。そんなことでは、とても、Javaは世の中に受け入れてもらえなかったでしょう。

JVM：そりゃあ、私だって、あなたが何の役にも立たないとは言いませんよ。でも、実際、あなたのやっていることって何でしょう。大したことは何もしていないんじゃないですか。プログラマはその気になれば、バイトコードを自分で書くことだってできるわけですし。そうなれば、あなたはすぐに失業ですよ。

コンパイラ：そんなことを言えば、あなたは自分の無知(ともちろん傲慢さも)をさらけ出すことになりますよ。百歩譲ってあなたの言うことが理論的には正しいとしても、プログラマが自分の手でバイトコードを書くなんてバカげていますよ。確かにフォーマットさえ完全に正しいバイトコードであれば、たとえコンパイラが作ったものでなくても、あなたは実行できるでしょう。でも、手でバイトコードで書くというのは、ポストスクリプトを手で書くのと同じくらい難しいことなんですよ。

JVM：(やれやれ理屈の多い人だ...)あなた、私の質問に答えていませんね。私は「あなたのやっていることって何でしょう」と聞いたんですよ。

コンパイラ：いいですか。Javaは「型付けが強い」言語なんですよ。私は、変数に間違った型の

This PDF is prepared for tools@maruhouse.org, personal use only.

データが格納されるのを防ぐことができます。プログラムが安全に実行できるようにと考えているんですよ。ですから、あなたがプログラムを実行する時には、使い方の間違った変数はほとんど残っていないはずです。それに ──

JVM：でも、完全になくなるってわけじゃないでしょう。私は時々、ClassCastExceptionという例外を発生させますが、これは、変数の使い方の間違いが放置されているという証拠ですよね ──

コンパイラ：すみません。私の話はまだ終わっていませんよ。もちろん、あなたのおっしゃるとおり、実行時に変数の使い方の間違いが発見されることもないわけではありません。でも、中には、Javaにとって重要な「ダイナミックバインディング」という機能をサポートするために、あえて見逃しているというケースもあるんです。この機能のおかげで、「実行するまでどのような型になるかがわからない」という種類のデータが扱えます。Javaというプログラミング言語の幅を広げるのに役立っているんですよ。とはいえ、私は、「こんなことをしても何の得もないし、絶対に実行時に問題のもとになる」というような間違いは決して見逃しません。プログラマはよく、Socketオブジェクトを使うべき時にButtonオブジェクトを使うというような間違いをしますが、私はこういう間違いを確実に見つけ、実行に問題が起きるのを防ぐことができます。

JVM：まあ、いいでしょう。確かにおっしゃるとおりかもしれませんが、じゃあ、セキュリティの問題はどう考えるんです。私はプログラムを実行する時に色々なセキュリティ対策を講じます。それにひきかえ、あなたは何ですか。セミコロンが入っているかどうか、とかくだらないチェックをしているだけじゃないですか。まったく、もっと大変な問題がたくさんあるというのに、そんなことをしている場合ですか。

コンパイラ：お言葉ですがね。私が最前線で守りを固めていることが、どれほどあなたの役に立っているかわかっているんですか。さっき話に出たデータ型のチェックにしたって、システムに大きな損害を与えるようなプログラムが作られるのを防ぐ意味があるんですよ。それに私は、private宣言されたメソッドが不正に呼び出されることや、セキュリティ上の理由から変更が許されていないメソッドに変更が加えられること、重要なデータに不正にアクセスするコードが作られることなんかも防ぐことができるんです。私がどれだけ重要な仕事をしているか、語り始めたら、何時間、いや何日かかるかわかりませんよ。

JVM：そうはおっしゃいますがね。結局私もあなたと同じチェックをしなくちゃならないんですよ。コンパイル後にバイトコードに手を加える輩もいるもんでね。

コンパイラ：そりゃそうです。時にはそういうこともあるでしょう。でも、私が問題の99%は解消しているんです。そうでなかったら、あなたはおそらくまったく機能しないはずですよ。そろそろ時間もないようですから、続きはまた改めて、ということにしましょう。

JVM：わかりました。では、またの機会に。

This PDF is prepared for tools@maruhouse.org, personal use only.

エクササイズ

プログラミングマグネット

バラバラになったJavaプログラムを並べ替え、下のような実行結果が得られるようにしてください。もし、{ 、} などが抜けているところがあれば付け加えてもかまいません。

```
if (x == 1){
  System.out.print("d");
  x = x - 1;
}
```

```
if (x == 2){
  System.out.print("b c");
}
```

```
class Shuffle1 {
  public static void main(String [] args){
```

```
if (x > 2){
 System.out.print("a");
}
```

```
int x = 3;
```

```
x = x - 1;
System.out.print("-");
```

```
while (x > 0){
```

実行結果

```
File  Edit  Window  Help  Sleep
%java Shuffle1
a-b c-d
```


This PDF is prepared for tools@maruhouse.org, personal use only.

エクササイズ

コンパイラになってみよう

A～Cのソースコードを、コンパイラになったつもりで見てみましょう。そして、それぞれのコードが果たして正しくコンパイルできるかどうかを確認してください。もし、「コンパイルできない」と思った場合は、その理由を答え、正しくコンパイルできるように修正もしてください。

A

```
class Exercise1b {
  public static void main(String [] args){
    int x = 1;
    while (x < 10){
      if (x > 3){
        System.out.println("big x");
      }
    }
  }
}
```

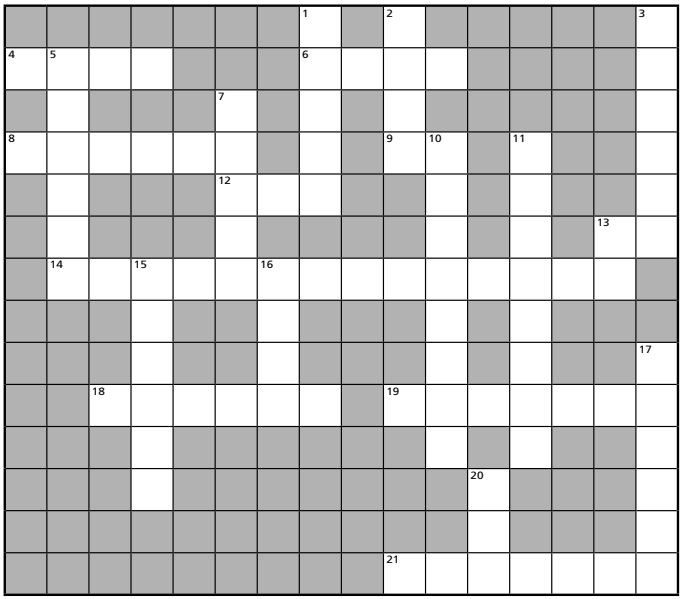
B

```
public static void main(String [] args){
  int x = 5;
  while (x > 1){
    x = x - 1;
    if (x < 3){
      System.out.println("small x");
    }
  }
}
```

C

```
class Exercise1b {
  int x = 5;
  while (x > 1){
    x = x - 1;
    if (x < 3){
      System.out.println("small x");
    }
  }
}
```


This PDF is prepared for tools@maruhouse.org, personal use only.



Javaクロスワード

Javaに関する用語のクロスワードです。1章をよく読んでいれば必ず解けるはずです。中には、直接Javaには関係のない用語も混じっていますが、基本的なコンピュータ関連の用語ですので、わかると思います（注：このクロスワードには英語で、すべて大文字で答えてください）。

ヨコのカギ

4. Javaプログラムを実行する際のコマンド
6. 同じ処理を繰り返す
8. 条件が満たされている場合に所定の処理が一度行われる
9. ノートパソコンの電源
12. 整数のデータ型
13. 集積回路
14. 画面への表示
18. 文字列
19. 新しいクラス、メソッド、変数などを作る
21. コンパイル、プログラムの実行の際に使用する

タテのカギ

1. 浮動小数点数
2. 「何も戻さない」という意味
3. 「どこからでも呼び出せる」という意味のアクセス修飾子
5. 同じ型のデータを複数格納する
7. 一定の条件が満たされている間、同じ処理を繰り返す
10. ソースコードをバイトコードに変換する
11. 値を代入する
13. 情報技術
15. 常に変わらない
16. アプリケーションに1つだけある
17. ステートメントから構成される
20. バイトコードを実行する

ヒント

分岐 = branch
宣言 = declare
配列 = arrays
変数 = variable

This PDF is prepared for tools@maruhouse.org, personal use only.

プログラムパズル

下のプログラムは一部分が欠落しています。欠落部分に入るコードの候補がいくつか示してありますが、それぞれのコードを当てはめた場合にどのような実行結果が得られるかを考え、コードと実行結果を線で結んでください（実行結果の中には、どのコードとも結び付かないものがあります）。

```
class Test {
  public static void main(String [] args) {
    int x = 0;
    int y = 0;
    while ( x < 5 ) {
      [                                    ]
      System.out.print(x + "" + y +" ");
      x = x + 1;
    }
  }
}
```

この部分にコードが入る

コードと実行結果を線で結ぶ

コードの候補

```
y = x - y;
```

```
y = y + x;
```

```
y = y + 2;
if( y > 4 ) {
  y = y - 1;
}
```

```
x = x + 1;
y = y + x;
```

```
if ( y < 5 ) {
  x = x + 1;
  if ( y < 3 ) {
    x = x - 1;
  }
}
y = y + 2;
```

実行結果

```
22 46
```

```
11 34 59
```

```
02 14 26 38
```

```
02 14 36 48
```

```
00 11 21 32 42
```

```
11 21 32 42 53
```

```
00 11 23 36 410
```

```
02 14 25 36 47
```


This PDF is prepared for tools@maruhouse.org , personal use only.

プールパズル

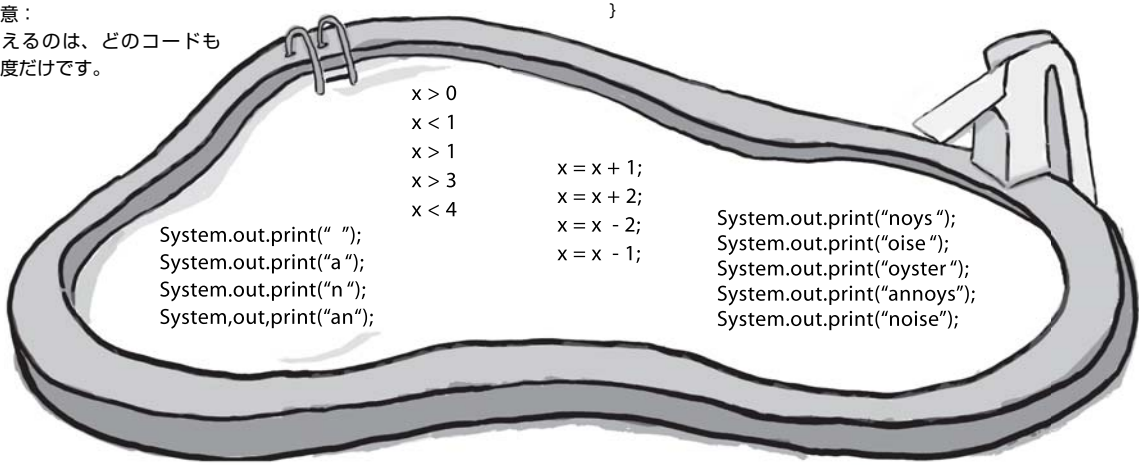
右のプログラムの欠落部分に、下のプールの中のコードを当てはめ、プログラムを完成させてください。同じコードを2回以上繰り返し使うことはできませんが、すべてのコードを使う必要はありません。また、実行結果が、以下に示すようなものになるようにしてください。一見やさしそうですが、意外に難しいので注意が必要です。

実行結果

```
File  Edit  Window  Help  Cheat
%java PoolPuzzleOne
a noise
annoys
an oyster
```

```
class PoolPuzzleOne {
  public static void main(String [] args) {
    int x = 0;

    while ( __________ ) {

      ____________________________
      if ( x < 1 ) {
        ____________________________
      }
      ____________________________

      if ( __________ ) {

        ____________________________

        __________
      }
      if ( x == 1 ) {

        ____________________________
      }
      if ( __________ ) {

        ____________________________
      }
      System.out.println("");

      ____________
    }
  }
}
```

注意：
使えるのは、どのコードも一度だけです。



This PDF is prepared for tools@maruhouse.org , personal use only.

エクササイズの答え

プログラミングマグネット

```
class Shuffle1 {
  public static void main(String [] args) {

    int x = 3;
    while (x > 0) {

      if (x > 2) {
        System.out.print("a");
      }

      x = x - 1;
      System.out.print("-");

      if (x == 2) {
        System.out.print("b c");
      }

      if (x == 1) {
        System.out.print("d");
        x = x - 1;
      }
    }
  }
}
```

File Edit Window Help Poet

```
%java Shuffle1
a-b c-d
```

コンパイラになってみよう

A

```
class Exercise1b {
  public static void main(String [] args) {
    int x = 1;
    while ( x < 10 ) {
      x = x + 1;
      if ( x > 3) {
        System.out.println("big x");
      }
    }
  }
}
```

このコードはコンパイルは正しく行え、実行もできますが（実行結果は出力されない）、「**x =x +1;**」という行を追加しなければ、永久にループが終了しません。

B

```
class Foo {
  public static void main(String [] args) {
    int x = 5;
    while ( x > 1 ) {
      x = x - 1;
      if ( x < 3) {
        System.out.println("small x");
      }
    }
  }
}
```

クラスの先頭に、「これはクラスである」という宣言と、クラスの名前がなければコンパイルはできません。また、**{** と **}** の数も一致していません。

C

```
class Exercise1b {
  public static void main(String [] args){
    int x = 5;
    while ( x > 1 ) {
      x = x - 1;
      if ( x < 3) {
        System.out.println("small x");
      }
    }
  }
}
```

while 文のコードはメソッドの中に入れる必要があります。クラスの中に入れるだけでは十分ではありません。

This PDF is prepared for tools@maruhouse.org, personal use only.

プールパズル

```
class PoolPuzzleOne {
  public static void main(String [] args) {
    int x = 0;

    while ( X < 4 ) {

      System.out.print("a");
      if ( x < 1 ) {
         System.out.print(" ");
      }
      System.out.print("n");

      if ( X > 1 ) {

         System.out.print("oyster");
               x = x + 2;
      }
      if ( x == 1 ) {

         System.out.print("noys");
      }
      if ( X < 1 ) {

         System.out.print("oise");
      }
      System.out.println("");

      X = X + 1;
     }
   }
}
```

```
File  Edit  Window  Help  Cheat
%java PoolPuzzleOne
a noise
annoys
an oyster
```

Javaクロスワード

							1 F		2 V						3 P
4 J	5 A	V	A				6 L	O	O	P					U
	R				7 W		O		I						B
8 B	R	A	N	C	H		A		9 D	10 C		11 V			L
	A				12 I	N	T			O		A			I
	Y				L					M		R		13 I	C
	14 S	Y	15 S	T	E	16 M	O	U	T	P	R	I	N	T	
			T			A				I		A			
			A			I				L		B			17 M
		18 S	T	R	I	N	G		19 D	E	C	L	A	R	E
			I							R		E			T
			C								20 J				H
											V				O
									21 C	O	M	M	A	N	D

プログラムパズル

```
class Test {
  public static void main(String [] args) {
    int x = 0;
    int y = 0;
    while ( x < 5 ) {
      [                                    ]
      System.out.print(x + "" + y +" ");
      x = x + 1;
    }
  }
}
```

コードの候補

```
y = x - y;
```

```
y = y + x;
```

```
y = y + 2;
if( y > 4 ) {
  y = y - 1;
}
```

```
x = x + 1;
y = y + x;
```

```
if ( y < 5 ) {
  x = x + 1;
  if ( y < 3 ) {
    x = x - 1;
  }
}
y = y + 2;
```

実行結果

```
22 46
```

```
11 34 59
```

```
02 14 26 38
```

```
02 14 36 48
```

```
00 11 21 32 42
```

```
11 21 32 42 53
```

```
00 11 23 36 410
```

```
02 14 25 36 47
```


This PDF is prepared for tools@maruhouse.org, personal use only.

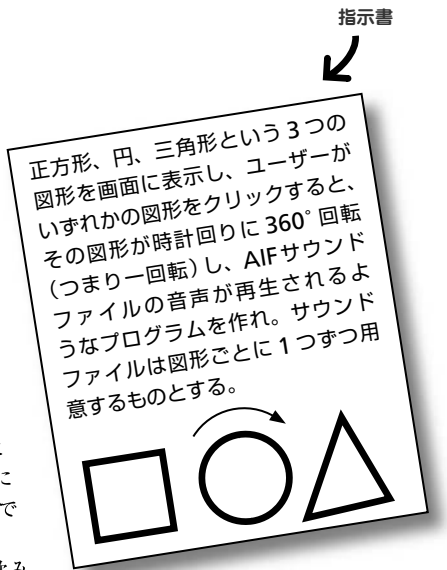
2章　クラスとオブジェクト

オブジェクト指向の国へ

2章では、前の章で予告したとおり、オブジェクト（クラス）について解説します。Javaのコードの例はすでにいくつか紹介しましたが、今までのコードはすべてmain()メソッドしかないものでした。これは正確には「オブジェクト指向」のプログラムとは言えません。まったく「オブジェクト指向的」ではないと言っても言い過ぎではないでしょう。PhraseOMaticプログラムに使っていたStringなどは実はオブジェクトの一種なのですが、Stringを使ったからと言ってオブジェクト指向的なプログラムを書いたことにはなりません。真にオブジェクト指向的なプログラムを書くためには、やはり自らの手で新しいオブジェクトを作る必要があります。すでに述べたとおり、オブジェクト指向でない言語のことを「手続き指向言語」などと呼びますが、これからは、どのようにすれば手続き指向的ではない、オブジェクト指向的なプログラムが書けるのかを具体的に学んでいきましょう。Javaでのオブジェクト指向プログラミングの楽しさを知ってもらえればと思います。「クラス」、「オブジェクト」という言葉はどちらも同じ意味に使われることがありますが、厳密には意味が少し違っています。この章では両者の違いについても触れます。ともかく、オブジェクト指向の良さを知ってしまえば、もう後戻りはできません。そのことがこの章でよくわかるでしょう。

This PDF is prepared for tools@maruhouse.org, personal use only.

対決！
オブジェクト指向対手続き指向



あるソフトウェア会社に、Larryと Bradという2人のプログラマがいました。Larryは手続き指向の、Bradはオブジェクト指向のプログラマです。ある日、イヤミな上司が2人に同じ指示書を渡し、「このとおりのプログラムを作れ」と命令してきました。2人で競争をし、先にプログラムを完成させた方に、シリコンバレーの技術者に人気のアーロンチェアを贈るというのです。2人はどちらも指示書を見てすぐに「これは楽勝だ」と思いました。

Larryは自分のオフィスの椅子に座り、まず、「このプログラムはどんなことをしなければならないものなのか（プログラムにどのような手続きが必要か）」ということを考えてみました。その結果、「図形を回転させる」、「音を鳴らす」という2つのことをする必要がある、ということに気付きました。そこで、「rotate（回転）」、「playsound（音を鳴らす）」という2つの関数を作ることにしました。「プログラムを作るというのは、必要な関数をすべて揃えることである」というのがLarryの考えだったのです。

Bradは、同じ頃、会社近くのカフェで休憩中でした。ただ、コーヒーを飲みながら、一応、指示書のプログラムについて考えてはいました。彼が考えたのは、「このプログラムにはどのような物や事が関わっているか」ということです。すぐにわかったのは、「これはいくつかの図形が関わるプログラムである」ということでした。また、図形以外にも、「ユーザー」、「音」、「クリック」などがプログラムに関わってきます。Bradはとりあえず、個々の図形を表す「クラス」を作ることにしました。以下にLarryと Bradの作業経過を簡単に示します。さて、アーロンチェアは誰のものになったでしょうか。

アーロンチェア

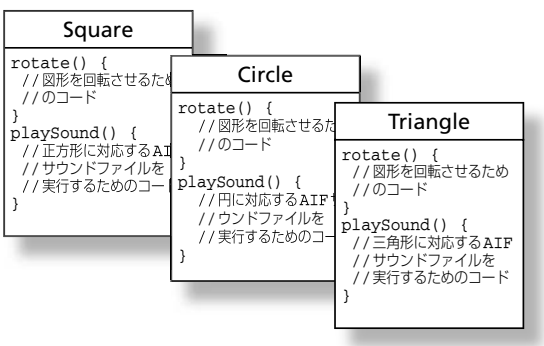
Larry のプログラム

Larryはいつものように、「必要な関数をすべて揃える」という方法でプログラムを作り始めました。まず彼が「必要」と判断したのは、rotate、playSoundという2つの関数でした。

```
rotate(shapeNum) {
   // 図形を 360° 回転させるためのコード
}
playSound(shapeNum){
   // 変数 shapeNum を使用して
   // 図形に対応する AIF サウンドファイルを探し、
   // 再生するためのコード
}
```

Brad のプログラム

Bradは各図形に対応するクラスを作ることにしました。



This PDF is prepared for tools@maruhouse.org, personal use only.

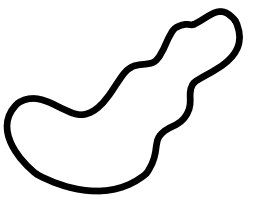
プログラムを先に完成させたのはLarryでした。「これでアーロンチェアは自分のもの」Larryはそう思ったのですが……

ところが、そこで信じられないことが……

上司が、「Larryの方が早かったようだが、実はプログラムの仕様を少しだけ変更したいんだよ」と言い出したのです。「まあ、君たちくらいの優秀な技術者にとっては、この程度の変更は何でもないだろうがね」と上司は言います。

「バカな。何てことを言うんだ」とLarryは思いました。仕様変更が簡単だったことなどいまだかつてなかったからです。しかし、Bradの方は上司の言葉を聞いても不思議に落ち着いていました。「彼が落ち着いているのはひょっとしてオブジェクト指向のせいか……」とも思いましたが、Larryは「オブジェクト指向は確かに魅力的だが、プログラミングに時間がかかる」という自分の考えを変える気はありませんでした。考えを変えてしまえば、これまで苦労して身に付けたことが無駄になるような気がしたのです。

正方形、円、三角形に加え、下の図のような「アメーバ形」の図形を追加する。アメーバ形についても、ユーザーがクリックすると他の図形と同様に回転し、.hifファイルの音声が再生されるようにして欲しい。



仕様変更の内容

Larry のプログラム

Larryは自分のオフィスに戻り、考えました。rotate関数はそのまま使えそうでした。アメーバ形に対応する変数shapeNumの値さえ決めれば問題はないからです。しかし、playSound関数はそうはいきません。アメーバ形の場合だけ再生するファイルを変えなくてはならないのです。

```
playSound(shapeNum) {
  //図形がアメーバ形でなければ
    //shapeNumの値を調べ
    //その値に対応するAIFファイルを再生する
  //さもなければ
    //.hifファイルを再生する
}
```

この変更自体は結局は簡単にできたのですが、どれほど簡単ではあっても、一度テストを済ませたコードに変更を加えるというのは気分のいいものではありません。とはいえ、上司が途中で仕様変更を命じてくるというのは珍しいことではなかったので、彼もあきらめの心境でした。

Brad のプログラム

同じ頃、Bradはビーチでマルガリータを飲んでいました。突然の仕様変更にも少しもあわてていなかったのです。ただ、新しいクラスを1つ追加すればいいだけのことだったからです。既に書いたコードには何の変更も要りません。彼は、これがオブジェクト指向の最大のメリットだと感じていました。「本当にこういう時、便利だよな……」いつもそうですが、今回もつくづくそう思いました。

Amoeba

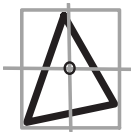
```
rotate() {
  //アメーバ形を回転させるための
  //コード
}
playSound() {
  //アメーバ形に対応する
  //.hifファイルを再生するため
  //のコード
}
```


This PDF is prepared for tools@maruhouse.org, personal use only.

ほんのわずかですが、LarryはBradより先に仕様変更を終えました

（やっぱりオブジェクト指向なんてナンセンスなんだよ……）Larryはそう思い、笑みを浮かべていましたが、2人のプログラムを見て上司が（いかにもがっかりした、というように）発した、「ああ、アメーバ形の回転はこうじゃないんだよ」という言葉に、笑顔は消えました。

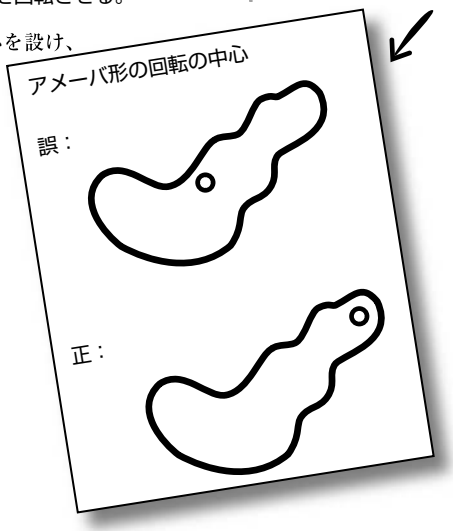
LarryとBradは図形を次のような方法で回転させていました。



1　図形を囲む長方形を作る
2　その長方形の中心の位置を計算し、その中心を使って図形を回転させる。

しかし、上司の意図では、アメーバ形だけは、一方の端に中心を設け、時計の針のように回転させるはずだったのです。

Larryはつぶやきました。「やれやれ困ったな。rotate関数にif-else文をもう1つ追加して、アメーバ形の場合だけ違う方法で回転の中心を決めるようなコードを書けばそれで何とかなるはずだけど……」とは思いましたが、「でも、また何か修正を命じられたらどうする？」という不安をどうしても抑えることができませんでした。



Larryのプログラム

Larryはオフィスに戻り、rotate関数に、アメーバ形の場合だけ回転の中心を変更するために、xPt、yPtという引数を追加しました。それによって影響を受けるコードがかなりあったので、何度もテストと再コンパイルを繰り返すはめになってしまいました。彼の手法にはどうも無理があるようです。

```
rotate(shapeNum,xPt,yPt){
  // 図形がアメーバ形でなければ、
     // 長方形を描き
     // その中心点を使って
     // 回転をする
  // さもなければ
     //xPt とyPt の値を使って
     // 中心点を決め
     // それを回転に使用する
}
```

Bradのプログラム

同じ頃、Bradは野外ロックフェスティバルに行っていました。芝生の上に座り、音楽を楽しみながら、彼はrotateメソッドに修正を加えました。ただし、彼が修正したのは、最後に追加したAmoebaクラスのrotateメソッドだけです。アメーバ形の中心点を決めるため、Amoebaクラスには2つの変数を追加しましたが、プログラムの他の部分への影響がないことは確実なので、既存のコードについてはテストは必要なく、再コンパイルを繰り返す必要もありません。修正は1組目の演奏が終わるまでには完了し、彼はノートパソコンから完成したプログラムを送信しました。

Amoeba

```
int xPoint
int yPoint
rotate() {
 //アメーバ形を回転させるための
 //コードxPointとyPointを
 //使用して中心点を決める
}
playSound() {
 //.hifファイルを
 //再生するためのコード
}
```


This PDF is prepared for tools@maruhouse.org, personal use only.

勝ったのはBrad、でしょうか……

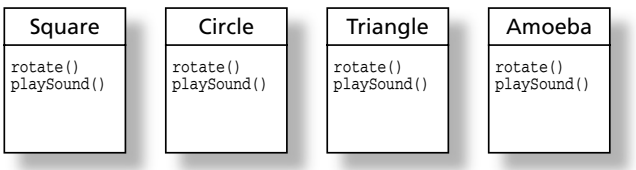
ことはそう簡単ではありません。LarryはBradのプログラム作成手法に重大な欠点を見つけていたのです（Larryはアーロンチェアを貰ったら、経理のLucyに見せると約束していたので、簡単に負けるわけにはいきませんでした）。

Larry：君のコードには重複が多いね。同じrotate関数が4つもあるじゃないか。
Brad：これは関数じゃないよ。「メソッド」って言うのさ。
Larry：何だっていいよ。こんなのバカげているよ。4つも同じような「メソッド」があるなんて。面倒なだけだろう。何かいいことがあるのかい？
Brad：もしかして、君は作りかけのプログラムを見たんじゃないのか。最終的にどうなったか見せてあげるよ。最終的にはオブジェクト指向で言う「継承」というテクニックを使ったんだ。

Lucy
（経理部の女性）

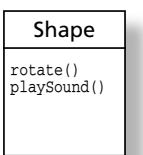
❶

Bradの最初のプログラムデザイン：4つのクラスに同じ名前のメソッドがある



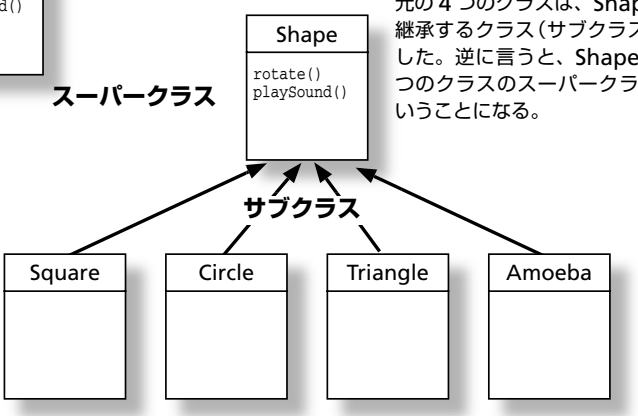
❷

Bradはその後、Shapeという新しいクラスを作り、元の4つのクラス（Square、Circle、Triangle、Amoeba）に共通するメソッドを持たせた。



❸

元の4つのクラスは、Shapeを継承するクラス（サブクラス）とした。逆に言うと、Shapeは4つのクラスのスーパークラスということになる。

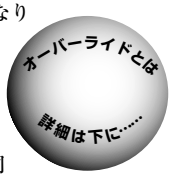


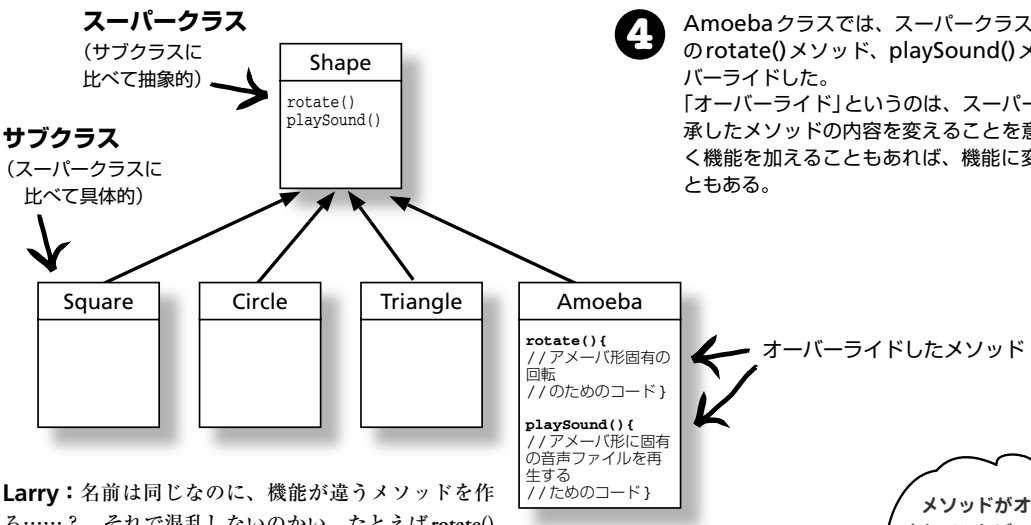
4つのクラスをSquareのサブクラスにしたため、rotate()、playSound()というメソッドはいったん削除しています。Larryの言う「重複」はなくなったわけです。
あるクラスが別のクラスを継承する（サブクラスになる）というのは、そのクラスの性質や機能を受け継ぐということです。ここでは、Shapeのサブクラスである4つのクラスが、Shapeの機能（つまりメソッド）を受け継ぐことになります。

This PDF is prepared for tools@maruhouse.org, personal use only.

アメーバ形は？

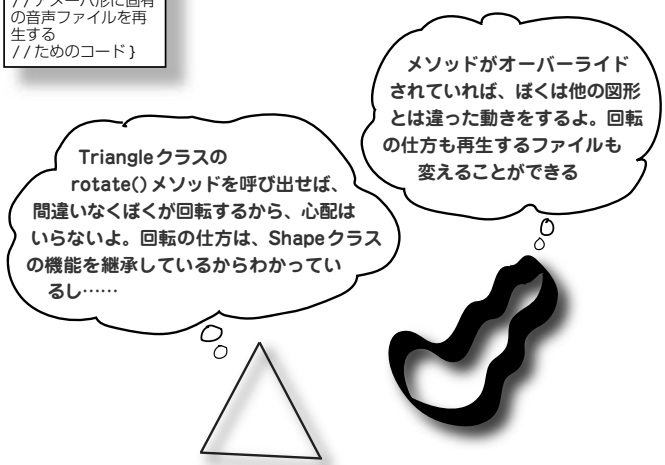
Larry：でも、アメーバ形の場合はrotate関数もplaySound関数も機能がかなり違うはずじゃないか。他とまったく同じというのは問題だろう。
Brad：関数じゃないよ、「メソッド」だって。
Larry：どっちでもいいよ。機能が違うはずなのに、Shapeクラスの機能を「継承」してしまうのはまずいよ。
Brad：話はこれで終わりじゃないんだよ。Amoebaクラスの場合だけは、同じ名前のメソッドでも内容を変える。「オーバーライド」って言うんだけど。そうすれば、アメーバ形の場合だけ、他の図形と違うrotate()メソッド、playSound()メソッドが実行されることになる。





4 Amoebaクラスでは、スーパークラスであるShapeのrotate()メソッド、playSound()メソッドをオーバーライドした。
「オーバーライド」というのは、スーパークラスから継承したメソッドの内容を変えることを意味する。新しく機能を加えることもあれば、機能に変更を加えることもある。

Larry：名前は同じなのに、機能が違うメソッドを作る……？　それで混乱しないのかい。たとえばrotate()関数、いやメソッドを呼び出してから、回転させるのがアメーバ形なのか、他の図形なのかを指示してやるとか、そういうことが必要になるのかな……。
Brad：オブジェクト指向の場合は、メソッドを呼び出す時に「Amoebaクラスのrotate()メソッド」、「Triangle形のrotate()メソッド」という具合に指定するから混乱しないんだよ。それで、機能の違うメソッドが実行されて、図形が違った風に回転する。Amoeba形以外にまた新しい図形を追加する場合は、単に新しいクラスを追加して、必要に応じてメソッドをオーバーライドすればいい。それで他の図形とは動きが変わることになる。
Larry：同じように「食べる」、「走る」と言っても人によって微妙に違うようなものかな……



This PDF is prepared for tools@maruhouse.org, personal use only.

アーロンチェアを手に入れたのは？

結局、アーロンチェアを手に入れたのはLarryでもBradでもありませんでした。上司は内緒で別のフロアのAmyにも同じ指示書を渡していたのです。ひどい話です。

オブジェクト指向言語の長所とは？

「実際の物事の動きをそのままプログラムに置き換えることができるので、プログラムの設計がしやすい」
— ソフトウェアアーキテクト、Joy（27才）

「機能を追加する時に、既存のコードへの影響がないのがいい」
— プログラマ、Brad（32才）

「データとそれを操作するメソッドを1つのクラスにまとめることができるので便利」
— プログラマ、Josh（22才）

「すでに作ったクラスを再利用できる。継承して機能を追加するということも簡単」
— プロジェクトマネージャ、Chris（39才）

「Chrisの発言には驚きです。もう5年もまったくプログラミングをしていない人の言葉とは思えません」
— Chrisの部下、Daryl（44才）

「椅子がもらえるということですね。その他ですか？うーん……」
— プログラマ、Amy（34才）

頭の体操

クラスとオブジェクトについて学ぶ前に……

手続き指向プログラマとオブジェクト指向プログラマの対決、いかがでしたか。この対決を通して、オブジェクト指向の重要な概念であるクラスやメソッドについても簡単に触れましたが、理解できたでしょうか。この後、クラス、オブジェクトについてさらに詳しく学ぶことになります（ただし、継承やオーバーライドの詳細については、また後の章で解説します）。
次ページ以降の説明を読む前に、前ページまでの内容（あるいはこれまでにオブジェクト指向言語を使った経験）を踏まえ、次のようなことについて少し考えてみてください。

- Javaのクラスを作る際にはどのようなことについて考慮しなければならないか。
- 考慮すべきことをチェックリストにするとどのようなものになるか。

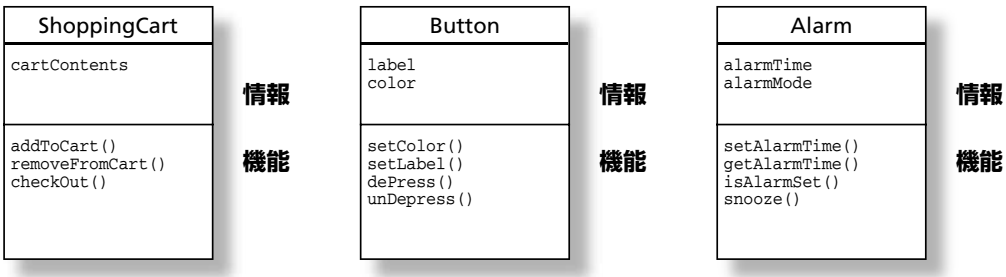
「認知科学的」アドバイス

何か考える時は、声に出した方がいいでしょう。声を出し、その声を聞くと、脳の中の、また違った部分が活性化するようです。もちろん、誰かと話しができればそれが一番いいのですが、手近に話し相手がいない場合は、ペットが相手でもいいと思います。おかげで私の犬は、今やオブジェクト指向のエキスパートです。

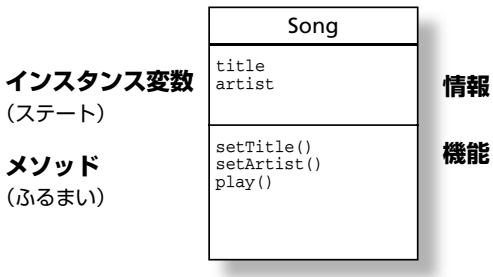
This PDF is prepared for tools@maruhouse.org, personal use only.

クラスを作る際に考えるべきこと

- **クラスに対応する物や事に必要な情報**
- **クラスに対応する物や事が果たすべき機能（役割）**



- **情報のことを「インスタンス変数」**
- **機能のことを「メソッド」と呼ぶ**



cartContents、label、colorなどをインスタンス変数と呼びます。インスタンス変数の値（データ）を変えることで、オブジェクト（インスタンス）の性質をさまざまに変えることができます。

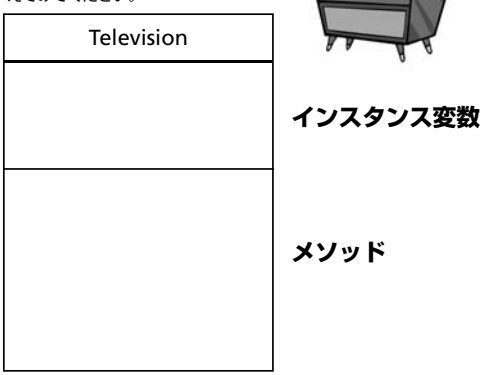
オブジェクトのことを「インスタンス」と呼ぶ場合もあります（オブジェクト、インスタンスといった言葉の厳密な意味については次ページ参照）。

addToCart()、setColor()、setAlarmTime()などをメソッドと呼びます。クラスを作る際には、インスタンス変数とともに、どのようなメソッドが必要か（クラスにどのような機能を持たせるべきか）を考えます。インスタンス変数は、メソッドによって操作されます。メソッドでは、インスタンス変数の値を読み込んだり、インスタンス変数に値を書き込んだりするのです。たとえば、Alarmクラスには、alarmTimeというインスタンス変数がありますが、この変数の値の読み込み、書き込みは、2つのメソッドによって行われます。

以上のように、クラスにはインスタンス変数とメソッドが必要なのですが、インスタンス変数、メソッドについてだけ考えればクラスができるわけではないという点には注意しなければなりません。

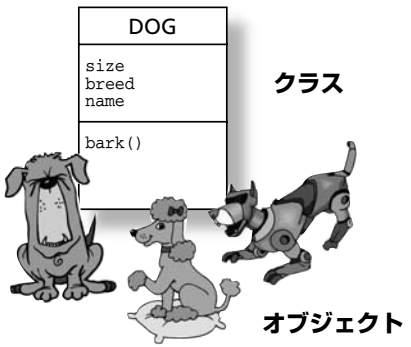
自分で考えてみよう

テレビを表すTelevisionクラスを作る場合、インスタンス変数、メソッドとしてはどのようなものが必要かを考えてみてください。



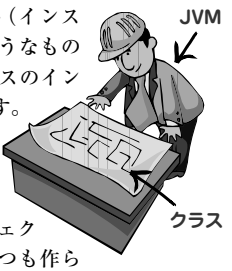
This PDF is prepared for tools@maruhouse.org, personal use only.

クラスと オブジェクトの 違い



クラスとオブジェクトは厳密には異なる（クラスはオブジェクトの設計図）

クラスは、Java仮想マシンがオブジェクト（インスタンスともいう）を作るのに使う設計図のようなものです。オブジェクトが作られる時には、クラスのインスタンス変数には特定の値が書き込まれます。たとえば、1つのButtonクラスから、インスタンス変数にさまざまな値が書き込まれ、色、サイズ、形、ラベルがそれぞれに異なるButtonオブジェクト（Buttonクラスのオブジェクト、あるいはインスタンスとも呼ぶ）がいくつも作られるということになります。



角度を変えて見ると……

オブジェクトはアドレス帳の中の、情報が書き込まれたページのようなもの

クラスとオブジェクトの関係はアドレス帳にたとえるとわかりやすいかもしれません。アドレス帳を買ったばかりの状態では、個々のページには空欄がいくつもあります。アドレス帳のページをクラスとすると、空欄はインスタンス変数と考えることができます。この空欄に何らかの情報を書き込むと、そのページはオブジェクト（インスタンス）になるのです。

この場合、メソッドは、カードの各欄に書き込まれた情報を取り出したり、各欄に情報を書き込んだり、といった役割を果たすことになります。このメソッドには、たとえば、getName()、changeName()、setName()といった名前が付けられるでしょう。

メソッドの果たす役割（機能）はどのオブジェクトでも同じですが、オブジェクトの持つ情報はそれぞれに異なります。

This PDF is prepared for tools@maruhouse.org, personal use only.

クラスの作成と使用

では、実際にクラスを作り、作ったクラスを使用してみましょう。クラスを使用するには、そのためのクラスをもう1つ作ることになります。たとえば、Dogクラスを使用するには、Dogクラスに加えてもう1つのクラスを作るということです。そのクラスは、とりあえず、mainメソッドだけのクラスにします。mainメソッドの中で、使用したいクラスのオブジェクト(インスタンス)を作成し、それを利用するのです。「利用する」というのは、ここでは具体的には、メソッドを実行し、インスタンス変数を使うということを指します。

この本の中では、これからも、このような2つのクラスの組み合わせが何度も出てきます。一方は作りたいクラスで、もう一方はそのクラスをテストするためのクラスです。テスト用のクラスの名前は、すべて「テスト対象のクラス名+TestDrive」というものにします。たとえば、テスト対象のクラスが「Bungee」ならば、テスト用のクラスの名前は「BungeeTestDrive」になるということです。テスト用のクラスには、先述のとおり、main()メソッドだけを持たせます。このmainメソッドの中で、テスト対象のクラスのオブジェクトを作り、「.」という演算子を使って、テスト対象のクラスのメソッドと変数を利用します。以下に例を示します。

「.」演算子

「.」演算子を使用すると、作成したオブジェクトのインスタンス変数、メソッドが利用できます。

```
//オブジェクトの作成
Dog d =new Dog();

//以下のように書くことで
//作成したDogオブジェクトの
//bark()メソッドが利用できる
d.bark();

//Dogオブジェクトのsize変数
//に値を指定するには以下のよう
//に書く
d.size =40;
```

1 テスト対象のクラスを作成する

```
class Dog {
  int size;
  String breed;
  String name;
  void bark() {
    System.out.println("Ruff!Ruff!");
  }
}
```

インスタンス変数

メソッド

2 テスト用のクラスを作成する

```
class DogTestDrive {
  public static void main (String [] args) {
    //Dogをテストするためのコード
  }
}
```

mainメソッドだけを作る(具体的にどのようなコードを入れるかは後述)。

3 テスト用のクラスのmainメソッドに、テスト対象のクラスのオブジェクトを作り、変数とメソッドを利用するコードを書く

```
class DogTestDrive {
  public static void main (String [] args) {
    Dog d =new Dog();
    d.size =40
    d.bark();
  }
}
```

Dogオブジェクトの作成

「.」演算子を使用して、Dogのsize変数の値を指定する

「.」演算子

bark()メソッドを使用する

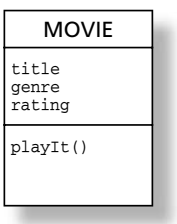
少しでもオブジェクト指向の知識がある人は、著者がここで「カプセル化」を行っていないことに気付いたと思います。「カプセル化」については4章で詳述します。

This PDF is prepared for tools@maruhouse.org, personal use only.

Movieクラスの作成とテスト

```
class Movie {
  String title;
  String genre;
  int rating;

  void playIt(){
    System.out.println("Playing the movie");
  }
}

public class MovieTestDrive {
  public static void main(String [] args) {
    Movie one =new Movie();
    one.title ="Gone with the Stock";
    one.genre ="Tragic";
    one.rating =-2;
    Movie two =new Movie();
    two.title ="Lost in Cubicle Space";
    two.genre ="Comedy";
    two.rating =5;
    two.playIt();
    Movie three =new Movie();
    three.title ="Byte Club";
    three.genre ="Tragic but ultimately uplifting";
    three.rating =127;
  }
}
```

自分で考えてみよう



MovieTestDriveクラスでは、Movieクラスのオブジェクト（インスタンス）を3つ作成し、「.」演算子を使って、インスタンス変数の値の設定を行います。また、オブジェクトのうち1つはメソッドの呼び出しにも使います。3つのオブジェクトそれぞれについて、title、genre、ratingという3つの変数の値がどうなるかを右の表にまとめてください。

オブジェクト1	title genre rating
オブジェクト2	title genre rating
オブジェクト3	title genre rating

This PDF is prepared for tools@maruhouse.org, personal use only.

mainから外へ飛び出す

main()メソッドだけを使っているうちは、まだ「オブジェクト指向プログラミングをしている」とは言えません。しかも、先述のテスト用のクラスのように、main()メソッドで他のクラスのオブジェクトを作成したり、メソッドを呼び出したりするだけでも十分ではありません。本格的なオブジェクト指向アプリケーションを作るためには、オブジェクトどうしがお互いを利用し合うようなことが必要になってくるのです。

main()メソッドの2つの役割

- **他のクラスのテスト**
- **Javaアプリケーションの起動**

Javaアプリケーションは、突き詰めれば、「オブジェクトどうしのやりとり」でなりたっていると言ってもいいでしょう。「やりとり」というのは、主に「互いのクラスのメソッドを呼び出す」ということを指します。メソッドの呼び出しに関しては4章で詳しく解説します。この章ですでに触れたとおり、main()メソッドは、他のクラスのテストに利用できます。しかし、一方で、6章でも詳述するとおり、main()メソッドには、Javaアプリケーションを起動させるという役割もあります。main()メソッドで他のクラスのメソッドを呼び出すと、そのメソッドがさらに別のクラスのメソッドを呼び出す……というようにして、アプリケーションが動いていくのです。

Javaアプリケーションがどのように動くかがわかるよう、次に簡単な例を紹介します。まだ基礎の段階なので、ここではプログラムの「効率」、「美しさ」ということはあまり考慮しません。そうしたことには、さらに後の章で触れることになります。ともかく、まずは、オブジェクトどうしがどのように「やりとり」するのか、ということをおおまかにつかんでください。

サンプルアプリケーション

アプリケーションの概要

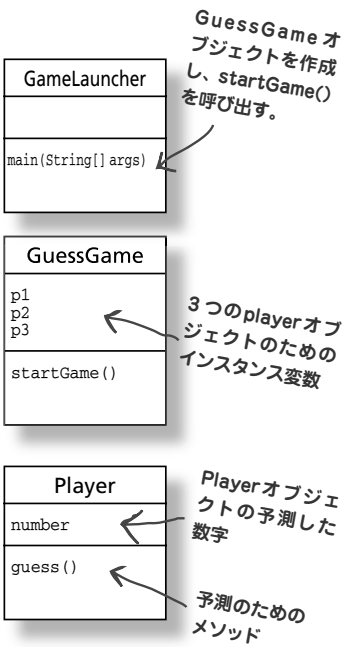
0から9までの中から無作為に(自動的に)数字が選ばれ、そして、その数字が何かをPlayerオブジェクトが予測する、というゲームのアプリケーションを作ります(あらかじめ断っておきますが、あまり面白いゲームではありません)。

必要なクラス

`GuessGame.class` `Player.class` `GameLauncher.class`

アプリケーションのロジック

1) まず、GameLauncherクラスからアプリケーションが起動される。GameLauncherクラスはmain()メソッドを持つクラス。
2) GameLauncherクラスのmain()メソッドで、GuessGameクラスのオブジェクトが作られ、GuessGameクラスのstartGame()メソッドが呼び出される。
3) startGame()メソッドが呼び出されるとゲームが開始される。このメソッドでは、3つのPlayerオブジェクトが作られ、無作為に数字が選ばれる。その後、Playerオブジェクトに数字を予測するよう指示が出され、選ばれた数字と予測された数字との比較が行われる。最後は、どの数字が選ばれたのか、どのPlayerオブジェクトがどの数字を予測したのか、どのPlayerオブジェクトが正解で、どのPlayerオブジェクトが不正解だったのか、といったことが画面に表示される。正解のPlayerオブジェクトがなかった場合は再度ゲームが繰り返される。



This PDF is prepared for tools@maruhouse.org, personal use only.

```
public class GuessGame {
   Player p1;
   Player p2;
   Player p3;

   public void startGame() {
     p1 = new Player();
     p2 = new Player();
     p3 = new Player();

     int guessp1 = 0;
     int guessp2 = 0;
     int guessp3 = 0;

     boolean p1isRight = false;
     boolean p2isRight = false;
     boolean p3isRight = false;

     int targetNumber = (int) (Math.random() * 10);
     System.out.println("I'm thinking of a number between 0 and 9...");

     while(true) {
       System.out.println("Number to guess is " + targetNumber);

       p1.guess();
       p2.guess();
       p3.guess();

       guessp1 = p1.number;
       System.out.println("Player one guessed " + guessp1);

       guessp2 = p2.number;
       System.out.println("Player two guessed " + guessp2);

       guessp3 = p3.number;
       System.out.println("Player three guessed " + guessp3);

       if (guessp1 == targetNumber) {
          p1isRight = true;
       }
       if (guessp2 == targetNumber) {
          p2isRight = true;
       }
       if (guessp3 == targetNumber) {
         p3isRight = true;
       }

       if (p1isRight || p2isRight || p3isRight) {

         System.out.println("We have a winner!");
         System.out.println("Player one got it right? " + p1isRight);
         System.out.println("Player two got it right? " + p2isRight);
         System.out.println("Player three got it right? " + p3isRight);
         System.out.println("Game is over.");
         break; // ゲーム終了。ループを抜ける
       } else {
         // 正解したplayerオブジェクトがないのでゲーム続行
         System.out.println( "Players will have to try again." );
       } // if/elseの終了
     } // ループの終了
   } // メソッドの終了
} // クラスの終了
```

3つのPlayerオブジェクトのためのインスタンス変数

Playerオブジェクトを3つ作成し、インスタンス変数に代入

Playerが予測した数字を代入するための変数の宣言

Playerが予測した数字が正解かどうかを示す値を代入するための変数の宣言

無作為に数字を選ぶ

Playerオブジェクトのguess()メソッドを呼び出す

Playerオブジェクトのnumber変数にアクセスし、Playerオブジェクトが予測した数字（guess()メソッドの実行結果）を得る

Playerオブジェクトの予測した数字と、無作為に選ばれた数字とを比較する。予測が正しければ、変数にtrueを代入する（何も代入しなければfalseになる点に注意）

いずれかのPlayerオブジェクトの予測が正しければ以下の処理が実行される（||という演算子は「OR」を意味する）

予測がいずれも正しくなければループの処理が続行され、Playerオブジェクトは新たに予測をすることになる

This PDF is prepared for tools@maruhouse.org, personal use only.

アプリケーションの実行

```
public class Player {
  int number =0; // 予測した数字が入る変数

  public void guess(){
    number =(int)(Math.random()*10);
    System.out.println("I'm guessing "
                       +number);
  }
}

public class GameLauncher {
  public static void main (String [] args) {
    GuessGame game =new GuessGame();
    game.startGame();
  }
}
```

ガーベージコレクション

Javaでは、オブジェクトが作られると必ず、「ヒープ」と呼ばれるメモリ領域に格納されます。その点は、オブジェクトがいつ、どこで、どのように作られるかを問わず同じです。ただ、作られたオブジェクトがいつまでもヒープに保管されるというわけではありません。Java(JVM)には「ガーベージコレクション」という機能があるからです。オブジェクトが作成される時には、そのオブジェクトに合った大きさのメモリ領域が割り当てられます。たとえば、15個のインスタンス変数を持つオブジェクトの方が、インスタンス変数を2つしか持たないオブジェクトより一般に大きな容量を必要とするので、そうしたことを考慮して割り当てる容量が決められるのです。ただ、いったん割り当てられた領域を別の目的に使用しなくてはならないということもあります。その場合はどうすれば割り当て解除ができるのでしょうか。実は、「ガーベージコレクション」というのは、その割り当て解除を自動的に行う機能なのです。JVMは、「今後使われることはない」と判断したオブジェクトを「ゴミ(ガーベージ)」として処理し、割り当てられたメモリ領域を空けることができるのです。空けられた領域は別の用途に使うことができます。ガーベージコレクションは、メモリが不足した時に行われます。「ガーベージコレクション」については、次章で詳述します。

サンプルアプリケーションの実行結果
(結果は毎回変化します)

```
File  Edit  Window  Help  Explode
%java GameLauncher
I'm thinking of a number between 0 and 9...
Number to guess is 7
I'm guessing 1
I'm guessing 9
I'm guessing 9
Player one guessed 1
Player two guessed 9
Player three guessed 9
Players will have to try again.
Number to guess is 7
I'm guessing 3
I'm guessing 0
I'm guessing 9
Player one guessed 3
Player two guessed 0
Player three guessed 9
Players will have to try again.
Number to guess is 7
I'm guessing 7
I'm guessing 5
I'm guessing 0
Player one guessed 7
Player two guessed 5
Player three guessed 0
We have a winner!
Player one got it right? true
Player two got it right? false
Player three got it right? false
Game is over.
```


This PDF is prepared for tools@maruhouse.org, personal use only.

素朴な疑問に答えます

Q: Javaの変数、メソッドは、いわゆる「グローバル」なものにはできないと聞きました。変数やメソッドをプログラム中のあらゆる箇所で使いたいという場合はあると思うのですが……

A: 確かに、Javaには「グローバル」な変数、メソッドというものはありません。ただ、現実には、変数やメソッドをプログラム中のあらゆる箇所で使えるようにする必要は出てくるでしょう。Phrase-O-Maticプログラムのrandom()メソッドなどはその例です。また、円周率「pi」などの定数もそうでしょう。10章で詳しく学びますが、Javaでも「public static」という宣言をすることで、メソッドを「グローバル」に近いものにすることはできます。「public static」と宣言されたメソッドは、アプリケーション中のあらゆるクラスで利用できます。また、「public static final」という宣言をすれば、変数（定数）をグローバルに近いものにすることも可能です。

Q: 「グローバルに近い」変数やメソッドを使用した場合、Javaのプログラムはオブジェクト指向ではなくなってしまうのですか。

A: いいえ。そんなことはありません。Javaでは結局、変数、メソッドがいずれかのクラスに属することになるからです。たとえば、定数「pi」やrandom()メソッドは、グローバルなもののように使用できますが、いずれもMathクラスに属しています。また、変数やメソッドをグローバル的に使用することは、Javaプログラミングではあくまで例外に属することであるという認識も必要でしょう。

Q: Javaのプログラムは結局クラスの集まりということですか。

A: そうです。Javaのプログラムはクラスを集めたもの（最低1つのクラスから構成される）と言えます。Javaアプリケーションでは、構成するクラスのうち、少なくとも1つはmain()メソッドを持っている必要があります。このmain()メソッドからアプリケーションは起動されます。プログラムを構成するクラスはそれぞれが個別のファイルになります。プログラムをユーザーに提供する際には、そのファイルを提供することになります。ただ、Javaのプログラムが機能するためには、エンドユーザーのコンピュータにJVMが必要です。したがって、Javaプログラムが CD-ROMなどのかたちで提供される際には、JVMとそのインストーラも併せてCD-ROMに納められるのが一般的です。通常、JVMは、対応するプラットフォームの異なる複数のバージョンが提供されるので、手持ちのコンピュータにJVMの入っていないエンドユーザーは、自分のコンピュータのプラットフォームに合ったJVMを選んでインストールすることになります。

Q: クラスが膨大な数になった場合はどうするのですか。膨大な数のファイルを提供するのは大変な手間だと思うのですが……。アプリケーションに関係するファイルを1つにまとめることはできないのですか。

A: できます。確かに、クラスの数が膨大な場合、それをエンドユーザーに提供するというのは困難でしょう。その場合は、.jarファイルというアーカイブにファイルをまとめることになります。.jarファイルのフォーマットはpkzipに似ています。jarファイルには、「マニフェスト」と呼ばれるテキストファイルが納められ、そのファイルの中に、「どのクラスがmain()メソッドを持っているか（つまりどのクラスからアプリケーションが起動されるか）」といった情報が納められます。

重要ポイント

- オブジェクト指向プログラミングでは、既存のコードに手を付けずにプログラムを拡張できる。
- Javaプログラムのコードは必ずいずれかのクラスに属する。
- クラスは、オブジェクト（インスタンスともいう）を作るための設計図のようなもの。
- オブジェクトを利用する際、中の仕組みがどのようになっているかを知る必要はない。仕方と結果だけがわかっていればよい。
- クラスには、情報と機能を持たせる。
- クラスの持つ情報を「インスタンス変数」と呼ぶ。
- クラスの持つ機能を「メソッド」と呼ぶ。
- クラスをテストするには、もう1つ別のクラスを作る。テストに際しては、テスト対象のクラスのオブジェクトを作成する。
- クラスは、他の（より抽象的な）クラスからインスタンス変数やメソッドを継承できる。
- Javaプログラムは、「オブジェクトどうしのやりとり」で成り立っている。

This PDF is prepared for tools@maruhouse.org, personal use only.

エクササイズ

コンパイラになってみよう

A、Bのソースコードを、コンパイラになったつもりで見てみましょう。そして、それぞれのコードが果たして正しくコンパイルできるかどうかを確認してください。もし、「コンパイルできない」と思った場合は、その理由を答え、正しくコンパイルできるように修正もしてください。コンパイルができると判断した場合は、実行結果がどのようになるかを答えてください。

A

```
class TapeDeck {

  boolean canRecord = false;

  void playTape() {

    System.out.println("tape playing");
  }

  void recordTape() {
    System.out.println("tape recording");
  }
}

class TapeDeckTestDrive {
  public static void main(String [] args){

    t.canRecord = true;
    t.playTape();

    if (t.canRecord == true) {
      t.recordTape();
    }
  }
}
```

B

```
class DVDPlayer {

   boolean canRecord = false;

  void recordDVD(){
    System.out.println("DVD recording");
  }
}

class DVDPlayerTestDrive {
  public static void main(String [] args) {

    DVDPlayer d = new DVDPlayer();
    d.canRecord = true;
    d.playDVD();

    if (d.canRecord == true) {
      d.recordDVD();
    }
  }
}
```


This PDF is prepared for tools@maruhouse.org, personal use only.

エクササイズ

プログラミングマグネット

バラバラになったJavaプログラムを並べ替え、下のような実行結果が得られるようにしてください。もし、{ 、} などが抜けているところがあれば付け加えてもかまいません。

```
d.playSnare();
```

```
DrumKit d =new DrumKit();
```

```
boolean topHat =true;
boolean snare =true;
```

```
void playSnare() {
  System.out.print("bang bang da-bang");
}
```

```
public static void main(String [] args) {
```

```
if (d.snare ==true) {
  d. playSnare();
}
```

```
d.snare =false;
```

```
class DrumKitTestDrive {
```

```
d.playTopHat();
```

```
class DrumKit {
```

```
void playTopHat() {
  System.out.print("ding ding da-ding");
}
```

```
File  Edit  Window  Help  Dance
%java DrumKitTestDrive
bang bang ba-bang
ding ding da-ding
```


This PDF is prepared for tools@maruhouse.org, personal use only.

プールパズル

右のプログラムの欠落部分に、下のプールの中のコードを当てはめ、実行結果が以下に示すようなものになるようなプログラムを完成させてください。同じコードを2回以上繰り返し使うこともできます。また、すべてのコードを使う必要はありません。

```
public class EchoTestDrive {
  public static void main(String [] args) {
    Echo e1 = new Echo();
    ______________________
    int x = 0;
    while (___________) {
      e1.hello();
      ________________________
      if (____________) {
        e2.count = e2.count +1;
      }
      if (____________) {
        e2.count = e2.count + e1.count;
      }
      x = x + 1;
    }
    System.out.println(e2.count);
  }
}
```

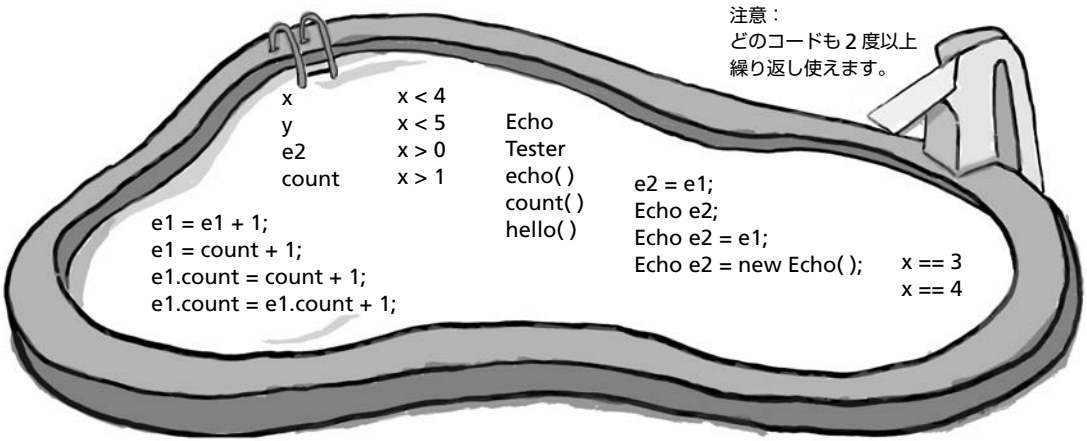
```
class ____________ {
  int _________= 0;
  void ___________ {
    System.out.println("helloooo...");
  }
}
```

実行結果

```
File  Edit  Window  Help  Implode
%java EchoTestDrive
helloooo...
helloooo...
helloooo...
helloooo...
10
```

ボーナスクエスチョン

実行結果の最後に表示される「10」を「24」にする場合、プログラムはどのように変わるか。



This PDF is prepared for tools@maruhouse.org, personal use only.

Javaプログラムはいくつかの要素から構成されます。下の説明文に当てはまる構成要素が何かを解答欄に書いてください。同じ答えを何度書いてもかまいません。

答えの候補は次の4つ：
クラス、メソッド、オブジェクト、インスタンス変数

「オブジェクト」に似ているが厳密には違う意味を持つ	**クラス**
クラスを基に作る	
設計図の役割を果たす	
クラスの持つ機能	
プログラムをユーザーに提供する際には、結局これを提供することになる	
クラスの持つ情報	
他のオブジェクトから呼び出される	
クラスは他のクラスからこれを継承できる	
ヒープに格納される	
インスタンスとも呼ばれる	
オブジェクトごとに値が異なる	
レシピに似ている	
メソッドによって値を変更できる	

This PDF is prepared for tools@maruhouse.org, personal use only.

エクササイズの答え

プログラミングマグネット

```
class DrumKit {

  boolean topHat = true;
  boolean snare = true;

  void playTopHat() {
    System.out.println("ding ding da-ding");
  }

  void playSnare() {
    System.out.println("bang bang ba-bang");
  }
}

class DrumKitTestDrive {
  public static void main(String [] args) {

    DrumKit d = new DrumKit();
    d.playSnare();
    d.snare = false;
    d.playTopHat();

    if (d.snare == true) {
      d.playSnare();
    }
  }
}
```

```
File  Edit  Window  Help  Dance
%java DrumKitTestDrive
bang bang ba-bang
ding ding da-ding
```

コンパイラになってみよう

A

```
class TapeDeck {
  boolean canRecord = false;
  void playTape() {
    System.out.println("tape playing");
  }
  void recordTape() {
    System.out.println("tape recording");
  }
}

class TapeDeckTestDrive {
  public static void main(String [] args) {

    TapeDeck t = new TapeDeck();
    t.canRecord = true;
    t.playTape();

    if (t.canRecord == true) {
      t.recordTape();
    }
  }
}
```

TapeDeckクラスのオブジェクトを作成しなければ、変数やメソッドを利用することはできません。

B

```
class DVDPlayer {
  boolean canRecord = false;
  void recordDVD() {
    System.out.println("DVD recording");
  }
  void playDVD () {
    System.out.println("DVD playing");
  }
}

class DVDPlayerTestDrive {
  public static void main(String [] args) {
    DVDPlayer d = new DVDPlayer();
    d.canRecord = true;
    d.playDVD();
    if (d.canRecord == true) {
      d.recordDVD();
    }
  }
}
```

DVDPlayerクラスに存在しないメソッドを使用していてはコンパイルできません。

This PDF is prepared for tools@maruhouse.org, personal use only.

プールパズル

```
public class EchoTestDrive {
  public static void main(String [] args) {
    Echo e1 = new Echo();
    Echo e2 = new Echo();
    int x = 0;
    while (x < 4 ) {
      e1.hello();
      e1.count = e1.count + 1;
      if (x == 3 ) {
        e2.count = e2.count + 1;
      }
      if (x > 0 ) {
        e2.count = e2.count + e1.count;
      }
      x = x + 1;
    }
    System.out.println(e2.count);
  }
}
```

```
  class Echo {
    int count = 0;
    void hello() {
      System.out.println("helloooo...");
    }
  }
```

ボーナスクエスチョンの答え

```
  Echo e2 =new Echo();
```

の代わりに

```
  Echo e2 =e1;
```

を入れる

```
File  Edit  Window  Help  Assimilate
%java EchoTestDrive
helloooo...
helloooo...
helloooo...
helloooo...
10
```

私は誰でしょう？

「オブジェクト」に似ているが厳密には違う意味を持つ　クラス

クラスを基に作る　オブジェクト

設計図の役割を果たす　クラス

クラスの持つ機能　メソッド

プログラムをユーザーに提供する際には、結局これを提供することになる　クラス

クラスの持つ情報　インスタンス変数

他のオブジェクトから呼び出される　メソッド

クラスは他のクラスからこれを継承できる　メソッド、インスタンス変数

ヒープに格納される　オブジェクト

インスタンスとも呼ばれる　オブジェクト

オブジェクトごとに値が異なる　インスタンス変数

レシピに似ている　クラス

メソッドによって値を変更できる　インスタンス変数

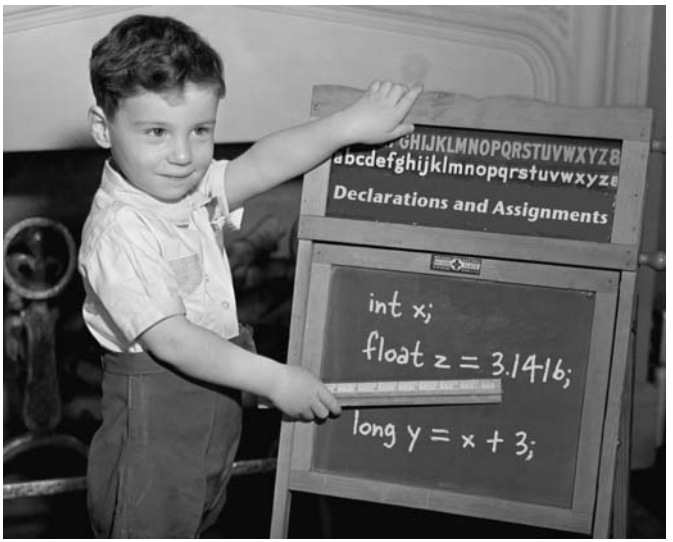
注意： クラス、オブジェクトはともに情報、機能を持ちます。厳密には、持つべき情報、機能がクラスで定義され、オブジェクトがその情報、機能を持つ、ということになりますが、今の時点ではあまり明確に区別しなくていいでしょう。

This PDF is prepared for tools@maruhouse.org, personal use only.

This PDF is prepared for tools@maruhouse.org, personal use only.

3章　プリミティブと参照

変数の基礎知識



変数には大きく分けて、プリミティブ型、参照型という2つの種類があります。また、メソッドの外で宣言されるインスタンス変数とローカル変数の2つに分けることもあります。後述するとおり、変数は、メソッドを呼び出す時の「引数」に使用することもあれば、メソッドの「戻り値」に使用することもあります。参照型の変数には、配列型、文字列（String）型、クラス型などの種類があります。こうした変数を使用すれば、整数、小数といった単なる数値だけを使用するよりも複雑なプログラムが作れます。この章では、Javaにおける変数がどのようなもので、どのように使うのか、といったことを解説します。また、前章で少し触れた「ガーベージコレクション」についても詳しく説明します。

This PDF is prepared for tools@maruhouse.org, personal use only.

注意：参照型の変数の場合、「型」というのは、オブジェクトの属する「クラス」とほぼ同義になります。たとえば、「Rabbit型のオブジェクト」と、「Rabbitクラスのオブジェクト」とは、ほぼ同じ意味である、ということです（このことについては後述します）。

変数の宣言

Javaの変数には「型」があります。この型によって、「どのような種類のデータを格納できるか」が決まるのです。たとえば、「ウサギ」型変数に「キリン」を格納するようなことはしてはいけません。「ウサギ」と「キリン」では、大きさも形も能力、行動も違います。「キリン」が格納された「ウサギ」型変数をそうとは知らずに使用するとプログラムが予測不能の動きをする恐れがあり非常に危険なのです。int型（整数型）として宣言された変数には、float型（浮動小数点数型）のデータを格納してはいけません。無理に格納すると、精度が下がり、小数点以下のデータがすべて失われます（それをわかった上で故意に格納することはある）。

変数の型と格納（代入）されるデータの型が異なっていると、多くの場合、コンパイラがそれを検知します。

```
Rabbit hopper = new Giraffe();
```

たとえば、上記のようなコードを書いた場合には、コンパイルエラーが発生します。

変数を作ることを「宣言」と呼びます。宣言に際しては、まず、その変数の型を決めることになります。先述したとおり、変数の型には大きく分けて、プリミティブ型と参照型の2種類があります。プリミティブ型というのは、整数、浮動小数点数、ブール値などの単純なデータを格納する型のことです。一方、参照型というのは、オブジェクトへの参照（と言ってもよくわからないかもしれません。詳しくは後述）を格納する型のことです。

この章では、まずプリミティブ型について解説し、次に参照型について解説します。ただ、型に関係なく、以下の2つの原則はすべての変数に当てはまるので覚えておいてください。

変数には型が必要

変数をプログラムの中で利用するためには、「どの変数を使いたいか」を指定するために名前が必要になります。型と名前は、変数の宣言に際して決めることになります。変数の宣言の例を以下に示します。

変数には名前が必要

```
int count;
```

型　名前

This PDF is prepared for tools@maruhouse.org, personal use only.

「ダブルモカをください。intサイズで」

変数は、コップにたとえるとわかりやすくなります。コップには、コーヒーカップ、ティーカップ、ビールジョッキなど、さまざまな種類があります。映画館でポップコーンを食べる時に使う大きな紙コップなどもあります。取っ手の付いているものもあれば、そうでなものもあり、陶器に金属で装飾したものなどもあります（電子レンジに絶対に入れてはいけないタイプです）。

ともかく、種類は色々ですが、コップのように何かを入れる容器であるという点はどの変数でも同じです。

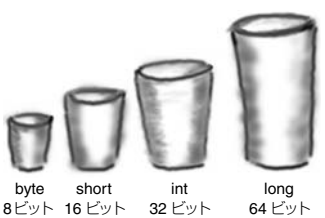
変数がプリミティブ型と参照型に分けられるということについてはすでに述べましたが、特にコップにたとえやすいのはプリミティブ型の方です。参照型をコップにたとえて説明すると、話が若干複雑になります。参照型にも後で触れますが、ここではまずプリミティブ型について見てみましょう。

プリミティブ型の変数は、「スターバックス」などのカフェで使うカップによく似ています。カフェの多くでは、カップに「ショート」、「トール」、「グランデ」などの名前を付けています。「ダブルモカのグランデ」という風に注文するわけです。

店先には、客が自分に合ったサイズを注文できるよう、各サイズのカップの見本が置いてあります。

Javaのプリミティブ型変数にも、同様に何種類ものサイズがあります。たとえば、整数を格納する変数の型には、左に示すような4つの種類がありますが、それぞれにサイズが異なっています。

コーヒーの場合のトールサイズ、ショートサイズなどと同じように、整数を格納する変数にもintサイズ、shortサイズなどがある、ということです。ただ、コーヒーの場合と少し異なるのは、変数には、個々に名前を付けるということです。コーヒーカップにコーヒーを入れる時、カップにいちいち名前は付けないのでその点では違っています。ショートサイズのカップにカフェラテを入れる代わりに、「height」という名前のint型の変数に「2486」という整数を入れる、ということをするわけです。Javaのプリミティブ型のサイズはそれぞれ、以下の図のようになっています。



プリミティブ型

型	サイズ（単位：ビット）	格納できる値の範囲
Boolean型とchar型		
boolean型		true、あるいはfalse（具体的にどのような値がtrue、falseになるかはJVMによって異なる）
char	16ビット	0～65535
数値型（すべて符号付き）		
整数		
byte	8ビット	−128～127
short	16ビット	−32768～32767
int	32ビット	−2147483468～2147483467
long	64ビット	−9223372036854775808～9223372036854775807
浮動小数点数		
float	32ビット	一定せず
double	64ビット	一定せず

プリミティブ型変数の宣言と値の代入

```
int x;
x = 234;
byte b = 89;
boolean isFun = true;
double d = 3456.98;
char c = 'f';
int z = x;
boolean isPunkRock;
isPunkRock =false;
boolean powerOn;
powerOn = isFun;
long big = 3456789;
float f = 32.5f ;
```

「f」が付いている点に注意。Javaでは、この「f」がなければ、浮動小数点数はすべてdouble型とみなされる。

This PDF is prepared for tools@maruhouse.org, personal use only.

変数に値を代入する際の注意

値が変数に合うものかどうか確認する。

サイズの小さい変数に大きい値を入れることはできません。

無理に入れようとするとデータの一部が失われることになります。イメージとしては左の写真のような具合にデータがあふれてしまうのです。コンパイラは、そうした事態の発生をできる限り防ぐため、変数と値の間の型の不一致を検知するとエラーを発生させます。

たとえば、以下のようなコードはエラーになります。

```
int x = 24;
byte b = x;
// エラーが発生
```

しかし、なぜ上記のコードを書いてはいけないのか疑問に思った人もいるでしょう。確かにint型の変数の値をbyte型の変数に代入しようとしているため、型は一致していませんが、xの値はわずか「24」なのです。これはbyte型の変数にも十分入るほどの小さい値です。これだけを見れば、このコードを書いていけない理由はないように思えます。しかし、それはただ人間がそう思うだけのことです。実は、コンパイラは実際にxの値がどのようになるかには関知しないのです。コンパイラはただ、「変数に入りきらない大きい値が代入される可能性がある」ということを検知するだけなのです。

変数に値を代入するには、以下のような方法を使用します。

- 「=」記号の後に、値をそのまま書く（x = 12、isGood = true など）
- 「=」記号の後に、値の代入された変数を書く（x = y など）
- 式を使用する（x = y + 43 など）

以下の例では、値をそのまま書いている部分を太字にしています。

`int size = 32;`	「size」という名前の int 型の変数を宣言し、値「32」を代入する。
`char initial ='j';`	「initial」という名前の char 型の変数を宣言し、値「j」を代入する。
`double d = 456.709;`	「d」という名前の double 型の変数を宣言し、値「456.709」を代入する。
`boolean isCrazy;`	「isCrazy」という名前の boolean 型の変数を宣言する（代入は行わない）。
`isCrazy = true ;`	先に宣言した変数「isCrazy」に値「true」を代入する。
`int y = x + 456;`	「y」という名前の int 型の変数を宣言し、式「x + 456」の演算結果を代入する。

自分で考えてみよう

サイズの大きい型の値を小さい型の変数に代入しようとするとコンパイラがエラーを発生させるということはすでに述べました。では、逆の場合はどうなのでしょうか。サイズの小さい型の値を大きい型の変数に代入しようとした場合は？　その場合、エラーは発生しません。
これまでにプリミティブ型の変数の型、サイズなどに関して学んだことを踏まえ、以下の練習をしてみてください。これまでに学んだことだけで何もかもがわかるというわけではありませんが、よく頭を働かせれば、少なくともこの練習問題は解けるはずです。

ヒント：コンパイラがエラーを発生させるのは、常に安全を考慮してのことです。

下の 1 ～ 13 のコードの中から、エラーが発生しないものを選び、そのコードを○で囲んでください。1 ～ 13 のコードはすべて 1 つの同じメソッドの中にあって、連続しているものとします。

```
1. int x = 34.5;
2. boolean boo = x;
3. int g = 17;
4. int y = g;
5. y = y + 10;
6. short s;
7. s = y;
8. byte b = 3;
9. byte v = b;
10. short n = 12;
11. v = n;
12. byte k = 128;
```


This PDF is prepared for tools@maruhouse.org, personal use only.

名前付けのルール

変数の宣言の際には型と名前を指定しなくてはならない、ということはすでに述べました。

型については、これまでの説明（プリミティブ型についての説明だけですが）で少しわかったと思います。

ここでは、もう一方の「名前」について触れておきましょう。名前はどのようなルールに基づいて付ければいいのでしょうか。実のところ、ルールは非常に簡単です。クラス、メソッド、変数に共通する名前付けのルールを以下にまとめておきます（多少のルール違反は許容される場合もありますが、安全のためこのルールを常に守るよう注意してください）。

- 先頭は英文字、下線（_）、あるいは「$」のうちのいずれかにする。数字で名前を始めることはできない。
- 2文字目以降には数字を使うこともできる。
- 「予約語」とされている言葉は使えない。

「予約語」とは、コンパイラが特殊な意味に解釈する言葉のことです。予約語を名前に使用するとコンパイラを混乱させることになります。

逐一説明は加えていませんが、本書にもすでに予約語が出てきています。たとえば、main()メソッドに使われていた以下のような言葉はすべて予約語です。

```
public    static    void
```

名前付けにこうした言葉は使用しない

また、プリミティブ型変数の型名を示す、以下のような言葉も予約語です。

```
boolean char byte short int long float double
```

予約語はこの他にも数多く存在します。下に一覧表を載せておきます。個々の言葉がどのように使うものなのかは今の段階では知る必要はありません。ともかく、名前付けには使えないということだけ覚えておいてください。また、今、予約語をすべて覚えなくてかまいません。無理に覚えようとしなくても、本書を読み終わる頃には自然にほとんどが頭に入っているはずです。

覚えておこう

プリミティブ型にどのようなものがあるかは頭文字をとって文章を作るなどすると覚えやすくなります。たとえば、

boolean char byte short int long float double

この頭文字から、次のような文章が作れます。

Be Careful! Bears Shouldn't Ingest Large Furry Dogs

（気を付けろ！クマに大きな毛むくじゃらのイヌを食わせるな）

自分でも作ってみてください。

B_C_B_S_I_L_F_D_

予約語一覧表（順不同）

boolean	byte	char	double	float	int	long	short	public	private
protected	abstract	final	native	static	strictfp	synchronized	transient	volatile	if
else	do	while	switch	case	default	for	break	continue	assert
class	extends	implements	import	instanceof	interface	new	package	super	this
catch	finally	try	throw	throws	return	void	const	goto	enum

予約語となっている言葉をクラス、メソッド、変数の名前に使用するとコンパイラを混乱させるので注意

This PDF is prepared for tools@maruhouse.org, personal use only.

オブジェクトを操作する

プリミティブ型の変数の宣言の仕方や、値の代入の仕方についてはすでに触れました。では、プリミティブ型でない「参照型」と呼ばれる変数の場合はどうなのでしょうか。参照型については、以下のことに注意しなくてはなりません。

- 参照型の変数に格納されるのは、オブジェクトそのものではない。
- 格納されるのは、「オブジェクトへの参照（オブジェクトリファレンスとも呼ばれる）」。
- 「参照」とは、厳密には「オブジェクトがメモリ中のどこに存在するか（オブジェクトのアドレス）」を知らせる情報のこと。
- 「参照」は、いわゆる「ポインタ」に似ている。ただ、Javaの参照の場合、プログラマは参照型変数に実際にどのような値が入っているかを知る必要はない。単に、その値が特定のオブジェクトを指し示していることを理解すればよい。JVMは参照型変数の値を使用して、オブジェクトを探すことができる。

変数にオブジェクトそのものを代入することはできません。つい、「System.out.println() メソッドにStringオブジェクト（文字列のオブジェクト）を渡す」、「このメソッドはDogオブジェクトを戻す」、「myFoo変数にFooオブジェクトを代入する」という言い方をしてしまいますが、これは正確ではないのです。オブジェクトそのものを格納しようとすれば、変数は非常に大きな、伸縮自在の「入れ物」でなければなりませんが、実際にはそうではありません。オブジェクトは常にメモリ（ヒープ）中の一か所だけに存在し、そこから動くことはありません（この点については後述）。

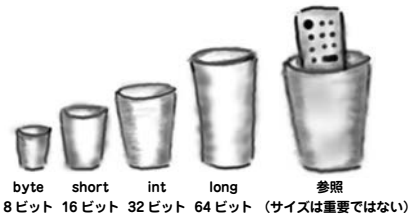
プリミティブ型の変数には、値そのものが格納されるのですが、参照型の変数に格納されるのは、「オブジェクトの存在場所」を知らせる情報（アドレス）だけです。

参照型の変数に「.」という演算子をつければ、その変数に対応するオブジェクトを利用することができます。以下に例を示します。

```
myDog.bark();
```

これは、「myDog変数に対応するオブジェクトのbark() メソッドを利用したい」という意味のコードです。「.」演算子を使用することを、オブジェクトを操作するリモコンのボタンを押すようなものと考えるとわかりやすいかもしれません。

This PDF is prepared for tools@maruhouse.org, personal use only.



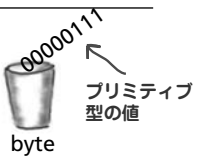
オブジェクトへの参照も変数の値である

ただ、この値は、単なる数値や文字などではなく、オブジェクトのリモコンのような役割を果たす。

プリミティブ型の変数

```
byte x = 7;
```

この場合、変数xには、10進数の「7」に対応する2進数の値「00000111」が格納される。

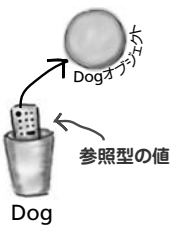


参照型の変数

```
Dog myDog = new Dog();
```

この場合、変数myDogには、Dogオブジェクトの存在場所を示す値が格納される。

Dogオブジェクトそのものは変数には格納されない。



プリミティブ型の変数の値は、数値や文字など（例：5、-26.7、a）です。
参照型の変数の値は、「オブジェクトの存在場所」を知らせる情報（アドレス）です。
アドレスも結局は数値には違いないのですが、具体的にどのような数値になるかは、JVMによって異なります。具体的な数値を知らなくてもプログラミングには支障がないので意識する必要はありません。ともかく、特定のオブジェクトへの参照を格納した変数を使えば、そのオブジェクトが利用できるということだけ理解しておけばいいのです。

参照型の変数のサイズも同様の理由から気にする必要はありません。

参照型の変数の宣言とオブジェクトの作成、参照の代入のためのコードを書く手順

```
Dog myDog = new Dog();
```

1　2　3

1 参照型の変数を宣言する

```
Dog myDog = new Dog();
```

このコードにより、「myDog」という名前の変数が作られ、JVMは、一定のメモリ領域をこの変数に割り当てます。上記の例の場合、変数の型は「Dog」になります。つまり、この変数には、Dogオブジェクトを操作するボタンを備えたリモコンだけが格納されるということです。

2 オブジェクトを作成する

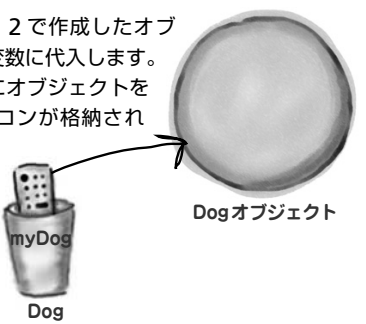
```
Dog myDog = new Dog();
```

このコードにより、JVMは、ヒープ上に新しいDogオブジェクトのための領域を作ります（この処理の詳細については9章で説明します）。

3 オブジェクトへの参照を変数に代入する

```
Dog myDog = new Dog();
```

「=」記号を使用して、2で作成したオブジェクトへの参照を変数に代入します。これによって、変数にオブジェクトを操作するためのリモコンが格納されるのです。



This PDF is prepared for tools@maruhouse.org, personal use only.

素朴な疑問に答えます

Q: 参照型の変数のサイズはどのくらいなのですか。

A: 一定していません。JVM開発チームに知り合いでもいない限り、それはわからないでしょう。参照型の変数に対応するデータがメモリ中のどこかにはあるはずですが、それにアクセスすることはできません。その必要もないでしょう(一応、64ビットくらいのデータだと思っておけばさほど間違いはないと思います)。「サイズを正確に知っておかないとメモリが心配だ」という人もいるかもしれませんが、その場合でも、変数のサイズを気にするよりは、作成するオブジェクトの数、あるいはオブジェクト自身に必要になるメモリ容量を気にした方が得策です。

Q: オブジェクトへの参照(オブジェクトリファレンス)のサイズは、対応するオブジェクトそのもののサイズにかかわらず一定ということですか。

A: そうです。使用するJVMが同じであれば、オブジェクトへの参照(オブジェクトリファレンス)のサイズは、対応するオブジェクトがどれであるかにかかわらず一定です。ただし、使用するJVMが異なれば、たとえ同じオブジェクトでもサイズは変わることがあります。オブジェクトへの参照が実際にどのようなデータになるかはJVMごとに違うからです。

Q: オブジェクトへの参照を算術演算に使うことはできますか。C言語のポインタのようにインクリメントなどは可能ですか。

A: できません。その点でJavaとC言語は大きく異なります。

Javaの真実

ゲスト:オブジェクトリファレンス

著者: オブジェクトリファレンスというのは、どのような仕事をするのですか。

オブジェクトリファレンス(以下、OR): 簡単な仕事ですよ。まあ、オブジェクトを操作するためのリモコンといった感じでしょうかね。

著者: クラスの違うオブジェクトに対応することもあるのですか。Dogオブジェクトに対応していたかと思えば、5分後にはCatオブジェクトに対応するなどということは……。

OR: もちろん、そんなことはありませんよ。Dogオブジェクトに対応すると決まれば、違うクラスのオブジェクトに対応することはありません。

著者: 同じDogクラスのオブジェクトの場合は? その場合も、特定のDogオブジェクトに対応するとなったら、他のDogオブジェクトに対応することはできないのですか。

OR: それは違います。同じDogクラスのオブジェクトであれば、DogオブジェクトAに対応していたかと思えば、5分後には別のDogオブジェクトBに対応する、ということはあり得ます。同じクラスならいいのです。たとえれば、同じ種類のテレビなら、複数あっても1つのリモコンですべて操作できるようなものです。ただし……、いいです。

著者: 何ですか。言いかけたのなら言ってください。

OR: 今、この話をするのは適切でないかもしれませんが……。まあ、ごく簡単にまとめてみましょう。実は、私が複数のオブジェクトに対応できると言いましたが、例外もあるのです。変数がfinalという修飾子を使って宣言された場合、オブジェクトリファレンスは、特定のオブジェクトにしか対応できません。たとえ、同じクラスのオブジェクトであっても、途中で対応するものを変えることはできないのです。

著者: すみません。確かにそれは今する話ではなかったですね。ともかく、finalを使って宣言をされていない限り、同じクラスの複数のオブジェクトに対応できるというわけですね。では、逆に、まったく何のオブジェクトにも対応させない、ということも可能なんでしょうか。

OR: 可能です。可能ですが……、あまりいいこととはいえませんね。

著者: と、言いますと?

OR: 変数の値をnullにするということです。これが気分のいいものじゃないんですよ。

著者: 変数にまったく値を入れない、ということですよね。

OR: いえ、nullも値ですから、値を入れないというわけではないんですよ。nullというのは、言ってみれば、何でも操作できるリモコンはあるけれど、家中にテレビが一台もない、というような状態です。結局何も操作できないわけです。一日中ボタンを押しても何も起きません。それでは、自分があまりに「役立たず」のように感じてしまって……。だいたい容量の無駄ですよ。そんなに大きな容量ではありませんが、無駄は無駄です。ただ、それだけが理由ではないんです。もし、あるクラスのオブジェクトに対応すべき変数が1つしかないにもかかわらず、その値がnullにされたら、そのクラスのオブジェクトはまったく利用できないということになってしまう。これが実は最も大きな問題なんです。

著者: なるほど、つまり……。

OR: わかるでしょう。オブジェクトと緊密な関係を結んでいたのに、突然、引き離されてしまうんですよ。その後には二度と会うことはできません。ガーベージコレクション機能によって破棄されてしまうからです。プログラマはそんなこと、想像もしないでしょう。こんなことなら、プリミティブ型の方がよかった。本当はオブジェクトリファレンスなんてイヤなんですよ。責任は重いし、仲良くなったオブジェクトとは別れなくてはならないし……。

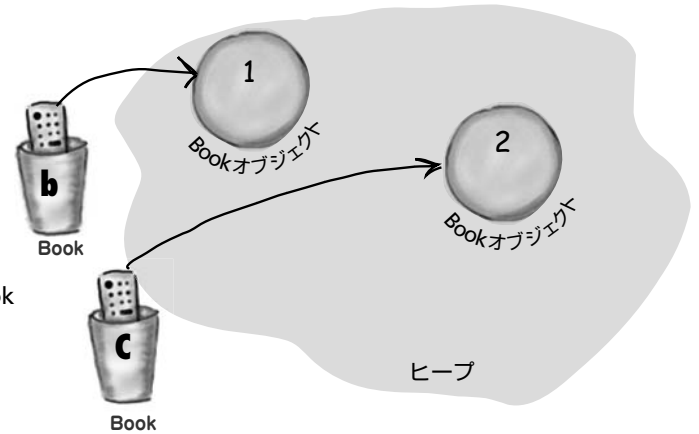
This PDF is prepared for tools@maruhouse.org, personal use only.

オブジェクト、参照、ガーベージコレクション

```
Book b = new Book();
Book c = new Book();
```

Bookクラスのオブジェクトへの参照を格納できる変数を2つ宣言し、Bookオブジェクトを2つ作成している。2つの変数にはそれぞれ、Bookオブジェクトへの参照を代入している。2つのBookオブジェクトはヒープ上に存在する。

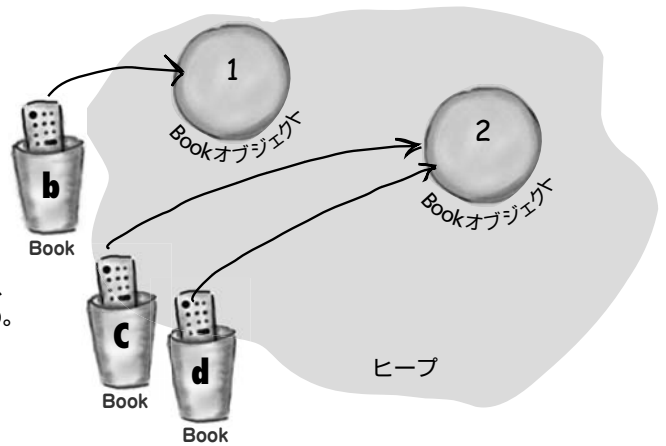
参照の数：2
オブジェクトの数：2



```
Book d = c;
```

参照型の変数dを新たに宣言し、変数cの値を代入している（新たなオブジェクトは作成していない）。これにより、変数cの値が変数dにコピーされることになる。
これで変数cとdに、同じオブジェクトへの参照が格納される。
変数cと変数dの値の内容はまったく同じ。いずれも、同じ1台のテレビを操作できるリモコンのようなもの。

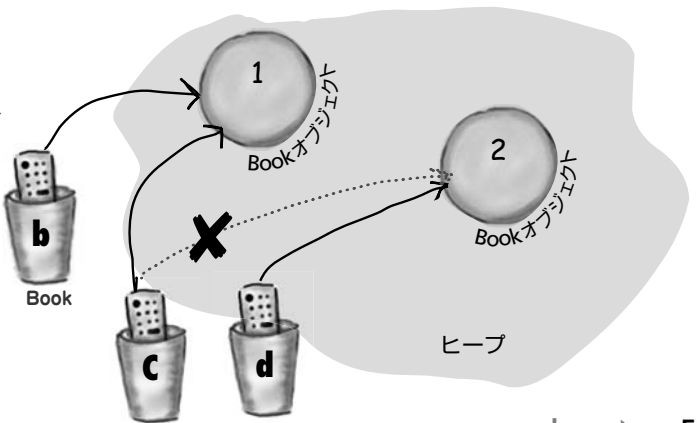
参照の数：3
オブジェクトの数：2



```
c = b;
```

変数cに変数bの値を代入している。これにより、変数bの値が変数cの値がコピーされることになる。
これで変数bとcに、同じオブジェクトへの参照が格納される。

参照の数：3
オブジェクトの数：2



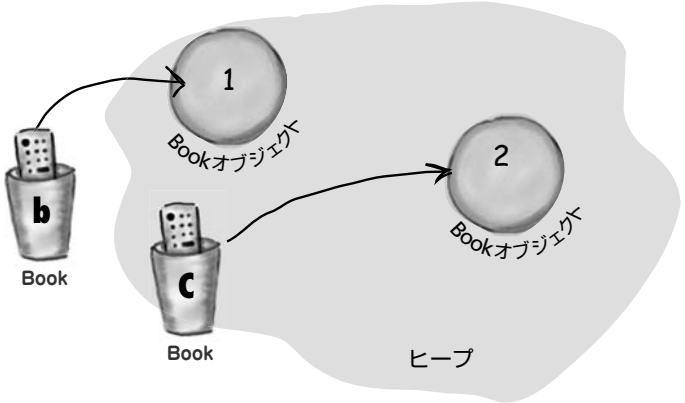
This PDF is prepared for tools@maruhouse.org, personal use only.

オブジェクトの生と死

`Book b = new Book();`

Bookクラスのオブジェクトへの参照を格納できる変数を2つ宣言し、Bookオブジェクトを2つ作成している。2つの変数にはそれぞれ、Bookオブジェクトへの参照を代入している。
2つのBookオブジェクトはヒープ上に存在する。

参照の数：2
オブジェクトの数：2

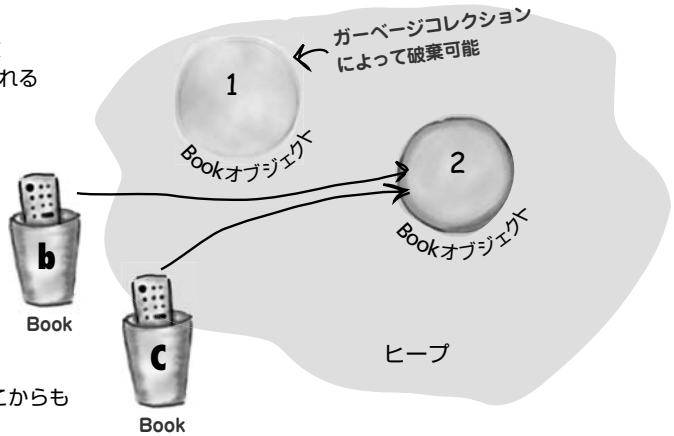


`b = c;`

変数cの値を変数bに代入している。これにより、変数cの値が変数bにコピーされ、両者には同じ値が格納されることになる。
これで変数bとcに、同じオブジェクトへの参照が格納される。オブジェクト1はどこからも参照されていない状態になる。この状態になると、ガーベージコレクション機能によって破棄することが可能。

アクティブな参照の数：2
利用可能なオブジェクトの数：1
破棄可能なオブジェクトの数：1

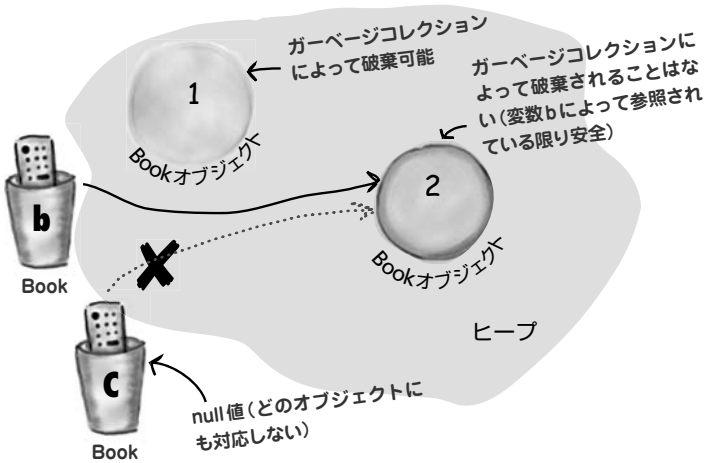
変数bによって参照されていたオブジェクト1は、どこからも参照されていない状態になり、利用できなくなる。



`c = null;`

変数cにnullを代入している。これにより、変数がどのオブジェクトにも対応しなくなる。ただし、変数はそのまま存在するので、後で新たに別の値（Bookオブジェクトへの参照）を代入することはできる。
オブジェクト2は、まだ変数bによって参照されているので、この状態のままであればガーベージコレクション機能によって破棄されることはない。

アクティブな参照の数：1
null参照の数：1
利用可能なオブジェクトの数：1
破棄可能なオブジェクトの数：1



This PDF is prepared for tools@maruhouse.org, personal use only.

配列はコップを並べたトレイのようなもの……

❶ たとえば、配列型の変数は以下のように宣言する。配列型の変数は、「配列オブジェクト」を操作するためのリモコンのようなもの。

```
int [] nums;
```

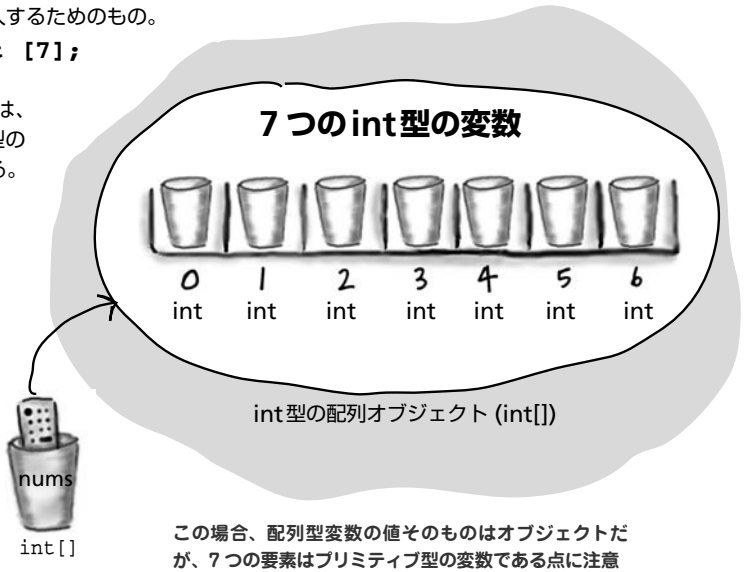
❷ 以下のコードは、要素数「7」のint型の配列を作り、先に宣言した変数に代入するためのもの。

```
nums = new int [7];
```

❸ 配列の各要素に値を代入するには、以下のようなコードを書く。int型の配列の各要素はint型の変数になる。

int型の変数が7つある

```
nums [0] = 6;
nums [1] = 19;
nums [2] = 44;
nums [3] = 42;
nums [4] = 10;
nums [5] = 20;
nums [6] = 1;
```



この場合、配列型変数の値そのものはオブジェクトだが、7つの要素はプリミティブ型の変数である点に注意

配列はオブジェクトの一種

Javaの標準ライブラリには、マップ、セット、ツリーといった優れたデータ構造（複数のデータをまとめるための手段のこと。マップ、ツリー、セットについては付録A参照）が用意されています。しかし、ともかくただ複数のデータを使いやすいかたちでまとめることができればよい、という場合は配列が便利です。配列に納めた各データは、要素の番号を指定するだけで簡単に利用できます。

配列の要素はすべてプリミティブ型、あるいは（Dogオブジェクトへの参照などを納める）参照型の変数です。同じ型の変数に代入できるものは、すべて配列の要素にも代入できます。たとえば、int型の配列（int []）の各要素には、int型の値が代入でき、Dog型の配列（Dog []）の各要素にはDogオブジェクトへの参照（Dogオブジェクトそのものではない）が代入できます。もちろん、Dogオブジェクトへの参照を代入するためには、Dogオブジェクトを作成する必要があります。詳細については次のページで解説します。

もう1つ注意すべきことは、配列の要素がプリミティブ型の変数であったとしても、配列自体は1つのオブジェクトであるという点です。

配列は、要素がプリミティブ型の変数でも、参照型の変数でも、常にオブジェクトです。要素が何であるかに関係なく、配列自体はオブジェクトなのです。プリミティブ型の値を格納する配列を使う場合には、そのことをつい忘れがちになるので気を付けなくてはなりません。

This PDF is prepared for tools@maruhouse.org, personal use only.

Dog型の配列を作る

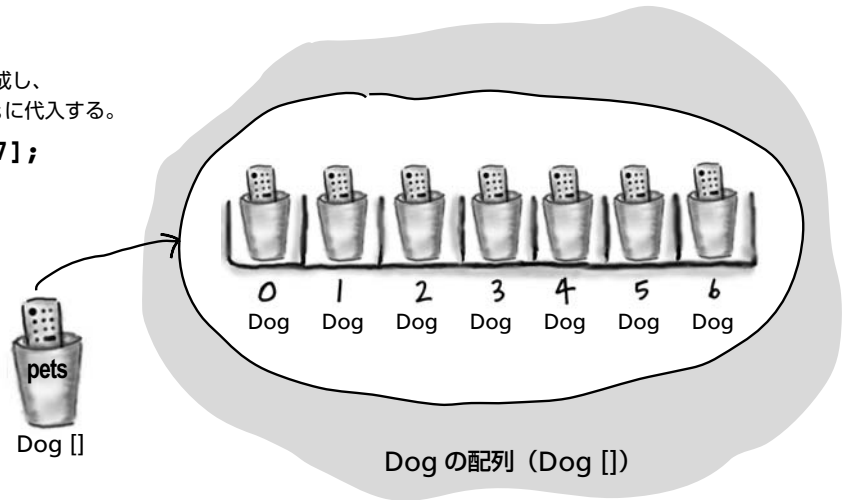
1 Dog型の配列変数を宣言する。

```
Dog [] pets;
```

2 要素数7のint型の配列を作成し、
それを先に宣言した変数petsに代入する。

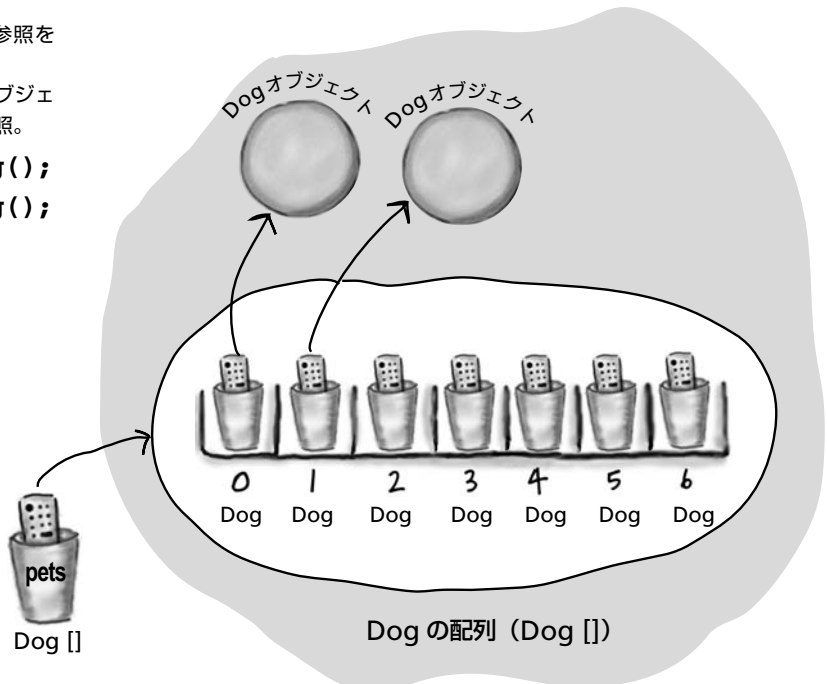
```
pets = new Dog [7];
```

注意！
Dogオブジェクトの配列には、Dogオブジェクトそのものではなく、Dogオブジェクトへの参照が格納されます。



3 Dogオブジェクトを作成し、参照を配列の要素に格納する。
配列の要素に格納されるのはオブジェクト自身ではなく、あくまで参照。

```
pets [0] = new Dog();
pets [1] = new Dog();
```



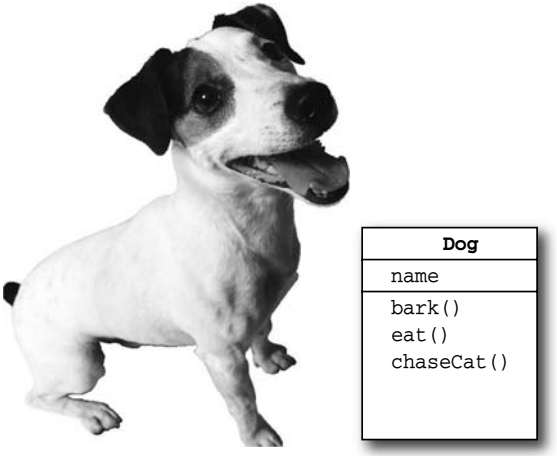
自分で考えてみよう

pets [2] にはどのような値が入るでしょうか＿＿＿＿＿＿

上記の手順「3」で作ったDogオブジェクトへの参照のうちいずれか1つをpets [3] に格納するには、どのようなコードを書けばよいですか

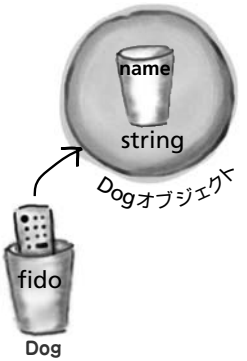
＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿

This PDF is prepared for tools@maruhouse.org, personal use only.



Dogオブジェクトの操作
（参照を利用）

```
Dog fido = new Dog();
fido.name = "Fido";
```



上記のコードでは、Dogオブジェクトを作成し、変数fidoに格納した参照と「.」演算子を利用して、name変数を操作しています*。

fido変数を利用すれば、Dogオブジェクトのbark()、eat()、chaseCat()といったメソッドも使用できます。

```
fido.bark();
fido.chaseCat();
```

Dog型の配列でのDogオブジェクトの扱い

Dogオブジェクトのインスタンス変数やメソッドを利用するには、すでに何度も述べたとおり、「.」演算子を使用します。Dogオブジェクト（への参照）が配列の要素になっている場合はどうでしょうか。その場合は、（fidoのような）変数名に直接「.」を付けるのではなく、以下のように、配列の要素の番号を書いてから「.」を付けることになります。

```
Dog [] myDogs = new Dog [3];
myDogs [0] = new Dog();
myDogs [0].name = "Fido";
myDogs [0].bark();
```

* ここでもやはり、説明を簡単にするため、「カプセル化」は行っていません。「カプセル化」については4章で詳述します。

宣言と違う型のデータを配列に格納することはできない

配列の要素には、宣言の際に指定した型の値のみ格納できます。違う型の値は格納できないのです。

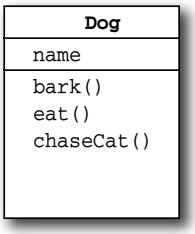
たとえば、Dog型の配列にCatオブジェクトへの参照を格納することはできません（もし、そういうことが許されるとしたら、Dogオブジェクトだと思って使ったものが、実はCatオブジェクトだった……、というようなことが起こり大変混乱してしまいます）。また、int型の配列の要素にdouble型の値を代入することはできません（先に述べたとおり、そういうことをするとデータが一部失われることになります）。ただ、int型の配列にbyte型の値を代入することは可能です。byte型の値のサイズは常にint型の器より小さいからです。これを「暗黙拡大変換」と呼びます。詳しくは後述しますが、ここでは、ともかくコンパイラは基本的に宣言と異なる型の値を許容しないと覚えておいてください。

This PDF is prepared for tools@maruhouse.org, personal use only.

```
class Dog {
  String name;
  public static void main (String [] args) {
    //Dog オブジェクトを作成し利用する
    Dog dog1 = new Dog();
    dog1.bark();
    dog1.name = "Bart";

    //Dog の配列を作る
    Dog [] myDogs = new Dog [3];
    // 配列に Dog への参照を代入する
    myDogs [0] = new Dog();
    myDogs [1] = new Dog();
    myDogs [2] = dog1;

    // 配列中の参照を利用して
    //Dog オブジェクトを使う
    myDogs [0].name = "Fred";
    myDogs [1].name = "Marge";

    //myDogs [2] の name 変数の値を画面に表示する
    System.out.print("last dog's name is ");
    System.out.println(myDogs [2].name);

    // ループを使用して
    //Dog オブジェクトの bark() メソッドを順に実行する
    int x = 0;
    while(x < myDogs.length){
      myDogs [x].bark();
      x = x + 1;
    }
  }

  public void bark() {
    System.out.println(name + "says Ruff!");
  }
  public void eat() { }
  public void chaseCat() { }
}
```

「length」を使えば、配列の要素数がわかる。

Dog クラス



実行結果

```
File  Edit  Window  Help  Howl
%java Dog
null says Ruff!
last dog's name is Bart
Fred says Ruff!
Marge says Ruff!
Bart says Ruff!
```

重要ポイント

- 変数には大きく分けて、プリミティブ型、参照型という 2 つの種類がある。
- 変数の宣言の際は名前と型を決める。
- プリミティブ型の変数とは、整数、浮動小数点数、ブール値などの単純な数値を格納する変数のこと。
- 参照型の変数には、「オブジェクトへの参照（オブジェクトリファレンスとも呼ばれる）」が格納される。「オブジェクトへの参照」とは「オブジェクトがメモリ中のどこに存在するか（オブジェクトのアドレス）」を知らせる情報のこと。
- 参照はオブジェクトを操作するためのリモコンのようなもの。「.」演算子をリモコンのボタンのように使って、オブジェクトのメソッドやインスタンス変数を利用できる。
- 参照型の変数に null 値が格納されると、その変数はどのオブジェクトにも対応しなくなる。
- 配列にはプリミティブ型の値も格納できるが、配列自身は常にオブジェクトになる。配列がプリミティブ型のデータになることはない。

This PDF is prepared for tools@maruhouse.org, personal use only.

エクササイズ

コンパイラになってみよう

A、Bのソースコードを、コンパイラになったつもりで見てみましょう。そして、それぞれのコードが果たして正しくコンパイルできるかどうかを確認してください。もし、「コンパイルできない」と思った場合は、その理由を答え、正しくコンパイルできるように修正もしてください。

A

```
class Books {
  String title;
  String author;
}

class BooksTestDrive {
  public static void main(String [] args) {

    Books [] myBooks = new Books [3];
    int x = 0;
    myBooks [0].title = "The Grapes of Java";
    myBooks [1].title = "The Java Gatsby";
    myBooks [2].title = "The Java Cookbook";
    myBooks [0].author = "bob";
    myBooks [1].author = "sue";
    myBooks [2].author = "ian";

    while (x < 3){
      System.out.print(myBooks [x].title);
      System.out.print(" by ");
      System.out.println(myBooks [x].Author);
      x = x + 1;
    }
  }
}
```

B

```
class Hobbits {

  String name;

  public static void main(String [] args) {

    Hobbits [] h = new Hobbits [3];
    int z = 0;

    while (z < 4) {
      z = z + 1;
      h [z] = new Hobbits();
      h [z].name = "bilbo";
      if (z == 1) {
        h [z].name = "frodo";
      }
      if (z == 2) {
        h [z].name = "sam";
      }
      System.out.print(h [z].name + " is a ");
      System.out.println("good Hobbit name");
    }
  }
}
```


This PDF is prepared for tools@maruhouse.org, personal use only.

エクササイズ

プログラミングマグネット

バラバラになったJavaプログラムを並べ替え、下のような実行結果が得られるようにしてください。もし、{ 、} などが抜けているところがあれば付け加えてもかまいません。

```
int y = 0;
```

```
ref = index [y];
```

```
islands [0] = "Bermuda";
islands [1] = "Fiji";
islands [2] = "Azores";
islands [3] == "Cozumel";
```

```
int ref;
while (y < 4) {
```

```
System.out.println(islands [ref]);
```

```
index [0] = 1;
index [1] = 3;
index [2] = 0;
index [3] = 2;
```

```
String [] islands = new String [4];
```

```
System.out.print("island = ");
```

```
int [] index = new int [4];
```

```
y = y + 1;
```

```
class TestArrays {

public static void main(String [] args) {
```

```
File  Edit  Window  Help  Bikini
%java TestArrays
island = Fiji
island = Cozumel
island = Bermuda
island = Azores
```


This PDF is prepared for tools@maruhouse.org, personal use only.

プールパズル

右のプログラムの欠落部分に、下のプールの中のコードを当てはめ、実行結果が以下に示すようなものになるようなプログラムを完成させてください。同じコードを2回以上繰り返し使うこともできます。また、すべてのコードを使う必要はありません。

実行結果

```
File  Edit  Window  Help  Bermuda
%java Triangle
triangle 0, area =4.0
triangle 1, area =10.0
triangle 2, area =18.0
triangle 3, area = ____
y = ____________________
```

ボーナスクエスチョン

実行結果にも欠落部分があるので、プールから適切なものを選んで埋めてください。

（ここでは、コードを書くスペースの都合上、他のクラスのテストのために使う「テスト対象のクラス名 + TestDrive」という名前のクラスは使っていません）

```
class Triangle {
  double area;
  int height;
  int length;
  public static void main(String [] args) {
    ______________
    ____________________
    while (__________){
      ________________________
      ________.height =(x + 1) * 2;
      ________.length = x + 4;
      ________________________
      System.out.print("triangle "+x+", area");
      System.out.println(" = " + _______.area);
      ______________
    }
    ____________
    x = 27;
    Triangle t5 = ta [2];
    ta [2].area = 343;
    System.out.print("y =" + y);
    System.out.println(",t5 area = "+ t5.area);
  }
  void setArea() {
    ____________=(height * length) / 2;
  }
}
```

注意：
どのコードも2度以上繰り返し使えます。

```
Triangle [ ] ta = new Triangle(4);
Triangle ta = new [ ] Triangle [4];
Triangle [ ] ta = new Triangle [4];
ta [x] = setArea();
ta.x = setArea();
ta [x].setArea();
28.0
30.0
ta = new Triangle();
ta [x] = new Triangle();
ta.x = new Triangle();
x <4
x <5
ta.x
ta(x)
ta [x]
x = x + 1;
x = x + 2;
x = x - 1;
int x;
int y;
int x = 0;
int x = 1;
int y = x;
4, t5 area = 18.0
4, t5 area = 343.0
27, t5 area = 18.0
27, t5 area = 343.0
area
ta.area
ta.x.area
ta [x].area
x
y
```

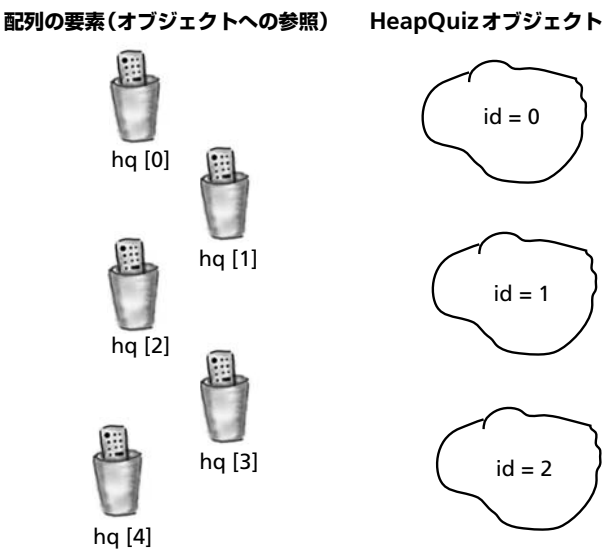

This PDF is prepared for tools@maruhouse.org , personal use only.

配列には何が……

右のJavaプログラムの実行が「//do stuff」という部分まで進んだ時、配列hq []の各要素にはどのようなデータが入るかを考えましょう。hq []はHeapQuiz型の配列なので、当然、要素にはHeapQuizオブジェクトへの参照が入るのですが、ここでは、各要素に格納される参照に対応するHeapQuizオブジェクトのid変数の値がどのようになるかを当ててもらいます。下の図で対応する要素とid変数の値を線で結んでください。対応するid変数の値がない要素に関しては線は引かなくてかまいません。

ヒント：線はだいたい、57、58ページのような感じで描くようにしてください（参照からオブジェクトに向かう矢印にする）。一度で正しい答えが出ない可能性も考えて鉛筆を使う方が賢明でしょう。

```
class HeapQuiz {
  int id = 0;
  public static void main(String [] args) {
    int x = 0;
    HeapQuiz [ ] hq = new HeapQuiz [5];
    while (x < 3 ) {
      hq [x] = new HeapQuiz();
      hq [x].id = x;
      x = x + 1;
    }
    hq [3] = hq [1];
    hq [4] = hq [1];
    hq [3] = null;
    hq [4] = hq [0];
    hq [0] = hq [3];
    hq [3] = hq [2];
    hq [2] = hq [0];
    //do stuff
  }
}
```



参照とオブジェクト（のid変数の値）を対応づける。すべての参照に対応するオブジェクトがあるわけではない。

This PDF is prepared for tools@maruhouse.org, personal use only.

消えた参照

暗い嵐の夜、Tawnyは自分のオフィスを出て、部下のプログラマたちのいる作業場に入った。彼らは例のごとく忙しそうだったが、彼女にはどうしても彼らに頼まなくてはならない仕事があった。顧客から携帯電話で動くJavaプログラムの開発を極秘で依頼されていたのだが、そのプログラムを構成するクラスにメソッドを1つ追加する必要があったのだ。携帯電話で動くため、当然、使えるメモリ容量は限られている。彼女が姿を現し、ホワイトボードに向かうと、騒がしい作業場が一瞬静かになった。Tawnyは、追加すべきメソッドがどのようなものかをホワイトボードに簡単に書き、ゆっくりと部屋を見回してから「時間がないのよ。ともかく、この中で一番メモリ容量が少なくて済むメソッドを書いた人は、明日私と一緒にマウイ島に行って、顧客先の主催する打ち上げパーティに出てもらうわ。新しいソフトウェアのインストールも手伝ってもらうけど」と言った。

翌朝、Tawnyは短いアロハドレスを着て作業場に現れ、「さあ、飛行機が出るまで時間がないから、急いで皆の作ったメソッドを確認しましょう」と笑顔で言った。まず、最初にホワイトボードに向かったのはBobだった。彼がメソッドの設計について書き始めると、Tawnyは「それはいいから核心だけを話して。問題はContactオブジェクトの扱いでしょう」と言った。そこでBobは次のようなコードをホワイトボードに書いた。

```
Contact [] ca = new Contact [10];
while (x < 10 ){   //Contactオブジェクトを10個作る
  ca [x] = new Contact();
  x = x + 1;
}
//配列caにContactオブジェクトへの参照を格納する
```

「メモリ容量が限られていることはわかっていますが、仕様からすれば10個のContactオブジェクトを個別に利用しなければならないため、これが最良の方法かと思います」とBobは言った。次に意見を述べたのはKentだ。彼は勝利を確信し、頭ではTawnyとココナッツのカクテルで乾杯しているところを思い浮かべていた。Kentは作り笑いをしながら「Bobのコードはどうもすっきりしていませんね。そう思いませんか。私のコードを見てください」と言った。

```
Contact refc;
while (x < 10 ){   //Contactオブジェクトを10個作る
  refc = new Contact();
  x = x + 1;
}
//変数refcにContactオブジェクトへの参照を格納する
```

「これでメモリがかなり節約できます。ハワイ行きは私に決定ですね」Kentはそう言って笑った。しかし、Tawnyの答えはKentの予想とは違っていたのである。「確かにあなたのコードでメモリは節約できるけど、残念ながら、一緒に来てもらうのはBobにするわ」

Kentの方がメモリを節約できるコードを書いたにもかかわらず、TawnyがBobを選んだのはなぜだろうか。

This PDF is prepared for tools@maruhouse.org , personal use only.

エクササイズの答え

プログラミングマグネット

```
class TestArrays {
  public static void main(String [] args) {
    int [] index = new int [4];
    index [0] = 1;
    index [1] = 3;
    index [2] = 0;
    index [3] = 2;
    String [] islands = new String [4];
    islands [0] = "Bermuda";
    islands [1] = "Fiji";
    islands [2] = "Azores";
    islands [3] = "Cozumel";
    int y = 0;
    int ref;
    while (y < 4) {
      ref = index [y];
      System.out.print("island = ");
      System.out.println(islands [ref]);
      y = y + 1;
    }
  }
}
```

```
File  Edit  Window  Help  Bikini
%java TestArrays
island = Fiji
island = Cozumel
island = Bermuda
island = Azores
```

コンパイラになってみよう

A

```
class Books {
  String title;
  String author;
}
class BooksTestDrive {
  public static void main(String [] args) {
    Books [] myBooks = new Books [3];
    int x = 0;
    myBooks [0] = new Books();
    myBooks [1] = new Books();
    myBooks [2] = new Books();
    myBooks [0].title = "The Grapes of Java";
    myBooks [1].title = "The Java Gatsby";
    myBooks [2].title = "The Java Cookbook";
    myBooks [0].author = "bob";
    myBooks [1].author = "sue";
    myBooks [2].author = "ian";
    while (x < 3){
      System.out.print(myBooks [x].title);
      System.out.print(" by ");
      System.out.println(myBooks [x].Author);
      x =x +1;
    }
  }
}
```

Booksオブジェクトそのものを作るコードが必要

B

```
class Hobbits {
  String name;
  public static void main(String [] args) {
    Hobbits [] h = new Hobbits [3];
      int z = -1;
      while (z < 2) {
      z = z +1;
      h [z] = new Hobbits();
      h [z].name = "bilbo";
      if (z == 1){
        h [z].name = "frodo";
      }
      if (z == 2){
        h [z].name = "sam";
      }
      System.out.print(h [z].name + " is a ");
      System.out.println("good Hobbit name");
    }
  }
}
```

配列の要素の番号は0から始まるという点に注意

This PDF is prepared for tools@maruhouse.org, personal use only.

プールパズル

```
class Triangle {
  double area;
  int height;
  int length;
  public static void main(String [] args) {
    int x = 0;
    Triangle [ ] ta = new Triangle [4];
    while (x < 4 ) {
      ta [x] = new Triangle();
      ta [x].height = (x + 1) * 2;
      ta [x].length = x + 4;
      ta [x].setArea();
      System.out.print("triangle "+x+", area");
      System.out.println(" = " + ta[x].area);
      x = x + 1;
    }
    int y = x;
    x = 27;
    Triangle t5 = ta [2];
    ta [2].area = 343;
    System.out.print("y =" +y);
    System.out.println(", t5 area = "+ t5.area);
  }
  void setArea() {
    area = (height * length) / 2;
  }
}
```

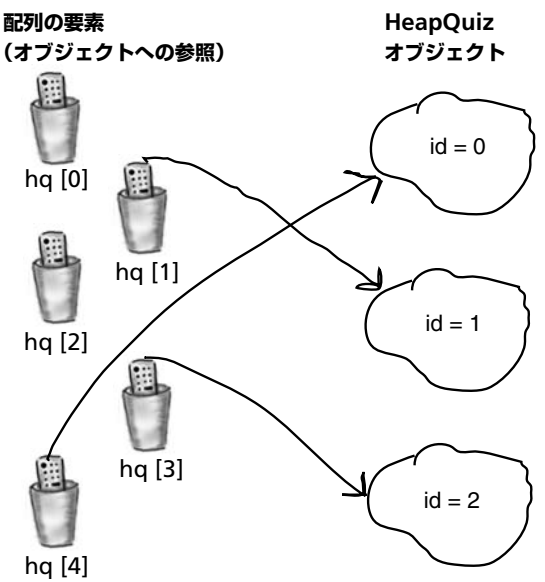
```
File  Edit  Window  Help  Bermuda
%java Triangle
triangle 0, area = 4.0
triangle 1, area =10.0
triangle 2, area =18.0
triangle 3, area =28.0
y = 4, t5 area = 343
```

消えた参照

Tawnyは、Kentの書いたコードに重大な欠陥を発見したのです。KentのコードがBobのコードよりも使用する参照型変数が少ないのは確かですが、せっかく10個のContactオブジェクトを作っても、Kentのようなコードを書いてしまっては最後に作ったContactオブジェクト以外、まったく利用できなくなります。Kentのコードでは、ループで処理が繰り返される度に、前に変数に格納されたオブジェクトへの参照が削除され、新たな参照が変数に代入されます。つまり、参照が削除されたオブジェクトは利用できなくなるのです。このコードは何の役に立たないと言ってもいいでしょう。

（Bobの書いたコードのおかげでソフトウェアは素晴らしいものになり、喜んだ顧客のはからいでTawnyとBobは予定より一週間長くハワイで過ごすことができました。本書を最後まで読めば、きっとあなたにもそんな幸運が舞い込むはずです）

配列には何が……



This PDF is prepared for tools@maruhouse.org, personal use only.

This PDF is prepared for tools@maruhouse.org, personal use only.

4章　メソッドとインスタンス変数

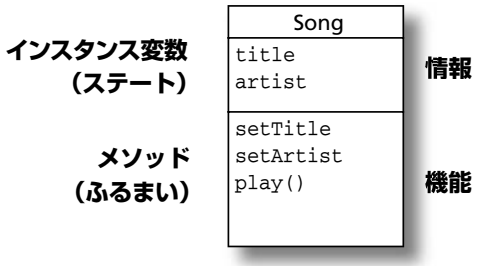
オブジェクトの「ふるまい」

オブジェクトにインスタンス変数とメソッドがあるということはすでに説明しました。インスタンス変数には、「そのオブジェクトがどのようなものであるか」という情報が納められるのですが、その情報のことを「ステート（あるいは状態）」と呼ぶことがあります。このステートは、メソッドの持つ機能に影響を与え、また逆にメソッドもステートに影響を与えます。この章では、ステートとメソッドの関係について詳しく見てみましょう。先述のとおり、インスタンス変数の値は、同じクラスのオブジェクトでも個々に変わります。たとえば、DogオブジェクトAのname変数の値は「Fido」で、weight変数の値は「700」だけれども、DogオブジェクトBでは、name変数の値が「Killer」で、weight変数の値が「9」だ、ということが起きるわけです。そして、Dogクラスにmake Noise()というメソッドがあった場合、DogオブジェクトAのmakeNoise()メソッドの方が、DogオブジェクトBのmakeNoise()メソッドより大きな音を出す、ということもあります（大きいイヌの方が通常、小さいイヌより声が大きいので）。つまり、インスタンス変数の値、ステートが、メソッドの機能に影響を与えているのです。見方を変えれば、メソッドがインスタンス変数の値を利用して機能しているとも言えます。メソッドのコードは、たとえば、weight変数の値が「5」増えるごとに鳴き声を少し大きくする、というようなものにしておきます。

This PDF is prepared for tools@maruhouse.org, personal use only.

クラスにはインスタンス変数とメソッドがある

クラスはオブジェクトの設計図のようなものです。オブジェクトは、クラスのコードに従って作られます。インスタンス変数には、オブジェクト（インスタンス）ごとに異なる値が格納される、ということはすでに述べました。では、メソッドはオブジェクトごとに異なるということはないのでしょうか。



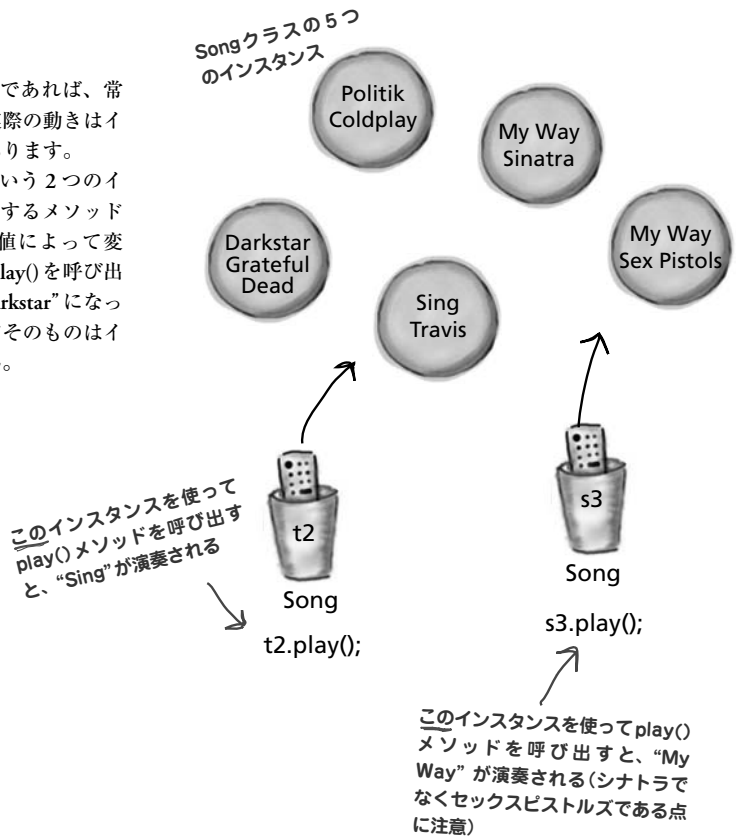
メソッドの機能はオブジェクトごとに異なるのか？

……異なることはあり得ます*。

同じクラスのオブジェクト（インスタンス）であれば、常に同じメソッドを持つのですが、メソッドの実際の動きはインスタンス変数の値によって異なる可能性があります。

右に示したSongクラスには、title、artistという2つのインスタンス変数があります。play()は曲を演奏するメソッドですが、どの曲を演奏するかは、title変数の値によって変わってきます。つまり、どのインスタンスのplay()を呼び出すかによって、曲が“Politik”になったり、“Darkstar”になったりするわけです。しかし、メソッドのコードそのものはインスタンスによって変わるわけではありません。

```
void play() {
  soundPlayer.playSound(title);
}
```




```
Song t2 = new Song();
t2.setArtist("Travis");
t2.setTitle("Sing");
Song s3 = new Song();
s3.setArtist("Sex Pistols");
s3.setTitle("My Way");
```

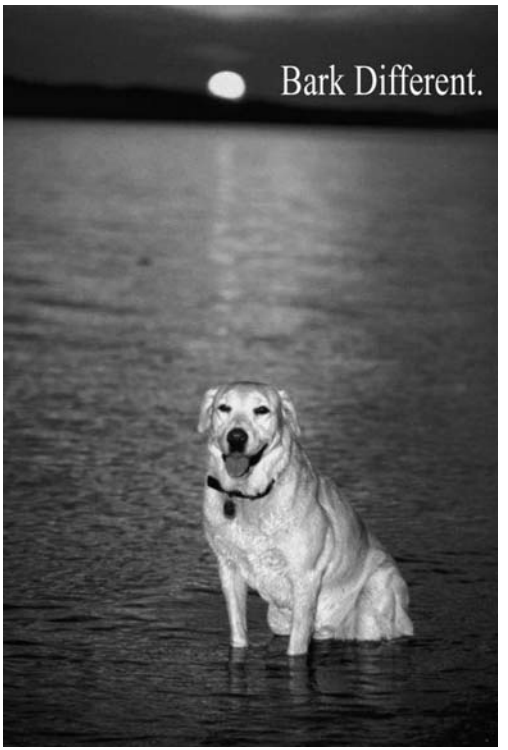
* 歯切れの悪い文ですが、意味はすぐにわかるはずです。

This PDF is prepared for tools@maruhouse.org, personal use only.

メソッドの機能に影響するインスタンス変数（ステート）

ステートがメソッドの機能に影響する例を見てみましょう。

以下のようなインスタンス変数とメソッドを備えたDogクラスがあるとします。bark()メソッドの動き（画面に表示される文字列）が、size変数の値によって変わるのがわかるでしょう。




```
class Dog {
  int size;
  String name;

  void bark() {
    if (size > 60) {
      System.out.println("Wooof!Wooof!");
    } else if (size > 14) {
      System.out.println("Ruff!Ruff!");
    } else {
      System.out.println("Yip!Yip!");
    }
  }
}
```

Dog
size name
bark()

```
class DogTestDrive {

  public static void main (String [] args) {
    Dog one = new Dog();
    one.size = 70;
    Dog two = new Dog();
    two.size = 8;
    Dog three = new Dog();
    three.size = 35;

    one.bark();
    two.bark();
    three.bark();
  }
}
```

```
File  Edit  Window  Help  Playdead
%java DogTestDrive
Wooof! Wooof!
Yip! Yip!
Ruff! Ruff!
```


This PDF is prepared for tools@maruhouse.org, personal use only.

メソッドに値を渡す

他の言語でも多くの場合は似たようなことができるのですが、Javaでは、メソッドに「値を渡す」ということができます。メソッドに渡す値は、以下のように()の中に書きます。

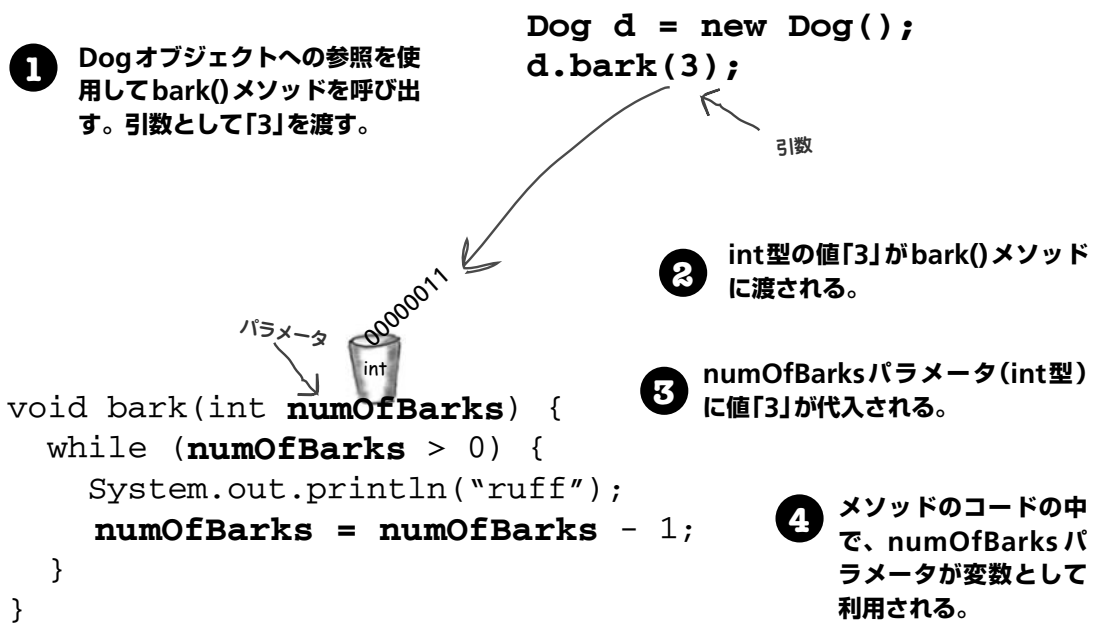
```
d.bark(3);
```

メソッドに渡す値のことを「引数」、あるいは「パラメータ」などと呼ぶこともあります。どちらの言葉を使うかは、その人のバックグラウンドや好みによって変わってきます。(本書を決して読むことのないような)コンピュータサイエンスの研究者であれば、言葉の厳密な定義も重要なのでしょうが、本書では特にそうしたことを重要視することはしません。今のところは、自分さえわかればどのような呼び方をしていてもいいでしょう(極端に言えば、引数やパラメータでなくてもよい)。ただ、一応、著者が本書で次のように区別をしているということだけは覚えておいてください。

メソッド自身のコードの中にあるものを「パラメータ」、
メソッドを呼び出すコードの中にあるものを「引数」と呼ぶ。

つまり、メソッドに渡す「値」そのものは「引数」と呼ぶことになります。この引数が、メソッドの中の「パラメータ」に代入されるのです。パラメータは変数の一種であるとも言えます。他の変数と同様、型と名前を決めて宣言されます。そして、この変数は、メソッドの中でも利用できます。

引数として渡す値の型は、対応するパラメータがどのように宣言されているかによって変わります。宣言の際に決められた型の値のみ引数として渡せるのです。



This PDF is prepared for tools@maruhouse.org, personal use only.

メソッドは値を戻す

メソッドは値を受け取るだけでなく、「値を戻す」ということもできます。メソッドはどれも基本的に値を戻し、メソッド作成の際には戻す値（戻り値という）の型を指定するのですが、以下のようにvoidという型を指定した場合だけは例外で何の値も戻さなくなります。本書でここまで紹介してきたメソッドは戻り値の型がvoidになったものばかりでした。

```
void go() {
}
```

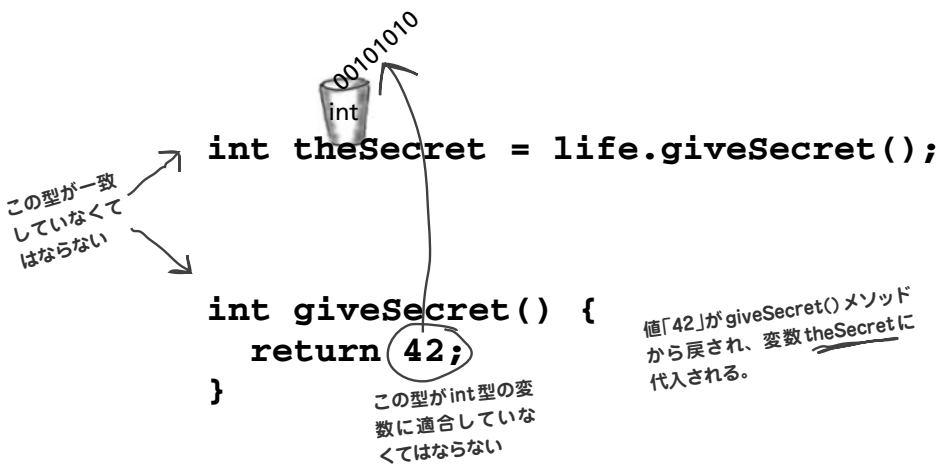
メソッドの戻り値の型はたとえば以下のように指定します。

```
int giveSecret() {
  return 42;
}
```

メソッドの戻り値の型を指定したら、必ずその型（あるいは指定された型と互換性のある型。詳細は7、8章を参照）の値を戻すことになります。

指定と異なる型の値を戻すことを基本的にコンパイラは許容しない

戻り値の型は指定どおりに！



This PDF is prepared for tools@maruhouse.org, personal use only.

メソッドには、複数のパラメータを宣言することもできる

メソッドには、複数のパラメータを宣言することもできます。その場合、パラメータとパラメータの間はカンマ(,)で区切ります。複数のパラメータが宣言されたメソッドには、複数の引数を渡すことができますが、その場合も引数と引数の間はカンマ(,)で区切ります。引数の型、指定の順序は、パラメータの宣言に合わせます。

2つのパラメータを持つメソッドを呼び出す(2つの引数を渡す)。

```
void go() {
  TestStuff t = new TestStuff();
  t.takeTwo(12,34);
}

void takeTwo(int x, int y) {
  int z = x + y;
  System.out.println("Total is " + z);
}
```

引数を渡す順序はパラメータの宣言に合わせる。最初に指定された引数は1つ目のパラメータに、2番目に指定された引数は2つ目のパラメータ……、という具合に代入されることになる。

引数の指定には、パラメータの宣言と型が合っていれば変数も使用できる。

```
void go() {
  int foo = 7;
  int bar = 3;
  t.takeTwo(foo, bar);
}

void takeTwo(int x, int y) {
  int z = x + y;
  System.out.println("Total is " + z);
}
```

変数foo、barの値がパラメータx、yに代入される。つまり、ここでは変数xの値とfooの値(いずれも7)、変数yとbarの値(いずれも3)は同じになるということ。

zの値は、変数fooの値とbarの値を足したものと同じになる。

This PDF is prepared for tools@maruhouse.org, personal use only.

引数として指定された値の「渡され方」

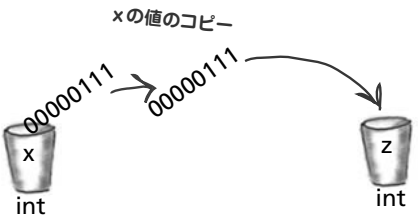
Java では指定された値のコピーが渡される（これを「値渡し」という）

```
int x = 7;
```

❶ int型の変数を宣言し、値「7」を代入する。すると、変数xに、値「7」に対応するビットパターンが格納される。

```
void go(int z) { }
```

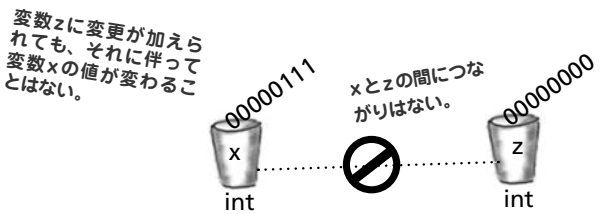
❷ int型のパラメータ「z」を持つメソッドを宣言する。




```
foo.go(x);
```

```
void go(int z){ }
```

❸ go()メソッドを呼び出し、変数xの値を引数として渡す。変数xのビットパターンは、パラメータzにコピーされる。




```
void go(int z){
z = 0;
}
```

❹ 引数として渡された値は、単なるコピーなので、goメソッドの中でzの値に変更を加えても、xの値が変化することはない。

go()メソッドの中で変数xの値を変更することはできない。

This PDF is prepared for tools@maruhouse.org, personal use only.

素朴な疑問に答えます

Q: 引数としてプリミティブ型の値ではなく、オブジェクトを指定するとどうなるのですか。

A: 詳しくは次章以降で説明しますが、ここまで読めば、答えはわかるのではないかと思います。Javaでは、引数として指定された値がすべて「コピー」されてメソッドに渡されます(これを「値渡し」と言う)。注意すべきなのは、「値」というのが厳密には変数に格納された「ビットのパターン」を意味するということです。また、すでに説明したとおり、変数にオブジェクトそのものを格納するということはできません。格納するのは「オブジェクトへの参照(オブジェクトリファレンス)」です。引数としてオブジェクトへの参照を指定すると、メソッドには、「参照のコピー」が渡されます。

Q: メソッドに複数の戻り値を指定することはできますか。また、何らかのかたちでメソッドから複数の値を戻すということはできますか。

A: 戻り値として指定できるのは常に1つですが、複数の値を戻すということは、配列を使えば不可能ではありません。たとえば、int型の値を3つ戻したいという場合は、int型の配列を戻り値として指定すればいいのです。配列に3つの値を代入すれば、結果的に3つの値が戻ります。実は、型の違う複数の値を戻すことも不可能ではないのですが、その方法はやや複雑になります。詳しくは次章以降に説明します。

Q: 戻り値の型は必ず宣言と同じでなければならないのですか。

A: 宣言とまったく同じでなくても、「暗黙拡大変換」の行われる型の値であれば使用できます。たとえば、宣言がint型の場合、byte型の値は戻せます。byte型のサイズは、必ずint型のサイズより小さくなるためです。宣言された型の方が、戻す型よりサイズが小さい場合は、明示的に「キャスト」という操作を行います。

Q: メソッドの戻り値に対しては必ず何か操作を行わなくてはならないのですか。戻り値を無視することもできるのですか。

A: Javaでは、メソッドが値を戻しても、それを利用せず、無視するということができます。void宣言されていないメソッドを呼び出せば、必ず値が戻されますが、その値は利用しなくてもかまわないのです。実際に、メソッドで行われる処理だけが必要で、戻り値は必要ではない、というケースはあります。Javaでは、そのような場合に、戻り値を変数に代入するなどの操作は必要ないのです。

覚えておこう

戻り値、引数(パラメータ)の型に注意

戻り値の型としてRabbitオブジェクトを指定した場合、Giraffeオブジェクトを戻すことはできません。引数(パラメータ)に関しても同じです。パラメータの型としてRabbitオブジェクトを指定した場合、Giraffeオブジェクトを引数として渡すことはできません。

重要ポイント

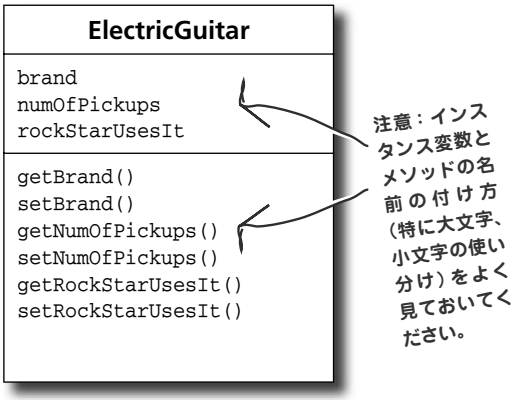
- クラスはステートと機能を持つ。
- ステートはインスタンス変数に格納される。
- 機能はメソッドのコードによって決められる。
- メソッドの動きはインスタンス変数の値で変わることがある。
- メソッドを呼び出す際は、メソッドに値を渡すことができる。この値を「引数」と呼ぶ。引数は複数渡せる場合もある。
- 引数の数と型は、メソッドのパラメータの宣言に合わせる。
- 引数、戻り値の型は、基本的にメソッドの宣言に合わせるが、宣言でint型がされた場合に、byte型の値を使用することは可能。型のサイズが宣言で使用されたものより小さく、「暗黙拡大変換」が行われるため。明示的に「キャスト」の処理を行えば、宣言の型よりサイズの大きい型の値を使用することも可能。
- 引数には、渡す値そのもの(2、'c' など)を指定することも、渡す値を格納した変数を指定することもできる。変数の型は宣言に合わせる。たとえば、宣言でint型が指定されていればint型の変数を使用する。引数として指定できるものは他にもあるが、それについては後述。
- メソッドを作成する際は戻り値の型を指定する必要がある。戻り値がない場合はvoid型を指定する。
- メソッドの戻り値の型がvoidでない場合は、必ず何らかの値(型は宣言で指定された型に適合させる)を戻す。

This PDF is prepared for tools@maruhouse.org, personal use only.

パラメータ、戻り値の有効な使い方

パラメータや戻り値がどのようなものであるかはひととおりわかったと思いますので、次にその有効な使い方を紹介しましょう。ここで紹介するのは、「ゲッター（Getter）」、「セッター（Setter）」などと呼ばれるタイプのメソッドです。専門家の間では「アクセサ」、「ミューテータ」などとも呼ばれるようですが、どの言葉が正しいかということはひとまず考えなくていいでしょう。取りあえず、本書では「ゲッター」、「セッター」と呼ぶことにします。

ゲッター、セッターはいずれもインスタンス変数の値に関係するメソッドです。簡単にまとめれば、ゲッターがインスタンス変数の現在の値を戻すメソッド、セッターがインスタンス変数に新たな値を設定するメソッド、ということになります。セッターを呼び出す際には引数として、インスタンス変数に設定したい値を指定します。すると、その値がインスタンス変数の新たな値となるのです。




```
class ElectricGuitar {

  String brand;
  int numOfPickups;
  boolean rockStarUsesIt;

  String getBrand() {
    return brand;
  }

  void setBrand(String aBrand) {
    brand = aBrand;
  }

  int getNumOfPickups() {
    return numOfPickups;
  }

  void setNumOfPickups(int num) {
    numOfPickups = num;
  }

  boolean getRockStarUsesIt() {
    return rockStarUsesIt;
  }

  void setRockStarUsesIt(boolean yesOrNo) {
    rockStarUsesIt = yesOrNo;
  }
}
```

This PDF is prepared for tools@maruhouse.org, personal use only.

カプセル化
データを危険から守る

本書でここまでに紹介してきたコード例には、オブジェクト指向言語のコードとして大きな問題がありました。言ってみれば、飲み物持参のパーティに飲み物を持っていかないのと同じくらいおかしなことをしていたのです(あまりよいたとえではないかもしれませんが……)。

問題とは何でしょうか？

それは「データを保護していない」ということです。

これまでに紹介したようなコードでは、他人に勝手にデータを覗かれたり、修正されたりすることを防止できないのです。

ここでの「データ」とは、正確には「インスタンス変数の値」を指します。つまり、「データを保護していない」というのは、インスタンス変数の値を誰にでも参照、修正できる状態で放置していたということです。たとえば、誰もが以下のようなコードを書くことで、インスタンス変数の値を修正できてしまう状態になっていたのです。

```
theCat.height = 27;
```

これは非常に危険なことです。仮に、悪意を持った人が勝手にインスタンス変数の値を以下のように変更したとしたらどうでしょうか。

```
theCat.height = 0;
```

このような操作をされると危険！

このような不正な操作を防ぐには、インスタンス変数を操作するための「セッター」を用意し、誰もが必ずそのセッターを使用しなければならないようにする必要があります。「.」演算子を使ってインスタンス変数を直接操作することはできないようにするのです。

誰もが「セッター（セッターメソッド）」を使うと決まっていれば、以下のようなコードで、変数に不適切な値が設定されることを防止できる。

```
public void setHeight(int ht) {
  if (ht > 9) {
    height = ht;
  }
}
```

変数の値が適切かどうかをチェックするコードを入れる。

This PDF is prepared for tools@maruhouse.org, personal use only.

データの保護

では、セッターを使う以外のデータ修正の手段をなくすには、具体的にどうすればいいのでしょうか。実のところ、それはさほど難しいことではありません。

それにはpublic、privateといった「アクセス修飾子」を使用します。「public」というアクセス修飾子は、本書でもすでにmainメソッドなどに使用しているので、何度も目にしているはずです。

データを不正な操作から保護することを「カプセル化」と言いますが、このカプセル化のためのアクセス修飾子の使い方には、基本となるルールがあります。それは、「インスタンス変数はprivateを使って宣言し、ゲッター、セッターはpublicを使って宣言する」というものです。常にこのルールが適用できるというわけではないのですが、まず、このルールを覚えてください。

デザインについての知識やJavaのプログラミングの技術のある方なら、もっと違ったアプローチも考えられるかも知れませんが、今のところは次のように考えておけば無難でしょう。

インスタンス変数はprivateを使って宣言する。

ゲッターとセッターはpublicを使って宣言する。

カプセル化を怠ると、前ページの「theCat.height = 0;」というような操作を許すことになる。

Javaの真実

ゲスト：オブジェクト　テーマ：カプセル化

著者：カプセル化はなぜそんなに重要なのですか。

オブジェクト：たとえば、500人の前で話していて、突然自分が裸であることに気付いたとします。「カプセル化していない」というのは、まさにそんな感じなんですよ。

著者：なるほど……。ちょっとピンと来ませんが……、まあ、いいでしょう。ともかく裸になったような気がするわけですね。でも、外部からちょっとデータが見えるくらいのことがそれほど大変なことなんでしょうか。

オブジェクト：「大変なことか？」ですって？これは驚いた（笑い出す）。皆さん、聞きましたか？　「大変なことか？」ですか！　こりゃまいった（笑い転げる）。

著者：何がそんなにおかしいんですか。もっともな質問だと思うんですが……。

オブジェクト：わかりました。説明しましょう。それはですね……（また笑い出す。止められない）。

著者：水でも持ってきましょうか。

オブジェクト：いやいや。いいです。失礼しました。深呼吸をすれば大丈夫です。さあ、続けましょう。

著者：カプセル化をすれば、どんなことから身を守れるんですか。

オブジェクト：簡単に言うと、カプセル化によって、インスタンス変数の値を勝手に不適切なものに変えることができなくなるということです。

著者：「不適切」というのはどういう意味ですか。具体的に言ってみてください。

オブジェクト：そうですね。一言では言いにくいのですが……。インスタンス変数というのは、ある想定のもとで使われていることが多いんです。たとえば、「値が負の数にはならない」とか。「バスルームの数」や「飛行機の速度」といったことを表す変数なら値が負の数になることはあり得ないですよね。「誕生日」、「バーベルの重さ」、「携帯電話の番号」などを表す変数でも事情は同じです。

著者：なるほどわかります。ということは、カプセル化というのは、変数の値に一定の制限を設けることでもあるんですね。

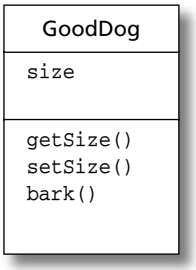
オブジェクト：そうです。インスタンス変数の操作を必ず「セッター」を使って行わせることで、そうした制限を設けることができるんです。セッターで値が適切かどうかをチェックするというか……。適切でない値が設定されそうになった場合には例外を発生させるか、適切な値（設定しようとした値に最も近いもの）に変更するわけです（値がクレジットカード番号などである場合には例外を発生させる）。すべてはセッターメソッドの作り方だいです。ただしインスタンス変数をアクセス修飾子「public」を使って宣言した場合は別ですが。

著者：でも、時々、何のチェックもしないセッターメソッドもありますよね。そういうセッターメソッドを作るのは、不要なオーバーヘッドを生じさせて、パフォーマンスを悪化させるだけではないのですか。

オブジェクト：セッター（ゲッターもそうですが）のメリットはそれだけではないんです。たとえチェックをするコードがなくても、取りあえずセッターを作っておけば、後でチェックが必要と判断した場合も、他人のコードに影響を与えることなしに改造ができるんです。たとえば、インスタンス変数を「public」を使って宣言したとして、その変数を社内の半分の人がプログラムに利用していたとしましょう。ある日突然そのインスタンス変数をセッターメソッドで操作するように変えてしまったとしたら、社内の半分の人のプログラムはそれで動かなくなってしまいます。でも、カプセル化を行っていれば、方針変更は非常にスムーズにできます。誰もそれで困ることはありません。パフォーマンスの問題はこのメリットに比べれば取るに足りないものでしょう。

This PDF is prepared for tools@maruhouse.org, personal use only.

GoodDogクラスをカプセル化する



インスタンス変数をprivate宣言する

ゲッター、セッターはpublic宣言する

```
class GoodDog {

  private int size;

  public int getSize() {
    return size;
  }

  public void setSize(int s) {
    size = s;
  }

  void bark() {
    if (size > 60) {
      System.out.println("Wooof! Wooof!");
    } else if (size > 14){
      System.out.println("Ruff! Ruff!");
    } else {
      System.out.println("Yip! Yip!");
    }
  }
}

class GoodDogTestDrive {

  public static void main (String [] args) {
    GoodDog one = new GoodDog();
    one.setSize(70);
    GoodDog two = new GoodDog();
    two.setSize(8);
    System.out.println("Dog one:" +one.getSize());
    System.out.println("Dog two:" +two.getSize());
    one.bark();
    two.bark();
  }
}
```

セッター、ゲッターは最初は単純なものにし、後から複雑なものに変えることができる。それによって他のプログラムに影響を与えることはない

メソッドを呼び出すコードは値の代わりに使用することができる。その場合、結果的にメソッドの戻り値を利用することになる(値とメソッドの戻り値の型は同じでなくてはならない)。

値を使ったコード：

int x =3 + 24;

メソッドの呼び出しを使ったコード：

int x =3 + one.getSize();

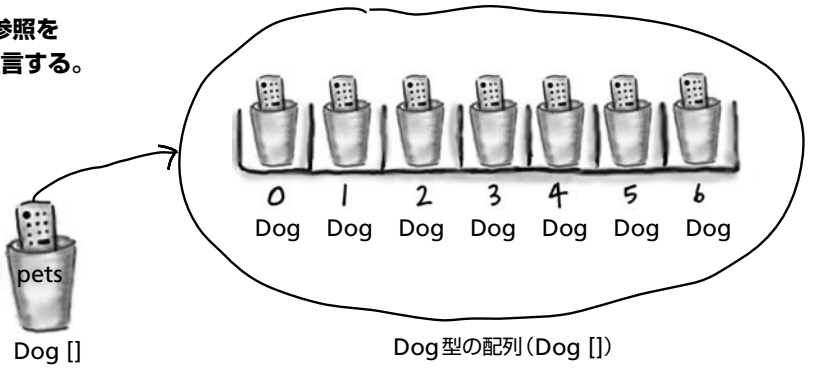
This PDF is prepared for tools@maruhouse.org, personal use only.

配列でのオブジェクトの扱い

配列でのオブジェクトの扱いは、通常の変数の場合とさほど変わりません。ただ、オブジェクトへの参照が変数に納められた場合と、配列の要素に納められた場合とでは利用の仕方が若干異なります。以下に例を示しますので確認してください。

1 Dogオブジェクトへの参照を7つ格納できる配列を宣言する。

```
Dog [] pets;
pets = new Dog[7];
```

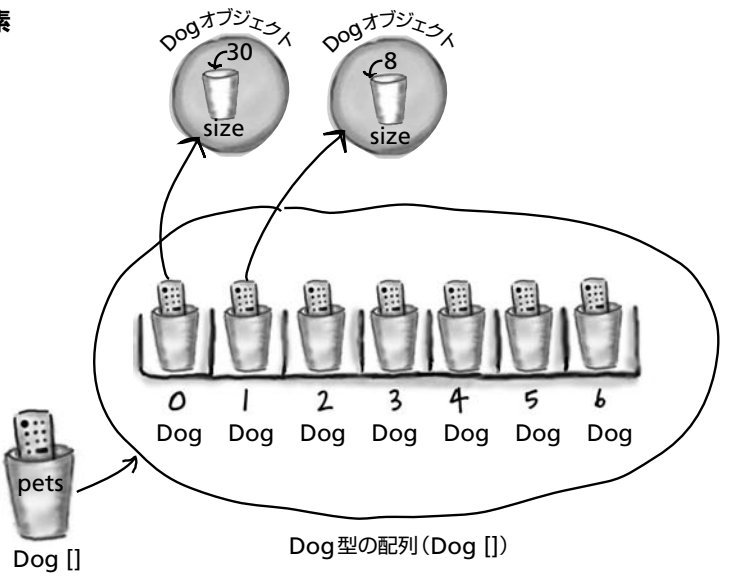


2 Dogオブジェクトを2つ作成し、参照を配列の最初の2つの要素に代入する。

```
pets[0] = new Dog();
pets[1] = new Dog();
```

3 2つのオブジェクトのメソッドを呼び出す

```
pets[0].setSize(30);
int x = pets [0].getSize();
pets[1].setSize(8);
```



This PDF is prepared for tools@maruhouse.org, personal use only.

インスタンス変数の宣言と初期化

すでに述べたとおり、変数を宣言する際には必ず名前と型を決定しなければなりません。

```
int size;
String name;
```

また、宣言と同時にその変数の初期値を設定する（代入する）ことも可能です。

```
int size = 420;
String name = "Donny";
```

では、もしインスタンス変数の初期値を設定せずに、ゲッターメソッドを使用してその変数の値を取得しようとした場合には、どのようなことが起きるのでしょうか。初期化する前のインスタンス変数の値はどのようになっているのでしょうか。

インスタンス変数にはデフォルト値がある。初期化を行わなくても、セッターメソッドを使用して値の設定を行わなくても、このデフォルト値が自動的に代入される。

デフォルト値：

整数を格納する変数	0
浮動小数点数を格納する変数	0.0
boolean型の変数	false
参照型の変数	null

```
class PoorDog {

  private int size;
  private String name;

  public int getSize() {
    return size;
  }
  public String getName() {
    return name;
  }

}

public class PoorDogTestDrive {
  public static void main (String [] args) {
    PoorDog one = new PoorDog();
    System.out.println("Dog size is " + one.getSize());
    System.out.println("Dog name is " + one.getName());
  }
}
```

2つのインスタンス変数を宣言し、初期化をしない

戻り値はどうなるか？

このようなコードは問題ないのか。コンパイルはできるのか

```
File  Edit  Window  Help  CallVet
%java PoorDogTestDrive
Dog size is 0
Dog name is null
```

インスタンス変数には、デフォルト値があるため、必ずしも初期化しなくてよい。数値を格納するプリミティブ型の変数の場合、デフォルト値は「0」になり、boolean型変数の場合は「false」になる。参照型の変数のデフォルト値は「null」（先述のとおり、参照型変数のnull値は、「参照に対応するオブジェクトがない」という意味。オブジェクトそのものが存在しないという意味ではない）

This PDF is prepared for tools@maruhouse.org, personal use only.

インスタンス変数とローカル変数の違い

1 インスタンス変数はメソッドの外で宣言する（ただし、必ず何らかのクラスの中で宣言しなければならない）。

```
class Horse {
  private double height = 15.2;
  private String breed;
  //more code...
}
```

2 ローカル変数はメソッドの中で宣言する。

```
class AddThing {
  int a;
  int b = 12;

  public int add() {
    int total = a + b;
    return total;
  }
}
```

3 ローカル変数は利用の前に初期化が必要。

```
class Foo {
  public void go() {
    int x;
    int z =x + 3;
  }
}
```

変数xには値が代入されていないので、利用するコードを書くとコンパイル時にエラーになる

```
File  Edit  Window  Help  Yikes
%javac Foo.java
Foo.java:4: variable x might
not have been initialized

    int z = x +3;
1 error      ^
```

ローカル変数にデフォルト値はないので、初期化が絶対に必要。初期化をしていないローカル変数を利用するコードを書くと、コンパイル時にエラーが発生する。

素朴な疑問に答えます

Q: メソッドのパラメータの場合はどうですか。ローカル変数と同じルールが適用されるのですか。

A: メソッドのパラメータは、ローカル変数と同様、メソッドの中（厳密に言えば、メソッドの引数のリストの中ですが）で宣言されますが、ローカル変数とは異なり、初期化は必要としません。初期化をしていないことでコンパイルエラーになることはないのです。

パラメータの宣言されたメソッドを呼び出す際は、宣言に従って適切な引数を指定しなければコンパイルエラーになります。この引数の指定によってパラメータは必ず初期化される（初期化されなければコンパイルエラーが発生する）と考えることもできます。

This PDF is prepared for tools@maruhouse.org, personal use only.

変数の比較

2つのプリミティブ型変数の内容が同じかどうかを確かめるのは簡単です。「==」という演算子を使えばいいだけです。また、2つの参照型変数の内容が同じかどうか（同じオブジェクトに対応しているかどうか）を確認するのにも「==」という演算子が使えます。では、2つのオブジェクトが同じものかどうかを確認するにはどうすればよいのでしょうか。それには、.equals()というメソッドを使う必要があります。注意すべきなのは、「同じである」という言葉の意味がオブジェクトのタイプによって異なる点です。たとえば、Stringオブジェクトの場合、2つのオブジェクトが「同じである」というのは、両者が同じ文字から構成されるということです。同じ文字から構成されていれば、比較対象となったオブジェクトがそれぞれヒープ上の異なる領域に格納されていたとしても、両者は「同じ」とみなされます。それでは、比較対象がDogオブジェクトであった場合はどうでしょうか。インスタンス変数の値がまったく同じであれば「同じ」とみなしてよいのでしょうか。おそらくそうではないでしょう。ともかく、何を「同じ」とみなせば妥当かはオブジェクトごとに違うのです。このことは次章以降で詳述しますが、ここでは、まず、「==」という演算子が、2つの変数の値のビットパターンが同じかどうかを確認するだけのものである、ということだけ覚えてください。そのビットが何を意味するかは関係がないのです。ビットパターンさえ同じであれば、「==」演算子にとっては「同じ」ということになってしまうのです。

2つのプリミティブ型変数の値が同じであること、参照型変数の値が同じオブジェクトに対応していることを確認するには「==」演算子を使用する。

2つのオブジェクトが「同じ」であることを確認するには、equals()メソッドを使用する。

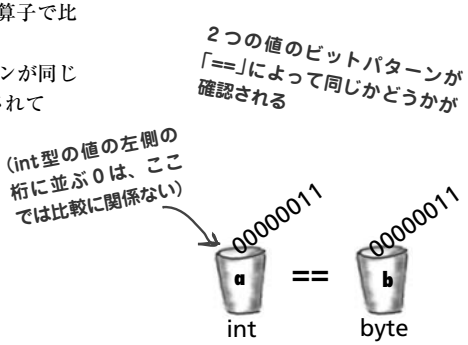
（たとえば、2つのStringオブジェクトを構成する文字がどちらも“Fred”であれば同じとみなされる）

2つのプリミティブ型変数を比較するには、「==」演算子を使用する

「==」演算子は、あらゆるプリミティブ型の変数の比較に使えます。この演算子で比較されるのは2つの変数の値のビットパターンです。

if (a == b){...}というようなコードを書くと、変数aとbの値のビットパターンが同じであれば、trueが戻されます（下の例の場合、byte型の値とint型の値が比較されていますが、値が小さいものであれば、「同じ」とみなされることはあります。int型にはbyte型に存在しない桁もあるのですが、そのビットが「0」であれば、比較の結果には影響しません）。

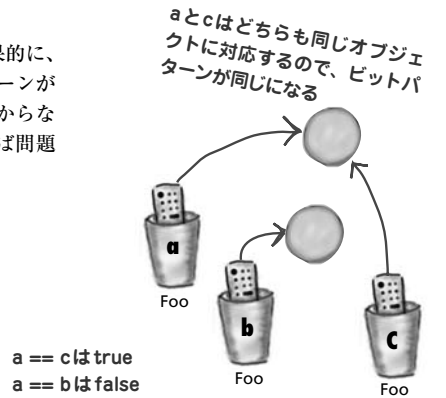
```
int a = 3;
byte b = 3;
if (a == b) { //true }
```



2つの参照型変数の比較（同じオブジェクトに対応しているかどうかの確認）にも「==」が使用できる

この場合もやはり、実際に比較されるのは、値の「ビットパターン」です。結果的に、変数の値が同じオブジェクトに対応していれば、trueが戻されます。ビットパターンがどのようなものになるかはわからない（JVMごとに異なる）のですが、それがわからなくても、対応するオブジェクトが同じであれば同じとみなされると覚えておけば問題はありません。

```
Foo a = new Foo();
Foo b = new Foo();
Foo c = a;
if (a == b) { //false }
if (a == c) { //true }
if (b == c) { //false }
```



This PDF is prepared for tools@maruhouse.org, personal use only.

覚えておこう

Javaは「値渡し」、
これは、引数として指定された値が
すべて「コピー」されて
メソッドに渡されるということ。

注意すべきなのは、「値」というのが厳密には変数に格納された「ビットのパターン」を意味するということ。

自分で考えてみよう

正しいコードはどれ？

以下のようなメソッドがある場合、右側に列挙したコードのうち正しいものはどれですか。正しいものの横に「✓」マークをつけてください（コードの中には、メソッドの呼び出しに必要な変数に値を代入するためのものもあるので注意）。

```
int calcArea(int height, int width) {
  return height *width;
}
```

```
int a = calcArea(7, 12);
short c = 7;
calcArea(c, 15);
int d = calcArea(57);
calcArea(2, 3);
long t = 42;
int f = calcArea(t, 17);
int g = calcArea();
calcArea();
byte h = calcArea(4, 20);
int j = calcArea(2, 3, 5);
```


This PDF is prepared for tools@maruhouse.org, personal use only.

エクササイズ

コンパイラになってみよう

A、Bのソースコードを、コンパイラになったつもりで見てみましょう。そして、それぞれのコードが果たして正しくコンパイルできるかどうかを確認してください。もし、「コンパイルできない」と思った場合は、その理由を答え、正しくコンパイルできるように修正もしてください。コンパイルができると判断した場合は、実行結果がどのようになるかを答えてください。

A

```
class XCopy {
  public static void main(String [] args) {
    int orig = 42;
    XCopy x = new XCopy();
    int y = x.go(orig);
    System.out.println(orig + " " +y);
  }
  int go(int arg){
    arg = arg * 2;
    return arg;
  }
}
```

B

```
class Clock {
  String time;

  void setTime(String t) {
    time = t;
  }

  void getTime() {
    return time;
  }
}

class ClockTestDrive {
  public static void main(String [] args) {

    Clock c = new Clock();

    c.setTime("1245");
    String tod = c.getTime();
    System.out.println("time: " + tod);
  }
}
```


This PDF is prepared for tools@maruhouse.org, personal use only.

エクササイズ

Javaに関連する用語には様々なものがあります。下の説明文に当てはまる用語が何かを解答欄に書いてください。同じ答えを何度書いてもかまいません。

答えの候補は次のとおり：
インスタンス変数、引数、戻り値、ゲッター、セッター、カプセル化、アクセス修飾子「public」、アクセス修飾子「private」、値渡し、メソッド

1つのクラスに複数含まれることがある	
メソッドはこれを1つしか持つことができない	
型は宣言と同じでなくてよい場合がある	
インスタンス変数をprivateで宣言する	
値をコピーすること	
更新にはセッターのみを使用する方がよい	
メソッドはこれを複数持つことがある	
インスタンス変数の現在の値を戻す	
インスタンス変数には使用しない方がよい	
複数の引数を持つことができる	
インスタンス変数に新たな値を設定する	
カプセル化に利用する	
指定できるのは常に1つ	

This PDF is prepared for tools@maruhouse.org, personal use only.

プログラムパズル

右のプログラムは一部分が欠落しています。欠落部分に入るコードの候補がいくつか示してありますが、それぞれのコードを当てはめた場合にどのような実行結果が得られるかを考え、コードと実行結果を線で結んでください。
実行結果の中には、どのコードとも結び付かないものがあります。また、複数のコードと結び付く実行結果もあります。

```
public class Mix4 {
  int counter = 0;
  public static void main(String [] args) {
    int count = 0;
    Mix4 [] m4a = new Mix4 [20];
    int x = 0;
    while ( [          ] ) {
      m4a [x] = new Mix4();
      m4a [x].counter = m4a [x].counter + 1;
      count = count + 1;
      count = count + m4a [x].maybeNew(x);
      x = x + 1;
    }
    System.out.println(count + " "
                       + m4a [1].counter);
  }

  public int maybeNew(int index) {
    if ( [            ] ) {
      Mix4 m4 = new Mix4();
      m4.counter = m4.counter + 1;
      return 1;
    }
    return 0;
  }
}
```

コードの候補	実行結果
`x < 9`	`14 7`
`index < 5`	`9 5`
`x < 20`	`19 1`
`index < 5`	`14 1`
`x < 7`	`25 1`
`index < 7`	`7 7`
`x < 19`	`20 1`
`index < 1`	`20 5`

This PDF is prepared for tools@maruhouse.org , personal use only.

プールパズル

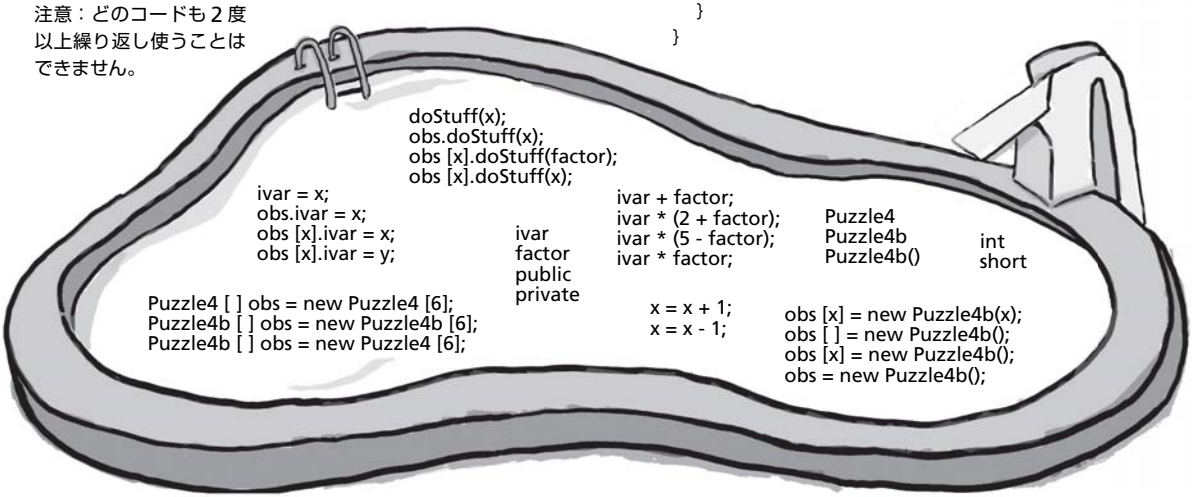
右のプログラムの欠落部分に、下のプールの中のコードを当てはめ、実行結果が以下に示すようなものになるようなプログラムを完成させてください。すべてのコードを使う必要はありませんが、同じコードを2回以上繰り返し使うことはできません。

実行結果

```
File  Edit  Window  Help  BellyFlop
%java Puzzle4
result 543345
```

```
public class Puzzle4 {
  public static void main(String [] args) {
    ____________________________________
    int y = 1;
    int x = 0;
    int result = 0;
    while (x < 6){
      ____________________________
      ____________________________
      y = y * 10;
      ________________
    }
    x = 6;
    while (x > 0) {
      ________________
      result = result +_________________
    }
    System.out.println("result " + result);
  }
}
class __________{
  int ivar;
   _____________doStuff(int  __________){
    if (ivar > 100){
      return ____________________________
    } else {
      return ____________________________
    }
  }
}
```

注意：どのコードも2度以上繰り返し使うことはできません。



This PDF is prepared for tools@maruhouse.org , personal use only.

データベースがハッキングされたのはなぜ？

Jayは、Buchananに銃を脇腹に突き付けられ、思わず身を硬くした。外見の恐ろしさもあるが、この愚かな大男にあまり関わりたくなかったのである。Buchananは、ボスのオフィスに入るよう促した。Jayは（少なくとも最近は）とがめられるような心当たりはなかったので、ボスに会って二言三言交わすくらいならそう悪いことでもないか……と思っていた。Jayはここのところウェストサイドで大量の覚醒剤をさばいており、そのことはボスも喜んでいるはずだった。覚醒剤の取引は、最高に儲かる、という商売でもなかったが、彼のさばいているクスリは害が少ないという点がよかった。クスリを打った人間のほとんどは、その後、少し集中力がなくなるくらいで、一応まともには暮らしていたのである。

ボスのオフィスは、以前はすっきりとしていたのだが、しばらくぶりに入ってみると、中の様子はかなり変わっていた。大ボスというわけではないのだから、「豪華」というほどではないが、それでも彼にしてみれば可能な限りの装飾を施したようだった。「Jay、よく来たな。また会えてうれしいよ」と言うボスの口調にかすかな怒りを感じ取ったJayは「ひょっとして、何かヘマをやらかしましたかね。それならはっきりそう言ってください」と答えた。するとボスからは「いや、君はよくやってくれているよ。儲けも増えているようだしね。ただ、最近ちょっとした、何と言うのかな……“ハッキング”というやつに遭って困っているんだよ」という答えが返ってきた。

5分間ミステリー

Jayは身構えた。彼はかつて名うてのハッカーだったのである。おかげでセキュリティで何か問題が起きると、ありがたくないことに必ず注目されてしまうことになる。「私じゃありませんよ。足を洗ったんです。今の私にはハッキングなどデメリットが大きすぎてやる意味がありません。もう、自分の部下を動かして自分のビジネスをする以外のことには関心がないんですよ」とJayが言うと、ボスは笑って「私も君のせいだとは思っていないよ。だがな、誰にしてもハッカーを退治しない限り、私は相当の儲けを失うことになるんだよ」と言った。そこでJayは、「今すぐ私を解放してくだされば、部下に言って、今日中には何とかできると思いますが」と答えた。

「残念ながら、ことはそう簡単じゃない。Buchananが教えてくれたんだが、君は今、J37NEクラスを使っているそうじゃないか」何とも遠まわしな物言いである。「確かに少し使っていますね。それが何か？」Jayは不安を隠すことができなかった。「クスリの取引場所を知らせるのに使うんだろうが、問題は、ジャンキーの中に、J37NEクラスを悪用して私のデータベースのハッキングをする輩がいるってことだ。しかし、J37NEクラスのメソッドやインスタンス変数を利用してデータベースに侵入するなんて芸当をやれるのは、君くらいキレる奴だけだと思うんだよ……」その時Buchananが叫んだ。「それは俺の書いたコードだ。それをJayのようなクズハッカーにいじられるんじゃ、やってられないぜ」それを聞いたJayは内心「しめた」と思った。「Buchanan、もちろん、君は優秀なプログラマだ。だからアクセス修飾子も十分に使いこなしているよね」「からかうのはよせ。取引場所が調べられるように、ジャンキーどもの使うメソッドはpublic宣言した。だが、データベースに関係する重要なメソッドは当然、すべてprivate宣言しているぜ。外部の人間は絶対に誰も使えないはずだ」「ボス、これで何が問題なのかがだいたいわかりましたよ。この大男はここに置いて、ちょっと出かけませんか」Buchananは銃に手を伸ばそうとしたが、それとほぼ同時にボスの素早い一撃が彼の首に命中した。「やめろBuchanan。銃を置いて外へ出ろ。後は私たちに任せるんだ」

Jayは何がわかったというのだろうか？
果たして彼はボスのデータベースを救うことができるのか。

This PDF is prepared for tools@maruhouse.org, personal use only.

エクササイズの答え

コンパイラになってみよう

A

このソースコードはコンパイルできる。実行結果は、“42 84” となる。Javaでは「値渡し」が行われる（引数として指定された値がコピーされる）ため、変数origの値自体がgo()メソッドによって変更されるわけではないという点に注意。

B

```
class Clock {
  String time;
  void setTime(String t) {
    time =t;
  }
  String getTime() {
    return time;
  }
}

class ClockTestDrive {
  public static void main(String [] args) {
    Clock c =new Clock();
    c.setTime("1245");
    String tod =c.getTime();
    System.out.println("time:" ++tod);
  }
}
```

ゲッターメソッドには戻り値の型の指定が必要。

私は誰でしょう

1つのクラスに複数含まれることがある	インスタンス変数、ゲッター、セッター、メソッド
メソッドはこれを1つしか持つことができない	戻り値
型は宣言と同じでなくてよい場合がある	戻り値、引数
インスタンス変数をprivateで宣言する	カプセル化
値をコピーすること	値渡し
更新にはセッターのみを使用する方がよい	インスタンス変数
メソッドはこれを複数持つことがある	引数
インスタンス変数の現在の値を戻す	ゲッター
インスタンス変数には使用しない方がよい	アクセス修飾子「public」
複数の引数を持つことができる	メソッド
インスタンス変数に新たな値を設定する	セッター
カプセル化に利用する	ゲッター、セッター、アクセス修飾子「public」、アクセス修飾子「private」
指定できるのは常に1つ	戻り値

This PDF is prepared for tools@maruhouse.org, personal use only.

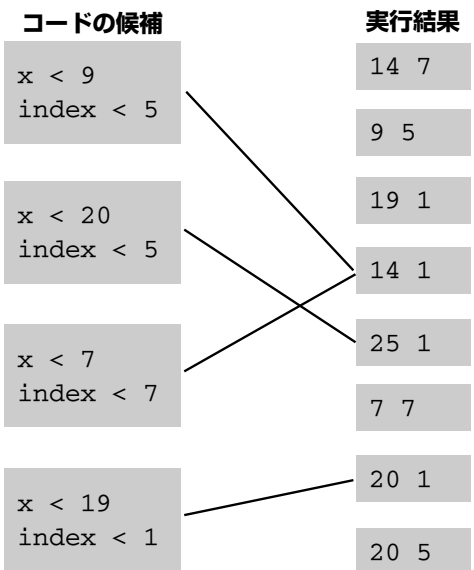
プログラミングパズル

```
public class Puzzle4 {
  public static void main(String [] args) {
    Puzzle4b [] obs = new Puzzle4b [6];
    int y = 1;
    int x = 0;
    int result = 0;
    while (x < 6) {
      obs [x] = new Puzzle4b();
      obs [x].ivar = y;
      y = y * 10;
      x = x + 1;
    }
    x = 6;
    while (x > 0) {
      x = x - 1;
      result = result + obs [x].doStuff(x);
    }
    System.out.println("result " + result);
  }
}
class Puzzle4b {
  int ivar;
  public int doStuff(int factor) {
    if (ivar > 100) {
      return ivar *factor;
    }else {
      return ivar *(5 -factor);
    }
  }
}
```

実行結果

```
File  Edit  Window  Help  BellyFlop
%java Puzzle4
result 543345
```

Jayは、Buchananがミスをしているのではないかと考えたのである。Buchananの話では、インスタンス変数については一切触れられていない。彼は確かにメソッドのコードは正しく書いたようだが、おそらくインスタンス変数をprivate宣言するのを忘れている。これでは簡単にデータが盗まれてしまい、ボスは大損をすることになる。



This PDF is prepared for tools@maruhouse.org, personal use only.

5章　本格的なプログラムの作成

本格的なプログラム（メソッド）を書く

この章では、実用に耐えうる本格的なプログラムを作成するのに必要ないくつかのことがらについて解説します。これまでにもプログラムの例は紹介してきましたが、いずれもごく簡単なもので、とても「実用的」とは言えませんでした。実用的なプログラムを書くには、使える演算子を増やし、ループなども使いこなせるようになる必要があります。ループでまず重要なのはforループです。whileループも大切ですが、本格的なプログラムを書こうとすればともかくforループを習得しなければなりません。forループはたとえば、乱数の生成に役立ちます。また、StringからintへのデータS変換などにも使えるでしょう。もちろん、forループの用途はそれ以外にも無数にあります。この章では、ゲームのプログラムを例にとり、本格的なプログラムをゼロから作り、テストするという作業がどのようなものになるのかを見ていきます。同様のプログラムは次の6章でも使用することになります。この章で使うものは比較的単純なバージョンで、次章ではそれをより高度なものにしていきます。

This PDF is prepared for tools@maruhouse.org, personal use only.

バトルシップゲームプログラムを作る

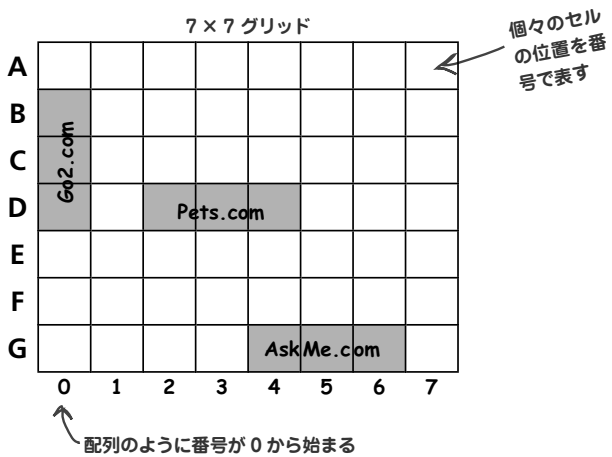
ここで作るのは、いわゆる「バトルシップゲーム」のプログラムです。人間とコンピュータが対戦するのですが、通常とは異なり、軍艦を持つのはコンピュータの側だけです。人間のプレーヤーがその軍艦の位置を推測し、沈めることになります。もちろん、推測の回数が少なくて済む方がよい、というのは言うまでもありません。コンピュータの軍艦には、それぞれ（本のテーマに合わせて）「～.com」という名前が付けられます。

このバトルシップゲームでは
7×7のグリッドと
3隻の「軍艦」を使用する。
個々の軍艦はセル3つ分の大きさ。

ゲームのルール：人間のプレーヤーがコンピュータの軍艦の位置を推測し、沈没させる。正解に到達するまでの推測の数が少ないほどよい。

ゲームの進行順序：ユーザーがプログラムを起動すると、縦7つ横7つのマス目を持つグリッドに3隻の軍艦（いずれにも「～.com」という名前が付けられる）が自動的に配置される。配置が完了したら、プレーヤーに最初の推測をするよう指示が出される。

プレーの仕方：このゲームはGUIを使わず、コマンドラインでプレーする（本書ではまだGUIを扱う方法を説明していないため）。画面に推測をするよう指示が出されたら、プレーヤーは軍艦が配置されていると思われる場所（グリッドのセル）を1つ選ぶ。どのセルを選んだかは、A3、C5といった番号を入力して知らせる。番号を入力すると、「Hit」、「Miss」、「You sunk Pets.com」といった文字列が表示される。「Hit」は推測が合っていることを、「Miss」は推測が外れていることを示す。「You sunk Pets.com（Pets.comの部分はさまざまに変わる）」は、推測が合っていて、軍艦を1隻沈没させることができたということを意味する。軍艦が3隻とも沈んだら、ゲームは終了し、終了までのプレーヤーの推測の回数が表示される。



プログラムの実行画面の例（一部）

```
File  Edit  Window  Help  Sell
%java DotComBust
Enter a guess A3
miss
Enter a guess B2
miss
Enter a guess C4
miss
Enter a guess D2
hit
Enter a guess D3
hit
Enter a guess D4
Ouch!You sunk Pets.com :(
kill
Enter a guess B4
miss
Enter a guess G3
hit
Enter a guess G4
hit
Enter a guess G5
Ouch!You sunk Go2.com :(
```


This PDF is prepared for tools@maruhouse.org, personal use only.

プログラムの設計

Javaのプログラムにクラスやメソッドが必要であるということはすでに説明しましたが、ここで作成するバトルシップゲームプログラムの場合、そのクラスやメソッドはどのようなものにすればよいのでしょうか。それを知るためには、まず、ゲームがどんなものであるかをよく考えなければなりません。

以下に、バトルシップゲームの進行の手順について簡単にまとめます。

1 ユーザーがゲームを開始させる

- **A** 3 隻の軍艦が作られる。
- **B** 軍艦がそれぞれグリッド上に配置される。

2 プレー開始

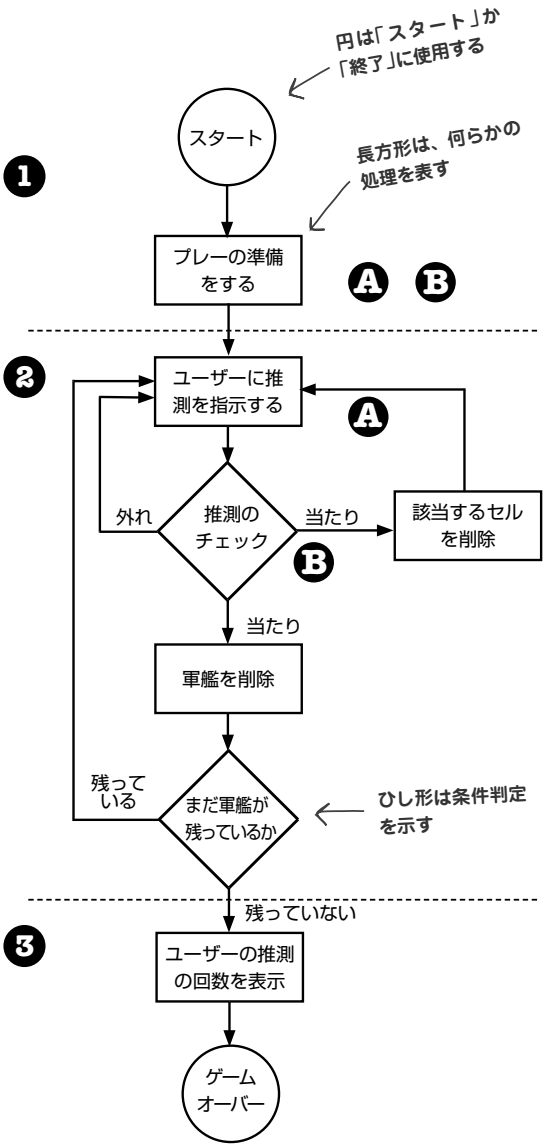
軍艦がすべて沈没するまで以下の手順が繰り返される。

- **A** 軍艦の位置を推測するようユーザに指示される。（位置は“A2”、“C0”といった番号で示す）
- **B** ユーザの推測が正しいかどうかが確認される。正しければ、該当するセルが削除される。特定の軍艦が配置されたセルがすべて当てられた場合は軍艦が削除される。

3 ゲーム終了

推測の回数が表示される。

これでプログラムが何をしなければならないかについて少しわかったと思います。次に考えるべきことは、すべきことをするためにどのようなオブジェクト（クラス）が必要かということです。すでに述べたとおり、Javaでは、手続き指向でなく、オブジェクト指向で考えるということが大切なのです。



フローチャート

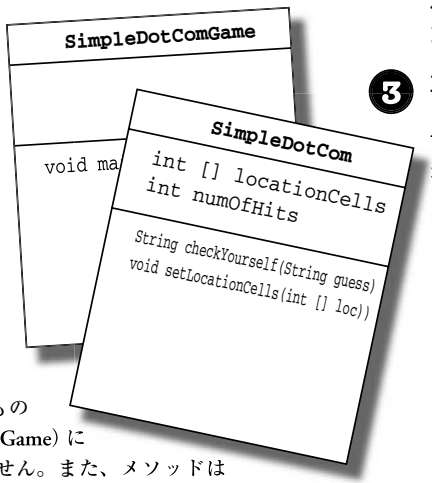
This PDF is prepared for tools@maruhouse.org, personal use only.

簡易版ゲームプログラム

前ページのような動きをするバトルシップゲームプログラムを作ろうとすれば、おそらく、ゲームそのものに対応するクラスと軍艦に対応するクラスが最低限必要になるでしょう。前ページに書いたとおりの機能を持つプログラムは次章で作成します。この章では取りあえず、それを単純にした「簡易版」を作ってみましょう。

「簡易版」では、軍艦を2次元のグリッドではなく、単なるセルの列に配置します。また、配置する軍艦も3隻ではなく1隻だけです。

プレーの仕方は簡易版でも基本的には同じです。したがって、軍艦に対応するクラス（ここではSimpleDotCom）はやはり必要ということになります。このクラスのインスタンスを、いずれかのセルに配置し、その位置と、プレーヤーが推測した位置とを比較するのです。位置が一致すればセルは削除され、軍艦オブジェクト（SimpleDotComクラスのオブジェクト）に対応するセルがすべて削除されれば「沈没」ということになり、ゲームは終了します。



簡易版は次章で作成する「完全版」のたたき台になります。簡易版を正しく作ることができれば、後はそれを少し複雑なものに改造するだけで完全版ができるのです。

簡易版の場合、ゲームそのものを表すクラス（SimpleDotComGame）にはインスタンス変数はありません。また、メソッドはmain()のみです。つまり、プログラムが起動されると、main()メソッドで軍艦オブジェクトのインスタンスが1つ作られ、その配置が決められるということになります。また、その後、プレーヤーに対して位置の推測をするよう指示が出され、その推測が正しいかどうかがチェックされる……という処理が、軍艦に対応する3つのセルがすべて削除するまで繰り返されます。

注意すべきなのは、セルの列はあくまで「仮想的」なものであるということです。列が7つのセルから構成されるということから、つい、7つの要素からなる配列を作り、3つの要素に軍艦オブジェクトへの参照を格納するということをしたくなりますが、その必要はありません。軍艦が配置される3つの（連続する）セルの番号を格納する配列があればいいのです。

❶ ゲームが開始されると、軍艦オブジェクトが1つ作成され、7つのセルのうちの3つに配置される。
軍艦の位置は、“A2”、“C4”などではなく、1、2、3などの単なる整数の番号で表される。

❷ プレーが開始されると、プレーヤーに軍艦の位置を推測するよう指示が出される。プレーヤーがセル番号を推測すると、それが軍艦オブジェクトが配置されている3つのセルの番号のいずれかと一致しているかどうかが確認される。一致していれば、numOfHits変数の値が1増やされる（インクリメントされる）。

❸ 3つのセルの番号がすべて当てられると（numOfHitsの値が3になると）、ゲームは終了する。その際、プレーヤーが合計で何回の推測をしたかが知らされる。

プログラムの実行画面の例

```
File  Edit  Window  Help  Destroy
%java SimpleDotComGame
enter a number 2
hit
enter a number 3
hit
enter a number 4
miss
enter a number 1
kill
You took 4 guesses
```


This PDF is prepared for tools@maruhouse.org, personal use only.

クラスの作成

プログラマはそれぞれ、コードを書くための自分なりの「方法論」のようなものを持っています。その点は著者も同じです。ここで、プログラム作成作業のだいたいの流れをつかんでもらうため、著者がどのような手順でプログラム（クラス）のコードを作っているかを紹介しておきましょう。もちろん、「誰もがこのとおりにしなければならない」というわけではなく、皆が自分に合った方法で作業をすればいいわけですが、著者がここで紹介する方法で非常にスムーズに作業を進められているというのも確かです。また、この方法はJavaプログラミングの学習にも適していると著者は考えています。著者のコード（クラス）作成手順は以下のとおりです。

- ☐ 作成するクラスの果たすべき役割を明らかにする。
- ☐ 必要なインスタンス変数、メソッドをリストアップする。
- ☐ メソッドの仮コードを書く（詳細は後述）。
- ☐ メソッドのテストコードを書く。
- ☐ クラスをひとまず完成させる（リアルコードを作成する）。
- ☐ 作ったクラスとそのメソッドをテストする。
- ☐ テストの結果を踏まえクラスをデバッグする。
- ☐ ユーザーに実際に運用してテストしてもらう（幸い*、学習用に作るプログラムではこの作業は不要）。

クラスのコードは 3 種類書く

仮コード	テストコード	リアルコード

次ページからは、「ここではどのコードについて話をしているのか」ということを示すのに、上の図を使います。たとえば、SimpleDotComクラスの仮コードについて説明している場合、図は以下のようになります。

SimpleDotComクラス

仮コード	テストコード	リアルコード

仮コード

プログラムの処理の流れ（ロジック）をおおまかに書いたの一種の「擬似コード」です。

テストコード

リアルコードが正しいものであるかどうかを確認するために使用するコードです。

リアルコード

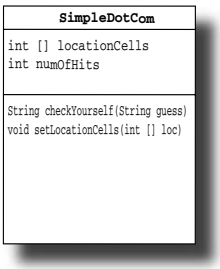
本来作成すべきコードのことです。作業の最終目的はこのリアルコードを作ることです。

頭の体操

プログラムに複数のクラスが必要な時、作る順序はどのような基準で決めますか。Javaのプログラムはよほど小規模なものでない限り、（オブジェクト指向プログラミングの常識に従い、1 つのクラスに数多くの機能を詰め込むということをしなければ）まず間違いなく複数のクラスから構成されます。つまり、どのクラスから始めるか、という問題は常につきまとうと言えるのです。

* ユーザーが運用すると思いがけない問題が数多く発生します。

This PDF is prepared for tools@maruhouse.org, personal use only.



まず、仮コードがどのようなものであるかを、例を使って説明してみましょう。仮コードというのは、Java言語と自然言語（英語、日本語など）を混ぜて書く、一種の「擬似コード」です。通常、この仮コードはインスタンス変数のリスト、メソッドのリスト、メソッドのロジックという3つの部分から構成されます。中でも最も重要なのがメソッドのロジックです。この部分に「このクラスでどのような処理を行うか」を書くことになるからです。

インスタンス変数

locationCells　軍艦の位置を表すセル番号を格納するint型の配列
numOfHits　推測が当たった回数を表す整数を格納するint型の変数。初期値は0にする

メソッド

checkYourself()　ユーザーの入力したセル番号（String型のパラメータを使って受け取る）を、軍艦が実際に配置されたセルの番号と比較し、その結果を表す文字列（「hit」、「miss」、「kill」など）を戻すメソッド
setLocationCells()　インスタンス変数locationCellsの値を設定するためのセッターメソッド

メソッドのロジック

メソッド：String checkYourself(String userGuess)
　値の取得　ユーザーが推測したセル番号をString型のパラメータを使って取得
　値の型変換　取得したセル番号をStringからintに変換
　繰り返し処理　intの配列の個々の要素について同じ処理を繰り返す
　　//ユーザーの推測したセル番号と軍艦が実際に配置されているセル番号の比較
　　if文　セル番号が一致しているかどうかの判定
　　インクリメント　推測が一致した回数を表す値を1増やす
　　//一致が3回目かどうかを確認。
　　if文　一致の回数が3に達していれば、文字列“kill”を戻す
　　else文　一致の回数が3に達していなければ“hit”を戻す
　　（内側の）if文の終了
　　else文　セル番号が一致していなければ、“miss”を戻す
　　（外側の）if文の終了
　繰り返し処理の終了
メソッドの終了

メソッド：void setLocationCells(int [] cellLocations)
　値の取得　パラメータ（int型の配列）を使って軍艦の配置を表すセル番号を取得
　値の設定　インスタンス変数locationCellsの値を設定
メソッドの終了

This PDF is prepared for tools@maruhouse.org, personal use only.

テストコードの作成

本来作成すべきコードを テストするためのコードを作る

仮コードは後でリアルコードに変えるのですが、その前にまず、リアルコードをテストするためのコードを作成します。驚く人がいるかもしれませんが、まだリアルコードが存在しない段階で、先にそれをテストするためのコードを書くのです。

これは、エクストリームプログラミング（eXtreme Programming = XP）と呼ばれる開発手法で採用されているやり方です。テストコードを先に書くことにより、リアルコードが短時間で容易に作成できるなどのメリットがあると言われています。著者は特にXPを読者に奨めるつもりはありませんが、この「テストコードを先に書く」という部分はよいのではないかと考えています。もちろん、XP自体も優れた開発手法だと思います。

エクストリームプログラミング（XP）

エクストリームプログラミングは 90 年代の後半に生まれた比較的新しいソフトウェア開発手法です。「プログラマの願望に応える手法」であるとして人気を集め、現在では零細企業からFordのような大企業にいたるまであらゆる種類の企業に採用されています。XPのメリットとしては、開発の工程がかなり進んだ段階でも大きな仕様変更に柔軟に対応でき、顧客が希望どおりの製品を手に入れられる可能性が高くなる、ということが大きいと言えるでしょう。

XPは、次に示すようないくつかの「プラクティス」と呼ばれるルールを基礎とした開発手法です。ただし、ここに示したプラクティスすべてではなく、そのうちから自分の合うものを一部だけ取り出して利用している人も少なくありません。

- 各バージョンの規模を小さくし、短いサイクルでリリースをする。
- 同様の作業の繰り返しで開発をすすめる。
- 必要最小限以上のコードは書かない（近い将来確実に必要になるであろうと思われるものも含む）。
- テストコードを先に書く。
- 残業をしない。
- 必要と気付いたらいつでも（また、プログラムのどの部分に関しても）設計の変更を行う。
- テストをすべてパスしないうちは絶対にリリースを行わない。
- スケジュールは、短いサイクルのリリースを基本として無理のないように立てる。
- 常に設計をシンプルに保つ。
- 2 人のプログラマがペアで作業を進める。また、担当者の異動を頻繁に行って、チーム全員がすべてのコードについて知っているという状態を保つ。

This PDF is prepared for tools@maruhouse.org, personal use only.

SimpleDotCom クラスのテストコードを作成する

それでは、SimpleDotCom クラスのテストコードを作成してみましょう。このコードでは、SimpleDotCom クラスのオブジェクトを作成し、そのメソッドを実行する必要があります。SimpleDotCom クラスの場合、checkYourself()、setLocationCells() という 2 つのメソッドがあるのですが、重要なのは checkYourself() メソッドのほうです（ただし、checkYourself() メソッドを正しく動作させるには、setLocationCells() が必要になる）。

まず、すでに紹介した checkYourself() メソッドの仮コードをもう一度よく見てみましょう（setLocationCells() メソッドは単純なセッターメソッドなので、ここではテスト対象にしていません。ただし、現実のソフトウェア開発プロジェクトでは、品質をより高めるため、このようなセッターメソッドもテスト対象とする場合があります）。

仮コードをよく見たうえで、「このメソッドが正しいかどうかを確認するにはどのようなテストコードを作ればよいか」を考えてみてください。

仮コード

メソッド：String checkYourself(String userGuess)
　値の取得　ユーザーが推測したセル番号を String 型のパラメータを使って取得
　値の型変換　取得したセル番号を String から int に変換
　繰り返し処理　int の配列の個々の要素について同じ処理を繰り返す
　　//ユーザーの推測したセル番号と軍艦が実際に配置されているセル番号の比較
　　if 文　セル番号が一致しているかどうかの判定
　　インクリメント　推測が一致した回数を表す値を 1 増やす
　　//一致が 3 回目かどうかを確認。
　　if 文　一致の回数が 3 に達していれば、文字列 “kill” を戻す
　　else 文　一致の回数が 3 に達していなければ “hit” を戻す
　　（内側の）if 文の終了
　　else 文　セル番号が一致していなければ、“miss” を戻す
　（外側の）if 文の終了
　繰り返し処理の終了
メソッドの終了

テストコードで行うべきこと

1. SimpleDotCom クラスをインスタンス化する。
2. SimpleDotCom オブジェクトの変数「locationsCell」に、値（{2,3,4} など、3 つの int 型の値から成る配列）を代入する。
3. ユーザーが推測したセル番号（“2”、“0” など）を格納するための String 型の変数を作る。
4. checkYourself() メソッドを呼び出す。引数にはダミーの値を指定する。
5. 比較の結果を知らせる値が正しいかどうか確認し、正しければ “passed”、誤っていれば “failed” という文字列を画面に表示する。

This PDF is prepared for tools@maruhouse.org, personal use only.

素朴な疑問に答えます

Q: 何か勘違いをしているかもしれないのですが……。存在しないリアルコードに対してテストコードを動かすというのはいくら何でもおかしいのではないですか。

A: 確かに、存在しないリアルコードに対してテストコードを「動かす」というのはおかしいのですが、そうは書いていません。テストコードはあくまで「作る」だけです。リアルコードを作る前にテストコードを作る、と書いたのです。仮に存在しないリアルコードに対してテストコードを動かそうとしても不可能でしょう。リアルコードを作らない限り、まずコンパイルができません。また、無理にコンパイルできるようなテストコードを作ったとしても、テストは常に失敗に終わってしまいます。

Q: それにしてもわからないのですが、テストコードをリアルコードより先に作ることに何の意味があるのでしょうか。

A: テストコードを作ると、リアルコードをどのようなものにすればよいかが明確になるのです。特にメソッドの機能をどうすればよいかがよくわかります。また、リアルコードができた時点でテストコードが既に存在していれば誰もがすぐにテストをするでしょうが、テストコードが存在しなければ、わざわざ改めて作る人は少ない、という理由もあります。テストコードを作ること自体はあまり面白いことではないからです。テストコードはできる限り単純なものにします。そして、そのテストをパスすることだけを目標にリアルコードを書くのです。テストをパスしたら、また新たに単純なテストコードを書きます。書けたら、最初のテストと新しいテストの両方をパスするようにリアルコードを改造します。このように「新たに単純なテストコードを書き、それに合わせてリアルコードを書き直す」という作業を、リアルコードが完成するまで繰り返すのです。注意すべきなのは、どの段階のリアルコードも、存在するすべてのテストコードにパスしなければならないということです。

SimpleDotCom クラスのテストコード

```
public class SimpleDotComTestDrive {

  public static void main (String [] args) {

    SimpleDotCom dot = new SimpleDotCom();

    int [] locations = {2,3,4};

    dot.setLocationCells(locations);

    String userGuess = "2";

    String result = dot.checkYourself(userGuess);

    String testResult = "failed";

    if (result.equals("hit")){

      testResult = "passed";

    }

    System.out.println(testResult);

  }

}
```

SimpleDotCom オブジェクトをインスタンス化する

配列を作成し、変数 locationCells に設定するための値を代入する（値は軍艦が配置されるセルの番号に使用されるため、配列の要素数は 3 になる）

変数 locationCells のセッターメソッドを呼び出す

本来ユーザーが入力する値の代わりにこの値を使う

SimpleDotCom オブジェクトの checkYourself() メソッドを呼び出す。引数には上で作成した「偽物」の推測値を指定する

checkYourself() メソッドの実行結果として、戻り値 "hit" が得られれば、メソッドは正常に動いていることになるので、"passed" を代入する。

テスト結果（"passed"、あるいは "failed"）を画面に出力する

自分で考えてみよう

次ページからは SimpleDotCom クラスのリアルコードを作ることになります。リアルコードができたら、テストコードを使ってそれをテストします。実はそのためには上のテストコードではまだ不十分なのですが、あとはどのようなコード（テスト）を足せば十分なものになるでしょうか。そのコードを余白に書いてみてください。

This PDF is prepared for tools@maruhouse.org, personal use only.

checkYourself() メソッド

仮コードをそのままJavaのコードに置き換えることができる、ということはまずありません。仮コードを基にJavaのコードを作る際には、多くの場合、何らかの修正が必要になります。とはいえ、仮コードを作ることで、「どのような処理を行うJavaコードを作らねばならないか」がよくわかるようになることは事実です。ここでは、先に示した仮コードを基に実際にJavaコードを作ってみましょう。

読者の中には、下のコードを見て「まだ改良の余地がある（改良しなければならない）」と感じる人もいるでしょう。また、本書ではまだ説明していないテクニックもいくつか使っているので理解しにくいかもしれません（まだ説明していないことがらには①などの番号を付けてある）。そうしたテクニックについては次のページで解説しています。

ユーザーが推測したセル番号を取得

セル番号を String から int に変換

int の配列の個々の要素について同じ処理を繰り返す

セル番号が一致しているかどうかの判定

推測が一致した回数を表す値を 1 増やす

// 一致が 3 回目かどうかを確認

一致の回数が 3 に達していれば、文字列 “kill” を戻す

3 に達していなければ、“hit” を戻す

“kill” でも “hit” でもなければ、“miss” を戻す

```
public String checkYourself(String stringGuess) {
  ①
  int guess = Integer.parseInt(stringGuess);
    String result ="miss";

    ②
    for (int cell : locationCells) {
      if (guess == cell) {
        result = "hit";
        ③
        numOfHits++;
        ④
        break;
      }
    } //if の終了
  } //for の終了
  if (numOfHits ==locationCells.length) {
    result = "kill";
  } //if の終了
  System.out.println(result);
  return result;
} // メソッドの終了
```

Stringをintに変換

初期値は“miss”にする（この値は「推測が外れた」ということを意味する）

配列locationCellsの各要素について同じ処理をする

配列中のセル番号とユーザーが推測したセル番号を比較する

推測が合っていた場合の処理

ループを抜ける（まだ比較をしていない要素が残っていたとしても、もう比較はしない）

ループを抜けた後、軍艦が沈んだかどうか（推測の正解が 3 回目に達したかどうか）を確認する。達していれば、変数resultの値は“kill”に変更する

実行結果を画面に表示する（resultの値が“hit”、あるいは“kill”に変更されていなければ、結果は“miss”になる）

実行結果を戻す

This PDF is prepared for tools@maruhouse.org, personal use only.

仮コード　テストコード　リアルコード

新しいテクニック

前ページのコードには、これまでにまだ解説していないテクニックが使われています。ここでは取りあえず、本書を読み進むのに支障がないよう、こうしたテクニックについておおまかに説明しておきましょう。詳しくはこの章の最後で説明します。

① String を int に変換する

Integer クラスのメソッド。文字列（String 型のデータ）を数値（int 型のデータ）変換する機能を持つ

Java の開発環境にあらかじめ用意されているクラス

引数は文字列

```
Integer.parseInt("3")
```

② for ループ

このforループは、配列locationCellsの各要素について同じ処理を繰り返し行うもの。locationCellsの要素を1つずつ取り出し、順に変数cellに代入していく

":"は、"in"という意味。つまり、このfor文全体は「配列locationCellsの中の個々のint値について同じ処理を行う」ということを意味する。

```
for(int cell : locationCells){ }
```

取り出した配列の要素を格納する変数を宣言する。処理が繰り返される度に、宣言された変数（ここでは、int型の変数cell）には、配列から取り出された要素が一つずつ格納される。配列の末尾の要素まで到達したら（あるいは4のような"break"のコードがあれば）ループの処理は終了する

ループの処理の対象となる配列。処理が繰り返される度に要素が1つ取り出され、変数cellに代入される（forループについては、章の終わりに詳述）

③ ポストインクリメント演算子

「++」は変数の値を1だけ増やす（インクリメントする）という意味

```
numOfHits++
```

numOfHits++は、以下のコードとほぼ同じ意味（効率という意味では前者がやや有利）
numOfHits = numOfHits + 1;

④ break ステートメント

```
break;
```

ループから即座に抜けるという意味。抜ける際に条件判定も繰り返し処理も行わない

This PDF is prepared for tools@maruhouse.org, personal use only.

素朴な疑問に答えます

Q: Integer.parseInt()に数字以外の引数を指定するとどのようになりますか。また、「3」などではなく「three」のような引数を指定した場合は「3」と認識されるのですか。

A: Integer.parseInt()に指定できるのは、ASCII文字セットの数字(0、1、2、3、4、5、6、7、8、9)だけです。"two"、"blurp"といった数字以外の文字を指定すると、エラー(正確には「例外」です。例外については後の章で詳述)が発生します。

Q: ここで紹介されているものとまったく違う形式のforループが1章に載っています。forループには2種類の形式があるということですか。

A: そうです。従来のJavaのforループ(forループ自体の詳細についてはこの章の中で後述)は一種類で、以下のような形式のものだけでした。

```
for int i = 0 ; i< 10 : i++ {
// 何らかの処理を10回繰り返す
}
```

この形式のforループも引き続き以前と同じように使用できますが、Java 5.0(Tiger)からは「拡張forループ(公式にこう呼ばれている)」という新しいforループも加わっています。配列の各要素について順に同じ処理を行う、という際にはこのforループが使えます(次章を読むとわかるとおり、配列以外のコレクションに関しても同様のことが可能)。同じ処理に従来のforループを使うこともできますが、拡張forループを使った方が簡単にコードが書けます。

SimpleDotCom クラスのコードとそのテストコードの最終形

```
public class SimpleDotComTestDrive {
  public static void main (String [] args) {
    SimpleDotCom dot = new SimpleDotCom();
    int [] locations = {2,3,4};
    dot.setLocationCells(locations);
    String userGuess = "2";
    String result = dot.checkYourself(userGuess);
  }
}
```

```
public class SimpleDotCom {

  int [] locationCells;
  int numOfHits = 0;

  public void setLocationCells (int [] locs) {
    locationCells = locs;
  }

  public String checkYourself (String stringGuess) {
    int guess = Integer.parseInt(stringGuess);
    String result = "miss";
    for(int cell : locationCells) {
      if (guess == cell) {
        result = "hit";
        numOfHits++;
        break;
    }
  } // ループ終了

  if (numOfHits ==
    locationCells.length) {
    result = "kill";
  }
  System.out.println(result);
    return result;
  } // メソッド終了
} // クラス終了
```

実は上記のコードにはバグが含まれています。一応、コンパイルも実行できるのですが、時々問題が起きるのです。その点については後で説明します。今の段階では気にしないでいいでしょう。

実行結果はどのようになるか

テストコードを実行すると、まずSimpleDotComオブジェクトが作成され、軍艦の配置を表すセル番号として、2、3、4が指定されます。そして、「偽の」ユーザー推測値「2」が設定され、checkYourself()メソッドが実行されます。SimpleDotComクラスのコードが正しければ、以下のような実行結果が得られるはずです。

```
java SimpleDotComTestDrive
hit
```


This PDF is prepared for tools@maruhouse.org, personal use only.

仮コード　テストコード　リアルコード

自分で考えてみよう

SimpleDotComクラスのコードとテストコードはすでに作りましたが、まだゲームそのものに対応するクラス（SimpleDotComGame）のコードは作っていません。これまでに紹介したSimpleDotComクラスのコード、テストコードと、すでに説明したゲームの仕様を踏まえ、SimpleDotComGameクラスの（mainメソッドの）仮コードを作ってみてください。以下にその一部がすでに書いてありますので、間を埋めていけばいいでしょう。答えは次のページにありますが、自分なりの仮コードができるまではページをめくらないようにしてください。仮コードは、すでに書かれているものも含め 12 ～ 18 行くらいになるはずです。

メソッド：public static void main (String [] args)

変数の宣言　ユーザーの推測の回数を表す数値を格納するint型の変数numOfGuessesの宣言

演算　軍艦の位置（セル番号）を指定するため、乱数（0 から 4 までの整数）を 1 つ発生させる

WHILEループ　軍艦が沈むまでの間、同じ処理を繰り返すループ

値の取得　ユーザーがコマンドラインで入力した値を取得

SimpleDotComGameクラスで行うべきこと

1. SimpleDotComを作成する。
2. 軍艦の位置を指定する（7 つのセル番号から 3 つの連続する番号を選ぶ）。
3. ユーザーにセル番号の推測を指示する。
4. 推測が正しいかどうかを確認する。
5. 軍艦が沈むまで推測、確認を繰り返す。
6. ユーザーが何回推測を行ったかを画面に表示する。

ゲームプログラムの実行画面の例

```
File  Edit  Window  Help  Runaway
%java SimpleDotComGame
enter a number 2
hit
enter a number 3
hit
enter a number 4
miss
enter a number 1
kill
You took 4 guesses
```


This PDF is prepared for tools@maruhouse.org, personal use only.

SimpleDotComGameクラス（のmain()メソッド）の仮コード

仮コードに関しては1つ注意して欲しいことがあります。たとえば、仮コードに「ユーザーのコマンドラインで入力した値を取得」とありますが、これは必ずしも「そのためのコードをゼロから書け」という意味ではありません。幸い、Javaはオブジェクト指向言語なので、既存のクラスの中に必要な機能を備えたものが見つかることが多いのです。そのクラスを使用すれば、中がどうなっているのかが厳密にわからなくても目的の処理を行うことができます。つまり、仮コードを書く際には、「自分にこんな難しいコード（リアルコード）が書けるだろうか」などと心配する必要はないのです。まず間違いなく、何らかの方法で書けるはずだからです。そういうことに頭を悩まさず、知力のすべてをロジックを考えることに向けてください。もちろん、そのためにはロジックを考える左脳が十分にはたらかねばなりませんが……。

public static void main (String [] args)

変数の宣言　ユーザーの推測の回数を表す数値を格納するint型の変数numOfGuessesの宣言。初期値は0にする
インスタンス化　SimpleDotComクラスのインスタンスを作成する
演算　軍艦の位置（セル番号）を指定するため、乱数（0から4までの整数）を1つ発生させる
配列の作成　int型の配列を作り、発生させた乱数とそれに1を加えた値、2を加えた値を格納する（例：3、4、5）
メソッド呼び出し　SimpleDotComのインスタンスを利用してsetLocationCells()メソッドを呼び出す
変数の宣言　軍艦が沈んでいないかどうかを示すboolean型変数isAliveを宣言する。初期値はtrue

WHILEループ　軍艦が沈むまでの間（isAlive == trueの間）、同じ処理を繰り返すループ
　値の取得　ユーザーがコマンドラインで入力した値を取得
　//ユーザーの推測した番号が正しいかどうかの確認
　メソッド呼び出し　SimpleDotComのインスタンスを利用してcheckYourself()メソッドを呼び出す
　インクリメント　変数numOfGuessesのインクリメント
　//軍艦が沈んだかどうかの確認
　if文　checkYourself()メソッドの戻り値が“kill”かどうかの確認
　　値の設定　isAliveにfalseを設定する（ループを抜けるという意味）
　　出力　ユーザーの推測回数を画面に出力
　if文の終了
 whileループの終了
メソッドの終了

「認知科学的」アドバイス

脳は右脳、左脳に分かれていますが、どちらか一方だけを長時間使い続けるのはよくありません。たとえば、左脳ばかりを使い続けるのは、右腕を使わずに左腕ばかりを使うのと同じくらい不自然なことです。どちらの脳も一定時間使ったら休ませるべきでしょう。しばらく一方の脳を使ったら、使う脳を切り替えるのです。人間が左脳を使うのは、何か順を追って進める仕事をする場合や論理的思考、分析を行う場合です。また右脳を使うのは、比喩表現を考える場合や創意工夫の必要がある仕事をする場合、パターンマッチングなど、視覚的な作業をする場合です。

This PDF is prepared for tools@maruhouse.org, personal use only.

重要ポイント

- Javaプログラムの作成は、まずおおまかな仕様を確認するところから始める。
- 新しいクラスを作る際には通常、以下の 3 つのコードを作成する。
  - **仮コード**
  - **テストコード**
  - **リアルコード**
- 仮コードは「このクラスでどのような処理を行うか」を記述するもの。その処理をどのような方法で行うかは意識する必要がない。それはリアルコードを書く際に考える。
- テストコードは仮コードを踏まえて作成する。
- テストコードはリアルコードより先に書く。
- 同じ処理を何回繰り返すかがわかっている場合には、whileループではなくforループを使用する。
- 変数の値を 1 だけ増やすにはプリインクリメント演算子、あるいはポストインクリメント演算子を使用する（例：x++;）。
- 変数の値を 1 だけ減らすにはプリデクリメント演算子、あるいはポストデクリメント演算子を使用する（例：x--;）。
- 文字列を数値に変換するには、Integer.parseInt()を使用する。
- Integer.parseInt()の引数には、数字（“0”、“1”、“2”など）のみ指定できる。
- 無条件（処理を続行する条件が満たされているか否かにかかわらず）に即座にループを抜けるにはbreakステートメントを使用する。

ある日のゴーストタウン

This PDF is prepared for tools@maruhouse.org, personal use only.

SimpleDotComGameクラスのmain()メソッド

SimpleDotComGameクラスのmain()メソッドのリアルコードは以下に示すようになります。SimpleDotComクラスの場合と同様、このコードを見て、改良の余地（必要）があると思う人もいるでしょう。また、本書をここまで読んだだけではわかりにくい箇所もあるので、それについては次のページで解説しています。テストコードの工程を飛ばしたことを疑問に思う人もいるでしょうが、このクラスに関しては、テストコードは必要ありません。このクラスそのものがテストコードのようなものだからです。main()メソッドを他のクラスから呼び出しても意味はありません。

```
public static void main(String [] args) {

  int numOfGuesses = 0;

  GameHelper helper = new GameHelper();


  SimpleDotCom theDotCom = new SimpleDotCom();
                                  ①
  int randomNum = (int)(Math.random() * 5);


  int [] locations = {randomNum, randomNum+1, randomNum+2};

  theDotCom.setLocationCells(locations);

  boolean isAlive = true;


  while(isAlive == true) {
                                  ②
    String guess = helper.getUserInput("enter a number");

    String result = theDotCom.checkYourself(guess);

    numOfGuesses++;

    if (result.equals("kill")) {

      isAlive = false;

      System.out.println("You took " + numOfGuesses + "guesses");

    } //if文の終了

  } //whileループの終了

} //mainの終了
```

ユーザーの推測の回数を記録するint型の変数の宣言。初期値は0にする。

ユーザーの推測回数を記録するための変数の宣言

著者が作ったクラス。ユーザーの入力した値を取得するためのメソッドを備えている。ここでは、あえてJavaの開発環境にあらかじめ用意されているかのように使っている

SimpleDotComクラスのインスタンスの作成。

SimpleDotCom オブジェクトの作成

乱数（0から4までの整数）を1つ発生させる。

乱数を発生させる。この乱数は軍艦の配置を表すセル番号の配列を作るのに利用する

int型の配列の作成し、3つのセル番号を格納。

SimpleDotComのsetLocationCells()メソッドを呼び出す。

配列の値をSimpleDotComオブジェクトのインスタンス変数に代入する

boolean型変数isAliveの宣言。

軍艦が沈んだかどうかを表すboolean型の変数を宣言する。この変数の値を調べることで、ループの処理を続行するか否かが決められる

isAlive == trueの間、同じ処理を繰り返すwhileループ。

ユーザ入力値を取得。

ユーザーの入力した値を取得する

// 入力値の確認。SimpleDotComのcheckYourself()メソッドを呼び出す。

ユーザーの推測した値が正しいかどうかを確認するメソッドの呼び出し。戻り値はString型変数に格納

変数numOfGuessesのインクリメント。

推測回数をインクリメント

戻り値が"kill"かどうかの確認。

戻り値が"kill"かどうかを確認。"kill"であれば、isAliveの値をfalseにする（ループを抜ける）。最後にユーザーの推測回数を画面に表示

isAliveにfalseを設定する。

ユーザーの推測回数を画面に出力。

This PDF is prepared for tools@maruhouse.org, personal use only.

仮コード　テストコード　リアルコード

random()とgetUserInput()

前ページのリアルコードには、補足説明が必要と思われる箇所が2つほどあります。GameHelperクラスに関しては本章の終わりで詳しく説明しますが、ここでも取りあえず読み進むのに支障がない程度に簡単に触れておきます。

これによって、値の型を強制的に変える「キャスト」という処理が行える。どの型に変えたいかは()の中に指定する。Math.random()というメソッドは本来double型の値を戻すのだが、ここではそれをint型にキャストしている(0～4の「整数」が必要なため)。これによって、小数点以下が切り捨てられる

Math.random()は0以上1未満の値を戻すメソッド。したがって、この式とキャストによって0～4の整数が得られることになる(キャストなしでは、得られる値は0～4.999...になる)

① 乱数の発生

```
int randomNum = (int)(Math.random() * 5)
```

発生させた乱数を格納するためのint型の変数を宣言する

Javaにあらかじめ用意されているクラス

Mathクラスのメソッド

② GameHelperクラスを使用してユーザー入力を取得する

このゲームプログラムのために著者が作ったGameHelperクラスのインスタンスへの参照。このクラスの詳細については後述

引数は、画面に表示するユーザーへの指示。ここに指定された文字列が画面に表示されると、ユーザーは推測したセル番号を入力することができる

```
String guess = helper.getUserInput("enter a number");
```

ユーザーの入力した値("3"、"5"などの文字列)を格納するString型の変数を宣言

ユーザーにコマンドライン入力を求めるGameHelperクラスのメソッド。ユーザーがリターンキーを押すと、入力された値を読み取り、それをString型の値として戻す

This PDF is prepared for tools@maruhouse.org, personal use only.

GameHelperクラス

この章で例に使用しているバトルシップゲームプログラムでは、軍艦に対応するクラス、ゲーム自身を表すクラスの他に、もう1つのクラスを利用します。ユーザーのコマンドライン入力を取得するためのgetUserInput()メソッドを備えたGameHelperクラスです。コマンドライン入力を取得するという処理に関しては、ここで詳しく説明することはしません。本書のかなり後の章（14章）で出てくることがらについて知識がなければ理解できないからです。

ここでは個々のコードの意味は特に気にせずに下に示したGameHelperクラスのコードをそのまま書き写し*、SimpleDotComクラス、SimpleDotComGameクラスとともにコンパイルしてください。コンパイル時にはすべてのクラスのコードを同じディレクトリに置き、そのディレクトリをワーキングディレクトリにします。

この「レディメードコード」というロゴは、「著者があらかじめコードを書いておいたので、書き写して使ってください」という意味です。個々のコードの意味などは後で解説することになります。

```
import java.io.*;
public class GameHelper {
  public String getUserInput(String prompt) {
    String inputLine = null;
    System.out.print(prompt + " ");
    try {
      BufferedReader is = new BufferedReader(
      new InputStreamReader(System.in));
      inputLine =is.readLine();
      if (inputLine.length() == 0 )return null;
    } catch (IOException e) {
      System.out.println("IOException:" +e);
    }
    return inputLine;
  }
}
```

* コードをタイピングするのは楽しい作業ですが、「そのための時間を他のことに使いたい」という人もいるでしょう。そういう人のために、headfirstjava.com にコードが用意してあります。

This PDF is prepared for tools@maruhouse.org, personal use only.

プログラムを実行してみる

これまでに作成したプログラムを実行してみましょう。以下に示すように、1、2、3、4、5、6 という具合に順に数字を入力するとプログラムは問題なく動作しているように見えます。

実行画面
（実行環境によって
若干異なることもあります）

```
File Edit Window Help Smile
%java SimpleDotComGame
enter a number 1
miss
enter a number 2
miss
enter a number 3
miss
enter a number 4
hit
enter a number 5
hit
enter a number 6
kill
You took 6 guesses
```

あれっ？　これはバグ？

先述のとおり、ここで紹介したコードにはバグが含まれています。たとえば、ユーザーが 1、1、1 という具合に同じ数字を続けて 3 回入力すると以下のような結果が得られます。

**同じ数字を続けて 3 回
入力した場合の実行結果**
（バグ）

```
File Edit Window Help Faint
%java SimpleDotComGame
enter a number 1
hit
enter a number 1
hit
enter a number 1
kill
You took 3 guesses
```

自分で考えてみよう

バグについて考える

どこがバグなのか
わかったでしょうか。
どうすれば修正できるでしょうか。
果たして……

このバグについては次章で詳しく説明します。ただ、何が問題なのか、どうすれば解決するのかを自分でも考えてみてください。

This PDF is prepared for tools@maruhouse.org, personal use only.

forループの基礎知識

この章では簡易版のゲームプログラムのコードについて解説してきました(この章で紹介したコードは、次章で「完全版」を作る際にも出てきます)。ただ、プログラミング言語の基本事項に関しては、ゲームプログラムそのものについての説明がかえってわかりにくくなる恐れがあるとの判断から、あまり取りあげませんでした。ここで「ループ」について一通りの説明をしておきますが、自信のある人(特にC++のプログラマ)は、飛ばして次に進んでもいいでしょう。

forループ

ポストインクリメント演算子

繰り返すべき処理のコード(ループの本体)

```
for(int i = 0; i < 100; i++){ }
```

初期化

条件判定

カウンタ変数のインクリメント

同じ処理を100回繰り返すforループ

上記のコードの意味

* 変数iを宣言し、初期値を0にする。
* iの値が100未満の間、同じ処理を繰り返す。
* 処理を一度行うごとにiの値を1増やす。

初期化

forループではまず、ループの「本体」で使用する変数を初期化します。この変数は多くの場合、「カウンタ変数」となります。複数の変数を初期化する場合もありますが、その点については後述します。

条件判定

次にループの処理を続行する条件を満たしているかを判定するコードを書きます。条件判定には、boolean型の値(true、あるいはfalse)を利用することになります。x >= 4というような式を書くか、trueかfalseを戻すメソッドを呼び出すのです。

繰り返し処理後に行うべきことの指定

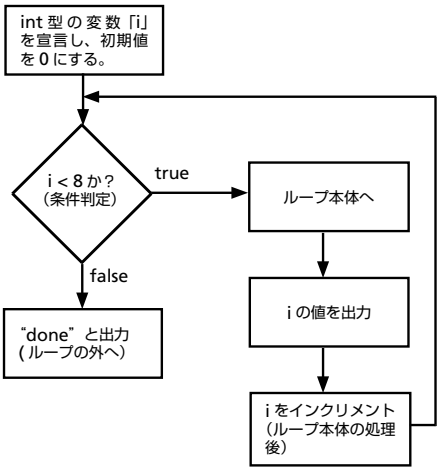
条件判定のコードの後には、「繰り返すべき処理を1回行うごとにどのようなことを行えばよいか」を指定するためのコードを書きます。行うことは1つとは限りません。また、ここで指定した処理は、繰り返すべき処理が行われた「後」に行われる、という点に注意が必要です。

腕立てふせを100回繰り返す

This PDF is prepared for tools@maruhouse.org, personal use only.

ループの処理

```
for (int i =0; i <8; i++) {
  System.out.println(i);
}
System.out.println("done");
```



for と while の違い

while文では条件判定のみを行います。初期化や、カウンタ変数のインクリメントなどはwhile文の外で行うことになります。while文を使ったループは、処理を繰り返す回数は一定していないが、ともかく、ある条件が満たされている間は同じ処理を繰り返したい、という場合に使うのに便利です。しかし、繰り返しの回数が明確にわかっている場合には、forループを使った方がいいでしょう。上記のforループのコードをwhileループに変えると以下のようになります。

```
int i = 0;          ← カウンタ変数の宣言と初期化
while (i < 8) {
  System.out.println(i);
  i++;              ← カウンタ変数のインクリメント
}
System.out.println("done");
```

実行結果

```
File  Edit  Window  Help  Repeat
%java Test
0
1
2
3
4
5
6
7
done
```

++　　--
インクリメント/デクリメント演算子

インクリメント、デクリメント演算子は、変数の値を 1 増やす、あるいは 1 減らすのに使用する演算子です。

```
x++;
```

というコードは以下のコードと同じ意味になります。

```
x = x + 1;
```

つまり、インクリメント演算子を使えば、変数の現在値に 1 を加えることができるのです。

また、

```
x--;
```

は、以下のコードと同じ意味です。

```
x = x - 1;
```

ただ、ここで注意しなければならないのは、インクリメント、デクリメント演算子（++、--）は位置によって（変数の前に置くか後に置くかによって）意味が異なることです。変数の前に置くと（例：++x）、まず変数をインクリメントしてからインクリメント後の値を利用する、という意味になります。演算子の配置が問題になるのは、以下のような複数の演算子を使った式においてです。

```
int x = 0;        int z = ++x;
```

このコードでは、最終的にxの値もzの値も 1 になります。しかし、以下のように++をxの後に置くと、結果が異なってきます。

```
int x = 0;        int z = x++;
```

このコードでは、最終的にxは 1 になりますが、zは 0 になります。インクリメントより先に代入が行われるからです。

This PDF is prepared for tools@maruhouse.org, personal use only.

拡張forループ

Java 5.0（Tiger）から「拡張forループ」と呼ばれる新しい種類のforループが使えるようになっています。これにより、配列などのコレクション（配列以外のコレクションに関しては次章を参照）の各要素について順に同じ処理を行う、というコードが簡単に書けるようになりました。それ以外のメリットがあるわけではないのですが、forループは多くの場合この目的で使用するので、新たな形式が追加された意味は大きいでしょう。拡張forループに関しては、配列以外のコレクションに関する説明の際に再度触れます。

配列の要素を代入する変数を宣言

コロン（：）は「in」を意味する

繰り返し実行するコードはここに書く（ループの本体）

```
for (String name : nameArray) { }
```

配列の要素の型は、宣言する変数に代入できるものでなければならない

処理を繰り返す度に、変数nameに新たな要素が代入される

繰り返し処理の対象となるコレクション。仮にこのコードより前に以下のように宣言されていたとすると、

String[] nameArray = {"Fred", "Mary", "Bob"};

変数nameにはまず"Fred"、次に"Mary"が代入されることになる

上記のコードの意味を言葉で書くと「nameArrayの各要素を順に変数nameに代入し、それぞれについてループ本体を実行する」となります。

コンパイラの処理

- String型の変数nameを作成し、nullを設定する。
- nameArrayの最初の要素をnameに代入する。
- ループの本体（{}の中のコード）を実行する。
- nameArrayの次の要素をnameに代入する。
- 配列の末尾に到達するまで同じ処理を繰り返す。

注：他の言語を学習してきた経験のある人には、この「拡張forループ」は「for eachループ」あるいは「for inループ」と同じようなもの、と言えばわかりやすいかもしれない。コレクション中の各要素について同じ処理を繰り返すという点では同じである

前半部分：要素を代入する変数の宣言

前半部分には、配列（コレクション）の要素を代入する変数の宣言と初期化のためのコードを書きます。ループの本体では、この変数を使用した処理を行うことになります。処理が繰り返される度に、変数には新たな要素が代入されます。変数の型は、配列の要素を代入できるものでなくてはなりません。配列がString型であれば、たとえばint型の変数を宣言することはできない、ということになります。

後半部分：コレクションの指定

後半部分には、ループ処理の対象となる配列（コレクション）を指定します。すでに述べたとおり、配列以外のコレクションに関しては次章で触れるので今わからなくても心配はいりません。

This PDF is prepared for tools@maruhouse.org, personal use only.

String を int に変換する

```
int guess = Integer.parseInt(stringGuess);
```

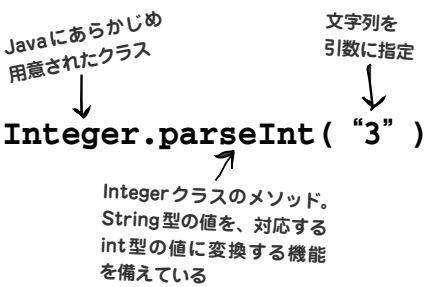
ここで紹介しているゲームプログラムでは、画面に指示が出ると、ユーザーが推測したセル番号を入力することになります。このセル番号ははじめString型の値（文字）になり、checkYourself()メソッドに渡されます。
しかし、軍艦の配置を表すものとして配列に格納されるセル番号はint型の値です。つまり、そのままではString型の値とint型の値を比較することになりますが、そういうことはできません。
たとえば、以下のようなコードは正しくないのです。

```
String num = "2";
int x = 2;
if (x == num) // エラー！
```

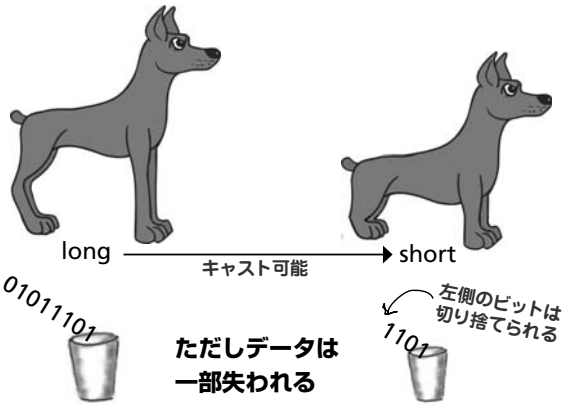
このコードをコンパイルしようとすると、次のようなコンパイルエラーが発生します。

```
operator == cannot be applied to
      int,java.lang.String
    if (x == num) { }
            ^
```

この問題を解決するには、String型の値をint型に変換する必要があります（たとえば、文字「2」を数値「2」に変換する）。幸い、Javaにはあらかじめ Integer（intではない）というクラスが用意されています。このクラスは、数字をそれに対応する数値に変換する機能を備えているのです。



プリミティブ型のキャスト



3 章で様々なプリミティブ型のサイズについて触れ、サイズの小さい変数に大きな値が格納できないということも述べました。

```
long y = 42;
int x = y; // コンパイルできない
```

この場合、long型はint型よりサイズが大きいので、コンパイラは安全を保証できず、エラーを発生させます。42 という値自体はさほど大きくなく、int型でも扱えるのですが、コンパイラはあくまで型のサイズに注目するのです。どうしてもサイズの大きい方の値をサイズの小さい方の変数に代入したい時は、「キャスト」という処理を行います。キャストはたとえば以下のように行います。

```
long y = 42; //long 型の変数 y に 42 が代入される
int x = (int)y; // 問題なく処理が行われる
```

上記のコードでは、変数yの値の一部（左側のビット）を切り捨て、intのサイズにしてxに代入します。ただ、yの値がint型の最大値より大きいと、思いもかけない値が得られる恐れがあります*。

```
long y = 40002;
//short 型は 16 ビットだが、40002 は 16 ビットには収まらない
short x = (short)y; //x の値は -25534 になってしまう！
```

しかし、キャストはうまく使えば非常に便利です。たとえば、次に示すように、浮動小数点数の整数部分だけが欲しいという場合などにも使えるのです。

```
float f = 3.14f;
int x = (int)f; //x は 3 になる
```

注意すべきなのは、boolean 型の値をキャストする、他の型の値をキャストして boolean 型の変数に代入する、といったことはできない、ということです。

*　この現象について理解するには、符号ビット、2 進数、2 の補数など、情報処理に関する基礎的な知識が必要になります。詳しくは付録 B 参照。

This PDF is prepared for tools@maruhouse.org, personal use only.

エクササイズ

JVMになってみよう

下のソースコードを、JVMになったつもりで見てみましょう。そして、実行結果が右の3つのうちどれになるかを答えてください。

```
class Output {

  public static void main(String [] args) {
    Output o = new Output();
    o.go();
  }

  void go() {
    int y = 7;
    for(int x = 1; x < 8; x++) {
      y++;
      if (x > 4){
        System.out.print(++y + " ");
      }
      if (y > 14){
        System.out.println(" x = " +x);
        break;
      }
    }
  }
}
```

```
File  Edit  Window  Help  OM
%java Output
12 14
```

```
File  Edit  Window  Help  Incense
%java Output
12 14 x = 6
```

```
File  Edit  Window  Help  Believe
%java Output
13 15 x = 6
```


This PDF is prepared for tools@maruhouse.org, personal use only.

エクササイズ

プログラミングマグネット

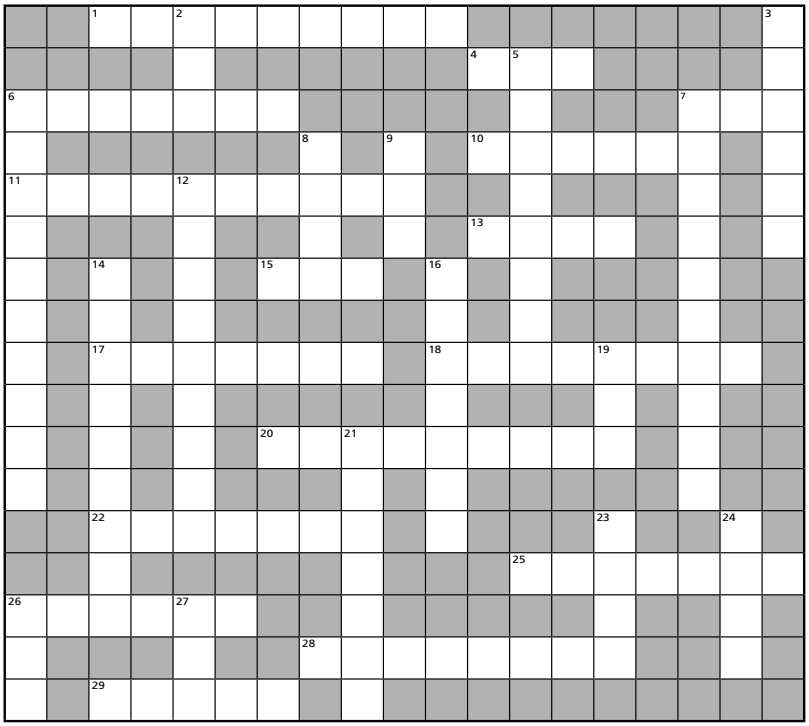
バラバラになったJavaプログラムを並べ替え、下のような実行結果が得られるようにしてください。もし、{ 、} などが抜けているところがあれば付け加えてもかまいません。

```
x++;
if (x == 1) {
System.out.println(x + " " + y);
class MultiFor {
for(int y = 4; y > 2; y--) {
for(int x = 0; x < 4; x++) {
public static void main(String [] args) {
```

```
File  Edit  Window  Help  Raid
%java MultiFor
0 4
0 3
1 4
1 3
3 4
3 3
```


This PDF is prepared for tools@maruhouse.org, personal use only.

Javaに関する用語のクロスワードです。本書をよく読んでいれば解けるはずです。多少ひねった問題もありますが、少し違った角度から用語を捉えることはJavaの習得に役立つはずです（注：このクロスワードには英語で、すべて大文字で答えてください）。



ヨコのカギ

1. 「実現する」という単語。Imp...
4. ループの一種
6 ソフトウェア開発技法
7. 32ビット
10. メソッドの実行結果
11. 自然言語によるコード
13. 型の変換
15. クラスの集まり
17. 配列を構成する
18. インスタンス____、ローカル____
20. インポートの必要がないパッケージ
22. Intではない
25. キャストできない
26. 乱数
28. Stringをintに
29. ループを抜ける

タテのカギ

2. ポストインクリメント、____インクリメント
3. クラスの機能
5. ++、--など
6. 「式」という単語。Exp...
7. 初めの値を決める
8. While、あるいはFor
9. 変数の値を決める
12. 減らす
14. ループ
16. 入出力のパッケージ
19. パラメータ
21. 「仮想」という単語。Vir...
21. 後
24. random()
26. 実行
27. ++でいくつ増える？

ヒント

仮コード＝Pseudo Code
要素＝Element
変数＝Variable
メソッド＝Method
演算子＝Operator
初期化＝Initialize
繰り返し＝Iteration
引数＝Argument（Argと略することもある）

This PDF is prepared for tools@maruhouse.org, personal use only.

プログラムパズル

下のプログラムは一部分が欠落しています。欠落部分に入るコードの候補がいくつか示してありますが、それぞれのコードを当てはめた場合にどのような実行結果が得られるかを考え、コードと実行結果を線で結んでください（実行結果の中には、どのコードとも結び付かないものがあります）。

```
class MixFor5 {
  public static void main(String [] args) {
    int x = 0;
    int y = 30;
    for (int outer = 0; outer < 3; outer++) {
      for(int inner = 4; inner > 1; inner--) {

        [                              ]   ← この部分にコードが入る

        y = y - 2;
        if (x == 6) {
          break;
        }
        x = x + 3;
      }
      y = y - 2;
    }
    System.out.println(x + " " + y);
  }
}
```

コードの候補

- `x = x + 3;`
- `x = x + 6;`
- `x = x + 2;`
- `x++;`
- `x--;`
- `x = x + 0;`

実行結果

- `45 6`
- `36 6`
- `54 6`
- `60 10`
- `18 6`
- `6 14`
- `12 14`

コードと実行結果を線で結ぶ

This PDF is prepared for tools@maruhouse.org , personal use only.

エクササイズの答え

JVM になってみよう

```
class Output {

  public static void main(String [] args) {
    Output o = new Output();
    o.go();
  }
  void go() {
    int y = 7;
    for(int x =1; x < 8; x++) {
      y++;
      if (x > 4) {
        System.out.print(++y + " ");
      }
      if (y > 14) {
        System.out.println(" x ="  + x);
        break;
      }
    }
  }
}
```

ポイントはbreakステートメントです。breakステートメントは実行結果にどのような影響を与えるでしょうか。

```
File  Edit  Window  Help  MotorcycleMaintenance
%java Output
13 15 x = 6
```

プログラムマグネット

```
class MultiFor {

  public static void main(String [] args) {

    for(int x = 0; x < 4; x++) {

      for(int y = 4; y > 2; y--) {
        System.out.println(x + " " +y);
      }

      if (x == 1) {
        x++;
      }
    }
  }
}
```

このコードが 2 つのforループの前にあったら、実行結果はどのように変わるでしょうか。

```
File  Edit  Window  Help  Motorcycle
%java MultiFor
0 4
0 3
1 4
1 3
3 4
3 3
```


This PDF is prepared for tools@maruhouse.org, personal use only.

Javaクロスワード

		1 I	M	2 P	L	E	M	E	N	T							3 M
				R							4 F	5 O	R				E
6 E	X	T	R	E	M	E						P				7 I	N	T
X							8 L		9 S		10 R	E	T	U	R	N		H
11 P	S	E	U	12 D	O	C	O	D	E			R				I		O
R				E			O		T		13 C	A	S	T		T		D
E		14 I		C		15 A	P	I		16 J		T				I		
S		T		R						A		O				A		
S		17 E	L	E	M	E	N	T		18 V	A	R	I	19 A	B	L	E	
I		R		M						A				R		I		
O		A		E		20 J	A	21 V	A	.	L	A	N	G		Z		
N		T		N				I		I						E		
		22 I	N	T	E	G	E	R		O				23 P			24 M	
		O						T				25 B	O	O	L	E	A	N
26 R	A	N	D	27 O	M			U						S			T	
U				N			28 P	A	R	S	E	I	N	T			H	
N		29 B	R	E	A	K		L										

コードの候補

- `x = x + 3;`
- `x = x + 6;`
- `x = x + 2;`
- `x++;`
- `x--;`
- `x = x + 0;`

実行結果

- `45 6`
- `36 6`
- `54 6`
- `60 10`
- `18 6`
- `6 14`
- `12 14`

This PDF is prepared for tdpc@maruhouse.net, personal use only.

This PDF is prepared for tools@maruhouse.org, personal use only.

6章　Java APIの基礎

Javaライブラリを使用する

Javaの開発環境にはあらかじめ数多くのクラスが用意されています。そうしたクラスの集まりをJavaライブラリ（Java API）と呼びます。作りたいプログラムに合うクラスをJavaライブラリで見つけることができれば、自分で同じようなクラスを改めて作る必要はなくなります。その分の時間や労力を他のことに振り向けることができるのです。Java APIを使用すれば、多くのプログラムに共通して必要なコードは書く必要がなくなり、それぞれのプログラムの、他のプログラムにない特徴的な部分だけを書けばよくなります。毎朝10時に会社に来て5時に帰るのに滞りなく仕事が進んでいる、というプログラマはJava APIを有効に利用しているのかもしれません。この章でも、このJava APIのクラスを利用することになります。Java APIの中でも特に重要なのが「コアライブラリ」です。通常は、プログラムのかなりの部分が、コアライブラリに用意された大量のクラスを「ブロック」のように使うだけでできあがります。Java APIのクラスは、前の章に出てきた「著者の作ったクラス（レディメードコード）」のように、改めてコードをタイプするような必要もありません。ただ、使い方さえ覚えれば、中のコードがどのようになっているかを知らなくても利用できるのです。

This PDF is prepared for tools@maruhouse.org, personal use only.

簡易版バトルシップゲームプログラムのバグ

プログラムの正しい動き

前章で紹介した簡易版バトルシップゲームプログラムは、1、2、3、4、5、6という具合に順に番号を入力すると、以下のように動作します。これは正しい動きです。

バグの出るケース

2、2、2というように同じ数字を続けて入力すると、以下のような誤った動きをする。

正しい実行結果

（実行環境によって多少異なる場合もある）

```
File  Edit  Window  Help  Smile
%java SimpleDotComGame
enter a number 1
miss
enter a number 2
miss
enter a number 3
miss
enter a number 4
hit
enter a number 5
hit
enter a number 6
kill
You took 6 guesses
```

誤った実行結果

（これはおかしい……）

```
File  Edit  Window  Help  Faint
%java SimpleDotComGame
enter a number 2
hit
enter a number 2
hit
enter a number 2
kill
You took 3 guesses
```

本来3つのセル番号を当てなければならないはずが、今のままでは同じセル番号を3回推測するだけで軍艦が沈没してしまう！

This PDF is prepared for tools@maruhouse.org, personal use only.

バグの原因は？

```
public String checkYourself(String stringGuess) {

  int guess = Integer.parseInt(stringGuess);

    String result = "miss";

    for(int cell : locationCells) {

      if (guess == cell) {

        result = "hit";

        numOfHits++;

        break;

      } //if文の終了

    } //forの終了

    if (numOfHits == locationCells.length) {

      result = "kill";

    } //if文の終了

    System.out.println(result);

    return result;

  } //メソッドの終了
```

- String を int に変換
- 戻り値となる値を格納するための変数を宣言。デフォルト値は“miss”にする（何の操作も加えられなければ戻り値は“miss”になる）
- 配列の各要素に関して同じ処理を繰り返す
- ユーザーの推測値と、配列の要素（いずれもセル番号）を比較する
- 推測が正解だった場合の処理
- 他のセル番号については確認することなく無条件でループを抜ける
- ループを抜け、軍艦が沈んでいるかどうか（正確が3回に達しているかどうか）を確認する。沈んでいれば戻り値を“kill”にする
- 戻り値を画面に表示する（“hit”でも“kill”でもなければ“miss”になる）
- メソッドの実行結果を呼び出し側に戻す

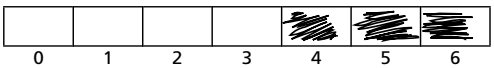
ここが問題の箇所。正解の回数を記録する変数の値を、「すでに同じセル番号が推測されていないか」を確認することなくインクリメントしている。

正しく動作させるには、ユーザーの入力したセル番号が、以前に入力されたものと重複していないかを何らかの方法で確認する必要がある。重複していれば、たとえ正解でも正解とカウントしないようにする。

This PDF is prepared for tools@maruhouse.org, personal use only.

バグの修正方法

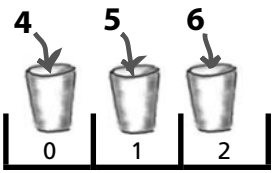
プログラムを正しく動作させるには、先述のとおり、ユーザーの入力したセル番号が、以前に入力されたものと重複していないかを何らかの方法で確認する必要があります。ここでは、どのような方法があるかを見てみましょう。ただ、その前に、プログラムに関してこれまでにわかっていることを再確認します。
このプログラムでは、軍艦が 7 つのセルのうちのいずれか 3 つに配置されます。ただし、3 つのセルの番号は常に連続します。たとえば、下の図では、セル番号 4、5、6 に配置されています。



7 つのセルの中の連続する 3 つのセルに軍艦が配置される

軍艦を表す SimpleDotCom オブジェクトは、インスタンス変数として、軍艦の配置（セル番号）を格納する int 型の配列を持っている

locationCells
（SimpleDotCom クラスのインスタンス変数）

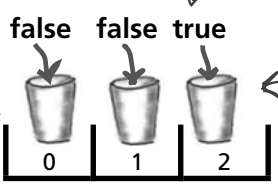


この配列には、軍艦が配置された 3 つのセル番号が格納される。ここでは 4、5、6 が格納されている。ユーザーはこの番号を推測しなければならない

① **方法 1**
もう 1 つ配列を作り、その配列に、各セル番号がすでに当てられているか否かを表す値を格納する。ユーザーがセル番号を当てたら、その配列の要素を逐一確認する。

“true” というのは、もう 1 つの配列（locationCells）の同じ番号に格納されているセル番号はすでに当てられているという意味。

配列 hitCells
（boolean 型の配列。この配列を新たに SimpleDotCom クラスのインスタンス変数とする）



この配列には、3 つのセル番号がそれぞれすでに当てられているかどうかを表す値が格納される。たとえば、locationCells 配列の要素番号 2 のセルがすでに当てられていれば、hitCells の要素番号 2 の値は true になる。

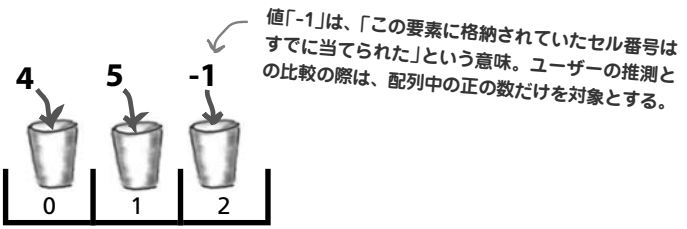
This PDF is prepared for tools@maruhouse.org, personal use only.

方法 1 はあまりに複雑

方法 1 で確かにバグは治るのですが、「意外に面倒」と感じた人が多いのではないでしょうか。ユーザーがセル番号を当てる度に、新たに作った配列（hitCells）で、同じセル番号がすでに当てられていないかを確認しなければなりません。まだ当てられていなければ、配列の値の変更も必要になります。これでもいいのでしょうが、もっと良い方法はないのでしょうか……。

② **方法 2**

新たに配列は作らない。セル番号が当てられた時に、その番号が格納されている配列の要素の値を「-1」に変更する。この方法だと確認、操作する配列が 1 つで済む。



方法 2 は方法 1 より少しはよいが……

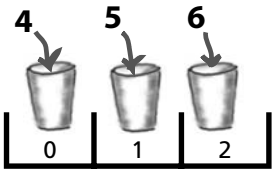
方法 2 は方法 1 に比べ、若干処理がシンプルになっています。しかし、それでもやはり非効率な部分が残ります。いくつか当たりが出て、配列の値が比較の対象外（-1）になっても、相変わらずループを使って 3 つの要素について確認を行うからです。もっとよい方法があるのではないでしょうか。

This PDF is provided for your personal use only.
ccplasr@dmfourth.com

③ **方法 3**

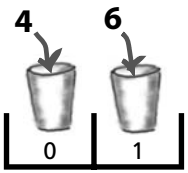
セル番号が当てられたら、その番号を削除し、配列を小さくする。厳密には、配列のサイズを変えるということはできないので、新たな配列を作り、そこに元の配列の残すべき要素をコピーすることになる。

「当たり」が出る前のlocationCells配列



はじめの要素数は 3 で、ループを使い、各要素とユーザーの推測したセル番号が一致しているかどうかを確認する

番号 1 の要素である「5」が当てられた後のlocationCells配列



「当たり」が出たら、要素数の少ない配列を新たに作り、古い配列中のまだ当たっていない要素をコピーする。

方法 3 は、もし、「新たな配列を作って、古い配列の要素をコピーする」ということが必要なく、単純に配列のサイズを小さくすることができれば、はるかに簡単になる。

checkYourself() メソッドの修正前の仮コード

繰り返し処理　int の配列の個々の要素について同じ処理を繰り返す

// ユーザーの推測したセル番号と軍艦が実際に配置されている
// セル番号の比較

if 文　セル番号が一致しているかどうかの判定

インクリメント　推測が一致した回数を表す値を 1 増やす

// 一致が 3 回目かどうかを確認

if 文　一致の回数が 3 に達していれば、文字列“kill”を戻す

else 文　一致の回数が 3 に達していなければ“hit”を戻す

（内側の）if 文の終了

else 文　セル番号が一致していなければ、“miss”を戻す

（外側の）if 文の終了

繰り返し処理の終了

理想的な修正案

繰り返し処理　int の配列の残っている要素すべてについて同じ処理を繰り返す

// ユーザーの推測したセル番号と軍艦が実際に配置されている
// セル番号の比較

if 文　セル番号が一致しているかどうかの判定

配列要素の削除　推測が当たっていたセル番号の格納されている配列要素を削除

// 一致が 3 回目かどうかを確認

if 文　配列が空であれば、文字列“kill”を戻す

else 文　一致の回数が 3 に達していなければ“hit”を戻す

（内側の）if 文の終了

else 文　セル番号が一致していなければ、“miss”を戻す

（外側の）if 文の終了

繰り返し処理の終了

This PDF is prepared for tools@maruhouse.org, personal use only.

This PDF is prepared for tools@maruhouse.org, personal use only.

Java APIのクラスを使う

ウソのようですが、彼女が夢見ているような配列に近いものはあります。
それは正確には配列ではなく、ArrayListというクラスです。
このクラスはJava APIのコアライブラリに用意されているものです。

Java 2 SE（Standard Edition）には、あらかじめ数多くのクラスが用意されています（小型機器向けのJavaであるME = Micro Editionでは事情が異なる）。
レディメードコードのマークがついたクラスに似ていますが、すでにコンパイルされ、すぐ使える状態になっているところが違っています。

コードをタイプする必要はありません。

ただ使えばいいのです。

Javaライブラリには多数（約2300）のクラスが用意されている

用意されたクラスは自分で書いたものと同じように利用できる

（注：add(Object elem)メソッドは、正確にはここでの説明のとおりのメソッドではない。詳細については後述するが、今の時点ではadd()を「指定のオブジェクトをリストに追加するメソッド」と覚えておいてかまわない）

ArrayList

add(Object elem)
引数に指定されたオブジェクトをリストに追加する。

remove(int index)
引数に指定されたオブジェクトを削除する。

remove(Object elem)
引数に指定されたオブジェクトを（そのオブジェクトがArrayList中にあれば）削除する。

contains(Object elem)
引数に指定されたオブジェクトと同じものがリストにあれば“true”を戻す。

isEmpty()
リストに要素がなければ“true”を戻す。

indexOf(Object elem)
引数に指定されたオブジェクトのリスト中での番号（インデックス）、あるいは-1を戻す。

size()
リスト中に存在する要素の数を戻す。

get(int index)
引数に指定された番号（インデックス）に対応するオブジェクトを戻す。

ArrayListのメソッドはこれですべてではない

This PDF is prepared for tools@maruhouse.org, personal use only.

ArrayListクラスの使い方

これでヒープ上にArrayListオブジェクトができる。作ったばかりの状態では空なのであまり容量を必要としない

① **ArrayListオブジェクトを作る**

```
ArrayList<Egg> myList = new ArrayList<Egg>();
```

<Egg>の部分は「Eggオブジェクトのリストを作る」というくらいの意味。現時点ではわからなくても心配は要らない。

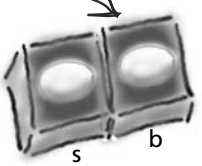
② **要素(オブジェクト)を追加する**

```
Egg s = new Egg();

myList.add(s);
```

左のように要素を追加すると、ArrayListオブジェクトのサイズは、Eggオブジェクトを格納できるように大きくなる

③ **要素(オブジェクト)をさらに追加する**

```
Egg b = new Egg();

myList.add(b);
```



さらにEggオブジェクトを追加すると、ArrayListオブジェクトはその分大きくなる

④ **ArrayList中の要素の数を確認する**

```
int theSize = myList.size();
```

ArrayListオブジェクトに2つのオブジェクトが格納されているため、size()メソッドは2を戻す

⑤ **ArrayListオブジェクトに特定のものが格納されているか確かめる**

```
boolean isIn = myList.contains(s);
```

ArrayListオブジェクトには引数に指定されたEggオブジェクトが格納されているため、contains()はtrueを戻す

⑥ **特定の要素がどこに格納されているかを(つまり要素のインデックスを)確かめる**

```
int idx = myList.indexOf(b);
```

ArrayListのインデックスは0から始まる。変数bに対応するオブジェクトは2番目の要素なので、indexOf()メソッドは1を戻す

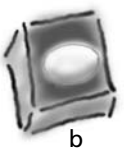
⑦ **ArrayListオブジェクトが空かどうか確かめる**

```
boolean empty = myList.isEmpty();
```

ArrayListオブジェクトは空ではないので、isEmpty()メソッドはfalseを戻す

⑧ **ArrayListオブジェクトから指定の要素を削除する**

```
myList.remove(s);
```



要素が減ると、ArrayListオブジェクトは小さくなる

This PDF is prepared for tools@maruhouse.org, personal use only.

自分で考えてみよう

以下の表はArrayListクラスと通常の配列を比較するためのものです。左側のArrayListクラスを使ったコードと同じ結果が得られるよう、通常の配列を使ってコードを書き、右側の空欄になっているところを埋めてみましょう。すぐに正解は得られないかもしれませんが、できる限り考えてみてください。

ArrayListクラス	通常の配列
`ArrayList<String> myList = new ArrayList<String>();`	**`String [] myList = new String [2];`**
`String a = new String("whoohoo");`	**`String a = new String("whoohoo");`**
`myList.add(a);`	
`String b = new String("Frog");`	**`String b = new String("Frog");`**
`myList.add(b);`	
`int theSize = myList.size();`	
`Object o = myList.get(1);`	
`myList.remove(1);`	
`boolean isIn = myList.contains(b);`	

This PDF is prepared for tools@maruhouse.org, personal use only.

素朴な疑問に答えます

Q: ArrayListが便利なのはわかりました。でも、こういったクラスの存在をどうやって知ればよいのですか。

A: おっしゃりたいことはわかります。「Java APIには無数のクラスの中から目的に合うものを探すのは困難だ」というのですね。ただ、一方で、APIから適切なクラスを見つけ出すということが、Javaプログラマとして成功するために非常に重要である、というのも事実です。APIをうまく活用すれば、時間をかけずに楽をして高度なソフトウェアが作れるからです。APIを使うことで、プログラムの中でも面倒で退屈な部分を人にまかせ、最も「面白い」部分だけを自分で作る、ということが可能になります。少し話がそれました……。APIにどのようなクラスが存在するかを知るには、APIについてある程度長い時間をかけて学習するしかありません。その具体的な方法については、この章の最後に説明します。

Q: それにしてもあまりに大変ではありませんか。まず、Java APIにArrayListというクラスが存在することを知らなければならないし、しかも、そのクラスが自分の目的に合うものかも確かめなくてはならないというのは……。結局、APIを使って「何をしなければならないか」がわかっても、「それをどうすればよいのか」がなかなかわかりません。

A: 確かにそのとおりだろうとは思います。しかし、本書を最後まで読み、Javaについての知識が深まれば、APIを活用し、自分で書くコードの量を最小限に抑えて目的のプログラムを作るということは徐々にできるようになってくるはずです。ともかく、APIの活用については、この章の最後に詳しく説明しますので、まずはそこまで頑張って読み進めてください。

Javaの真実

ゲスト：ArrayListクラス

著者：あなたと配列とは似ていますね。

ArrayList：向こうはそう思っているかもしれません。ただ、おかげさまで私はオブジェクトですから。やはり配列とは違うと思いますよ。

著者：私の記憶では、配列オブジェクトの一種ですよね。他のオブジェクトと同様にヒープに置かれますし。

ArrayList：配列がヒープに置かれるというのは事実ですが、完全なオブジェクトとは言えません。オブジェクトにはインスタンス変数とメソッドがあるというのはご存知だと思います。ArrayListには確かにその両方がありますが、「配列のメソッドを呼び出す」という話を聞いたことがありますか。

著者：言われてみればそのとおりですが、そもそも配列のメソッドを呼び出す必要があるのでしょうか。配列の中のデータを利用するために何らかのクラスのメソッドを呼び出すということはあっても、配列自身のメソッドを呼び出すという発想はないですね。配列自身のメソッドがなくても、配列にデータを入れたり、配列からデータを取り出したりできればいいわけですし。

ArrayList：ほう、あなたは、本当に配列からデータを取り出すことができると思っているのですか。

著者：もちろんです。たとえば、「Dog d = dogArray[1]」というようなコード書けば、配列のインデックス「1」に対応するデータが取り出せますよね。

ArrayList：ゆっくり説明しますので、よく聞いてください。このようなコードでは、厳密には「データを取り出している」とは言えないんですよ。この場合は、ただ、オブジェクトへの参照を別の変数に「コピー」しているだけです。

著者：それは確かにおっしゃるとおりです。これではデータを「取り出している」とは言えませんね。取り出したはずのDogオブジェクトへの参照は元どおり配列に存在するわけですから。でも、配列の要素にnullを指定するということはできますよね。

ArrayList：それはそうですが。ArrayListクラスなら、ただnullを指定するというのではなく、Dogオブジェクトへの参照を本当に削除することのできるメソッドが用意されているんですよ。サイズをダイナミックに変えるということもできます。配列にそれは無理でしょう。

著者：こんなことは言いたくありませんが、「ArrayListは過大評価されているが、実は効率の悪い配列にすぎない」という噂もありますよね。プログラマが自分で書こうと思えば簡単に書けるサイズ変更のメソッドなどを加えただけの単なる配列のラッパだという声もあります。それに、何よりプリミティブ型の値を格納できないという問題もあります。これは大きいんじゃないですか。

ArrayList：そんな噂に惑わされるとは信じられませんね。私が「効率の悪い配列」だなどということはありませんよ。まれに配列を使った方が若干、あくまで若干ですが、処理速度が高くなる場合があることは私も認めます。しかし、ArrayListの便利さを考えれば、そうしたパフォーマンスの問題は微々たるものです。そんなことを理由にArrayListを使わないのはバカげたことですよ。それからプリミティブ型の値のことですが、どうしても必要なら別のオブジェクトにプリミティブ型の値を格納して、そのオブジェクトをArrayListに入れることもできます（詳細については10章を参照）。Java 5.0の場合、プリミティブ型の値を他のオブジェクトに入れる「ラッピング」は（またプリミティブ型の値をオブジェクトから取り出すアンラッピングは）自動的に行われます。その処理の分、ArrayListでプリミティブ型の値を扱うと、配列より速度の面で不利になることは確かでしょう。でも、今時、プリミティブ型の値をそんな風に使う人がいるんでしょうかね。いけない。もうこんな時間だ。ピラティスに遅れてしまう。続きはまた日を改めてということで……。

This PDF is prepared for tools@maruhouse.org, personal use only.

ArrayListと配列の比較

ArrayList	配列
`ArrayList<String> myList = new ArrayList<String>();`	`String [] myList = new String[2];`
`String a = new String("whoohoo");`	`String a = new String("whoohoo");`
`myList.add(a);`	`myList [0] = a;`
`String b = new String("Frog");`	`String b = new String("Frog");`
`myList.add(b);`	`myList[1] = b;`
`int theSize = myList.size();`	`int theSize = myList.length;`
`Object o = myList.get(1);`	`String o = myList[1];`
`myList.remove(1);`	`myList[1] = null;`

`boolean isIn = myList.contains(b);`	`boolean isIn = false;`
	`for (String item : myList) (`
	`if (b.equals(item)) (`
	`isIn = true;`
	`break;`
	`}`
	`}`

両者の本当の違いはこのあたりから出てくる……

ArrayListはクラスなので、他のクラスと同様の方法でオブジェクトを作成することができ、「.」演算子を使ってメソッドを呼び出すこともできる。

配列を使うには、他では使用しない配列独特の文法（myList [0] = fooなど）を使う必要がある。配列はオブジェクトなのだが、他のオブジェクトと違い、メソッドを呼び出すことはできない。唯一、lengthというインスタンス変数を利用することはできる。

This PDF is prepared for tools@maruhouse.org, personal use only.

ArrayListと通常の配列の比較

① 通常の配列は、宣言の時に型とサイズを決めなくてはならない。

しかし、ArrayListを使う場合は、ただArrayListオブジェクトを作成するだけでよい。格納するオブジェクトが増えたり減ったりすることで大きさが変わるため、サイズを決める必要もない。

```
new String[2]
```

サイズの指定が必要

```
new ArrayList<String>()
```

サイズの指定は不要
（サイズを指定することは可能）

② 通常の配列にデータを格納する場合は、個々に番号（インデックス）を割り当てる必要がある（番号は0から始める。したがって、最後の番号は配列のサイズより1小さくなる）。

```
myList[1] = b;
```

インデックスが必要

配列のサイズを超えるインデックスを使おうとすると（たとえば、配列のサイズが「2」であるにもかかわらず、インデックス「3」を使おうとすると）、実行時にエラーが発生する。ArrayListの場合は、add(anInt, anObject)メソッドを使ってインデックスを指定することもできるが、単にadd(anObject)のように書くこともできる。これだけで、ArrayListは自動的に大きくなり、追加する要素のために必要なスペースは自動的に確保される。

```
myList.add(b);
```

インデックスは不要

③ 配列の構文は、Javaでは配列だけに使用される特殊なもの。

一方のArrayListはJavaの一般のオブジェクトと何ら変わりはないので特殊な構文は使用しない。

```
myList[1]
```

配列の［］は、Javaでは配列にしか使用しない

④ Java 5.0のArrayListにはパラメータが使える。

配列と違い、ArrayListに特殊な構文は使用しないと上に書いたが、Java 5.0（Tiger）では、ArrayListに使える特殊な構文が追加されており、以下のようにパラメータとして型が指定できるようになっている。

```
ArrayList<String>
```

<String>は型を指定するパラメータ。ArrayList<String>は、「Stringのリスト」を意味する。ArrayList<Dog>なら「Dogのリスト」ということになる

Java 5.0より前は、ArrayListの保持対象となるものの型を宣言することはできなかった。従って、コンパイラはArrayListを単なる異種のオブジェクトの寄せ集めとして扱うしかなかった。しかしJava 5.0からは、<typeGoesHere>という構文を使うことで、ArrayList作成時にあらかじめ保持するものの型を指定（制限）しておくことができるようになった。パラメータ付きのArrayListに関しては、16章でコレクションについて触れる際に詳述するので、今の時点では、ArrayListに付けられた<>についてはあまり深く考える必要はない。これによって、ArrayListに特定の型（<>内に指定された型）のオブジェクトのみを入れることができるようになるとだけ覚えておけばよい。

This PDF is prepared for tools@maruhouse.org, personal use only.

ArrayListを使用したバグの修正

前章で紹介したコードには以下のようなバグが含まれていました。

前章のSimpleDotComクラスのコードに改良を加える。名前はDotComクラスに変更

```
public class DotCom {

  int [] locationCells;
  int numOfHits = 0;

  public void setLocationCells(int [] locs) {
    locationCells = locs;
  }

  public String checkYourself(String stringGuess)
{
    int guess = Integer.parseInt(stringGuess);
    String result = "miss";

    for(int cell : locationCells) {
      if (guess == cell) {

        result = "hit";
        numOfHits++;

        break;
      }
    } // ループを抜ける

    if (numOfHits == locationCells.length) {
      result = "kill";
    }
    System.out.println(result);
      return result;
  } // メソッドの終了
} // クラスの終了
```

この部分がバグ。ユーザーの推測が当たるごとにnumOfHitsの値を無条件にインクリメントしている。同じセル番号がすでに当てられたかどうかをチェックしていない。

This PDF is prepared for tools@maruhouse.org, personal use only.

仮コード　テストコード　リアルコード

ArrayListを使って修正したコード

ArrayListを使うとこのように修正できる！

```
import java.util.ArrayList;

public class DotCom {

  private ArrayList<String> locationCells;
  //private int numOfHits;（不要なのでコメントにしている）

  public void setLocationCells(ArrayList<String> loc) {
    locationCells =loc;
  }

  public String checkYourself(String userInput) {
    String result = "miss";
      int index = locationCells.indexOf(userInput);
      if (index >= 0) {
        locationCells.remove(index);

        if (locationCells.isEmpty()) {

          result = "kill";
        } else {
          result = "hit";
        } //if 文の終了
      } //（外側の）if 文の終了

    return result;
  } // メソッドの終了
} // クラスの終了
```

このコードについては本章で後述するので、今は無視してよい

Stringの配列を、Stringを保持するArrayListに変更

引数名を新しいものに変える

インデックスを指定して、ArrayList中にユーザーが推測したセル番号があるかどうかを確かめる。セル番号がリスト中になければ-1が戻される。

変数indexの値が0以上であれば、ユーザーの推測したセル番号がリスト中にあるということなので、そのセル番号をリストから削除

リストが空であれば、軍艦が沈んだということになる

This PDF is prepared for tools@maruhouse.org, personal use only.

「完全版」バトルシップゲームプログラムの作成

ここまではバトルシップゲームプログラムの「簡易版」について書いてきましたが、いよいよ「完全版」の作成に取りかかりましょう。簡易版は、セルが一列に並んでいるところに軍艦が1隻だけというものでしたが、完全版は7×7のグリッドに軍艦が3隻配置されることになります。ゲームの概要は前章でも紹介しましたが、もう一度確認しておきましょう。

このバトルシップゲームでは
7×7のグリッドと
3隻の「軍艦」を使用する。
個々の軍艦はセル3つ分の大きさ。

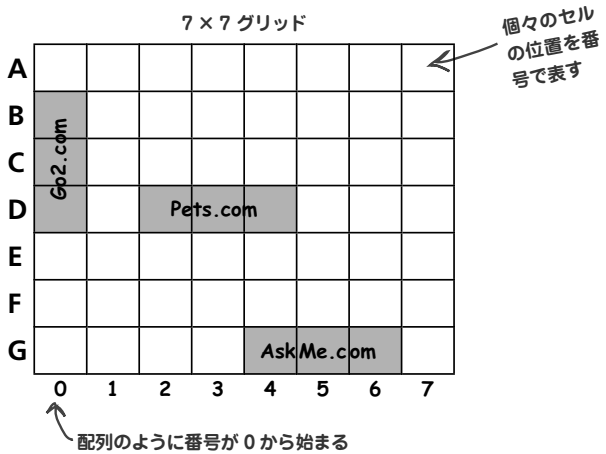
ゲームのルール：プレーヤーがコンピュータの軍艦の位置を推測し、沈没させる。正解に到達するまでの推測の数が少ないほど良い。

ゲームの進行順序：ユーザーがプログラムを起動すると、縦7つ横7つのマス目を持つグリッドに3隻の軍艦（いずれにも「～.com」という名前が付けられる）が自動的に配置される。配置が完了したら、プレーヤーに最初の推測をするよう指示が出される。

プレーの仕方：このゲームはGUIを使わず、コマンドラインでプレーする（本書ではまだGUIを扱う方法を説明していないため）。画面に推測をするよう指示が出されたら、プレーヤーは軍艦が配置されていると思われる場所（グリッドのセル）を1つ選ぶ。どのセルを選んだかは、A3、C5といった番号を入力して知らせる。番号を入力すると、「hit」、「miss」、「You sunk Pets.com」といった文字列が表示される。「hit」は推測が合っていることを、「miss」は推測が外れていることを示す。「You sunk Pets.com（Pets.comの部分は様々に変わる）」は、推測が合っていて、軍艦を1隻沈没させることができたということを意味する。軍艦が3隻とも沈んだら、ゲームは終了し、終了までのプレーヤーの推測の回数が表示される。

プログラムの実行結果の例

```
File  Edit  Window  Help  Sell
%java DotComBust
Enter a guess A3
miss
Enter a guess B2
miss
Enter a guess C4
miss
Enter a guess D2
hit
Enter a guess D3
hit
Enter a guess D4
Ouch!You sunk Pets.com :(
kill
Enter a guess B4
miss
Enter a guess G3
hit
Enter a guess G4
hit
Enter a guess G5
Ouch!You sunk Go2.com :(
```



This PDF is prepared for tools@maruhouse.org, personal use only.

簡易版から完全版への変更ポイント

完全版プログラムは軍艦を表すDotCom（簡易版ではSimpleDotComだったもの）、ゲームそのものを表すDotComBust、GameHelperという3つのクラスから構成されます。ここではまず、DotCom、DotComBustクラスの変更点について見てみましょう。GameHelperクラスについては後述します。

A DotComクラス

◎ 変数nameを追加する。
軍艦（DotComオブジェクト）の名前を格納するためのもの。この変数を使用することで、軍艦が沈没した時にその名前（“Pets.com”、“Go2.com”など）を表示することができる（前ページの実行結果を参照）。

B DotComBustクラス（ゲームそのものに対応するクラス）

◎ 軍艦に対応するDotComオブジェクトを1つではなく3つ作成する。

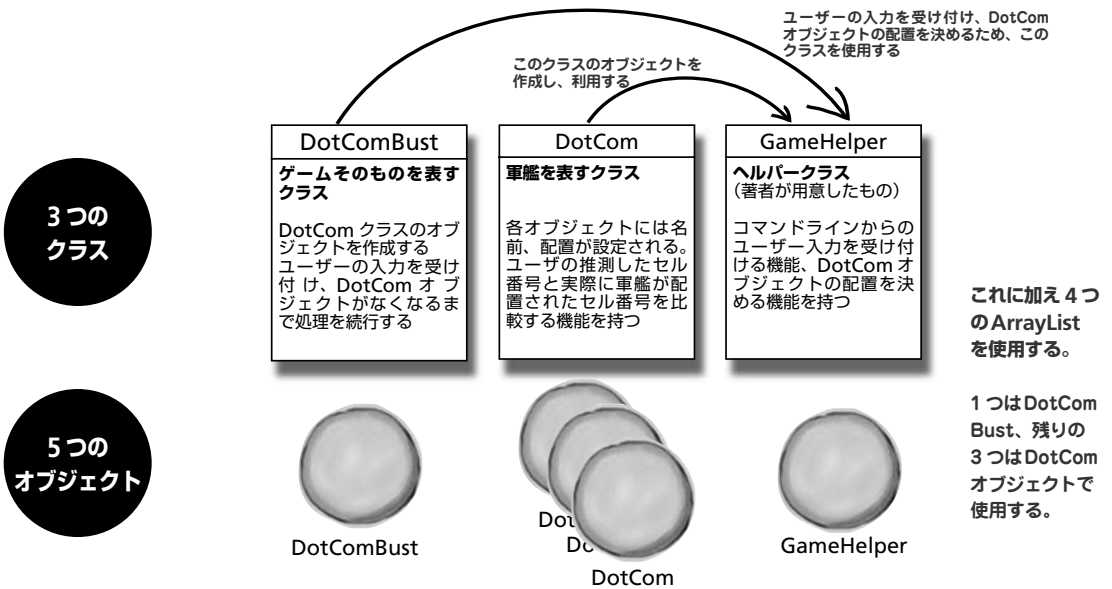
◎ それぞれのDotComオブジェクトに名前を付ける。
DotComオブジェクトのセッターメソッドを呼び出し、インスタンス変数nameに名前を設定する。

◎ 3つのDotComオブジェクトはセルの列ではなくグリッドに配置する。
配置の手順は簡易版より複雑になる。配置は無作為に決定するため、乱数発生の処理が必要になるが、そうした処理について説明するのは本章の目的ではない。そこで、著者があらかじめ用意したGameHelperクラスに配置に関するコードをまとめ、細かい部分を意識せずに使えるようにしてある。

◎ ユーザーの推測が正しいかどうかの確認は3隻の軍艦すべてについて行う。

◎ ゲームは、軍艦がすべて沈没するまで続く（1隻でも残っていればユーザーの推測したセル番号と軍艦が実際に配置されたセル番号の比較が続行される）。

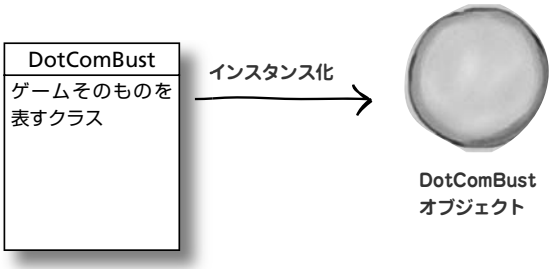
◎ main()メソッドは簡単にする。
簡易版では、説明を簡単にするため、ゲームそのものに対応するクラスですべての処理をmain()メソッドにまとめていた。完全版ではそういうことはしない。



This PDF is prepared for tools@maruhouse.org, personal use only.

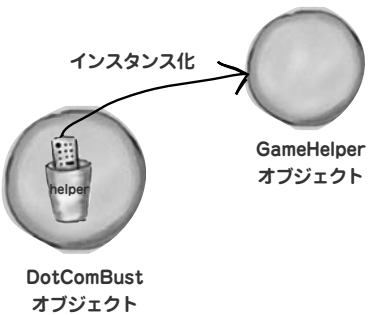
DotComBustクラスの処理

❶



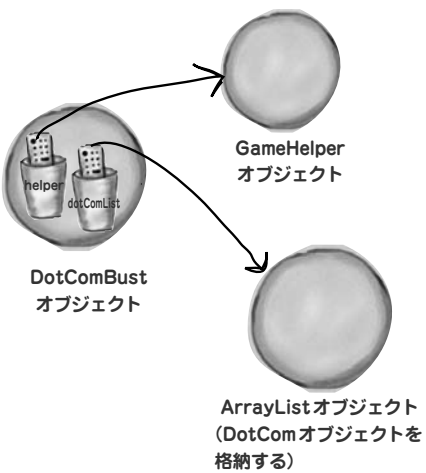
DotComBustクラスのmain()メソッドでインスタンス化によりDotComBustオブジェクトを作成し、そのオブジェクトによってゲームのすべての処理を進める。

❷



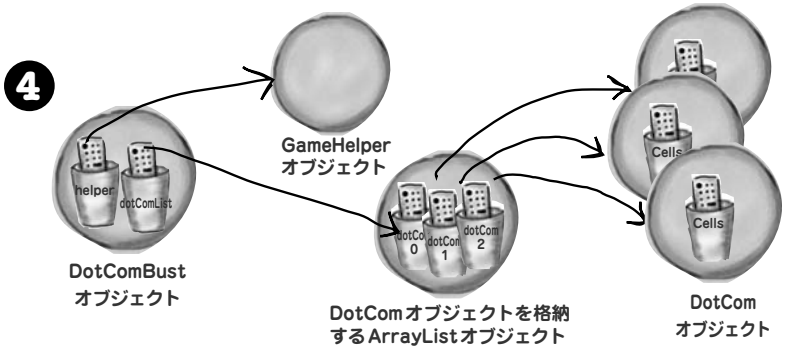
DotComBustオブジェクトは、GameHelperクラスのインスタンスを作成する。GameHelperオブジェクトは、DotComBustオブジェクトの処理を助ける。

❸



DotComBustオブジェクトは、3つのDotComオブジェクトを格納するためArrayListクラスのインスタンスを作成する。

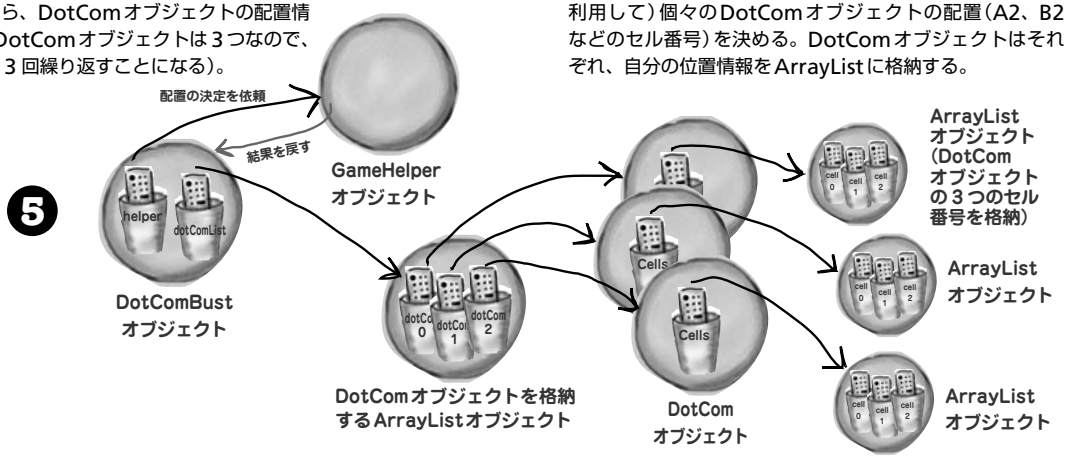
This PDF is prepared for tools@maruhouse.org, personal use only.



DotComBustオブジェクトは、3つのDotComオブジェクトを作成する（そして、それをArrayListオブジェクトに格納する）。

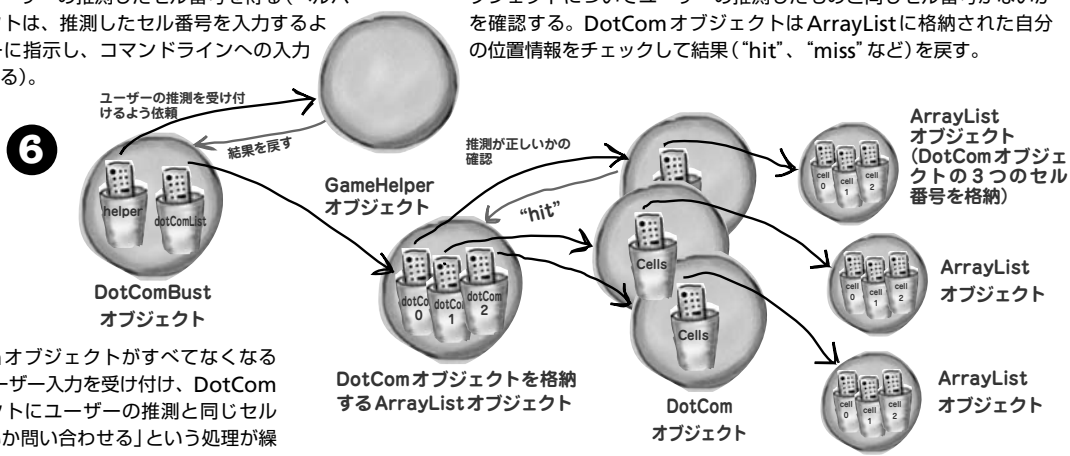
DotComBustオブジェクトは、ヘルパーオブジェクトから、DotComオブジェクトの配置情報を得る（DotComオブジェクトは3つなので、同じ処理を3回繰り返すことになる）。

DotComBustオブジェクトは、（ヘルパーオブジェクトを利用して）個々のDotComオブジェクトの配置（A2、B2などのセル番号）を決める。DotComオブジェクトはそれぞれ、自分の位置情報をArrayListに格納する。



DotComBustオブジェクトは、ヘルパーオブジェクトから、ユーザーの推測したセル番号を得る（ヘルパーオブジェクトは、推測したセル番号を入力するようユーザーに指示し、コマンドラインへの入力を受け付ける）。

DotComBustオブジェクトはループを使用し、個々のDotComオブジェクトについてユーザーの推測したものと同じセル番号がないかを確認する。DotComオブジェクトはArrayListに格納された自分の位置情報をチェックして結果（“hit”、“miss”など）を戻す。



DomComオブジェクトがすべてなくなるまで、「ユーザー入力を受け付け、DotComオブジェクトにユーザーの推測と同じセル番号がないか問い合わせる」という処理が繰り返され、ゲームが続行される。

This PDF is prepared for tools@maruhouse.org, personal use only.

DotComBust
GameHelper helper ArrayList dotComList int numOfGuesses
setUpGame() startPlaying() checkUserGuess() finishGame()

DotComBustクラスの仮コード

DotComBustクラスには、ゲーム開始の準備、ゲームの進行、ゲームの終了という3つの役割があります。この役割を単純に3つのメソッドに振り分けるという方法もあるのですが、ここではより細かく、「ゲームの進行」の処理を2つのメソッドに分けます。メソッドを小さくすれば（受け持つ機能を少なくすれば）、テストやデバッグ、コードの修正が楽になるためです。

インスタンス変数のリスト

helper　GameHelperのインスタンスを作成し、参照を格納する

dotComList　DotComオブジェクト（はじめは3つ）を格納するArrayListのインスタンスを作成し、参照を格納する

numOfGuess　ユーザーの推測の数を記録する（ゲーム終了時に表示するため）int型変数。初期値は0にする

メソッドのリスト

setUpGame()　DotComオブジェクトを作成し、名前と配置を決定するメソッド。ゲームのプレー方法についての簡単な説明を表示する機能も備える

startPlaying()　ユーザーに推測したセル番号の入力を求め、checkUserGuess()メソッドを呼び出す、という処理をDotComオブジェクトがすべて削除されるまで繰り返すメソッド

checkUserGuess()　その時点で残っているDotComオブジェクトすべてについて、（そのDotComオブジェクトの）checkYourself()メソッドを呼び出す、という処理を行うメソッド

finishGame()　「すべてのDotComオブジェクトを削除するまでに何回セル番号を入力したか」を表示し、ユーザーのプレーに対するコメントも表示するメソッド

メソッドのロジック

メソッド：void setUpGame()

//DotComオブジェクトを作成し名前を付ける

オブジェクトの作成　DotComオブジェクトを3つ作る

名前の指定　個々のDotComオブジェクトの名前を指定

ArrayListへの追加　dotComList（ArrayList）にDotComオブジェクトを追加

繰り返し処理　dotComList中の個々のDotComオブジェクトについて同じ処理を繰り返す

メソッド呼び出し　ヘルパーオブジェクトのplaceDotCom()メソッドを呼び出し、DotComオブジェクトの配置（7×7のグリッドから無作為に選択。各DotComオブジェクトの配置には、縦、あるいは横に並ぶ3つのセルが必要）を決めるため

配置の指定　placeDotCom()の実行結果を基に各DotComオブジェクトの配置を指定

繰り返し処理の終了

メソッドの終了

This PDF is prepared for tools@maruhouse.org, personal use only.

仮コード　テストコード　リアルコード

メソッドのロジック（続き）

メソッド：void startPlaying()
　繰り返し処理　DotComオブジェクトが1つでも存在する間、同じ処理を繰り返す
　　データの取得　ヘルパーオブジェクトのgetUserInput()を呼び出してユーザー入力データの取得
　　確認　checkUserGuess()メソッドを呼び出してユーザーの推測が正しいかの確認
　繰り返し処理の終了
メソッドの終了

メソッド：void checkUserGuess(String userGuess)
　//ユーザーの推測が当たっているか（すべてのセル番号を当てられたDotComがあるか）の確認
　インクリメント　変数numOfGuessesのユーザー推測数を表す数値をインクリメント
　値の設定　ローカル変数result（String型）の値を“miss”にする。“miss”はユーザーの推測が間違っていたことを示す値
　繰り返し処理　dotComsList（ArrayList）中の個々のDotComオブジェクトについて同じ処理を繰り返す
　　確認　DotComオブジェクトのcheckYourself()メソッドを呼び出してユーザーの推測が正しいかの確認
　　値の設定　必要に応じresult変数の値を“hit”、“kill”にする
　　if文　resultの値が“kill”であるかの確認
　　削除　dotComListからのDotComの削除
　繰り返し処理の終了
　表示　resultの値を表示

メソッド：void finishGame()
　表示　ゲーム終了のメッセージを表示
　　if文　ユーザーの推測数が一定数以下かどうかの確認
　　　表示　プレーの評価が高いことを示すメッセージを表示
　　else文
　　　表示　プレーの評価が低いことを示すメッセージを表示
　　if文の終了
メソッドの終了

自分で考えてみよう

仮コードからリアルコードを作る前にはテストコードを作るべきであるということはすでに述べました。テストコードを作って、それを利用してリアルコードを少しずつ作っていくのです。ただ、この章のプログラムに関してテストコードを逐一紹介することはしません。是非、テストコードをどのようなものにすればよいか自分の頭で考えてみてください。まず、どのメソッドについてのテストコードを作ればいいでしょうか。テストコードのための仮コードを作ってもいいし、テストコードの概要を箇条書きにまとめるだけでも練習にはなると思います。ただ、やはりできれば頑張って完全なテストコードを作るべきでしょう。

This PDF is prepared for tools@maruhouse.org, personal use only.

仮コード　テストコード　リアルコード

DotComBustクラスのリアルコード

```
import java.util.*;
public class DotComBust {

  private GameHelper helper = new GameHelper();
  private ArrayList<DotCom> dotComsList = new ArrayList<DotCom>();
  private int numOfGuesses = 0;

  private void setUpGame() {
    //DotComオブジェクトを作成し位置を指定
    DotCom one = new DotCom();
    one.setName("Pets.com");
    DotCom two = new DotCom();
    two.setName("eToys.com");
    DotCom three = new DotCom();
    three.setName("Go2.com");
    dotComsList.add(one);
    dotComsList.add(two);
    dotComsList.add(three);

    System.out.println("Your goal is to sink three dot coms.");
    System.out.println("Pets.com, eToys.com, Go2.com");
    System.out.println("Try to sink them all in the fewest number of guesses");

    for (DotCom dotComToSet : dotComsList) {
    ArrayList<String> newLocation = helper.placeDotCom(3);
    dotComToSet.setLocationCells(newLocation);
  } //ループ終了
} //setUpgameメソッド終了

private void startPlaying() {
  while(!dotComsList.isEmpty()) {
    String userGuess = helper.getUserInput("Enter a guess");
    checkUserGuess(userGuess);
  } //whileループ終了
    finishGame();
} //startPlayingメソッド終了
```

1 必要な変数を宣言、初期化する

2 3つのDotComオブジェクトを作成し、名前を指定し、ArrayListに追加する

3 プレー方法に関する簡単な説明を表示する

4 DotComのリストが空でない間同じ処理を繰り返す

5 DotComオブジェクトを配置すべきセルをヘルパーオブジェクトに問い合わせる

6 DotComオブジェクトのセッターメソッドを呼び出し、ヘルパーオブジェクトからの情報を基にDotComオブジェクトの配置を指定する

7 リスト中の個々のDotComオブジェクトについて同じ処理を繰り返す

8 ユーザー入力を取得する

9 checkUserGuessメソッドを呼び出す

10 finishGameメソッドを呼び出す

This PDF is prepared for tools@maruhouse.org, personal use only.

仮コード　テストコード　リアルコード

```
private void checkUserGuess(String userGuess) {

    numOfGuesses++;    (11)
    String result = "miss"; (12)

    for (DotCom dotComToTest : dotComsList) {    (13)
      result = dotComToTest.checkYourself(userGuess); (14)
      if (result.equals("hit")) {
        break;
      }                (15)
      if (result.equals("kill")) {
        dotComsList.remove(dotComToTest); (16)
        break;
      }
    } //forループの終了
    System.out.println(result); (17)
  } //メソッドの終了

  private void finishGame() {
    System.out.println("All Dot Coms are dead! Your stock is now worthless");
    if (numOfGuesses <= 18) {
      System.out.println("It only took you " + numOfGuesses + " guesses.");
      System.out.println("You got out before your options sank");
    } else {
      System.out.println("Took you long enough. "+numOfGuesses + " guesses.");
      System.out.println("Fish are dancing with your options.");
    }                                                          (18)
  } //メソッドの終了
  public static void main (String [] args) {
    DotComBust game = new DotComBust(); (19)
    game.setUpGame(); (20)
    game.startPlaying(); (21)
  } //メソッドの終了
}
```

自分で考えてみよう

このページと次のページのコードには番号が振ってありますが、各番号の該当箇所に対応する説明文を下から探してみましょう。説明文の前に適切な番号を付けていってください。

どの説明文も使えるのは一回だけです。また、説明文はすべて使用します。

答えは次のページにあります。

全問できるまで決して見ないでください。

- リスト中のすべてのDotComオブジェクトについて同じ処理を繰り返す
- プレーを評価するメッセージを表示する
- resultの値を表示する
- リストからDotComオブジェクトを取得する
- 対応する「軍艦」が沈没したので、DotComオブジェクトをリストから削除し、ループを抜ける
- 特に指定のない限り、値が“miss”になるようにする
- ゲームの開始準備を行うためのメソッドを呼び出す
- ユーザーの推測数をインクリメントする
- ゲームのプレーのためのループ処理（ユーザーに入力を求め、それをチェックする処理）を行うためのメソッドを呼び出す
- DotComオブジェクトのメソッドを呼び出し、ユーザーの推測が正しいかを確認する。結果に応じてresultの値が“hit”、“kill”になる
- 条件判定を経ることなくループを抜ける
- DotComBustオブジェクトを作成する

This PDF is prepared for tools@maruhouse.org, personal use only.

DotComBustクラスのコード

仮コード　テストコード　**リアルコード**

```
import java.util.*;
public class DotComBust {

  private GameHelper helper = new GameHelper();
  private ArrayList<DotCom> dotComsList = new ArrayList<DotCom>();
  private int numOfGuesses = 0;

  private void setUpGame() {
    //DotComオブジェクトを作成し位置を指定
    DotCom one = new DotCom();
    one.setName("Pets.com");
    DotCom two = new DotCom();
    two.setName("eToys.com");
    DotCom three = new DotCom();
    three.setName("Go2.com");
    dotComsList.add(one);
    dotComsList.add(two);
    dotComsList.add(three);

    System.out.println("Your goal is to sink three dot coms.");
    System.out.println("Pets.com,eToys.com,Go2.com");
    System.out.println("Try to sink them all in the fewest number of guesses");

      for (DotCom dotComToSet : dotComsList){

        ArrayList<String> newLocation = helper.placeDotCom(3);
        dotComToSet.setLocationCells(newLocation);

    } //ループ終了
  } //setUpgameメソッド終了

  private void startPlaying() {

    while(!dotComsList.isEmpty()) {

      String userGuess = helper.getUserInput("Enter a guess");
      checkUserGuess(userGuess);
    } //whileループ終了
      finishGame();
  } //startPlayingメソッド終了
```

必要な変数を宣言、初期化する

DotComオブジェクトを保持するArrayListを作成（DotCom [] というコードでDotComオブジェクトの配列を作るのに似ている）

3つのDotComオブジェクトを作成し、名前を指定し、ArrayListに追加する

プレー方法に関する簡単な説明を表示する

リスト中の個々のDotComオブジェクトについて同じ処理を繰り返す

DotComオブジェクトを配置すべきセルをヘルパーオブジェクトに問い合わせる

DotComオブジェクトのセッターメソッドを呼び出し、ヘルパーオブジェクトからの情報を基にDotComオブジェクトの配置を指定する

DotComのリストが空でない間同じ処理を繰り返す

ユーザー入力を取得する

checkUserGuessメソッドを呼び出す

finishGameメソッドを呼び出す

This PDF is prepared for tools@maruhouse.org, personal use only.

仮コード　テストコード　**リアルコード**

```
private void checkUserGuess(String userGuess) {
    numOfGuesses++;
    String result = "miss";
    for (DotCom dotComToTest : dotComsList) {
      result = dotComToTest.checkYourself(userGuess);
      if (result.equals("hit")) {

        break;
      }
      if (result.equals("kill")) {

        dotComsList.remove(dotComToTest);
        break;
      }
    } //for ループの終了
    System.out.println(result);
  } //メソッドの終了

  private void finishGame(){
    System.out.println("All Dot Coms are dead! Your stock is now worthless");
    if (numOfGuesses <= 18) {
      System.out.println("It only took you " +numOfGuesses +"guesses.");
      System.out.println(" You got out before your options sank");
    } else {
      System.out.println("Took you long enough. "+ numOfGuesses + " guesses.");
      System.out.println("Fish are dancing with your options");
    }
  } //メソッドの終了

  public static void main (String [] args) {
    DotComBust game = new DotComBust();
    game.setUpGame();
    game.startPlaying();
  } //メソッドの終了
}
```

ユーザーの推測数をインクリメントする

特に指定のない限り、値が“miss”になるようにする

リスト中のすべてのDotComオブジェクトについて同じ処理を繰り返す

DotComオブジェクトのメソッドを呼び出し、ユーザーの推測が正しいかを確認する。結果に応じてresultの値が“hit”、“kill”になる

条件判定を経ることなくループを抜ける

対応する「軍艦」が沈没したので、DotComオブジェクトをリストから削除し、ループを抜ける

resultの値を表示する

プレーを評価するメッセージを表示する

DotComBustオブジェクトを作成する

ゲームの開始準備を行うためのメソッドを呼び出す

ゲームのプレーのためのループ処理（ユーザーに入力を求め、それをチェックする処理）を行うためのメソッドを呼び出す

This PDF is prepared for tools@maruhouse.org, personal use only.

DotComクラスのリアルコード

```
import java.util.*;

public class DotCom {
  private ArrayList<String> locationCells;
  private String name;

  public void setLocationCells (ArrayList<String> loc) {
    locationCells = loc;
  }

  public void setName (String n) {
    name =n;
  }

  public String checkYourself (String userInput) {
    String result = "miss";
    int index = locationCells.indexOf(userInput);
    if (index >= 0) {
      locationCells.remove(index);
      if (locationCells.isEmpty()) {
        result = "kill";
        System.out.println("Ouch! You sunk " + name + " : ( ");
      } else {
        result = "hit";
      } //if文の終了
    } //if文の終了
    return result;
  } //メソッドの終了
} //クラスの終了
```

DotComクラスのインスタンス変数：
- オブジェクトの配置を記録するArrayList
- DotComオブジェクトの名前

DotComオブジェクトの配置を指定するセッターメソッド（GameHelperクラスのplaceDotCom()メソッドが無作為に選ぶセル番号を使用する）

DotComオブジェクトの名前を指定するセッターメソッド

ArrayListクラスのindexOf()メソッドを使用している。ユーザーの推測したセル番号がリスト中にあれば、ArrayListでのそのセル番号のインデックスが戻される。なければ、-1が戻される

ArrayListのremove()メソッドを使用してリストのエントリを削除する

isEmpty()メソッドを使用して、すべてのセル番号が当てられたかどうかを確認する

「軍艦」が沈没したというメッセージを表示する

"miss"、"hit"、"kill"のいずれかを戻す

This PDF is prepared for tools@maruhouse.org, personal use only.

高度な条件判定式

これまでにも、ループやif文などで条件判定式は使ってきましたが、いずれもごく簡単なものでした。次に著者がバトルシップゲームプログラムの完全版のために用意したクラスを紹介しますが、そこではより高度な条件判定式を使っています。読者の中には、「まだ少し早いのでは」と感じる人もいるでしょうが、よい機会ですので、そうした条件判定式について簡単に説明をしておきましょう。

AND演算子（&&）、OR演算子（||）

たとえば、いくつかのルールに基づいてカメラを選択するchooseCamera()というメソッドを書くとしましょう。仮に、選択の基準となるルールが「価格」であれば、次のような条件判定式が使えます。

以下の条件判定式は、「価格が300ドル以上400ドル未満であれば、カメラXを選択する」というものです。

```
if (price >= 300 && price < 400) {
  camera = "X";
}
```

また、ブランドA、あるいはBのカメラだけについて特定の処理を行いたい、という場合には以下のようなコードを書きます。

```
if (brand.equals("A") || brand.equals("B")) {
  // ブランドA、Bのカメラにのみ当てはまるコードを書く
}
```

条件判定式は、次のように長く複雑になることもあります。

```
if ((zoomType.equals("optical") &&
     (zoomDegree >= 3 && zoomDegree <= 8)) ||
     (zoomType.equals("digital") &&
     (zoomDegree >= 5 && zoomDegree <= 12))) {
  // 条件が満たされた場合に行う処理
}
```

このようにいくつもの演算子を使う場合、問題となるのは「優先順位」です。優先順位については非常に複雑なルールがあるのですが、それを無理して覚えるよりは括弧を使って優先順位を明確にするテクニックを使った方がいいでしょう。

!=演算子、!演算子

たとえば、いくつかあるカメラのうち、「モデル2000」というものを除き、すべてについて同じ処理をしたい、という場合、次のようなコードを書くことになります。

```
if (model != 2000) {
  // モデル2000以外に当てはまる処理
}
```

また、ブランドX以外のすべてのカメラについて一定の処理を行いたい、という場合、次のようなコードが書けます。

```
if (!brand.equals("X")) {
  // ブランドX以外に当てはまる処理
}
```

ショートサーキット演算子（&&、||）

すでに紹介した、&&、||といった演算子は「ショートサーキット演算子」と呼ばれています。「&&」は両辺の式が同時にtrueである場合のみtrueとなる、という演算子です。ただし、最初に評価されるのは左辺で、左辺の式がfalseであれば右辺の式の評価は行われないので注意が必要です。
「||」は、両辺の式のうちいずれかがtrueである場合にtrueとなる、という演算子です。先に評価されるのはやはり左辺の式で、左辺の式がtrueであれば、右辺の式を評価することなくtrueとなります。
ショートサーキット演算子のこの特性は、たとえば、ある変数にオブジェクトへの参照が格納されるかどうかが明確にはわからない、といった場合に利用できます。変数の値がnull（つまり対応するオブジェクトがない）であるにもかかわらず、メソッドを呼び出そうとするとNullPointerExceptionという例が発生します。次のようなコードを書けば、そういった問題は起きません。

```
if (refVar != null &&
    refVar.isValidType() ) {
//isValidType() がtrueを戻した場合の処理
}
```

&演算子、|演算子

&、|といった演算子も、条件判定式においてショートサーキット演算子とほぼ同様の使い方ができます。ただし、必ず両辺の式が評価されるという点が違っています。また、&、|はともに、通常は「ビット演算」という別の用途に使われます。

This PDF is prepared for tools@maruhouse.org, personal use only.

GameHelperクラス

レディメードコード

それでは、著者の用意したGameHelperクラスのコードを紹介しましょう。ユーザーにコマンドラインからの入力を求めるためのメソッドもありますが、このクラスの主な役割は、DotComオブジェクトを配置するセル番号を作ることです。少し大変かもしれませんが、是非このコードを自分でタイプし、コンパイルしてください。タイプの負担が軽くなるようコードはできる限り少なくしましたが、それがコードの読みやすさにつながっているとは言えません。また、重要なことは、このクラスがないと、DotComBustクラスもコンパイルできないということです。

注意：余裕のある人は、placeDotCom()中のSystem.out.printlnに付いているコメントを外して動きがどうなるか試してみてください。このprint文が実行されると軍艦の場所がわかってしまうのでゲームにはならなくなりますが、コードのテストには役立ちます。

```
import java.io.*;
import java.util.*;

public class GameHelper {

  private static final String alphabet = "abcdefg";
  private int gridLength = 7;
  private int gridSize = 49;
  private int [] grid = new int [gridSize];
  private int comCount = 0;

  public String getUserInput (String prompt) {
    String inputLine = null;
    System.out.print (prompt + " ");
    try {
      BufferedReader is = new BufferedReader (
      new InputStreamReader (System.in));
      inputLine = is.readLine();
      if (inputLine.length() == 0) return null;
    } catch (IOException e) {
      System.out.println ("IOException: " + e);
    }
    return inputLine.toLowerCase ();
  }

  public ArrayList<String> placeDotCom (int comSize) {
    ArrayList<String> alphaCells = new ArrayList()<String>;
    String [] alphacoords = new String [comSize];          //'f6' などのコードを保持する
    String temp = null;                                    //後で連結する文字列を一時的に保持
    int [] coords = new int [comSize];                     //候補となるセル番号を保持する
    int attempts = 0;                                      //試行回数のカウンタ
    boolean success = false;                               //配置が適切かを示すフラグ
    int location = 0;                                      //その時点でどのセルが検討対象になっているかを示す

    comCount++;                                            //配置対象の DotCom オブジェクトを変える
    int incr = 1;                                          //オブジェクトの配置を横向きにする
    if ((comCount % 2) == 1) {                             //奇数番目の DotCom オブジェクトの場合は
      incr = gridLength;                                   //オブジェクトの配置を縦向きにする
    }

    while (!success & attempts++ < 200 ) {                 //メインのループ
      location = (int) (Math.random() * gridSize);         //乱数を得る
      //System.out.print (" try " + location);
      int x = 0;                                           //いくつ目のセルか、を示す値
        success = true;                                    //true は番号の指定が成功したことを示す
        while (success && x < comSize) {                   //隣から使われていない場所を探す
          if (grid [location] == 0) {                      //まだ使われていないかどうかを判定
```


This PDF is prepared for tools@maruhouse.org , personal use only.

```
        coords [x++] = location;                          // 得られた配置情報を保存
        location += incr;                                 // 処理対象を隣のセルに移す
        if (location >= gridSize) {                       // グリッドから下にはみ出ていないか確認
          success = false;                                // 失敗
        }
        if (x >0 && (location % gridLength == 0)) {       // グリッドから右にはみ出ていないか確認
          success = false;                                // 失敗
        }
      } else {                                            // すでに使われているセルだった場合
        //System.out.print ( " used " + location);
        success = false;                                  // 失敗
      }
    }
  }                                                       //while ループの終了

  int x = 0;                                              // 最終的な戻り値を得る
  int row = 0;
  int column = 0;
    //System.out.println ( "\n" );
    while (x < comSize) {
    grid [coords [x]] = 1;                                // セルを「使用済み」にする
    row = (int) (coords [x] /gridLength);                 // 横方向の番号を得る
    column = coords [x] % gridLength;                     // 縦方向の番号を得る
    temp = String.valueOf (alphabet.charAt (column));     // 番号をアルファベットに変換

    alphaCells.add (temp.concat (Integer.toString(row)));
    x++;
  // System.out.print( " coord "+x+" = " + alphaCells.get(x-1));
  }
  //System.out.println ( "\n" );

  return alphaCells;
 }
}
```

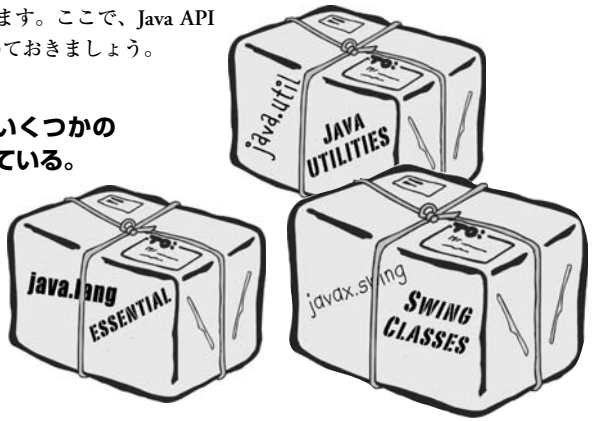
このコードが実行されると、軍艦の正確な位置がわかる

This PDF is prepared for tools@maruhouse.org, personal use only.

ライブラリ（Java API）の使用

DotComBustクラスは、すでに見てきたとおり、Java APIに用意されたArrayListというクラスを利用することでできています。ここで、Java API（Javaライブラリ）の基本的な使い方についてまとめておきましょう。

Java APIでは、クラスがいくつかの「パッケージ」にまとめられている。



APIのクラスを使用するには、そのクラスがどのパッケージに属するかを知る必要がある。

Java APIのクラスはすべて、いずれかの「パッケージ」に属します。パッケージにはそれぞれ、「javax.swing（これは、後で触れるSwing GUIに関連するクラスの属するパッケージの名前です）」というような名前が付けられています。ArrayListが属するのは、「ユーティリティクラス」と呼ばれるプログラマを支援するためのクラスが数多く納められた「java.util」というパッケージです。16章では、このパッケージのクラスを多数取りあげます（その他、17章では、プログラマが自ら作ったクラスを独自のパッケージにまとめる方法なども紹介します）。しかし、ここではまず、ともかくJava APIのクラスをどのように使うのかだけを覚えてください。

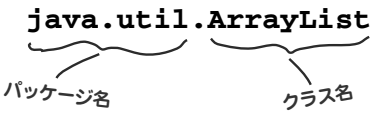
APIのクラスは非常に簡単に使うことができます。自分で書き、コンパイルしたクラスとほぼ同じように使用できるのです。いつでも使われるのを待っていると言ってもいいでしょう。ただ、注意すべきなのは、使いたいクラスの名前だけでなく、そのクラスの属するパッケージもプログラム中に明記しなくてはならないということです。

意識はしていなかったと思いますが、本書で紹介したプログラムでもすでに、あるパッケージに属するクラスを使用していました。System（System.out.println()などが属するクラス）、String、Math（Match.random()などが属するクラス）といったクラスがそれにあたります。いずれのクラスも「java.lang」というパッケージに属します。

This PDF is prepared for tools@maruhouse.org, personal use only.

Java APIのクラスをプログラムの中で使うには、そのクラスのフルネームを指定する必要がある*。

たとえば、「ArrayList」はArrayListクラスのフルネームではありません。「ArrayList」という名前だけを使うのは、ファミリーネームを使わず「Kathy」とだけ呼ぶようなものです。ArrayListクラスのフルネームは以下のようになります。



フルネームを指定するには、2種類の方法がある。

A インポートする

以下に示すように、「import」というステートメントをソースコードの先頭に書くという方法です。

```
import java.util.ArrayList;
public class MyClass {...}
```

または

B タイプする

以下に示すように、クラスを使用する度にフルネームを書くという方法です。

変数の宣言、インスタンス化の場合

```
java.util.ArrayList<Dog> list = new java.util.ArrayList<Dog>();
```

引数の型として使用する場合

```
public void go (java.util.ArrayList<Dog> list) { }
```

戻り値の型として使用する場合

```
public java.util.ArrayList<Dog> foo () {...}
```

* java.lang パッケージのクラスは例外で、フルネームを指定する必要がない。

素朴な疑問に答えます

Q: なぜパッケージというものがあるのですか。

A: パッケージの存在理由は大きく分けて3つです。まず、ライブラリ中のクラスを整理しやすくする、ということがあげられます。膨大な数のクラスがあるとそのままでは必要なものを見つけるが大変ですが、機能ごと（GUIに関係するもの、データベースに関係するもの、といった具合）に細かく分かれていれば、比較的簡単に見つけることができます。

また、名前の混乱を避ける、ということもあげられます。何人もプログラマがいると、違う目的で作られたにもかかわらず偶然に同じ名前を持つクラスが数多くできてしまう可能性があります。そのままではそれぞれを区別することはできませんが、属するパッケージが個々に異なれば、違うクラスであると認識することができます。たとえ、Java APIに属するクラスと同じ名前のクラスを知らずに作ってしまったとしても、パッケージが異なれば、区別ができます。

もう1つはセキュリティ上の理由です。クラスがパッケージに分かれていると、「同じパッケージに属するクラスでのみ利用できるクラス」などを作ることができるのです。この点は17章で詳しく説明します。

Q: パッケージは名前の混乱を避けるのに役立つということですが、本当でしょうか。2人の人間が同じパッケージに同じ名前のクラスを作るということもあるのではないですか。

A: Javaの名前付け規則では、それはできないことになっています。プログラマがこの規則を守る限り、問題は起きません。詳しくは17章を参照してください。

This PDF is prepared for tools@maruhouse.org, personal use only.

javaxの「x」はどういう意味？（javaxで始まるパッケージ名にはどういう意味があるのか）

Javaの最初のバージョン、2つ目のバージョン（1.02と1.1）では、ライブラリ（標準ライブラリ）として提供されるクラスはすべて、「java」という名前で始まるパッケージに属していました。インポートの必要がないjava.langパッケージをはじめ、java.net、java.io、java.utilパッケージ（当初はArrayListというクラスはなかった）、そしてGUI関連のクラスを納めたjava.awtパッケージなど、すべてのパッケージ名が「java」から始まっていたのです。

しかし、その後、標準ライブラリに属さないパッケージが徐々に提供されるようになりました。それらは「拡張パッケージ」と呼ばれ、大きく「標準拡張パッケージ」とそれ以外のものに分かれます。「標準拡張パッケージ」はSunが公式のものとして認めている拡張パッケージです。公式のものとなる以前のものとして、試験版、アーリーアクセス版、ベータ版などと呼ばれる拡張パッケージもありますが、結局、公式なものにならないパッケージも多くあります。

「javax」というのは、標準拡張パッケージに使われる名前です。慣例として、標準拡張パッケージの名前は「javax」から始めます。最も初期から存在する標準拡張パッケージであるjavax.swingの名前が「javax」から始まっていたために、他のパッケージもそれにならうようになったのです。

標準拡張パッケージの中には、標準ライブラリに属するパッケージとまったく変わらないほど一般的になってしまうものもあります。そのようなパッケージはJavaの開発環境には必ずはじめから用意されることになります。javax.swingもそうしたパッケージの例で、Javaのバージョン1.2（Java 2）からは標準ライブラリのパッケージと変わらなくなっています。Swingのようにパッケージが標準ライブラリと変わらなくなれば、もう使用する際に「パッケージをどのように入手するか」といったことを考える必要はありません。

問題なのは、それほど一般的になれば、Sunとしてはパッケージ名をjavaxではなく、javaから始まるものに変えたくなるということです。確かに、標準ライブラリのパッケージの名前はすべて「java」で始まっており、パッケージ名に「x」が付くとどうしても「拡張」というイメージがつきまといます。SunはJava 2の完成直前に、実際に一度パッケージ名から「x」を取っています。その当時に作られた本も、それに沿った内容になっており、その中ではSwingパッケージは新しい名前で使用されています。クラス名の重複が起きるわけでもなかったので、何も困ることはないはずでした。ところが、すでに2万を超えていたデベロッパにとっては、その小さな変更が重大な問題だったのです。Swingを利用したコードをすべて改造しなければならなかったからです。それは極めて大変な作業です。「javax」という名前を使用しているimportステートメントがいったいどのくらいあるのか、考えてみるとわかります。

結局、最後の最後になって、Sunはデベロッパたちの必死の説得に応じ、従来どおり「javax」で始まるパッケージ名を残すことにしました。今でも「javax」で始まる名前は使われています。開発環境にはじめから用意されているパッケージの中に名前が「javax」で始まるものを見つけたら、「はじめは拡張パッケージだったが、その後、標準ライブラリと変わらないくらい一般的になったもの」と考えて間違いありません。

重要ポイント

- ArrayListはJava APIのクラス。
- ArrayListへのオブジェクトの追加にはadd()メソッドを使用する。
- ArrayListからオブジェクトを削除するにはremove()メソッドを使用する。
- あるオブジェクトがArrayList中に存在するかを確認する。存在する場合はそのインデックスを確認するためには、indexOf()メソッドを使用する。
- ArrayListが空かどうかを確認するには、isEmpty()メソッドを使用する。
- ArrayListのサイズ（要素数）を確認するには、size()メソッドを使用する。
- 配列のサイズ（要素数）を確認するには、length変数を使用する。
- ArrayListはダイナミックにサイズを変えることができる。オブジェクトが追加されると大きくなり、削除されると小さくなる。
- ArrayListに保持するものの型は、<>を使ったパラメータで指定できる。たとえば、ArrayList<Button>なら、ArrayListにはButton型のオブジェクト（あるいは次章以降で解説するとおりButtonのサブクラスのオブジェクト）のみ保持できることになる。
- ArrayListが保持するのは基本的にオブジェクトであり、プリミティブ型の値ではない。ただし、コンパイラには、プリミティブ型の値を自動的にラッピング（値を取り出す時は自動的にアンラッピング）する機能がある。つまり、Object型のオブジェクトに自動的にプリミティブ型の値を入れることができる。プリミティブ型の値は、Object型のオブジェクトに入った状態でArrayListに保持される（詳細については後述）。
- クラスはいずれかのパッケージに属する。
- クラスには、パッケージ名とクラス名を組み合わせた「フルネーム」がある。ArrayListクラスのフルネームはjava.util.ArrayList。
- java.lang以外のパッケージに属するクラスを使用するには、プログラム中にフルネームを指定する必要がある。
- フルネームを指定するには、プログラムの冒頭にimportステートメントを入れる、クラスを使用する度にフルネームを書く、という2つの方法がある。

This PDF is prepared for tools@maruhouse.org, personal use only.

素朴な疑問に答えます

Q: 他のクラスをインポートすると、クラスの規模が大きくなるのではないですか。コンパイル時には、インポートしたクラス（あるいはパッケージ）も同時にコンパイルすることになるのですか。

A: 質問者の方はおそらくC言語のプログラマではないかと思われます。Javaの「インポート」は、Cの「インクルード」とは違います。したがって、この質問のようなことはありません。インポートはあくまでクラスのフルネームを逐一タイプする手間を省くために行うものです。インポートのし過ぎによってプログラムが大きくなる、パフォーマンスが低下するといったことを心配する必要はありません。

Q: StringクラスやSystemクラスをインポートなしで使えるのはなぜですか。

A: いずれもjava.langパッケージに属するクラスですが、java.langパッケージは何も指定しなくてもあらかじめインポートされていると考えてください。java.langパッケージのクラスは基本的なものばかりで使用頻度も高いためです。インポートをしたり、フルネームを書いたりといったことは必要ありません。java.lang.Stringクラス、java.lang.Systemクラスなどは開発環境に必ず1つしか存在しないようになっているので、「フルネームを指定していないから見つからない」ということはありません。

Q: 自分で作ったクラスをパッケージにまとめることはできるのですか。できるとしたら、その方法は？

A: できます。また、現実にJavaを使う際にはそういうことはよく行われます。詳細については17章で説明します。今のところはあまり考えなくていいでしょう。

覚えておこう

importステートメントを使わない場合は、クラスのフルネームを必ず書く

java.lang以外のパッケージに属するクラスを使用する場合は、必ずフルネームを指定しなくてはなりません。importステートメントをプログラムの冒頭に入れれば、それが簡単にできます。さもなければ、クラスを使う度にフルネームを書くことになります。

もうわかったと思いますが、ここで要点を図にしておきましょう。

This PDF is prepared for tools@maruhouse.org, personal use only.

「java.utilパッケージにArrayListというクラスがあることがわかったのはよかったと思います。でも、自分の力でこのようなクラスを見つけるにはどうしたらいいのですか？」

— Julia（31才）、職業ハンドモデル

APIの使い方

ここでは次の2点について説明します。

❶ **ライブラリにどのようなクラスがあるかを調べる方法**

❷ **適切なクラスを見つけた後、そのクラスの使い方を知る方法**

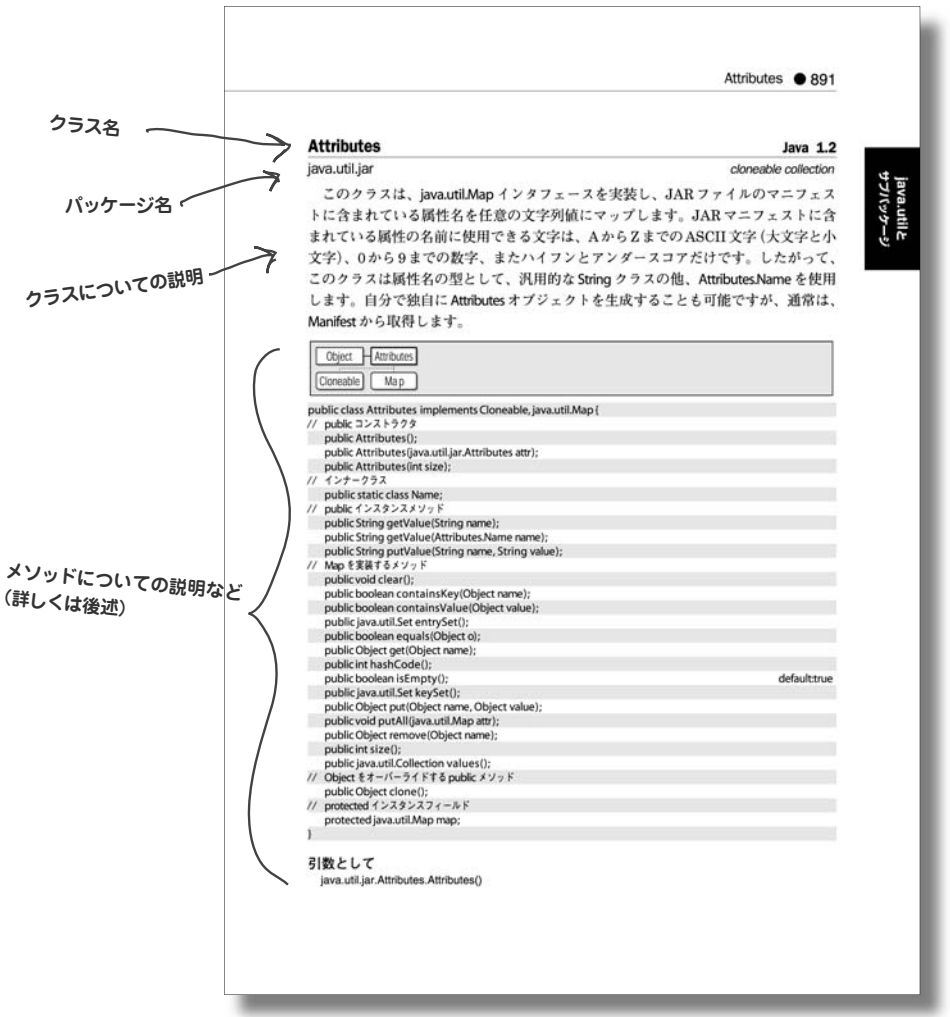
❶ 本で調べる

❷ HTMLのAPIドキュメントで調べる

This PDF is prepared for tools@maruhouse.org, personal use only.

❶ 本で調べる

Javaライブラリにどのようなクラスがあるかを知るには、リファレンスブック（左に表紙が載っている『Javaクイックリファレンス』など）で調べるのが最もいいでしょう。適当にページをめくっているだけでも、役に立つクラスが見つかることが多いものです。



Attributes ● 891

java.utilとサブパッケージ

Attributes Java 1.2

java.util.jar *cloneable collection*

このクラスは、java.util.Map インタフェースを実装し、JAR ファイルのマニフェストに含まれている属性名を任意の文字列値にマップします。JAR マニフェストに含まれている属性の名前に使用できる文字は、A から Z までの ASCII 文字（大文字と小文字）、0 から 9 までの数字、またハイフンとアンダースコアだけです。したがって、このクラスは属性名の型として、汎用的な String クラスの他、Attributes.Name を使用します。自分で独自に Attributes オブジェクトを生成することも可能ですが、通常は、Manifest から取得します。

Object ─ Attributes
Cloneable　Map

```
public class Attributes implements Cloneable, java.util.Map {
// public コンストラクタ
   public Attributes();
   public Attributes(java.util.jar.Attributes attr);
   public Attributes(int size);
// インナークラス
   public static class Name;
// public インスタンスメソッド
   public String getValue(String name);
   public String getValue(Attributes.Name name);
   public String putValue(String name, String value);
// Map を実装するメソッド
   public void clear();
   public boolean containsKey(Object name);
   public boolean containsValue(Object value);
   public java.util.Set entrySet();
   public boolean equals(Object o);
   public Object get(Object name);
   public int hashCode();
   public boolean isEmpty();                                  default:true
   public java.util.Set keySet();
   public Object put(Object name, Object value);
   public void putAll(java.util.Map attr);
   public Object remove(Object name);
   public int size();
   public java.util.Collection values();
// Object をオーバーライドする public メソッド
   public Object clone();
// protected インスタンスフィールド
   protected java.util.Map map;
}
```

引数として
java.util.jar.Attributes.Attributes()

This PDF is prepared for tools@maruhouse.org, personal use only.

❷ HTMLのAPIドキュメントで調べる

Javaの開発環境には、「API仕様」という充実したライブラリのオンラインドキュメントが用意されています。API仕様は、「Java 5, Standard Editionドキュメント(「Java 2 Standard Edition 5.0」と呼ばれることもある)」というドキュメントの構成要素の1つとなっています。ただ、このドキュメントは、Java 5のダウンロードファイルには含まれていないので、別にダウンロードする必要があります。高速のインターネット接続が可能で、しかも忍耐力に自信のある人であれば、Sunのサイトに逐一アクセスして参照することもできますが、やはりローカル環境に持っておいた方が便利でしょう。

API仕様は、クラスやメソッドの詳細について知るには最適なドキュメントと言えます。自分の目的に合ったクラスを探すには、紙のリファレンスブックの方が便利なのですが、リファレンスブックには個々のクラスについての情報はあまり載っていません。どのクラスを使えばよいかがわかったら、後はオンラインドキュメントで調べた方がよいのです。

たとえば、先に紹介した『Javaクイックリファレンス』でArrayListクラスについて調べれば、メソッドの引数や戻り値などはわかります。また、DotComクラスに使用したindexOf()などのメソッドがあることもわかります。しかし、それだけの情報では、メソッドを使いこなすには不十分なのです。引数に指定されたオブジェクトがリスト中に存在しなかった場合にはどうなるのか、といったことがわからないからです。その点、オンラインドキュメントならそういうことについても知ることができます(多くの場合、ですが)。オンラインドキュメントでindexOf()メソッドについて調べると、引数に指定されたオブジェクトがリストに存在しない場合は値「-1」が戻されるという記述が見つかります。これで、私たちは、このメソッドをリスト中のオブジェクトのインデックスを知る以外に、特定のオブジェクトが存在するか否かを確認する際にも使用できることを知るのです。オンラインドキュメントがなければ、「存在しないオブジェクトを引数に指定するとエラーになる」などと思い込む人がいても不思議はないでしょう。

This PDF is prepared for tools@maruhouse.org, personal use only.

エクササイズ

プログラミングマグネット

バラバラになったJavaプログラムを並べ替え、下のような実行結果が得られるようにしてください。注意：API仕様を見るとわかりますが、ArrayListには、次のような2つの引数を指定するadd()メソッドもあります。
このエクササイズでは、このadd()メソッドを使っています。
add(int index, Object o)
このメソッドでは、追加するオブジェクト、そして追加する位置を指定できます。

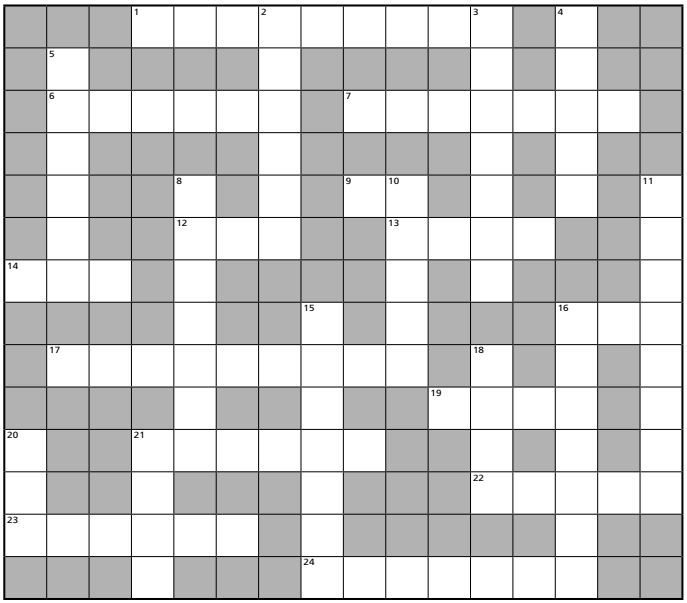
```
a.remove(2);
printAL(a);
a.add(0,"zero");
a.add(1, "one" );
printAL(a);
public static void printAL(ArrayList<String> al) {
if (a.contains("two")) {
   a.add("2.2");
}
a.add(2,"two");
public static void main (String[] args) {
System.out.print(element + "  ");
}
System.out.println(" ");
if (a.contains("three")) {
  a.add("four");
}
public class ArrayListMagnet {
if (a.contains("four") != 4) {
  a.add(4,"4.2");
}
}
}
}
import java.util.*;
printAL(a);
ArrayList<String> a = newArrayList<String>();
for (String element : al) {
a.add(3, "three") ;
printAL(a);
```

```
File  Edit  Window  Help  Dance
% java ArrayListNagnet
zero  one  two  three
zero  one  three  four
zero  one  three  four 4.2
zero  one  three  four 4.2
```


This PDF is prepared for tools@maruhouse.org, personal use only.

Java クロスワード

Javaに関する用語のクロスワードです。本章をよく読んでいれば必ず解けるはずです。ただ、Javaには関係のない用語も混じっているので注意してください（注：このクロスワードには英語で、すべて大文字で答えてください）。



ヨコのカギ

1. 単純な数値データ
6. 現実世界の「もの」に対応する
7. 要素がどこにあるかを探す
9. 条件によって動きを変える
12. リストにデータを追加する
13. 大きい整数
14. 値のコピー
16. 整数
17. 配列に似たクラス
19. リスト中の要素の数を知る
21. 配列の要素の数を知る
22. スペイン料理
23. タイプが面倒な時に使う
24. パッケージの集まり

タテのカギ

2. クラスの機能
3. 配列やArrayListを構成する
4. 2番目に小さいデータ
5. 浮動小数点数のデフォルト
8. クラスの集まり
10. ビット数の少ない浮動小数点数
11. 特定のオブジェクトが存在するかを確かめる
15. 現実のようだが
16. リストは空か
18. 配列は値の
20. ライブラリ
21. 同じことを繰り返す

ヒント

ヨコのカギ

1. 8種類ある
7. ArrayListのメソッド
16. ごく一般的なプリミティブ型
21. 長さ
22. Javaには関係ない

タテのカギ

2. オーバーライドできる
3. 要素
4. と10. どちらもプリミティブ型
16. ArrayListのメソッド
18. 一覧表

This PDF is prepared for tools@maruhouse.org, personal use only.

エクササイズの答え

```
File  Edit  Window  Help  Dance
% java ArrayListNagnet
zero  one  two  three
zero  one  three  four
zero  one  three  four 4.2
zero  one  three  four 4.2
```

```
import java.util.*;
public class ArrayListMagnet {
    public static void main (String[] args) {
        ArrayList<String> a = new ArrayList<String>();
        a.add(0,"zero");
        a.add(1,"one");
        a.add(2,"two");
        a.add(3,"three");
        printAL(a);
        if (a.contains("three")) {
            a.add("four");
        }
        a.remove(2);
        printAL(a);
        if (a.indexOf("four") != 4) {
            a.add(4, "4.2");
        }
        printAL(a);
        if (a.indexOf("two") != 4) {
            a.add("2.2");
        }
        printAL(a);
    }
    public static void printAL(ArrayList<String> al) {
        for (String element : al) {
            System.out.print(element + "  ");
        }
        System.out.println(" ");
    }
}
```


This PDF is prepared for tools@maruhouse.org, personal use only.

Javaクロスワード

			1 P	R	I	2 M	I	T	I	V	3 E		4 S		
	5 D					E					L		H		
	6 O	B	J	E	C	T		7 I	N	D	E	X	O	F	
	U					H					M		R		
	B			8 P		O		9 I	10 F		E		T		11 C
	L			12 A	D	D			13 L	O	N	G			O
14 G	E	T		C					O		T				N
				K			15 V		A				16 I	N	T
	17 A	R	R	A	Y	L	I	S	T		18 L		S		A
				G			R			19 S	I	Z	E		I
20 A			21 L	E	N	G	T	H			S		M		N
P			O				U				22 T	A	P	A	S
23 I	M	P	O	R	T		A						T		
			P				24 L	I	B	R	A	R	Y		

自分で考えてみよう

上と同じ答えが出るよう、自分でもヨコのカギとタテのカギを考え、下の空欄に書き込んでみてください。著者が作ったものより問題を易しくしたり難しくしたりするのも面白いでしょう。

タテのカギ

2.
3.
4.
5.
8.
10.
11.
15.
16.
18.
20.
21.

ヨコのカギ

1.
6.
7.
9.
12.
13.
14.
16.
17.
19.
21.
22.
23.
24.

This PDF is prepared for tools@maruhouse.org, personal use only.

7章　継承とポリモーフィズム

オブジェクト指向をより深く知る

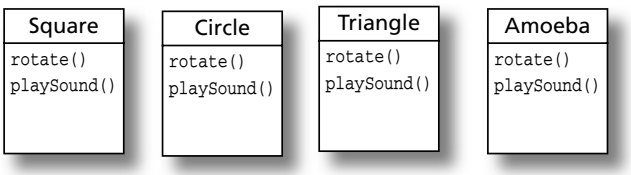
プログラムを作る際には、将来のことを考えて設計をしなければなりません。たとえば、後で改造が必要になったとしても、作業に追われて休みも取れない、などということはできれば避けたいものです。また、自分以外の人が簡単に改造できるというのも重要です。「もうすぐ完成」という段階になっての突然の仕様変更などにも柔軟に対応できれば言うことはないでしょう。この章では、そのために役立つ、オブジェクト指向言語の「ポリモーフィズム」、「継承」といったことがらについて解説します。ポリモーフィズムと継承について十分に理解すれば、より柔軟性のある、改造の容易なプログラム（クラス）を設計することができるようになります。しかも、理解するのがさほど難しいというわけではないので、心配することはありません。思い切ってすぐにでも勉強を始めてください。必ず頑張って取り組むだけの価値があるはずです。

This PDF is prepared for tools@maruhouse.org, personal use only.

2 章の復習

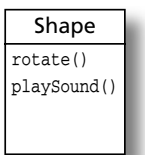
手続き指向プログラマであるLarryとオブジェクト指向プログラマであるBradが対決をする話を 2 章で紹介しました。ここで、もう一度 2 人の対決についておさらいしてみましょう。2 人の会話の中に、「継承」という概念について触れた、以下のような箇所がありました。

Larry：君のコードには重複が多いね。同じ rotate 関数が 4 つもあるじゃないか。
Brad：これは関数じゃないよ。「メソッド」って言うのさ。
Larry：何だっていいよ。こんなのバカげているよ。4 つも同じような「メソッド」があるなんて。面倒なだけだろう。何かいいことがあるのかい？
Brad：もしかして、君は作りかけのプログラムを見たんじゃないのか。最終的にどうなったか見せてあげるよ。最終的にはオブジェクト指向で言う「継承」というテクニックを使ったんだ。

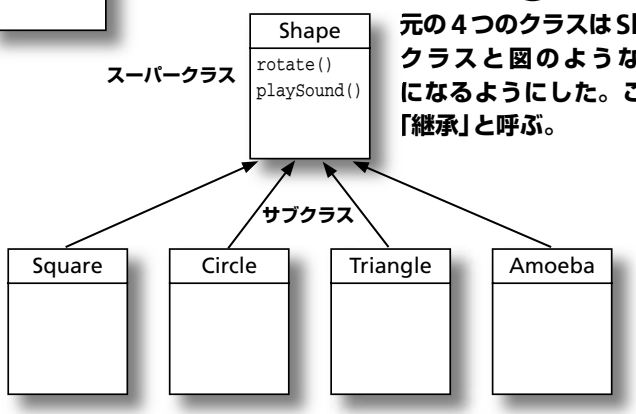


❶ 4 つのクラスに共通のメソッドがある。

❷ 4 つのクラスに対応するものはすべて「図形」で、回転する機能、音を鳴らす機能を備えている。そこで、共通する部分だけを取り出して（この作業を「抽象」と呼ぶ）新しい「Shape」というクラスを作った。



❸ 元の 4 つのクラスは Shape クラスと図のような関係になるようにした。これを「継承」と呼ぶ。



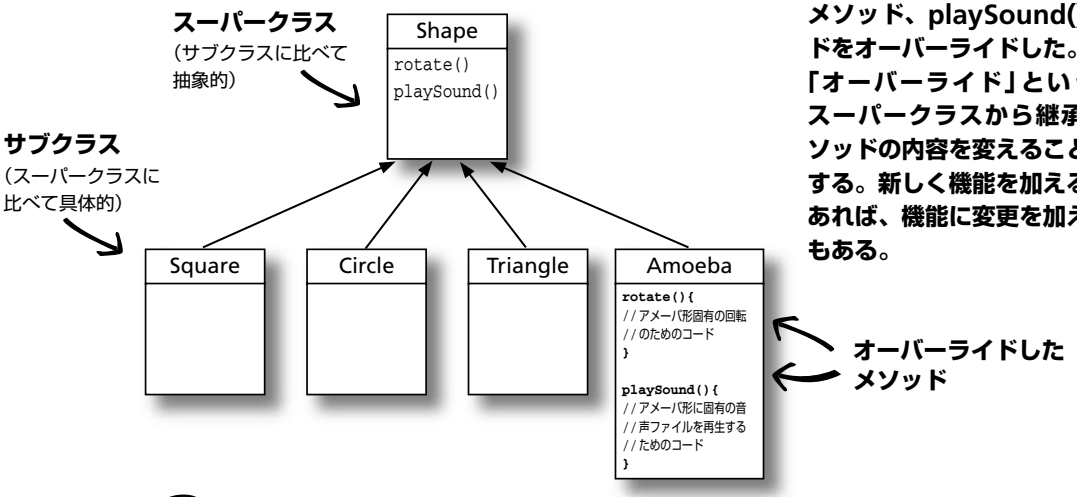
この場合、たとえば、「Square クラスは Shape クラスを継承する」、「Circle クラスは Shape クラスを継承する」というような言い方をします。Square クラス、Circle クラスなどからは、Shape と共通するメソッドを削除します。これで、同じ名前のメソッドが重複して存在するということはなくなります。
Shape のようなクラスを「スーパークラス」と呼びます。また、その他の 4 つのクラスは、「Shape のサブクラス」と呼びます。サブクラスは、スーパークラスのメソッドを利用することができます（これを「メソッドを継承する」という）。つまり、サブクラスは自動的にスーパークラスと同じ機能を持つことになる、ということです。

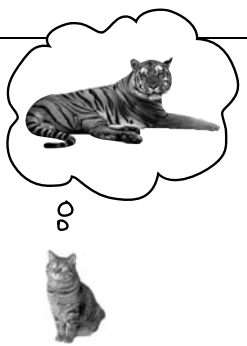
This PDF is prepared for tools@maruhouse.org, personal use only.

アメーバ形のメソッドはどうするか？

Larry：でも、アメーバ形の場合はrotate関数もplaySound関数も機能がかなり違うはずじゃないか。他とまったく同じというのは問題だろう。
Brad：関数じゃないよ、「メソッド」だって。
Larry：どっちでもいいよ。機能が違うはずなのに、Shapeクラスの機能を「継承」してしまうのはまずいよ。
Brad：話はこれで終わりじゃないんだよ。Amoebaクラスの場合だけは、同じ名前のメソッドでも内容を変える。「オーバーライド」って言うんだけど。そうすれば、アメーバ形の場合だけ、他の図形と違うrotate()メソッド、playSound()メソッドが実行されることになる。

❹ Amoebaクラスでは、スーパークラスであるShapeのrotate()メソッド、playSound()メソッドをオーバーライドした。「オーバーライド」というのは、スーパークラスから継承したメソッドの内容を変えることを意味する。新しく機能を加えることもあれば、機能に変更を加えることもある。





頭の体操

「飼いネコ」を表すクラスと「トラ」を表すクラスがあったとしたら、両者の関係はどのようになるでしょうか。どちらがスーパークラスでどちらがサブクラスでしょうか。たとえば、「飼いネコ」が「特殊なトラ」と言えるものであれば、「飼いネコ」クラスは「トラ」クラスのサブクラスであると言えます。それとも、両者とも、何か別のクラスのサブクラスなのでしょうか。
また、2つのクラスにはそれぞれどのようなメソッドがあるでしょうか。他のクラスから継承するメソッド、オーバーライドするメソッドはどれでしょうか。
次のページに進む前によく考えてください。

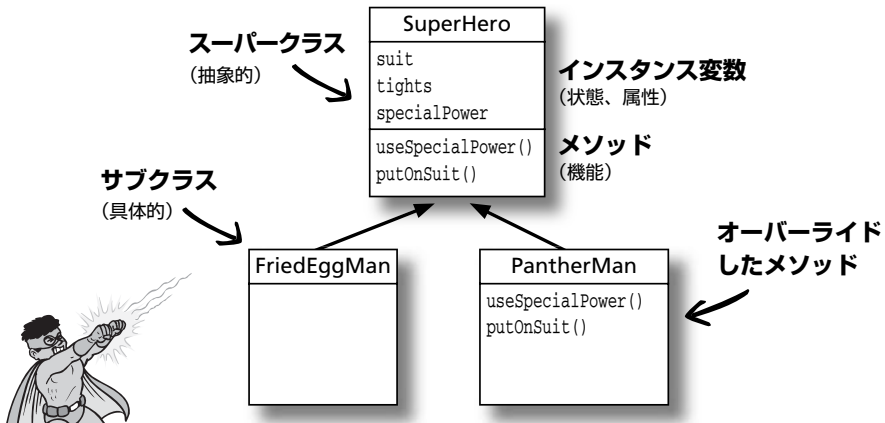
This PDF is prepared for tools@maruhouse.org, personal use only.

継承について理解する

「継承」というのは、あるクラスの特性、機能を引き継ぎ、より「特殊な」クラスを作ることです。引き継ぐ方のクラスを「サブクラス」、引き継がれる方のクラス（より「一般的な」クラス。一般的であることを「抽象的」ともいう）を「スーパークラス」と呼びます。サブクラスはスーパークラスを継承して作るわけです。

Javaでは、「別のクラスを継承している（つまり、他のクラスのサブクラスである）」ということを、「extends」という言葉で表します。「継承している」というのは、簡単にまとめれば、「メンバ」を引き継いでいる、ということです。「メンバ」というのは、インスタンス変数とメソッドを指します。

たとえば、以下のように、PantherManクラスがSuperHeroのサブクラスである場合には、PantherManクラスは、SuperHeroクラスからsuit、tights、specialPowerなどのインスタンス変数、useSpecialPower()などのメソッドを継承することになります。ただ、注意すべきなのは、サブクラスであるPantherManでは、スーパークラスであるSuperHeroにない新しいメソッドやインスタンス変数を追加することができ、また継承したメソッドやインスタンス変数に変更を加えることもできる（オーバーライドもできる）ということです。



FriedEggManは何も特別な能力を持っていないため、メソッドのオーバーライドは行いません。メソッドもインスタンス変数もSuperHeroクラスと共通のものでよいのです。

しかし、PantherManは、SuperHeroにない特別な能力を持つため、useSpecialPower()メソッド、putOnSuit()メソッドのオーバーライドが必要になっています。

インスタンス変数はその必要がないため、オーバライドしていません。この場合、サブクラスはスーパークラスとまったく同じインスタンス変数を持ち、それぞれに独自の値を設定することになります。たとえば、PantherManでは、tights変数に“purple”という値を設定し、FriedEggManでは同じ変数に“white”という値を設定するといったことができるのです。

This PDF is prepared for tools@maruhouse.org, personal use only.

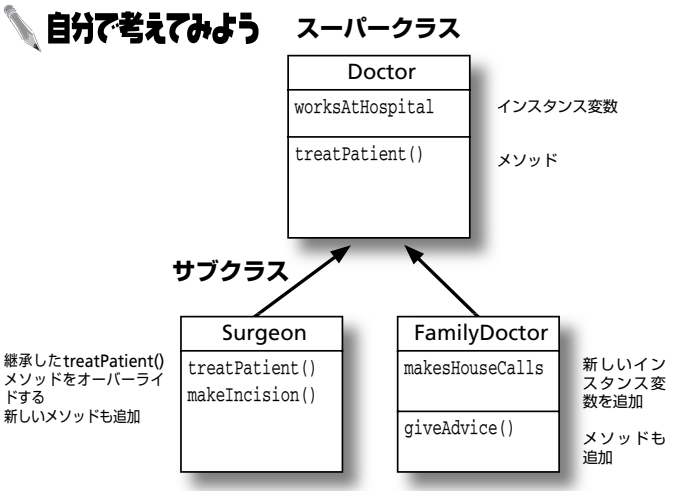
継承の例

```
public class Doctor {
  boolean worksAtHospital;

  void treatPatient() {
    //診察をする
  }
}
```

```
public class FamilyDoctor extends
Doctor {

  boolean makesHouseCalls;
  void giveAdvice() {
    //助言をする
  }
}
```

```
public class Surgeon extends Doctor{

  void treatPatient() {
    //手術する
  }

  void makeIncision(){
    //切開する
  }
}
```

自分で考えてみよう



- Surgeonクラスにはインスタンス変数がいくつあるか。
- FamilyDoctorクラスにはインスタンス変数がいくつあるか。
- Doctorクラスにはメソッドがいくつあるか。
- Surgeonクラスにはメソッドがいくつあるか。
- FamilyDoctorクラスにはメソッドがいくつあるか。
- FamilyDoctorクラスにはtreatPatient()の機能はあるか。
- FamilyDoctorクラスにはmakeIncision()の機能はあるか。

This PDF is prepared for tools@maruhouse.org, personal use only.

動物シミュレーションプログラムを構成する

クラスの継承関係を考える

何種類かの動物を同じ環境の中に置いた時、どのようなことが起きるかを調べるシミュレーションプログラムを設計するよう依頼されたとします。今のところはまだ実際のコーディングに入る必要はありません。ただ設計だけを考えるのです。

プログラムに含めるべき動物はすでに一部（全部ではない）わかっています。この動物をすべてクラス（オブジェクト）に置き換えることになります。そして、各クラスには、対応する動物にふさわしい特性と機能を持たせます。

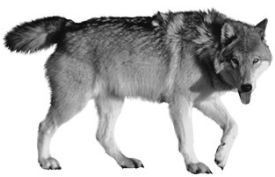
ここで重要なことは、必要な動物が増えた時に、誰でもが簡単に新しい動物（に対応するクラス）を追加できるようにしなければならない、ということです。

まず考えるべきなのは、すべての動物に共通する性質やふるまいにはどのようなものがあるか、ということでしょう。そして、その性質やふるまいを反映して、すべての動物に対応するクラスを作るのです。他の動物に対応するクラスは、そのクラスを継承して作ることになります。

1 すべての動物に共通する性質、ふるまいを考える

ここに示した 6 種類の動物に共通する性質、ふるまいとは何でしょうか。それがわかると、すべての動物に対応するクラスの作成（ステップ 2）に役立ちます。

また、その他、動物間の関係についても考えます。それがわかるとクラスどうしの関係（クラスＡがクラスＢを継承する、など）を知る（ステップ 4、5）のにも役立ちます。



This PDF is prepared for tools@maruhouse.org, personal use only.

重複を防ぐのに役立つ継承

すべてのクラスに共通すると思われるインスタンス変数は次の5つ。

picture ── 動物の画像を納めたJPEGファイルの名前
food ── 動物の食べ物の名前。値は“meat(肉)”と“grass(草)”の2つ。
hunger ── 動物の空腹度を表す。動物がいつ、どのくらいの量の食べ物を食べたかによって値が変化する。
boundaries ── 動物の行動範囲。縦の長さと横の長さで表す(例：640 × 480)。
location ── 動物の現在地。X座標、Y座標で表す。

すべてのクラスに共通すると思われるメソッドは次の4つ。

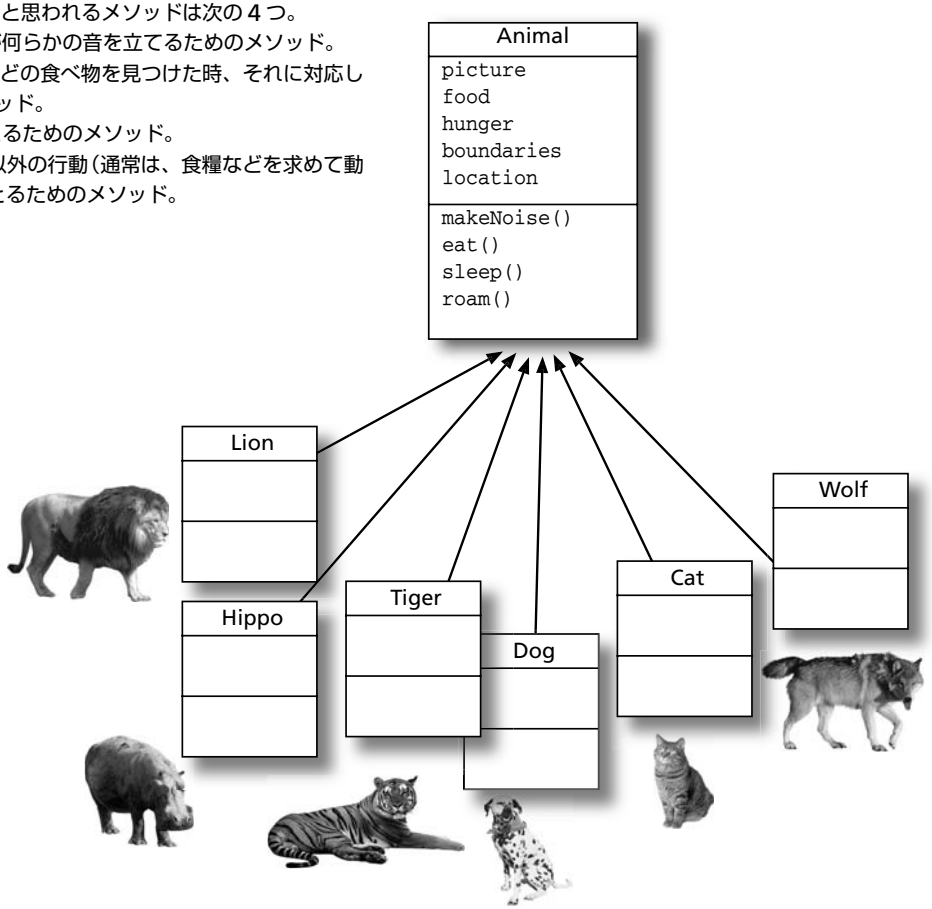
makeNoise() ── 動物が何らかの音を立てるためのメソッド。
eat() ── 動物が肉や草などの食べ物を見つけた時、それに対応して行動するためのメソッド。
sleep() – 動物が睡眠をとるためのメソッド。
roam() – 食べる、眠る以外の行動(通常は、食糧などを求めて動き回ることを指す)をとるためのメソッド。

2

すべての動物に対応するクラスを作成する

すべての動物に対応するクラスということから、名前は「Animal」にします。

Animalクラスには、他のすべてのクラスに必要となりそうなメソッドとインスタンス変数を持たせます。



This PDF is prepared for tools@maruhouse.org, personal use only.

動物の「食べ方」はみな同じか

動物に対応するすべてのクラスに共通するインスタンス変数を前のページに示しましたが、これについては問題がないと思います。もちろん、使用する画像ファイルも、食べ物も、行動範囲も、現在地も動物ごとに異なるわけですが、それぞれに対応するインスタンス変数の「値」を変えれば済むからです。変数自体はどのクラスでも同じでいいのです。しかし、メソッドについてはどうでしょうか。

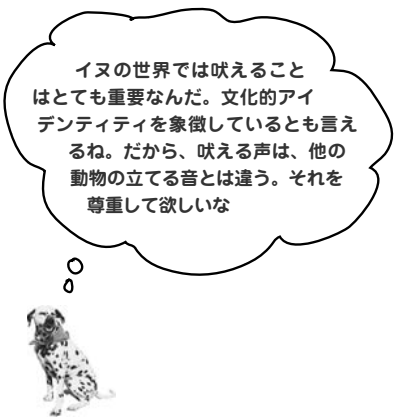
オーバーライドすべきメソッド

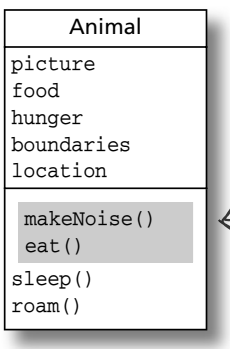
Animalクラスには、makeNoise()というメソッドを持たせますが、果たしてライオンとイヌの立てる音（あるいは音の立て方）は同じでしょうか。ネコとカバではどうでしょうか。おそらく、makeNoise()メソッドとeat()メソッドは動物ごとに変える必要があるでしょう。動物の種類によって違う音声ファイルを再生する、といった方法を採れば（ファイル名はインスタンス変数で指定する）、メソッドを共通にすることもできなくはありません。しかし、それではクラスが実際の動物とはかけ離れたものになってしまうでしょう。動物は、さまざまな状況でさまざまな種類の音を立てます（ものを食べる時の音もあれば、敵に遭遇した時に立てる音もある）。1つのメソッドでそれに対応するのは無理があるのです。

2章で、Shapeクラスのrotate()メソッドをAmoebaクラスでオーバーライドしたように、Animalクラスのメソッドもサブクラスでオーバーライドすることになります。

3 メソッドをサブクラスでオーバーライドすべきかどうか見極める

Animalクラスのメソッドのうち、eat()、makeNoise()はサブクラスごとにオーバーライドする必要があります。





ものの食べ方、音の立て方は動物ごとに違うので、eat()、makeNoise()メソッドはサブクラスごとにオーバーライドすべき。今のところ、sleep()メソッド、roam()メソッドは共通にしておく

This PDF is prepared for tools@maruhouse.org, personal use only.

継承関係を再検討する

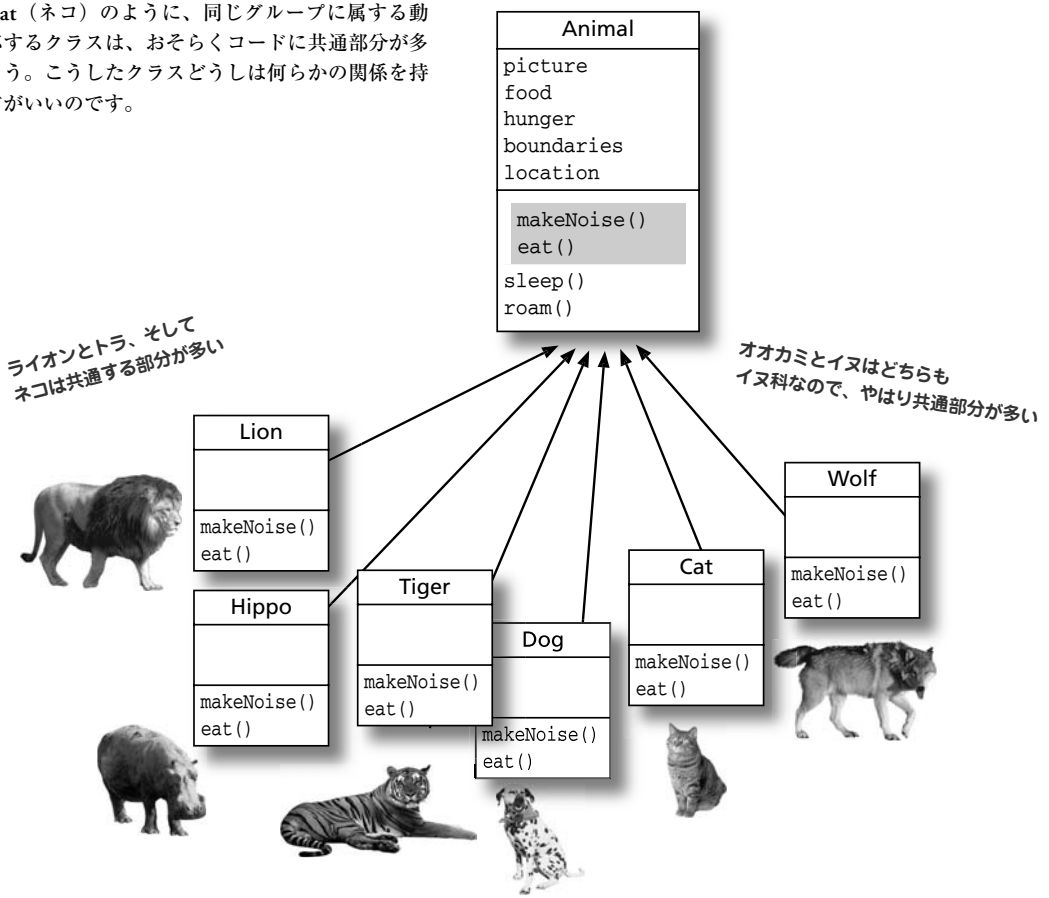
クラスどうしの継承関係は、ほぼ明確になってきました。また、eat()メソッド、makeNoise()メソッドはオーバーライドすべき、ということもわかりました。そうすれば、イヌとネコの鳴き声を間違えることもないでしょうし、カバがライオンのようにものを食べることもないでしょう。

しかし、ここで再度クラスの継承関係をよく見てみると、まだ改良の余地があることに気付きます。今のところは、すべてのクラスがAnimalクラスのサブクラスになっているだけです。しかし、たとえば、Wolf（オオカミ）とDog（イヌ）、Lion（ライオン）とTiger（トラ）とCat（ネコ）のように、同じグループに属する動物に対応するクラスは、おそらくコードに共通部分が多いでしょう。こうしたクラスどうしは何らかの関係を持たせた方がいいのです。

4

継承関係に改良の余地がないかを確認する。似た特徴を持つサブクラスがないかを探す。

WolfとDog、LionとTigerとCatは同じ科の動物に対応するため、共通部分が多いと判断します。



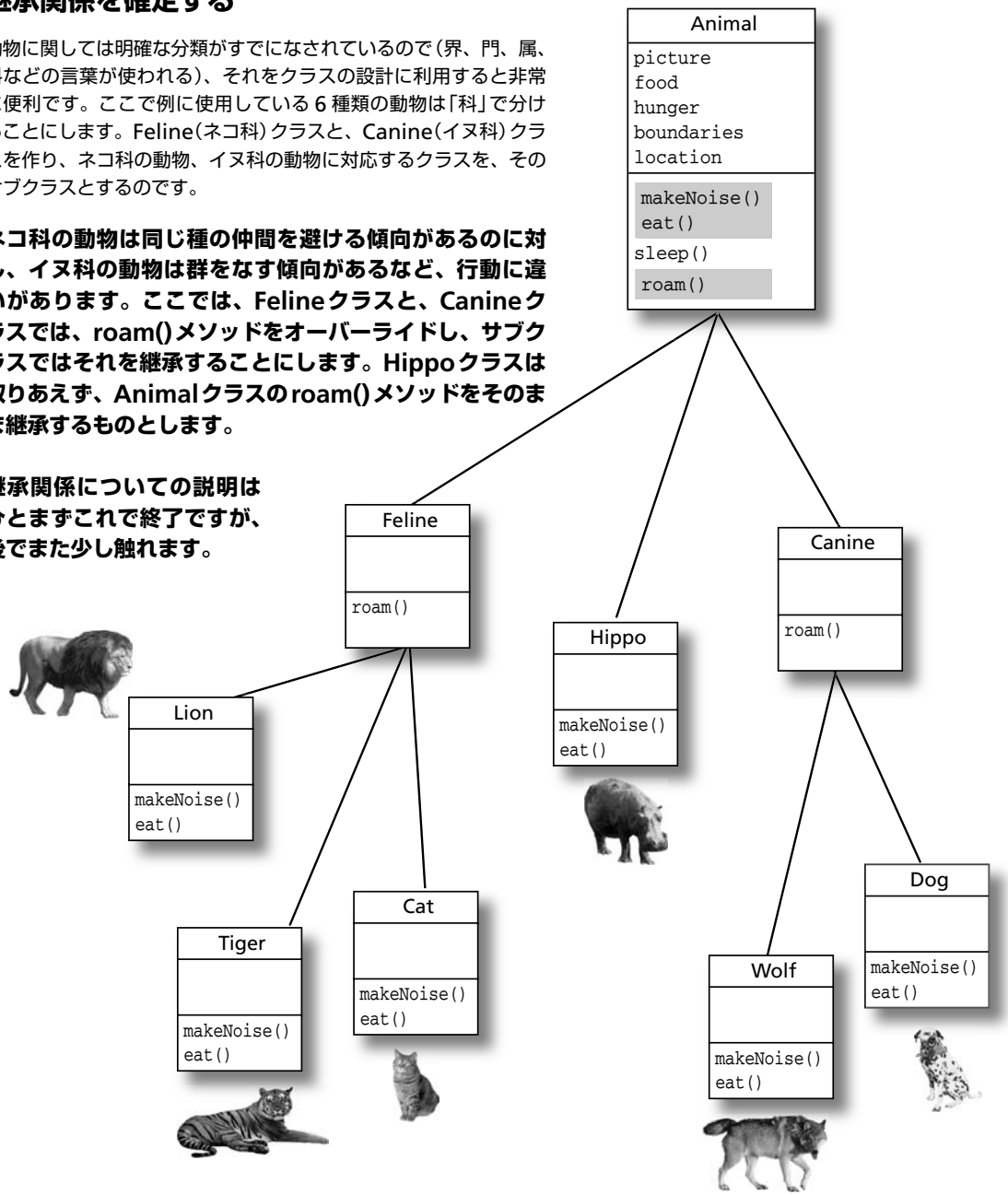
This PDF is prepared for tools@maruhouse.org, personal use only.

❺ 継承関係を確定する

動物に関しては明確な分類がすでになされているので（界、門、属、科などの言葉が使われる）、それをクラスの設計に利用すると非常に便利です。ここで例に使用している 6 種類の動物は「科」で分けることにします。Feline（ネコ科）クラスと、Canine（イヌ科）クラスを作り、ネコ科の動物、イヌ科の動物に対応するクラスを、そのサブクラスとするのです。

ネコ科の動物は同じ種の仲間を避ける傾向があるのに対し、イヌ科の動物は群をなす傾向があるなど、行動に違いがあります。ここでは、Felineクラスと、Canineクラスでは、roam()メソッドをオーバーライドし、サブクラスではそれを継承することにします。Hippoクラスは取りあえず、Animalクラスのroam()メソッドをそのまま継承するものとします。

継承関係についての説明はひとまずこれで終了ですが、後でまた少し触れます。



This PDF is prepared for tools@maruhouse.org, personal use only.

呼び出されているメソッドは？

Wolfクラスは4つのメソッドを持つことになります。1つはAnimalクラスから継承したもの、また1つはCanineクラスから継承したもの（厳密には、Animalクラスから継承し、Canineクラスでオーバーライドされたメソッドを継承したもの）です。4つのメソッドは、まず、以下のように、Wolfクラスのオブジェクトを作成し、そのオブジェクトへの参照を変数に代入してから、「.」演算子を使用して呼び出します。それぞれのコードで実際に呼び出されるメソッドがどれなのかを下の図で確かめてください。

Wolfオブジェクトを作成する

```
Wolf w = new Wolf();
```

Wolfクラスでオーバーライドされたもの

```
w.makeNoise();
```

Canineクラスでオーバーライドされたもの

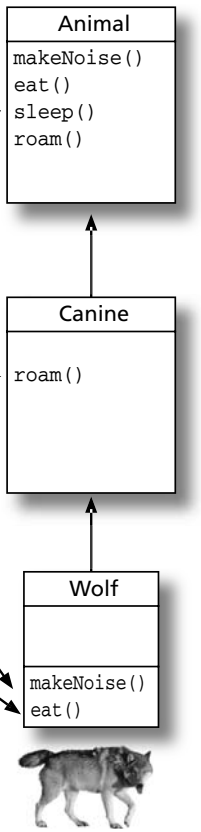
```
w.roam();
```

Wolfクラスでオーバーライドされたもの

```
w.eat();
```

Animalクラスのメソッドそのまま

```
w.sleep();
```



同じ名前のメソッドが複数存在する際には、最も「特殊な」メソッドが呼び出されることになります。

言い換えれば、できる限り下位のクラスのメソッドが呼び出されるということです（ただし、呼び出しに使用したオブジェクトより下位のクラスのメソッドが呼び出されることはない）。

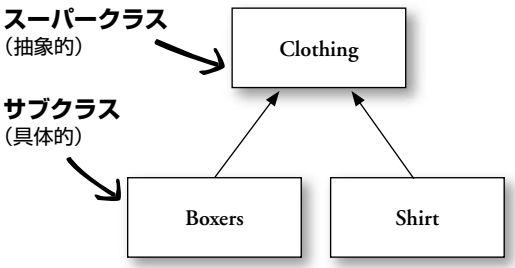
「下位」というのは、「継承関係の中で下位」という意味です。CanineクラスはAnimalクラスよりも下位で、WolfクラスはCanineクラスより下位です。Wolfオブジェクト（への参照）を利用してあるメソッドが呼び出されると、（JVMによって）該当するメソッドがまずWolfクラスで探されます。Wolfクラスで見つからない時は、次にCanineクラス、その次にAnimalクラスというように、見つかるまで順に検索対象クラスが上位に移されていきます。

This PDF is prepared for tools@maruhouse.org, personal use only.

継承関係を図に表す

クラス	スーパークラス	サブクラス
Clothing	---	Boxers、Shirt
Boxers	Clothing	
Shirt	Clothing	

継承関係の票



継承関係を表す図（クラス図）

自分で考えてみよう

継承関係の図も書いてみましょう

表の中のクラスどうしの関係を考え、空欄を埋めてください。

クラス	スーパークラス	サブクラス
Musician		
Rock Star		
Fan		
Bass Player		
Concert Pianist		

ヒント：どのクラスにも必ずスーパークラス、サブクラスがあるとは限りません
ヒント：必要であれば、新たなクラスを追加したり、表中のクラスを別のものに変えてもかまいません

素朴な疑問に答えます

Q：メソッドを呼び出すと、JVMによって、該当するメソッドが（Wolfオブジェクトを利用して呼び出した時はWolfクラスを起点として）下位のクラスから順に探されるということですが、該当するクラスが結局どこにも見付からなかった場合はどうなるのですか。

A：いい質問です。しかし、メソッドが存在するかどうかはコンパイル時に確認されるため、実行時に急に「メソッドが見付からない」という事態になることはないので心配は要りません。ただ、コンパイラは、呼び出すべきメソッドが厳密にどのクラスに属するものか、ということは問題にしません。たとえば、Wolfオブジェクトのsleep()メソッドを呼び出す場合、正確には、Animalクラスのsleep()メソッドを呼び出すことになるのですが、それはコンパイラにとってはどうでもいいことなのです。スーパークラスから継承したメソッドは、コンパイラから見ればサブクラスのメソッドでもある、と考えて間違いはないでしょう。

一方、JVMはコンパイラとは違い、「呼び出すべきメソッドはどのクラスのものか」を厳密に確認します。下位のクラスと上位のクラスに同じ名前のメソッドがあれば、下位のクラスのものが優先されます。ただし、呼び出しに使用されたオブジェクトより下位のクラスでメソッドが探されることはありません。

This PDF is prepared for tools@maruhouse.org, personal use only.

IS-A関係とHAS-A関係

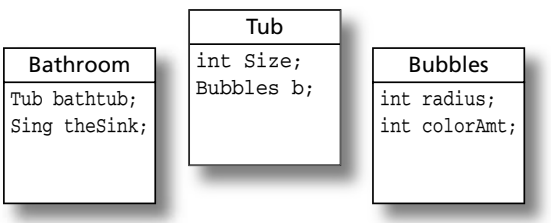
2つのクラスが継承関係にあるかどうかを確認するには、両者に対応する「もの（こと）」どうしの関係を英語の「A is a B」という文で表せるかどうかを調べる、という方法が有効です。表せれば、Aに対応するクラスはBに対応するクラスを継承する（前者は後者のサブクラス）ということになります。この文は日本語では「AはBの一種である」という意味になります。このような文で表せる関係を「IS-A関係」と呼びます。

たとえば、「Triangle（三角形）」と「Shape（図形）」の関係は「Triangle is a Shape」という文で表せるので、TriangleクラスはShapeクラスを継承する、ということになります。また、「Cat（ネコ）is a Feline（ネコ科）」という文も書けるので、CatクラスはFelineクラスのサブクラスということになります。その他、「Surgeon（外科医）is a Doctor（医者）」という文も書けるので、SurgeonクラスはDoctorクラスのサブクラスと考えていいでしょう。

では、Tub（浴槽）クラスとBathroom（風呂場）クラスはどうでしょうか。一見すると、「TubクラスはBathroomクラスを継承する」と考えてよさそうな気もしてきます。それが正しいかどうかは、両者が「IS-A関係」にあるかどうかを確認すればわかります。

両者の関係はどうも、「Tub is a Bathroom」という文では表せないようです。したがって、TubクラスがBathroomクラスを継承するという考えは誤りと言えます。このように、「IS-A関係」かどうかの確認は、「継承関係によく似ているが実はそうではない」という関係を的確に見つけ出すのに役立つのです。

BathroomとTubが逆であればどうでしょうか。「Bathroom is a Tub」という文もやはりおかしいので、BathroomクラスがTubクラスを継承するということもないようです。



この図の場合は、「Bathroom has a Tub」、「Tub has a Bubbles」という文は成り立つが、IS-A関係はまったく成り立たない。

とはいえ、TubとBathroomに何らかの関係があることは確かです。ただそれが継承関係でないだけのことです。TubとBathroomのような関係を「HAS-A関係」と言います。「Bathroom has a Tub（風呂場には浴槽がある）」という文が成り立つからです。このような関係が成り立つ場合、たとえば、BathroomクラスはTubというインスタンス変数を持つことになります。「Bathroomオブジェクトは、Tubオブジェクトへの参照を持つ」と言ってもいいでしょう。

This PDF is prepared for tools@maruhouse.org, personal use only.

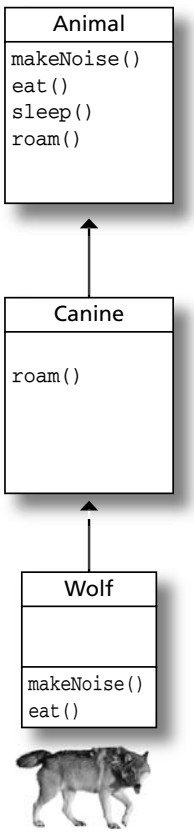
IS-A関係に関する注意事項

IS-A関係は、直接のスーパークラス、サブクラスだけとの間に成り立つわけではありません。設計が正しければ、間接的なスーパークラス、サブクラスとの間にもIS-A関係は成り立ちます。

クラスBがクラスAを継承するのであれば、「B is a A」が成り立つ。また、同様に、クラスCがクラスBを継承するのであれば、「C is B」だけでなく、「C is A」も成り立つ。

CanineはAnimalを継承する
WolfはCanineを継承する
WolfはAnimalを継承する

Canine IS-A Animal
Wolf IS-A Canine
Wolf IS-A Animal



たとえば、クラス間に左の図のような関係がある時には、「WolfはAnimalを継承する」、「Wolf is an Animalが成り立つ」と言ってまったく問題ありません。AnimalはWolfの直接のスーパークラスではなく、「スーパークラスのスーパークラス」ですが、それでもかまいません。間接的にでも継承関係が成り立っていればよいのです。

左の図と同じことを言葉で書くと以下のようになります。

「WolfはCanineの一種であり、WolfはCanineにできることをすべてできる。また、WolfはAnimalの一種でもあり、Animalにできることはすべてできる」

WolfクラスでAnimalクラス、Canineクラスのメソッドがオーバーライドされているかどうかはこの際問題ではありません。Wolfクラスは、一部をオーバーライドしていますが、Animalから makeNoise()、eat()、sleep()、roam() という4つのメソッドを継承しており、事実上4つのメソッドを持つことになります。

This PDF is prepared for tools@maruhouse.org, personal use only.

クラスの継承関係が正しいことを確認する方法

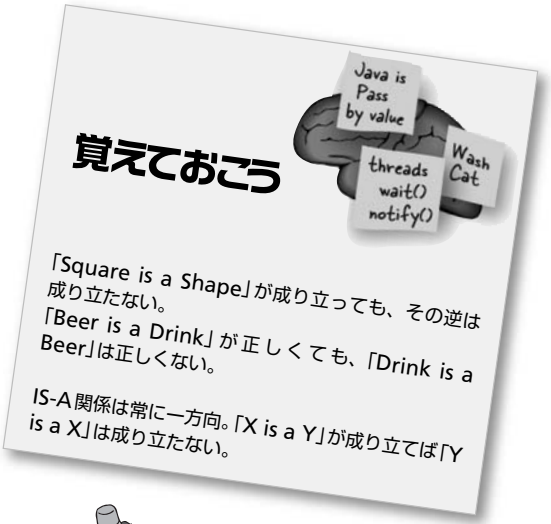
ここまで読んだ人であれば、クラスの継承関係が正しいかどうかを確かめるにはどうすればよいか、ほぼ理解できていると思います。この点については、次の章でさらに詳しく解説することになります（オブジェクト指向について、クラスの設計について、さらに理解を深めることができるはずです）。

今のところは、「IS-A関係が成り立っているかどうか」を基準にして、継承関係の正しさを確認する、ということさえわかっていれば問題ないでしょう。「X is a Y」という文が書ければ、2つのクラスに、直接的、間接的な継承関係を持たせることができます。多くの場合、両者の間には機能的な重複があるでしょう。

IS-A関係は常に「一方向」にしか成り立たない点に注意！

たとえば、「Triangle is a Shape」という文は正しいので、「TriangleはShapeを継承する」と考えてかまいません。

しかし、その逆の「Shape is a Triangle」は誤っています。つまり、「ShapeはTriangleを継承しない」ということです。「X is a Y」が成り立てば「Y is a X」は成り立たないということを常に念頭に置いていてください。



自分で考えてみよう

以下に示す継承関係がそれぞれ正しいかどうか確認してください。

- ☐ OvenはKitchenを継承する。
- ☐ GuitarはInstrument（楽器）を継承する。
- ☐ Person（人間）はEmployee（従業員）を継承する。
- ☐ FerrariはEngineを継承する。
- ☐ FriedEgg（卵焼き）はFoodを継承する。
- ☐ Beagle（ビーグル犬）はPetを継承する。
- ☐ Container（容器）はJar（瓶）を継承する。
- ☐ Metal（金属）はTitanium（チタン）を継承する。
- ☐ GratefulDeadはBandを継承する。
- ☐ Blonde（ブロンドの女性）はSmart（賢明）を継承する。
- ☐ Beverage（飲み物）はMartini（マティーニ）を継承する。

ヒント：「IS-A関係」が成り立っているかどうかを見る

This PDF is prepared for tools@maruhouse.org, personal use only.

素朴な疑問に答えます

Q：サブクラスがスーパークラスのメソッドを引き継ぐということはわかったのですが、逆にスーパークラスでサブクラスのメソッドを利用するということはできないのですか。

A：できません。クラスの継承というのは、いつ、誰が行うかわかりません。ずっと以前に作ったクラスを誰かがまったく知らないところで継承し、オーバーライドするということが多いのです。もちろん、誰がいつ、どのように継承をしたかということをたまたま知っていることもありますが、その場合でも、通常と逆に上位のクラスから下位のクラスのメソッドを継承するということは一切できないのです。子が親の性質を引き継ぐことはあっても、その逆はないのと同じことです。

Q：サブクラスで、スーパークラス中の特定のメソッドの機能をそのまま引き継ぎ、なおかつそのメソッドに独自の機能も追加する、ということはできますか。つまり、スーパークラスのメソッドを完全に別のものに変えてしまわずに、独自の機能も付け加えたい、ということなのですが……。

A：できます。これはJavaの大きな特徴の一つです。クラスを継承する際に「extends（拡張するという意味）」というコードを使用しますが、この言葉の意味から考えれば、スーパークラスのメソッドの機能を残してた上で独自の機能を追加できるのも当然と言えます。

```
public void roam(){
  super.roam();
    // 独自の機能
}
```

このコードによって、スーパークラスのroam()メソッドが呼び出せる。その上で、サブクラス独自のコードを追加すればよい

上記のように、「super」というキーワードを使用すれば、サブクラスのメソッドの中で、スーパークラスのメソッドが呼び出せます。言い換えれば、スーパークラスのメソッドの機能をそのまま引き継げる、ということです。このようにしてスーパークラスのメソッドを呼び出した上で、新たなコードを付け加えれば、独自の機能も追加できます。この方法なら、オーバーライドしても、元のメソッドの機能が失われることはありません。

サブクラスが継承できるもの、できないもの
継承できるメンバを見分ける

サブクラスはスーパークラスのメンバを継承できます。「メンバ」には、インスタンス変数、メソッドなどが含まれるのですが、本書でも後述するとおり、その中にはサブクラスが継承できるものとできないものがあります。それは、メンバごとに「アクセスレベル」の指定ができるからです。このアクセスレベルをどのように指定するかで、それぞれのメンバをどこでどのように利用できるかが決まります。

本書では、4種類のアクセスレベルについて取りあげます。4つのアクセスレベルを制約の大きいものから順に並べると以下のようになります。

private　default　protected　public

アクセスレベルの指定の仕方は、Javaプログラムの設計の善し悪し、信頼性に大きく影響します。まずは、4つのアクセスレベルのうち、publicとprivateについて見てみましょう。この2つについて簡単にまとめると次のようになります。

publicを指定されたメンバは継承できる
privateを指定されたメンバは継承できない

サブクラスがメンバを継承すると、そのメンバは事実上、サブクラス自身で定義されたものと同じになります。たとえば、先に例にあげたSquareクラスはShapeクラスからrotate()、playSound()というメソッドを継承していますが、これは外から見れば、Squareクラスがrotate()、playSound()メソッドを持っているのと同じことです。

これは、「クラスのメンバには、そのクラス自身で定義された変数、メソッド以外に、スーパークラスから継承した変数、メソッドも含まれる」ということを意味します。

注意：default、protectedについては、付録Bで詳しく触れます。

This PDF is prepared for tools@maruhouse.org, personal use only.

継承の適切な使用

継承を行うにあたっては次に示すようなルールを守る必要があります。なぜこのようなルールがあるかということについては後で詳述します。ここではともかくルールを覚えてください。

● 継承によって新たに作るクラスは、元のクラスよりも「具体的」なものにする

たとえば、Tree（木）クラスを継承して、Willow（ヤナギ）クラスを作る、というようなことをします。

●機能のかなりの部分が共通している同種のクラスを複数作る必要がある場合に継承を利用する

何度も例に取りあげているSquare、Circle、Triangleといったクラスは、すべて同様の機能を持つrotate()、playSound()などのメソッドを備えています。このような場合には、すべてのクラスに共通するスーパークラス（Shape）を作り、それを継承する、という方法を採ると、保守や拡張が容易になります。「継承」は、オブジェクト指向プログラミングにおいて重要な意味を持つものです。ただし、注意しなくてはならないのは、コードを再利用する方法として、常に「継承」が最良であるとは限らない、ということです。プログラムを作っていく際の叩き台にはなるし、デザイン上、継承をするのが正しい選択であることも多いのですが、デザインパターンについての知識があれば、もっと柔軟に様々な選択肢を見つけることができるはずです。デザインパターンについて詳しく学びたい方は、同じシリーズの『Head Firstデザインパターン』を次に読むといいでしょう。

● 他のクラスのコードを再利用するだけの目的で継承をしない

継承の目的は「コードの再利用」にあるのですが、スーパークラスとサブクラスが同種のものでない限りは、継承を利用すべきではありません。すでに述べたとおり、サブクラスはスーパークラスをより「具体的に」、より「特殊に」したものである、という関係になっている必要があるのです。たとえば、Alarmというクラスを作る時に、Pianoというクラスのコードが一部再利用できる、ということがわかったとしても、AlarmクラスとPianoクラスが同種のものでなければ、継承をすべきではないのです。同種のものでなければ、「AlarmクラスはPianoクラスをより具体的に、より特殊にしたものである」という関係にはなり得ません（この場合は、再利用できるコードだけを別のクラスにまとめ、そのクラスをインスタンス変数によって利用する、というかたちを採るべきでしょう）。

● サブクラスとスーパークラスは必ず「IS-A関係」にする

クラスを継承する場合には、「2つのクラスはIS-A関係になっているか」を必ず確認するようにします。たとえば、「Tea is a Beverage」は成り立ちますが、「Beverage is a Tea」は成り立たないので、TeaをBeverageのサブクラスにできても、その逆はできません。

重要ポイント

- 継承をしていることは「extends」というコードで表される。
- サブクラスは、スーパークラスの、publicを指定されたすべてのインスタンス変数とメソッドを継承するが、privateを指定されたインスタンス変数とメソッドは継承しない。
- 継承したメソッドはオーバーライドできるが、インスタンス変数はオーバーライドできない（サブクラスで同じ名前のインスタンス変数を再定義することはできるが、それはオーバーライドとは異なる。また、インスタンス変数のオーバーライドが必要になることはほとんどない）。
- 継承が可能かどうかは、2つのクラスの間に「IS-A関係」が成り立つかどうかによって判断する。「X is a Y」という文が成り立てば、XはYを継承できる。
- IS-A関係は一方向にしか成り立たない。「Hippo is an Animal」は成り立つが、「Animal is a Hippo」は成り立たない。
- サブクラスでメソッドをオーバーライドした場合、サブクラスのオブジェクトを利用してそのメソッドを呼び出すと、スーパークラスの元のバージョンでなく、オーバーライドされたバージョンのメソッドが実行される（下位のクラスのメソッドが優先）。
- クラスBがクラスAを継承し、クラスCがクラスBを継承する場合は、「クラスB is a クラスA」、「クラスC is a クラスB」、「クラスC is a クラスA」という関係が成り立つ。

This PDF is prepared for tools@maruhouse.org, personal use only.

継承は何のために行う？

オブジェクト指向のメリットを十分に享受するには、クラスの継承が重要なポイントになります。継承を利用すれば、まず、複数のクラスに共通する機能をスーパークラスにまとめることにより、コードの重複を減らすことができます。また、共通する機能を一か所にまとめるということは、機能に対する改良も一か所に対してのみ行えばよいということになります。そうすれば、すべてのサブクラスにその改良の内容が反映されるのです。ただ、スーパークラスだけを修正してコンパイルをすれば、それで改良は完了です。サブクラスに関しては何の作業も必要になりません。

スーパークラスだけが修正されれば、サブクラスは自動的に新しいバージョンのスーパークラスを利用するようになります。

この場合、サブクラスには再コンパイルの必要もないのです。スーパークラスに対し、サブクラスに影響するような修正を加えない限り、サブクラスはまったくそのままで使い続けることができます（この「影響するような修正」の具体的な意味については、後述します。今のところは、メソッドの引数や戻り値の型、メソッドの名前など、サブクラスでスーパークラスを利用するために書くコードに関係する修正、と考えておけばいいでしょう）。

① コードの重複を減らす。

いくつものクラスに共通しそうなコードは 1 つのクラスにまとめ、そのクラスを継承してサブクラスを作っていくようにします。そうすれば、共通のコードに修正が必要になった時も一か所についてのみ作業するだけで済みます。サブクラスには一切触れることなく、修正が反映できます。

② 複数のクラスに共通のプロトコルを適用する。

This PDF is prepared for tools@maruhouse.org, personal use only.

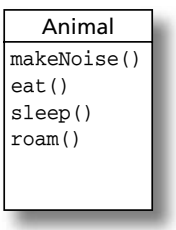
同じスーパークラスを継承して複数のサブクラスを作れば、そのすべてに確実に同じメソッドを持たせることができる*

これは、すべてのサブクラスに同じプロトコルを適用することにつながる。

スーパークラスのメソッドを継承するというのは、別の角度から見れば、スーパークラスと同じプロトコル（ルール、規則）に基づいて動く、という意味になります。同じスーパークラスを継承するサブクラスはすべて同じプロトコルの下に機能するのです。

継承というのは、ある種の契約を交わすようなもの、と言ってもいいでしょう。

たとえば、Animalクラスには以下のようなメソッドがあります。そして、Animalクラスを継承するサブクラスは、すべてこの4つのメソッドに定められたプロトコルに基づいて動くのです。



Animalを継承するクラスは、4つのメソッドのプロトコルに従って動く。プロトコルでは引数や戻り値などが定められる

この場合、「Animalを継承するサブクラス」には、Animalの直接のサブクラスだけでなく、サブクラスのサブクラス、さらにそのサブクラスなど、間接的なサブクラスも含まれます。

ここまでは理解できたでしょうか。実はまだ重要なことに触れていません。それは、「ポリモーフィズム」という概念です。

ポリモーフィズムというのは、サブクラスのオブジェクトをスーパークラスのオブジェクトの代わりとして利用できる、というJavaの特性のことです。

言っている意味がわかるでしょうか。

今、わからなくても心配はいりません。2ページ後で詳しく説明します。その説明を読めば、必ず理解できるはずです。

重要なのは……

ポリモーフィズムを利用できるようになるということ。

ポリモーフィズムを利用する方法とは……

スーパークラスのオブジェクトリファレンスを格納すべく宣言された変数に、サブクラスのオブジェクトリファレンスを代入する。

ポリモーフィズムを利用できるとどのようなメリットが……

非常にシンプルかつフレキシブルで、効率的なコードが書けるようになる。開発が容易になるだけでなく、拡張もこれまでよりはるかに容易になる。

同僚がプログラムのアップデートをしている間に自分だけ休暇をとり、南の島へ……ということが可能になるかもしれない。これはアップデートに際して他人の書いたコードを見る必要はないため。

詳しくは次のページを参照。

読者にも色々な人がいるはずだが、「南の島で休暇」という言葉に惹かれる人は多いはず……

* ただし、サブクラスが持てるのは、スーパークラスのメソッドのうち、おおまかに言えば「public」が指定されたものだけです。この点に関しては後で詳述します。

This PDF is prepared for tools@maruhouse.org, personal use only.

ポリモーフィズムについて解説する前に、まず、オブジェクトの作成、リファレンスの変数への代入、といった操作を通常はどのような手順で行うか、ということについておさらいしておきましょう。

**オブジェクトの作成と
リファレンスの変数への代入**

```
Dog myDog = new Dog();
```

1 3 2

❶ 変数を宣言する。

```
Dog myDog = new Dog();
```

これによって、メモリ上に、オブジェクトへの参照を格納するためのスペースが確保されます。上記の場合、変数の型は常に「Dog」で変わることはありません。この変数に格納されるのは、（CatでもButtonでもSocketでもなく）Dogオブジェクトを操作するためのリモコンということになります。

**❷ オブジェクトを
作成する。**

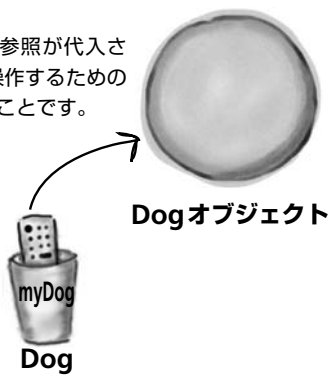
```
Dog myDog = new Dog();
```

新たにDogオブジェクトが作成され、メモリ（ヒープ）上のスペースが割り当てられます。

**❸ 変数にリファレンス
を代入する。**

```
Dog myDog = new Dog();
```

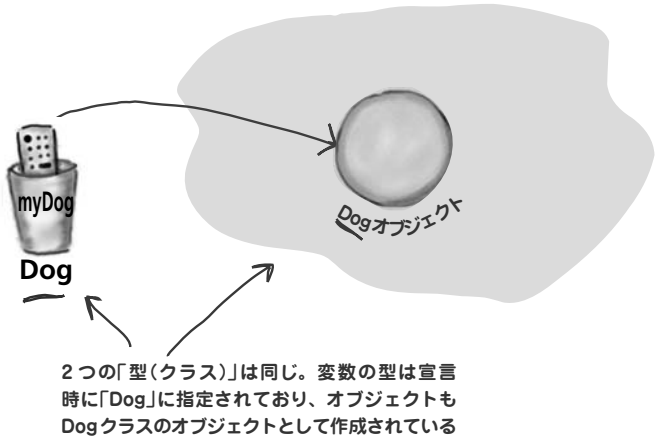
変数myDogにDogオブジェクトへの参照が代入されます。つまり、Dogオブジェクトを操作するためのリモコンがmyDogに格納されるということです。



This PDF is prepared for tools@maruhouse.org, personal use only.

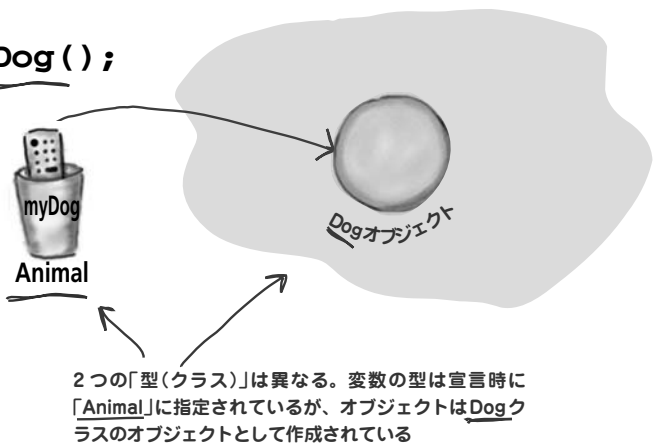
重要なのは、変数とオブジェクトの「型（クラス）」が同じであるということです。

先の例の場合、「型（クラス）」はいずれもDogです。



しかし、ポリモーフィズムを利用すると、以下のようなことができます。変数に、異なる型のオブジェクトリファレンスが代入できるのです。

```
Animal myDog = new Dog();
```



This PDF is prepared for tools@maruhouse.org, personal use only.

ポリモーフィズムを利用すれば、スーパークラスのオブジェクトリファレンスを代入するはずの変数に、サブクラスのオブジェクトリファレンスを代入できます。

参照型の変数には、宣言されたクラスのオブジェクトリファレンス以外に、そのサブクラスのオブジェクトリファレンスも代入できます。宣言されたクラスを直接、間接的に継承しているクラスのオブジェクトリファレンスなら、どのようなものでも代入できるのです。つまり、以下に示すような配列も作れるということです。

ポリモーフィズムを利用した配列の例

```
Animal [] animals = new Animal [5];

animals [0] = new Dog();

animals [1] = new Cat();

animals [2] = new Wolf();

animals [3] = new Hippo();

animals [4] = new Lion();

for (int i = 0; i < animals.length; i++) {

animals [i].eat();

animals [i].roam();

}
```

Animal型の配列、つまりAnimalオブジェクトへの参照を格納する配列を宣言する

配列はAnimal型だが、各要素には、Animalクラスのサブクラスのオブジェクトリファレンスが格納できる

このように書けるのもポリモーフィズムの特徴（このコード例の中でこの部分が特に重要）。この場合も、ループを使用して、配列の要素１つ１つを順に利用できる。各サブクラスから同じ名前のメソッドを１回ずつ呼び出すということも可能。どのメソッドを実行するか、混乱が起きるようなことはない

iの値が０の時には、Dogオブジェクトのeat()メソッドが呼び出される。iが１であれば、Catオブジェクトのeat()メソッドが呼び出される

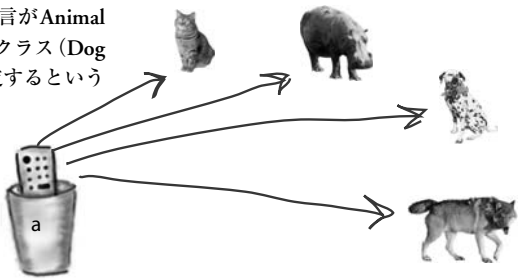
roam()メソッドでも同じ

This PDF is prepared for tools@maruhouse.org, personal use only.

ポリモーフィズムには他の利用方法も

引数、戻り値においても ポリモーフィズムは利用できる。

引数や戻り値の場合も、たとえば、宣言がAnimal型になっていたとしても、Animalのサブクラス（Dogなど）のオブジェクトリファレンスを指定するということができます。以下に例を示します。




```
class Vet {
  public void giveShot(Animal a) {
    //a というパラメータには、Animal クラスのオブジェクトだけでなく
    // サブクラスのオブジェクトへの参照が指定できる
    a.makeNoise();
  }
}
```

パラメータ「a」には、Animalクラスだけではなく、Animalクラスを継承するすべてのクラスのオブジェクトリファレンスが指定できる。giveShot()を呼び出すと、その中でmakeNoise()メソッドが呼び出されるが、パラメータ「a」に指定されるリファレンスによって、どのクラスのmakeNoise()になるかが変わる。

```
class PetOwner {
  public void start() {
    Vet v = new Vet();
    Dog d = new Dog();
    Hippo h = new Hippo();
    v.giveShot(d);
    v.giveShot(h);
  }
}
```

giveShot()メソッドの引数には、Animalクラスのサブクラスのオブジェクトリファレンスが指定できる

Dogクラスのmake Noise()が実行される

Hippoクラスのmake Noise()が実行される

This PDF is prepared for tools@maruhouse.org, personal use only.

プログラムにサブクラスが新たに追加されてもコードを修正する必要はない

前ページのVetクラスでは、すでに述べたとおり、giveShot()メソッドの引数に、Animalクラスだけでなく、そのサブクラスのオブジェクトリファレンスが指定できます。これは、オブジェクトのクラスがAnimalクラスを継承するものであることさえ確認できれば、後は何も考えずにVetクラスを使用できるということです。Animalクラスを継承するクラスのオブジェクトを使用する限り、Vetクラスは問題なく機能します。また、Vetクラスを書く人も、Animalクラスのサブクラスが具体的にどのようなものになるかをあらかじめ予想する必要はありません。

頭の体操

なぜ、引数に、宣言で指定されたクラスでなく、そのサブクラスのオブジェクトリファレンスを指定しても問題が起きないのでしょうか。サブクラスにはスーパークラスと同じメソッドが必ずあると考えていいのでしょうか。

This PDF is prepared for tools@maruhouse.org, personal use only.

素朴な疑問に答えます

Q：継承の「レベル」に限界はあるのですか。サブクラスのサブクラス、そのサブクラス、というようにずっと継承を続けてもよいものなのでしょうか。

A：Java APIを見てみると、確かにあまり「低い」レベルまで継承をしている例は少ないようです。同じレベルに多数のサブクラスが存在する、ということは多いようですが。仮にAというクラスを基準にしたとすれば、クラスAのサブクラス、あるいはさらにそのサブクラス……という程度のレベルでとどめている場合がほとんどでしょう。もちろん例外はあります（特にGUI関連のクラスに例外が多い）。このように、継承は「浅い」レベルでとどめておく方が無難ですが、特に制限が設けられているわけではありません（少なくとも著者は知りません）。

Q：ソースコードは見ることができないが、メソッドの機能を修正したい、というクラスがある場合は、そのクラスを継承すればいいのですか。できの悪いクラスを継承し、メソッドをオーバーライドして改良するということは可能なのでしょうか。

A：可能です。むしろ、それがオブジェクト指向のメリットの1つだと言ってもいいでしょう。できの悪いクラスがあったとしても、それをゼロから書き直さずに済むのです。また、他人の作ったクラスを修正する場合、作者に連絡をとってソースコードを入手する、ということは必要ありません。

Q：継承できないクラスというのはないのですか。クラスのメンバの中にはアクセスレベルを「private」にされ、継承できないものもあります。クラスそのものの場合はどうなのですか。

A：たとえば「privateクラス」というようなものはありません。それに近いもので「内部クラス」と呼ばれるものはありますが（詳しくは後述）。ただ、クラスを継承できないようにするには、以下のような3つの方法が使えます。

まず1つは、アクセスレベルの設定です。クラス自身のアクセスレベルを「private」にする、ということはできないのですが、「public」にしない、ということはできます。「public」と指定されていないクラスを継承できるのは、同じパッケージに属するクラスのみです。属するパッケージが異なれば、継承できない（継承だけでなく利用もできない）のです。

「final」という修飾子を使う方法もあります。finalを指定すると、「このクラスはこれ以上継承できない」という意味になるのです。finalの指定されたクラスのサブクラスは誰も作ることができません。

その他、クラスのコンストラクタ（詳しくは9章参照）にすべて「private」を指定する、という方法もあります。この方法でもクラスは継承できなくなります。

Q：なぜfinal修飾子などを使って継承できないクラスを作るのですか。何かメリットがあるのでしょうか。

A：わざわざfinal修飾子を使ってクラスを継承できなくすることは通常ありません。ただ、セキュリティ上の理由からそうしなければならないこともあります。クラスが継承できると、悪意の人間がオーバーライドによって勝手にメソッドの機能を変えてしまうということもできるからです。Java APIのクラスの多くにfinalが指定されているのは同様の理由からです。

たとえば、Stringクラスにはfinalが指定されていますが、その理由は、Stringクラスの機能に知らない間に変更が加えられていたらどのようなことになるか、を想像すればわかるはずです。

Q：クラス全体ではなく、特定のメソッドだけにfinalを指定することはできますか。

A：できます。それによって特定のメソッドのオーバーライドを防止できます。クラス全体にfinalを指定した場合は、そのクラスのすべてのメソッドがオーバーライドできなくなります。

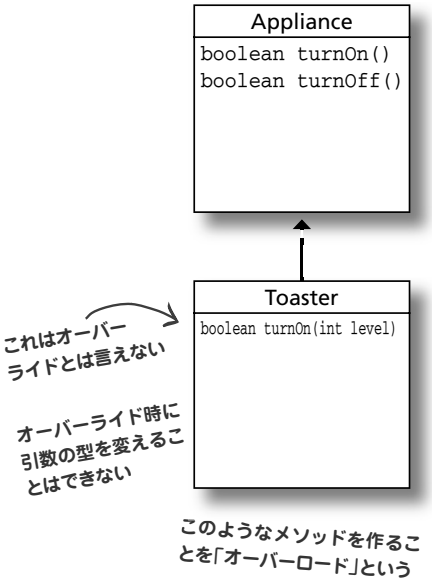
This PDF is prepared for tools@maruhouse.org, personal use only.

オーバーライドに関するルール

メソッドをオーバーライドする際には守らねばならないルールがあります。それは引数と戻り値の型を、元のメソッドと同じにしなければならないということです。言い換えれば、メソッドの外の世界との関わり方を同じにする必要があるということにもなります。

メソッドはたとえオーバーライドしても、外からの使い方が変わってはいけないのです。

たとえば、Applianceというクラスがあり、それを継承してToasterというクラスを作るとします。Applianceオブジェクトへの参照が格納されるはずの変数にToasterオブジェクトへの参照が格納され、その変数を利用してToasterクラスのメソッドが呼び出されていた場合、コンパイラは同じ名前のメソッドがApplianceにあるかどうかだけを確認します。同じ名前のメソッドがあれば、コードは正しいものとみなされるのです。しかし、プログラム実行時には、JVMによって、Toasterクラスのメソッドと、Applianceクラスの同名メソッドの引数と戻り値の型が同じであるかどうかが確認されるのです。右の図のように、Applianceクラスのturn On()メソッドに引数がなく、Toasterクラスのturn On()メソッドにはint型の引数がある、というような場合は、結局、Applianceクラスのturn On()メソッドが呼び出されることになります。これは、事実上、turn On()メソッドがオーバーライドされていない、ということを意味します。

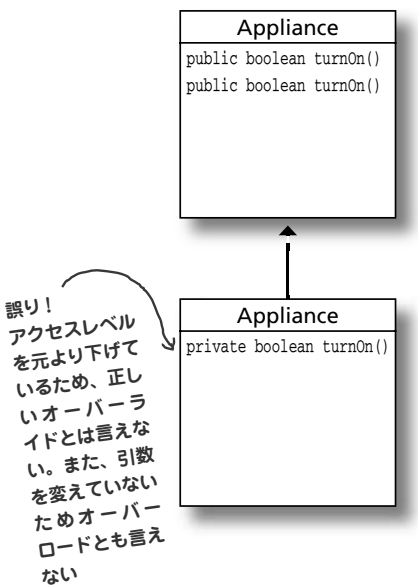


① **メソッドをオーバーライドしても引数の型は変えず戻り値の型は元と同じか互換性のあるものにする。**

スーパークラスのメソッドをオーバーライドする際、サブクラスのメソッドでは、すべての引数の型を元と同じにし、戻り値の型は元と同じにするか、そのサブクラスにします。これによって、外部からのメソッドの利用の仕方を変えずに済みます。サブクラスのオブジェクトには、必ず、スーパークラスのオブジェクトと同じ機能があるため、スーパークラスのオブジェクトの代わりに戻しても問題は起きません。

② **オーバーライド時にアクセスレベルを下げない。**

つまり、オーバーライドして作るメソッドのアクセスレベルは、元と同じか、元よりも高くするということです。たとえば、publicが指定されていたメソッドにprivateを指定する、というようなことはできません。このようなアクセスレベルの矛盾がプログラムにあっても、コンパイル時には発見されないので、実行時に突然問題が発覚して困ることになります。



これまでに解説したのは、privateとpublicという2つのアクセスレベルだけです。残り2つのアクセスレベルについては17章と付録Bを参照してください。例外処理のコードを書く際にもオーバーライドに関するルールに注意する必要がありますが、その点については11章「危険をともなうプログラムの作成」で詳述します。

This PDF is prepared for tools@maruhouse.org, personal use only.

メソッドのオーバーロード

「メソッドのオーバーロード」とは、「名前が同じで指定できる引数の異なるメソッドを作る」というだけの意味です。オーバーロードには、ポリモーフィズムは関係ありません。

オーバーロードは、メソッドを呼び出す人の利便性を上げるため、同じメソッドの引数の違う複数のバージョンを作ること、と考えることもできるでしょう。たとえば、メソッドの引数がint型のもの1つだけであれば、double型の値を指定したい場合には型の変換が必要になります。しかし、オーバーロードにより、double型の引数を指定できるメソッドが作ってあれば、呼び出しが容易になるでしょう。このことは、後の章で触れる「コンストラクタ」という特殊なメソッドにおいて特に重要になります。

オーバーロードは、すでに述べたとおりポリモーフィズムとは関係なく、オーバーライドに比べ、行う際のルールもゆるやかです。

オーバーロードによって作られたメソッドは、元のメソッドと同じでも、まったく別のメソッドと考えるべき。継承、ポリモーフィズムとは関係がない。オーバーロードとオーバーライドを混同しないこと。

① オーバーロードでは、戻り値の型も変えることができる。

オーバーロードの場合は、戻り値も変えることができます。ただし、次に触れるように、引数を変えずに戻り値だけを変えるということはできません。

② 戻り値だけを変えるということはできない。

戻り値の型を変えただけでは正しいオーバーロードとは言えません。戻り値の型だけが変わっている場合、コンパイラは「オーバーライドをしようとしている」と判断します。ただし、戻り値が元の戻り値のサブクラスでなければ、オーバーライドとしても不適切になります。メソッドをオーバーロードするには、必ず、引数に変更を加えなくてはなりません。そうすれば、戻り値の型はどのようにでも変えることができます。

③ アクセスレベルは上げることも下げることもできる。

オーバーロードの場合、メソッドのアクセスレベルは元より上げることも下げることもできます。元のメソッドよりアクセス制限の厳しいメソッドを作っても問題ないのです。

正しいオーバーロードの例

```
public class Overloads {

  String uniqueID;

  public int addNums (int a, int b) {
    return a + b;
  }

  public double addNums (double a, double b) {
    return a + b;
  }

  public void setUniqueID (String theID) {
    //ID確認のためのコードが入る
    uniqueID = theID;
  }

  public void setUniqueID (int ssNumber) {
    String numString = "" + ssNumber;
    setUniqueID (numString);
  }
}
```


This PDF is prepared for tools@maruhouse.org, personal use only.

エクササイズ

プログラムパズル

```
a = 6;        56
b = 5;        11
a = 5;        65
```

右のプログラムは一部分が欠落しています。欠落部分に入るコードの候補がいくつか示してありますが、それぞれのコードを当てはめた場合にどのような実行結果が得られるかを考え、コードと実行結果を線で結んでください。実行結果の中には、どのコードとも結び付かないものがあります。また、複数のコードと結び付く実行結果もあります。

プログラム

```
class A {
  int ivar = 7;
  void m1 () {
    System.out.print ("A's m1, ");
  }
  void m2 () {
    System.out.print ("A's m2, ");
  }
  void m3 () {
    System.out.print ("A's m3, ");
  }
}

class B extends A {
  void m1 () {
    System.out.print ("B's m1, ");
  }
}

class C extends B {
  void m3 () {
    System.out.print ("C's m3, "+ (ivar + 6));
  }
}

public class Mixed2 {
  public static void main (String [] args) {
    A a = new A ();
    B b = new B ();
    C c = new C ();
    A a2 = new C ();

    [                    ]
  }
}
```

この部分の欠落を埋める（3行）

コードの候補

```
b.m1 ();
c.m2 ();
a.m3 ();
```

```
c.m1 ();
c.m2 ();
c.m3 ();
```

```
a.m1 ();
b.m2 ();
c.m3 ();
```

```
a2.m1 ();
a2.m2 ();
a2.m3 ();
```

実行結果

```
A's m1, A's m2, C's m3, 6
B's m1, A's m2, A's m3,
A's m1, B's m2, A's m3,
B's m1, A's m2, C's m3, 13
B's m1, C's m2, A's m3,
B's m1, A's m2, C's m3, 6
A's m1, A's m2, C's m3, 13
```


This PDF is prepared for tools@maruhouse.org , personal use only.

エクササイズ

コンパイラになってみよう

左のソースコードを、コンパイラになったつもりで見てみましょう。そして、右側の1～4に示したコードのうち、A、Bの空白部分（AはMonsterクラス、BはVampireクラスに属する）に当てはめた時に、コンパイルが正しく行え、なおかつ下に示した実行結果が得られるものはどれかを答えてください。

```
public class MonsterTestDrive {
  public static void main (String [] args) {
    Monster [] ma = new Monster [3];
    ma [0] = new Vampire ();
    ma [1] = new Dragon ();
    ma [2] = new Monster ();
    for (int x = 0; x < 3; x++) {
      ma [x].frighten(x);
    }
  }
}

class Monster {
A
}

class Vampire extends Monster {
B
}

class Dragon extends Monster {
  boolean frighten (int degree) {
    System.out.println ("breath fire");
    return true;
  }
}
```

```
File  Edit  Window  Help  SaveYourself
% java MonsterTestDrive
a bite?
breath fire
arrrgh
```

1

A
```
boolean frighten (int d) {
  System.out.println ("arrrgh");
  return true;
}
```

B
```
boolean frighten (int x) {
  System.out.println ("a bite?");
  return false;
}
```

2

A
```
boolean frighten (int x) {
  System.out.println ("arrrgh");
  return true;
}
```

B
```
  int frighten (int f) {
  System.out.println ("a bite?");
return 1;
}
```

3

A
```
boolean frighten (int x) {
  System.out.println ("arrrgh");
  return false;
}
```

B
```
boolean scare (int x) {
  System.out.println ("a bite?");
  return true;
}
```

4

A
```
boolean frighten (int z) {
  System.out.println ("arrrgh");
  return true;
}
```

B
```
boolean frighten (byte b) {
  System.out.println ("a bite?");
  return true;
}
```


This PDF is prepared for tools@maruhouse.org, personal use only.

プールパズル

右のプログラムの欠落部分に、下のプールの中のコードを当てはめ、実行結果が以下に示すようなものになるようなプログラムを完成させてください。すべてのコードを使う必要はありません。同じコードを 2 回以上繰り返し使うこともできます。意外に難しいので慎重に考えてください。

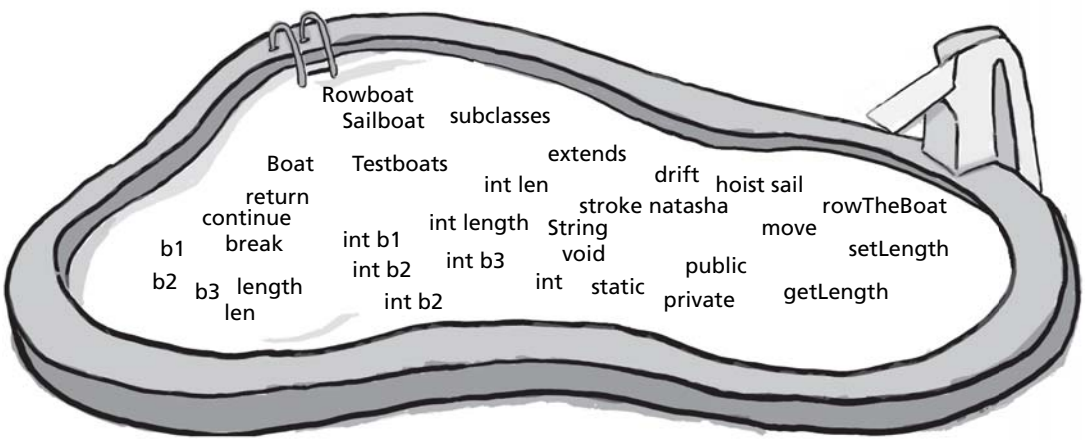
```
public class Rowboat ________________ {
  public ___________ rowTheBoat () {
    System.out.print ("stroke natasha");
  }
}
```

```
public class __________ {
  private int ___________ ;
  _______ void ___________ (_______) {
    length = len;
  }
  public int getLength () {
    _________ _________;
  }
  public __________ move () {
    System.out.print ("___________");
  }
}
```

```
public class TestBoats {
  _______ ______ ______ main (String [] args) {
    _________b1 = new Boat ();
    Sailboat b2 = new __________ ();
    Rowboat _______ = new Rowboat();
    b2.setLength (32);
    b1.__________ ();
    b3.__________ ();
    ________.move ();
  }
}
```

```
public class ___________________ Boat {
  public ________________ () {
    System.out.print ("___________");
  }
}
```

実行結果： `drift drift hoist sail`



This PDF is prepared for tools@maruhouse.org , personal use only.

エクササイズの答え

コンパイラになってみよう

コンパイルが正しく行え、なおかつ下に示した実行結果が得られるのは、1です。

2は、コンパイルができません。戻り値が不適切なためです。Vampireクラスのfrighten()メソッドは、Monsterクラスのfrighten()メソッドをオーバーライドしたものにも、オーバーロードしたものにもなっていません。戻り値の型を変えただけでは適切なオーバーロードとは言えません。また、intとbooleanには互換性がないため、適切なオーバーライドにもなっていません(適切なオーバーライドにするためには、戻り値の型を元と同じか、サブクラスなど元と互換性のある型にしなくてはならない)。

3、4はコンパイルはできますが、実行結果が、
arrrgh
breath fire
arrrgh
になってしまいます。

この場合、Vampireクラスのfrighten()メソッドはMonsterクラスのfrighten()メソッドをオーバーライドしたものにはなっていません(4のVampireクラスでは、frighten()メソッドの引数がint型ではなくbyte型になっている)。

コードの候補

```
b.m1 ();
c.m2 ();
a.m3 ();
```

プログラムパズル

```
c.m1 ();
c.m2 ();
c.m3 ();

a.m1 ();
b.m2 ();
c.m3 ();

a2.m1 ();
a2.m2 ();
a2.m3 ();
```

実行結果

```
A's m1, A's m2, C's m3, 6
B's m1, A's m2, A's m3,
A's m1, B's m2, A's m3,
B's m1, A's m2, C's m3, 13
B's m1, C's m2, A's m3,
B's m1, A's m2, C's m3, 6
A's m1, A's m2, C's m3, 13
```


This PDF is prepared for tools@maruhouse.org, personal use only.

プールパズル

```
public class Rowboat extends Boat {
  public void rowTheBoat () {
    System.out.print ("stroke natasha");
  }
}
public class Boat {
  private int length;
  public void setLength (int len) {
    length = len;
  }
  public int getLength () {
    return length;
  }
  public void move () {
    System.out.print ("drift");
  }
}
```

```
public class TestBoats {
  public static void main (String [] args) {
    Boat b1 = new Boat ();
    Sailboat b2 = new Sailboat ();
    Rowboat b3 = new Rowboat();
    b2.setLength (32);
    b1.move ();
    b3.move ();
    b2 .move ();
  }
}
public class Sailboat extends Boat {
  public void move () {
    System.out.print ("hoist sail");
  }
}
```

実行結果

```
drift drift hoist sail
```


This PDF is prepared for tools@maruhouse.org, personal use only.

8章　インターフェースと抽象クラス

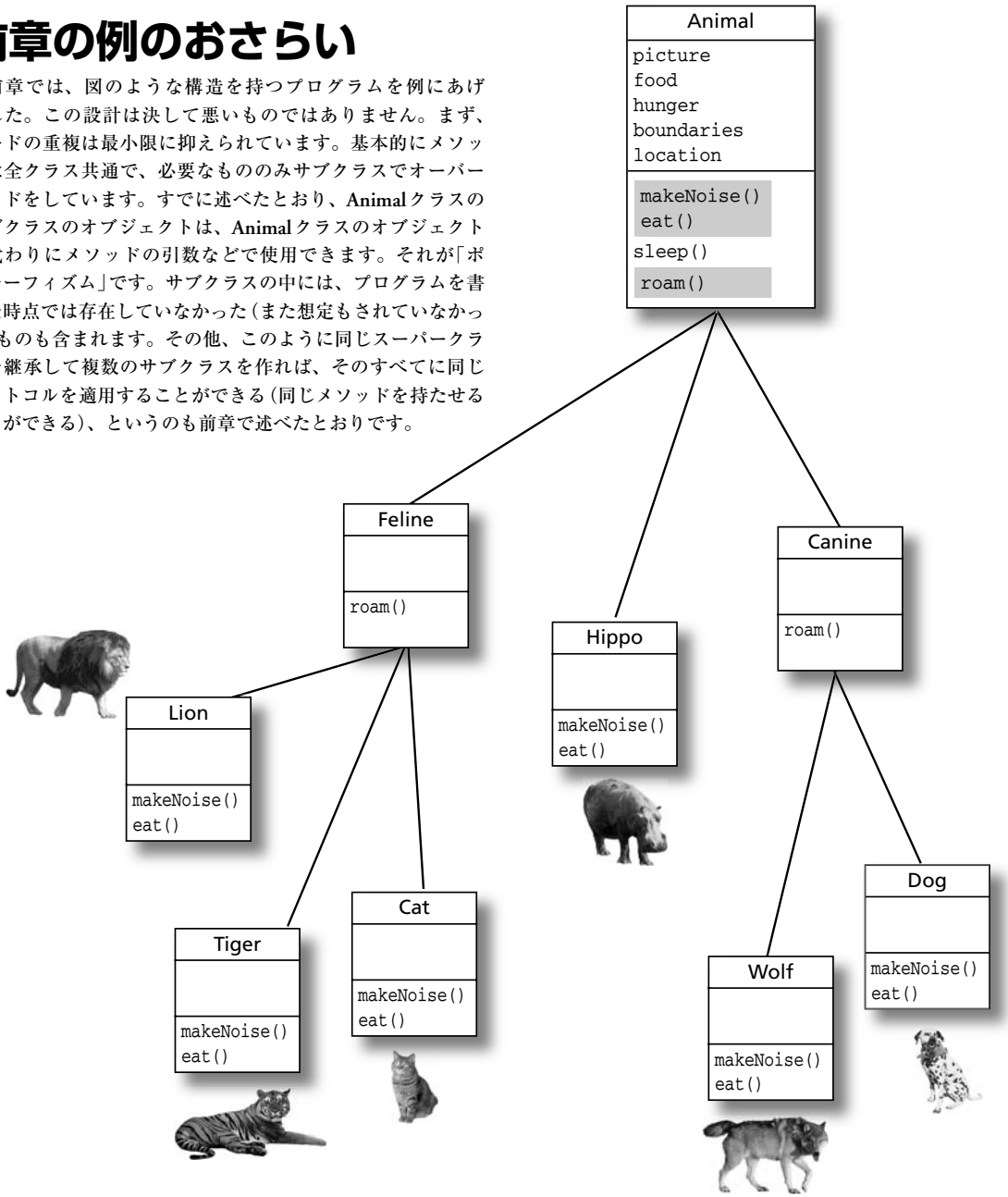
ポリモーフィズムの高度な利用方法

前の章では、継承とポリモーフィズムについて解説しました。次は、その継承やポリモーフィズムをより有効に利用するために必要な、「インターフェース」というものについて解説しましょう。注意すべきなのは、この「インターフェース」が、「ユーザーインターフェース」などとは無関係のものであるということです。インターフェースをうまく利用すれば、単にクラスを継承しただけの場合より、さらに柔軟性、拡張性の高いプログラムが書けます。インターフェースを使わなくてもプログラムは作れるのですが、インターフェースを使わなければJavaを使うメリットは半減すると言っても過言ではありません。また、大切なのは、インターフェースを使う場合には、設計の段階からそのことを念頭に置いておくことです。設計でインターフェースについて考えず、コーディングの段階になって突然インターフェースを使うということでは困るのです。ともかく、一度インターフェースについて知れば、もうインターフェースなしでプログラミングをしようとは考えられなくなるはずです。では、インターフェースとはいったい何なのでしょうか。インターフェースは、「抽象クラス」と呼ばれる種類のクラスに似ています。抽象クラスというのは、後で詳しく説明しますが、インスタンス化ができないクラスのことです。インターフェース、抽象クラスについて理解すれば、前章で紹介したポリモーフィズムの利用例が、ごく一面的なものにしか過ぎなかったということがわかるでしょう。インターフェースを知ってはじめてポリモーフィズムを十分に活かせるということが言えます。

This PDF is prepared for tools@maruhouse.org, personal use only.

前章の例のおさらい

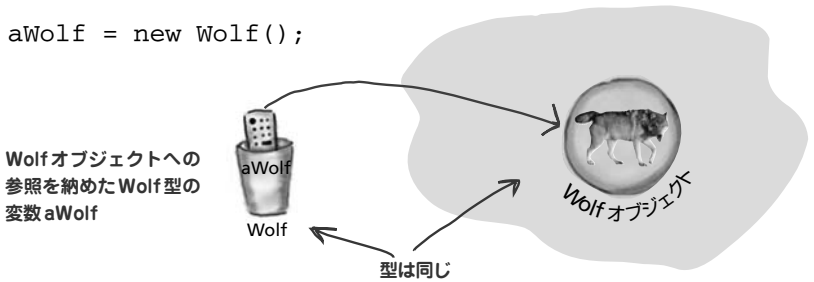
前章では、図のような構造を持つプログラムを例にあげました。この設計は決して悪いものではありません。まず、コードの重複は最小限に抑えられています。基本的にメソッドは全クラス共通で、必要なもののみサブクラスでオーバーライドをしています。すでに述べたとおり、**Animal**クラスのサブクラスのオブジェクトは、**Animal**クラスのオブジェクトの代わりにメソッドの引数などで使用できます。それが「ポリモーフィズム」です。サブクラスの中には、プログラムを書いた時点では存在していなかった(また想定もされていなかった)ものも含まれます。その他、このように同じスーパークラスを継承して複数のサブクラスを作れば、そのすべてに同じプロトコルを適用することができる(同じメソッドを持たせることができる)、というのも前章で述べたとおりです。



This PDF is prepared for tools@maruhouse.org, personal use only.

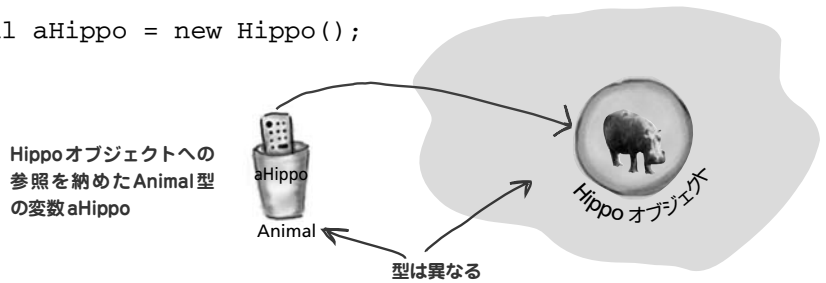
これはOK

```
Wolf aWolf = new Wolf();
```



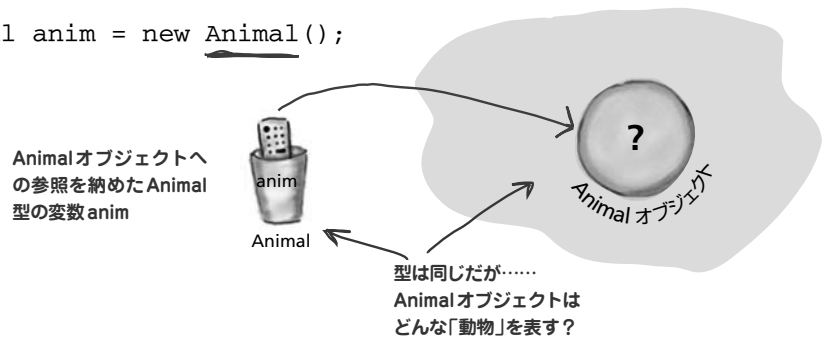
これはOK

```
Animal aHippo = new Hippo();
```



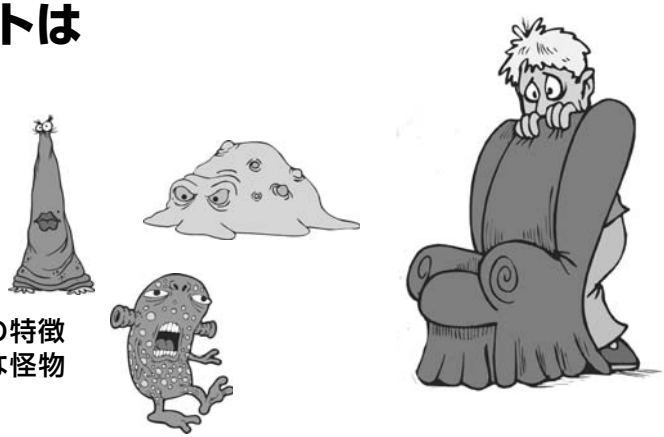
これは？

```
Animal anim = new Animal();
```



This PDF is prepared for tools@maruhouse.org, personal use only.

Animalオブジェクトはどんな動物を表す？



あらゆる動物の特徴を備えたこんな怪物かも……

Animal オブジェクトのインスタンス変数の値は何を意味する？

クラスの中にはインスタンス化すべきでないものもある

Wolfオブジェクト、Tigerオブジェクトなどは、具体的に対応するものが明確なので、作る意味があります。しかし、Animalオブジェクトはどうでしょうか。このオブジェクトに対応する「動物」はどのような形をしていて、どのような色、大きさで、足は何本あるのでしょう……。

Animalクラスのオブジェクトが作られるというのは、スタートレックに出てくる「トランスポーター」の事故のようなものです。トランスポーターが物体をバラバラにして転送し、再度組み立て直す際にバッファ内で問題が起き、ワケのわからないものができてしまうという……。

Animalのようなクラスは確かに必要なのですが、通常のクラスのようにそれ自体をインスタンス化すべきではないのです。この種のクラスは、あくまで継承するため、ポリモーフィズムを利用するために使用します。インスタンス化するのは、Animalクラスを継承した、より抽象性の低い（具体的なものを表す）クラスです。TigerオブジェクトやLionオブジェクトは作っても、決してAnimalオブジェクトは作らないのです。

インスタンス化をすべきではなくても、Animalのようなクラスは無理にインスタンス化をしようと思えばできてしまうので、その点が問題です。ただ、幸いなことに、クラスをインスタンス化できないようにする方法もあります。その方法とは、クラスのコードに「abstract」という宣言を入れる、というものです。この宣言があると、そのクラスはインスタンス化ができなくなります。

「abstract」という宣言をされたクラスを「抽象クラス」と呼びます。抽象クラスはインスタンス化はできませんが、変数、引数、戻り値、配列などの型を抽象クラスにすることはできます。この点が、ポリモーフィズムを有効に活かす上で非常に重要になるのです。

プログラムの設計を行う際には、どのクラスを抽象クラスにするか、その抽象クラスを継承してどのようなクラスを作るか、を正しく判断する必要があります。抽象クラスにしないクラス（この種のクラスを「具象クラス」と呼ぶ）は、そのクラスをインスタンス化するのが適切かどうかをよく考えなくてはなりません。

抽象クラスを作るには、以下のように、「abstract」というキーワードをクラスのコードの中に入れます。

```
abstract class Canine extends Animal {
  public void roam () { }
}
```


This PDF is prepared for tools@maruhouse.org, personal use only.

抽象クラスのインスタンスは作成できない

抽象クラスは、誰であろうとそのインスタンスが作れないクラスです。ポリモーフィズムを利用する目的で変数などの型として使用することはできますが、インスタンスは決して作れないのです。もし、インスタンス作成のコードを書くと、コンパイルエラーが発生します。

```
abstract public class Canine extends Animal
{
  public void roam () { }
}
```

```
public class MakeCanine {
  public void go () {
    Canine c;
    c = new Dog ();
    c = new Canine ();
    c.roam ();
  }
}
```

変数の型をCanineにすることは可能。ただし、代入するのは、常にサブクラスのオブジェクトリファレンス

Canineは抽象クラスなので、このようなコードを書くとコンパイルエラーが発生する

```
File  Edit  Window  Help  BeamMeUp
% javac MakeCanine.java

MakeCanine.java:5: Canine is abstract;
cannot be instantiated
      c = new Canine();
          ^
1 error
```

抽象クラスは、それ自体を利用することはなく、単独ではまったく価値のないものです*。ただ継承するためだけのものです。

たとえ抽象クラスが存在しても、利用するのは常にそのサブクラスのインスタンスということになります。

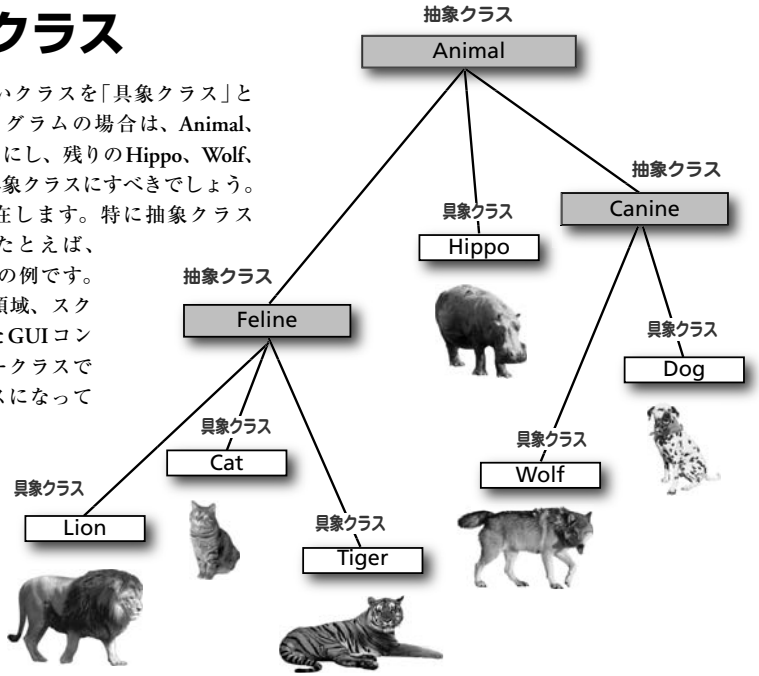
* 例外もあります。抽象クラスにもスタティックメンバを持たせることができるからです。

This PDF is prepared for tools@maruhouse.org, personal use only.

抽象クラスと具象クラス

すでに述べたとおり、抽象クラスでないクラスを「具象クラス」と呼びます。ここで例に使用しているプログラムの場合は、Animal、Canine、Felineといったクラスを抽象クラスにし、残りのHippo、Wolf、Dog、Tiger、Lion、Catといったクラスを具象クラスにすべきでしょう。

抽象クラスは、Java APIにも数多く存在します。特に抽象クラスが多いのが、GUIのライブラリです。たとえば、Componentクラスはそうした抽象クラスの例です。Componentクラスは、ボタン、テキスト領域、スクロールバー、ダイアログボックスといったGUIコンポーネントを表す多数のクラスのスーパークラスですが、その点を考えると、なぜ抽象クラスになっているのかがわかるでしょう。漠然とあらゆるGUIコンポーネントに対応するクラスのインスタンスを作っても意味がないからです。やはり作るとすれば、ボタンなどの具体的なコンポーネントに対応するサブクラスのインスタンスということになるでしょう。



頭の体操

抽象的？　具体的？

クラスを抽象クラスにすべきかどうかはどこで判断すればいいのでしょうか。たとえば、漠然と「ワイン」全般に対応するクラスは抽象クラスでしょう。では、「赤ワイン」、「白ワイン」に対応するクラスはどうでしょうか。人によって意見は分かれるかもしれませんが、これもやはり抽象クラスにすべきです。

「ピノノワール」に対応するクラスとなると、「具象クラスでもいいんじゃないか」という人も増えるはずです。さらに「キャメロット・ピノノワールの1997年もの」に対応するクラスとなると、ますますそういう人は増えるでしょう。しかし、それでも全員が賛成するとは限りません。プログラムの目的などによっても変わるでしょう。

上記のAnimalクラスの例の場合、抽象クラスと具象クラスの使い分けは絶対にこれでいいと言えるでしょうか。他の可能性はないか考えてみてください（たとえば、Catクラスにさらにサブクラスを追加する場合にはどうなるでしょうか）。

This PDF is prepared for tools@maruhouse.org, personal use only.

抽象メソッド

クラスの他、メソッドを「abstract」宣言することもできます。つまり、「抽象クラス」ではなく「抽象メソッド」を作ることもできるということです。抽象クラスは必ず継承する必要がありましたが、抽象メソッドは必ずオーバーライドしなければなりません。抽象メソッドを作るのは、「このメソッドの機能は、より具体性の高い下位のクラスに持たせるべき」と判断した場合です。メソッドによっては、クラスが抽象的すぎると、機能が明確にならず、作りようがない（作ってもあまり意味がない）、ということがあるのです。そのことは、たとえば、動物全般に対応するAnimalクラスのeat()メソッドの機能がどのようになるかを少し考えてみればわかるでしょう。

抽象メソッドには「本体」がない

抽象メソッドには、以下に示すとおり、「本体」となるコードがありません。先述のとおり、コードを「作りようがない（作ってもあまり意味がない）」からです。抽象メソッドには、{}も使いません。宣言を書いたらすぐにセミコロン（;）を書きます。

```
public abstract void eat ();
```

「本体」のコードはない
最後にはセミコロンを付ける

メソッドを１つでも抽象メソッドにしたクラスは、必ず抽象クラスにしなければならない。クラスが抽象クラスでないのに、メソッドだけを抽象メソッドにすることはできない。

抽象メソッドを１つでも含むクラスは、必ず抽象クラスにします。ただ、抽象クラスの中に、抽象メソッドと、通常のメソッドを混在させることは可能です。

素朴な疑問に答えます

Q：抽象メソッドはなぜ必要なのですか。クラスを抽象クラスにして、そのメソッドをサブクラスに継承するようにすれば済む話なのではないですか。

A：もちろん、そうする意味がある場合には、抽象クラスに「本体」のコードがある通常のメソッドを持たせ、それをサブクラスに継承するという方法で何も問題はありません。ただ、抽象クラスの場合、メソッドによっては、時折、作っても意味がない（作りようがない）ということが起きるのです。あらゆるサブクラスにおいて妥当であるようなコードというのは作るのが難しいことが多いからです。抽象メソッドはそういう場合に使うと便利です。抽象メソッドには何のコードも入れる必要がありません。そして、なおかつ、「この抽象クラスのサブクラスには、こういうメソッド（機能）を持たせる」という枠組みを決めることができるのです。

Q：それがポリモーフィズムの利用に役立つということですか。

A：そのとおりです。ポリモーフィズムを利用する場合には、すべてのサブクラスに同じメソッドを持たせるということが重要なのです。そうすれば、既存のプログラムに何ら変更を加えることなく、新しいサブクラスを次々に追加していくことができます。もし、サブクラスを新たに追加するたびに、それに対応するコードを書かなければならないとしたらどうでしょうか。Lionクラスを作ったら、そのためのコード、Wolfクラスを作ったら、そのためのコード……という具合に改造するとしたら大変なことになります。抽象メソッドを使えば、何もコードを書くことなく、ともかく「サブクラスは皆このメソッドを持つ」ということが決められるので非常に簡単です。

This PDF is prepared for tools@maruhouse.org, personal use only.

抽象メソッドの本体は サブクラスで必ず作る

抽象メソッドの本体の作成は、メソッドのオーバーライドとほぼ同じ

抽象メソッドには本体のコードはないのですが、この本体は、必ず、直下のサブクラス（具象クラス）の中で作らねばなりません。

ただ、直下のサブクラスも抽象クラスである、という場合には、本体を作る必要はありません。ともかく、最上位のクラスに最も階層が近い具象クラスでは本体を作るようにします。Animalクラスの例で言えば、Canineクラスでは本体を作る必要はないが、その下のDogクラスでは本体を作らねばならないということです。

抽象クラスには、抽象メソッドと通常のメソッドを混在させることができるので、たとえば、Canineクラスで一部の抽象メソッドの本体を作り、残りをDogクラスで作るということも可能です。Canineクラスで本体を作った場合にはDogクラスで本体を作る必要はありません。Dogクラスで作るのは、Canineクラスで作らなかったものだけです。

抽象メソッドの本体を作る（これを「抽象メソッドをインプリメントする」ということもあります）際には、宣言、引数、戻り値の型などが元と同じになるよう注意します。本体自身はプログラマが自由に作ることができます。

This PDF is prepared for tools@maruhouse.org, personal use only.

抽象クラスと具象クラス

自分で考えてみよう

たとえ同じもの（こと）を表すクラスであっても、利用するプログラムの種類によって抽象クラスにすべき場合と具象クラスにすべき場合とがあります。以下の表の中央の列に示したクラスそれぞれについて、抽象クラスにすべきプログラム、具象クラスにすべきプログラムとしてどのようなものがあり得るかを考え、表の空欄を埋めてください。ヒントとして、表の一部の欄はすでに埋めてあります。たとえば、木を表す「Treeクラス」は、「樹木生育プログラム」では、木の種類が重要な意味を持つので抽象クラスにすべきでしょう。しかし、ゴルフのシミュレーションプログラムの場合、木はおそらく単なる障害物の一種とみなされ、木の種類は重要な意味を持たないので具象クラスにできるでしょう（唯一の正解はありません。デザインによっても変わってきます）。

具象クラスにできるプログラムの例	クラス	抽象クラスにすべきプログラムの例
ゴルフコースシミュレーション	Tree	樹木生育プログラム
	House	建築プログラム
衛星写真プログラム	Town	
	FootballPlayer	コーチングプログラム
	Chair	
	SalesOrder	
	Book	
	Store	
	Supplier	
	GolfClub	
	Carburetor（キャブレター）	
	Oven	

This PDF is prepared for tools@maruhouse.org, personal use only.

ポリモーフィズムの応用例

まず、複数のDogオブジェクト（への参照）を格納できるリストクラスについて考えてみましょう。今のところは、ArrayListクラスについては何も知らないと仮定して話を進めます。取りあえずメソッドは、Dogオブジェクトをリストに追加するためのadd()だけということにします。この場合は、オブジェクトリファレンスの格納のためにDog型の配列（Dog []）を使うということができます。サイズは仮に「5」ということにしておきましょう。配列に5つのDogオブジェクトへの参照が格納された後も、add()メソッドを呼び出すことはできますが、その際には結局何の処理も行われません。通常は、add()メソッドによって、配列中の適切な位置にDogオブジェクトへの参照が追加され、インデックス（nextIndex変数の値）がインクリメントされます。

Dogオブジェクト専用のリストクラス

（この後、これをArrayListクラスに近いものに改良していく）

```
public class MyDogList {
  private Dog [] dogs = new Dog [5];
  private int nextIndex = 0;

  public void add (Dog d) {
    if (nextIndex < dogs.length) {
      dogs [nextIndex] = d;
      System.out.println ("Dog added at " + nextIndex);
      nextIndex++;
    }
  }
}
```

簡単なDog型の配列を中で使用する

Dogオブジェクトへの参照が追加されるたびにインクリメントする

格納しているDogオブジェクトリファレンスの数が「5」に達していなければ、Dogオブジェクトリファレンスを新たに追加し、メッセージを出力する

オブジェクトリファレンスの格納位置を変えるため、インクリメントする

This PDF is prepared for tools@maruhouse.org, personal use only.

クラスの異なるオブジェクトを混在させる

仮に、Dogオブジェクトに加え、Catオブジェクト（への参照）も格納しなければならないとすると、そのために採りうる方法としては次のようなものが考えられます。

1）MyCatListという、もう1つのリストクラスを作る。ただ、この方法は非効率。
2）DogAndCatListというリストクラスを作り、2つの配列（インスタンス変数）を使ってDogオブジェクトとCatオブジェクトを同時に格納できるようにする。リストへのオブジェクトの追加には、addCat(Cat c)、addDog(Dog d)というメソッドを使用する。これもあまり良い方法とは言えない。
3）Animalクラスのあらゆるサブクラスのオブジェクトを格納できるAnimalListというリストクラスを作成する。3つ目の方法が改造にも柔軟に対応できるので最も優れていると言えます。この方法なら、後でDogオブジェクト、Catオブジェクトだけでなく、さらに他のクラスのオブジェクトを格納する必要が出てきたとしても、問題は起きません。では、先のMyDogListクラスを改良して実際にAnimalListクラスを作ってみましょう。AnimalListクラスのコードは以下に示しましたが、MyDogListクラスからの主な変更箇所には線をひいてあります（ロジックに変更はないが、配列や変数の型がDogからAnimalに変わっている点に注意）。

AnimalListクラスのコード

バージョン 2

```
public class MyAnimalList {

  private Animal [] animals = new Animal [5];
  private int nextIndex = 0;

  public void add (Animal a) {
    if (nextIndex < animals.length) {
      animals [nextIndex] = a;
      System.out.println ("Animal added at " + nextIndex);
      nextIndex++;

    }
  }
}
```

混乱しやすいが、これはAnimalオブジェクトではなく、Animal型の配列を作るためのコード（抽象クラスのインスタンスは作成できないのだが、この種の配列は宣言できるという点に注意）

MyAnimalList
Animal [] animals int nextIndex
add (Animal a)

```
public class AnimalTestDrive {
  public static void main (String[] args) {
    MyAnimalList list = new MyAnimalList();
    Dog a = new Dog();
    Cat c = new Cat();
    list.add(a);
    list.add(c);
  }
}
```

```
File  Edit  Window  Help  Harm
% java AnimalTestDrive
Animal added at 0
Animal added at 1
```


This PDF is prepared for tools@maruhouse.org, personal use only.

Animalのサブクラス以外のオブジェクトを格納するにはどうするか

ArrayListと同等の機能を持たせるためには、Animalのサブクラスに対応するだけでは不十分です。配列にはあらゆるクラスのオブジェクトを格納できるようにし、add()メソッドの引数にもあらゆるクラスのオブジェクトを指定できるようにする必要があるのです。つまり、配列や引数の型を、あらゆるクラスの上に位置する（どのクラスよりも抽象的な）「究極のスーパークラス」とでも呼ぶべきものにしなければならないのです。そのようなクラスはあるのでしょうか。たとえば、Animalクラスにはスーパークラスはないはずですが……。

しかし、実際にはスーパークラスは存在するのです。

ここで、ArrayListクラスのメソッドについて思い出してみてください。add()、contains()、indexOf()といったメソッドの引数はどうなっていたでしょうか。型は「Object」となっていたはずです。

ここで紹介しているArrayListのメソッドは一部。メソッドは他にも多数存在する

ページ 3

ArrayList

boolean remove(Object elem)
引数に指定されたオブジェクトを削除する。指定のオブジェクトがリスト中に存在すれば"true"を戻す。

boolean contains(Object elem)
引数に指定されたオブジェクトと同じものがリストにあれば"true"を戻す。

boolean isEmpty()
リストの要素がなければ"true"を戻す。

int indexOf(Object elem)
引数に指定されたオブジェクトのリスト中での番号（インデックス）、あるいは-1を戻す。

Object get(int index)
引数に指定された番号（インデックス）に対応するオブジェクトを戻す。

boolean add(Object elem)
引数に指定されたオブジェクトをリストに追加する（さらに"true"を戻す）。

// メソッドは以上ですべてではない

ArrayListのメソッドの多くでは究極のスーパークラスであるObjectを使用する。Javaのあらゆるクラスは Objectのサブクラスなので、ArrayListのメソッドの引数にはあらゆるオブジェクトへの参照が指定できる（注意：Java 5.0のget()メソッド、add()メソッドは正確には上記と少し違うが、今のところは「だいたいこういうもの」と思っておいて良い。詳しくは後述）

Javaのクラスはすべて Objectクラスを継承したもの。

このObjectクラスが、究極のスーパークラスです。

ポリモーフィズムを活用しようとすれば、メソッドの引数、戻り値の型をそれに適したものにする必要があります。Javaには、あらゆるクラスの上に位置する究極のスーパークラスが存在します。そのおかげで、ArrayListクラスは、プログラマが独自に作るクラス（型）にも対応できます。プログラマがどのようなクラスを作るかをあらかじめ予測することは不可能なので、もし究極のスーパークラスがなかったとしたら、対応はできないでしょう。

厳密に言えば、Javaで作るクラスは、作った本人がそう思っていなくてもすべて「サブクラス」なのだ、ということになります。そのことをコードの中に明記する必要はありませんが、すべてはObjectのサブクラスなのです。たとえば、Dogクラスのコードは以下のように書いてもいいのでは、と思う人もいるでしょう。

```
public class Dog extends Object { }
```

ただ、思い出して欲しいのは、DogはCanineのサブクラスだ、ということです。また、CanineはAnimalのサブクラスです。AnimalはObjectクラスの直接のサブクラスなのですが、そのことを明記する必要はありません。このことを簡単にまとめると次のようになるでしょう。

明示的に何かのクラスを継承していないクラスは、自動的にObjectを直接継承しているとみなされる。

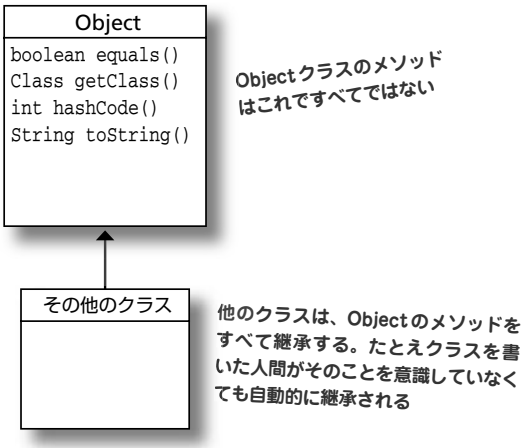
DogはCanineを継承していて、Objectは直接継承しているわけではありません（間接的には継承している）。同様のことはCanineにも言えます。しかし、AnimalはObjectを直接継承しています。

This PDF is prepared for tools@maruhouse.org, personal use only.

Objectクラスのメソッド

Objectはすべてのクラスのスーパークラスなのですが、このクラスにはいったいどのようなメソッドがあるのでしょうか。Objectクラスのメソッドとしてまずあげられるのが、2つのオブジェクトを比較して両者が同じものであるかどうかを確かめるメソッドです。また、あるオブジェクトがどのクラスのものであるかを調べるメソッドもあります。その他、オブジェクトのハッシュコード（ハッシュコードの詳細については17章と付録Bを参照してください）を戻すメソッドや、オブジェクトのIDのような文字列を戻すメソッドなどもあります。

Objectクラスのメソッドは他にもありますが、取りあえずここでは、この4つのメソッドについて触れておきましょう。



① equals (Object o)

```
Dog a = new Dog();
Cat c = new Cat();

if (a.equals(c)) {
  System.out.println ("true");
} else {
  System.out.println ("false");
}
```

```
File  Edit  Window  Help  Stop
% java TestObject
false
```

2つのオブジェクトが同じものであるかどうかを確かめる（「同じ」の定義については付録Bを参照）

② getClass()

```
Cat c = new Cat();
System.out.println (c.getClass());
```

```
File  Edit  Window  Help  Faint
% java TestObject
class Cat
```

オブジェクトがどのクラスのものかが知らされる

③ hashCode()

```
Cat c = new Cat();
System.out.println (c.hashCode());
```

```
File  Edit  Window  Help  Drop
% java TestObject
8202111
```

オブジェクトのハッシュコード（詳しくは17章、付録Bを参照）が出力される

④ toString()

```
Cat c = new Cat();
System.out.println (c.toString());
```

```
File  Edit  Window  Help  LapseIntoComa
% java TestObject
Cat@7d277f
```

クラス名と奇妙な数字（この数字の意味については今のところ知らなくてよい）を組み合わせた文字列が出力される

This PDF is prepared for tools@maruhouse.org, personal use only.

素朴な疑問に答えます

Q：Objectは抽象クラスなのですか。

A：Javaの定義からすれば抽象クラスとは言えません。Objectクラスのメソッドには「本体」のコードがあるからです。Objectを継承するクラス（つまりすべてのクラス）では、オーバーライドすることなく、Objectのメソッドをそのまま利用できます。

Q：Objectのメソッドをオーバーライドするということもできますか。

A：できるものもありますが、「final」と宣言されているものはオーバーライドできません。中には、hashCode()、equals()、toString()のように、オーバーライドして使用するのが普通、というメソッドもあります（詳しくは後述）。しかし、getClass()のようにfinal宣言されたメソッドはオーバーライドせず常に同じ使い方をします。その中には、ネイティブコード（特定のプラットフォームに固有のコードのこと）を利用して機能するメソッドなどもあります。

Q：ArrayListがObjectに対応できるだけの汎用性を持っているのなら、なぜArrayList<DotCom>のような記述をするのですか。これだと、使えるオブジェクトの型をDotComに限定してしまうことになると思うのですが。

A：確かに限定をしていることになります。Java 5.0より前は、限定ができませんでした。Java 5.0以降の場合は、ArrayList<Object>のように書けば、過去のバージョンと同様の意味になります。この記述だと、正確には「対応するのをObjectクラスのオブジェクトに限定する」という意味になりますが、Objectクラスのオブジェクトというのは、Javaのあらゆるクラスのオブジェクトというのと同じ意味になります。新たに加わった<>という構文については後で詳述します。

Q：話を元に戻します。Objectクラスは抽象クラスではない（つまり具象クラスである）ということですが、Objectクラスのオブジェクトを作るというのは妙ではありませんか。Animalクラスのオブジェクトを作るのがおかしいと言うのであれば、Objectクラスのオブジェクトを作るのはもっとおかしいと思うのですが。

A：良い質問です。確かにObjectクラスのオブジェクトを作るというのは奇異な印象を受けます。それでもObjectクラスのオブジェクトが作れるようになっているのは、今の時点では「その必要があるから」としか言えません。15章で詳しく説明しますが、Objectオブジェクトの用途として最も一般的なのは「スレッドの同期」です。取りあえずのところは、「Objectオブジェクトは作ることはできるが、めったに使うことはない」という認識でもかまいません。

Q：Objectクラスは、ArrayListのように、主に、ポリモーフィズムの応用のために引数や戻り値の型として使われると考えていいのでしょうか。

A：Objectクラスの役割は大きく分けて2つあります。1つは、ご質問にもあるように、ポリモーフィズムの応用のため、メソッドの引数や戻り値の型となることです。これにより、引数や戻り値にはあらゆるクラスのオブジェクト（への参照）が使用できるようになります。また、Objectクラスには、多くのクラスに共通して必要になるメソッドを提供するという役割もあります（Objectクラスのメソッドはすべてのクラスが継承するため）。

Q：Objectクラスが便利なのはよくわかりましたが、それでなぜ、すべてのメソッドの引数、戻り値の型をObjectにする、ということをしないのですか。

A：それはあまり得策とは言えません。その理由としてはまず、Javaが「タイプセーフ」な言語である、ということがあげられます。「タイプセーフ」というのはJavaの特徴の中でも特に重要とされているものの一つで、「あるクラスのメソッドを、別のクラスのオブジェクトを使って呼び出すことはできない」ということを意味します。たとえば、Ferrariオブジェクトを使って、ToasterクラスのcookメソッドをObjectを呼び出す、といったことはできません。Objectをあらゆる場面ですべてのクラスの代用として使うことは、この「タイプセーフ」という特性があるために不可能になっているのです。
たとえば、Object型の変数を作ったとして、その変数に、他のクラスのオブジェクトリファレンスを代入したとしても、Obejctクラスにないメソッドは呼び出すことができません。つまり、以下のようなコードは書くことができないのです。

```
Object o = new Ferrari();
o.goFast(); // 誤り
```

このコードはまずコンパイルができません。ObjectクラスにはgoFast()というメソッドはないためです。
Javaのコンパイラは、「メソッドが正しいオブジェクトを使って呼び出されているか」を逐一チェックします。そのため、変数の型がObjectであれば、Objectクラスにないメソッドは決して呼び出すことができないのです。このことについては後で詳しく説明しますので安心してください。

This PDF is prepared for tools@maruhouse.org, personal use only.

Objectクラスを使用する際の注意

先述のとおり、ObjectクラスはJavaのあらゆるクラスのスーパークラスであり、引数や戻り値の型としてObjectを使うと、応用範囲を最大限に高めることができます。ただ、Objectクラスを使う際には、その変わった特性について注意をする必要があります。と言っても、Objectクラスのインスタンスに何か変わったところがあるわけではありません。問題は、他のクラスのインスタンスへの参照を、Object型の変数に代入した場合です。

ArrayList<Dog>にはDog型のオブジェクトを保持することになりますが、このオブジェクトはArrayListから取り出した後もDog型のオブジェクトになります。

```
ArrayList<Dog> myDogArrayList = new ArrayList<Dog>();
Dog aDog = new Dog();
myDogArrayList.add(aDog);
Dog d = myDogArrayList.get(0);
```

- Dogオブジェクトを保持するArrayListを宣言
- Dogオブジェクトの作成
- Dogオブジェクトをリストに追加
- ArrayListに保持していたDogオブジェクトを、別のDog型の変数に代入（ArrayList<Dog>というコードでArrayListを作成したので、getメソッドの戻り値がDog型になっている）

では、以下のように、ArrayList<Object>という宣言をして、あらゆる型のオブジェクトを保持できるようにした場合にはどうなるでしょうか。

```
ArrayList<Object> myDogArrayList = new ArrayList<Object>();
Dog aDog = new Dog();
myDogArrayList.add(aDog);
```

- あらゆる型のオブジェクトを保持できるArrayListを作成
- Dogオブジェクトの作成
- Dogオブジェクトをリストに追加
- （この2行は上記と同じ）

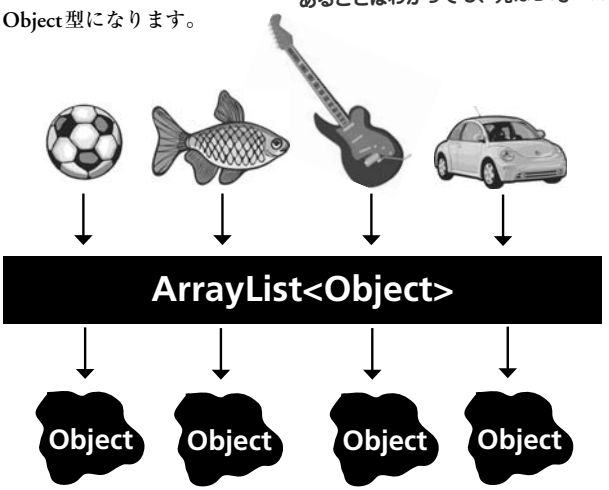
次のようなコードで、ArrayListに保持したDogオブジェクトを取り出そうとした場合にはどうなるでしょうか。

```
Dog d = myDogArrayList.get(0);
```

このコードはコンパイルできない。ArrayList<Object>というコードで宣言をした場合、get()メソッドはObject型のオブジェクトを戻す。コンパイラには、このオブジェクトがObjectクラスを継承するものであることはわかっても、元はDog型だったということはわからない。

ArrayList<Object>から取り出されたオブジェクトは、元の型が何であろうと、すべてObject型になります。

ArrayListに、SoccerBall、Fish、Guitar、Carなどさまざまなクラスのオブジェクトを格納する



取り出される時にはすべてObjectクラスのオブジェクトになる

ArrayList<Object>に格納されたオブジェクトの型は、リストから取り出される時、元の型に関係なく、すべて「Object」になる。コンパイラは、取り出されたオブジェクトの型をすべて「Object」とみなし、他の型であるとはみなさない。

This PDF is prepared for tools@maruhouse.org, personal use only.

全然わからない……
おすわり？
待て？
まったく思い出せないなあ……

DogオブジェクトがDogクラスのオブジェクトでなくなるとは？

Object型の変数や引数を使う際の問題は、代入したオブジェクトリファレンスの型が（一時的に）すべてObjectになってしまったように見えるという点です。たとえば、DogオブジェクトはDogクラスのオブジェクトとしての特性を失います。引数として渡されたオブジェクトをObject型のオブジェクトとして戻すメソッドを例に、その点について見てみましょう。

BAD ☹

```
public void go() {
  Dog aDog = new Dog() ;
  Dog sameDog = getObject(aDog) ;
}

public Object getObject(Object o) {
  return o ;
}
```

この行のコードは誤り。メソッドは引数として渡されたDogオブジェクトへの参照を戻すのだが、戻り値の型がObjectになるため、Object以外の型の変数に代入することはできない

引数として渡されたDogオブジェクトへの参照を戻すのだが、型がObjectになる。このコード自体にまったく問題はない。ArrayList<Dog>ではなく、ArrayList<Object>のget()メソッドを使った場合に似ている

```
File  Edit  Window  Help  Remember
DogPolyTest.java:10: incompatible types
found   : java.lang.Object
required: Dog
       Dog sameDog = takeObjects(aDog);
1 error
```

戻されたオブジェクトが元はDog型だったということはコンパイラにはわからない。したがって、それをDog型の変数に代入するとエラーになる（この点については次ページで詳述）

GOOD ☺

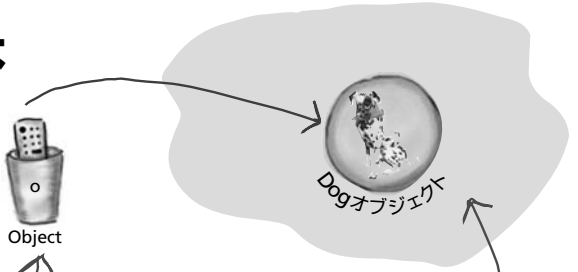
```
public void go() {
  Dog aDog = new Dog() ;
  Object sameDog = getObject(aDog) ;
}

public Object getObject(Object o) {
  return o ;
}
```

このコードだと問題は起きない（ただし、後述するとおり、これではあまり有用とは言えない）。Object型の変数ならどのような型のオブジェクトでも代入できるため。Objectは最上位のスーパークラスであり、Javaのあらゆるクラスは Objectのサブクラスなので、Javaのすべてのオブジェクトは Objectクラスのインスタンスであると言ってもよい。

This PDF is prepared for tools@maruhouse.org, personal use only.

元のクラスのメソッドは使えない

オブジェクトは、いったんObject型の変数に代入されると、元の型の変数に再度代入することはできなくなります。戻り値や引数の型がObject型になっていると、そういうことが起きます。ArrayList<Object>という宣言によって作られた、Object型のArrayListに保持されたオブジェクトについても同じことが言えます。しかし、それは何を意味するのでしょうか。Object型の変数にDogオブジェクトを決して代入してはならないということでしょうか。コンパイラにはObject型とみなされるオブジェクトを使ってDogクラスのメソッドを呼び出すとどうなるでしょうか。



ArrayList<Object>からオブジェクトリファレンスを取り出した場合（あるいは戻り値の型がObjectになっているメソッドを呼び出した場合）、得られるのはObject型のオブジェクトリファレンスである。たとえ元の型がDogであってもObjectになる。

```
Object o = al.get(index);
int i = o.hashCode();
o.bark();
```

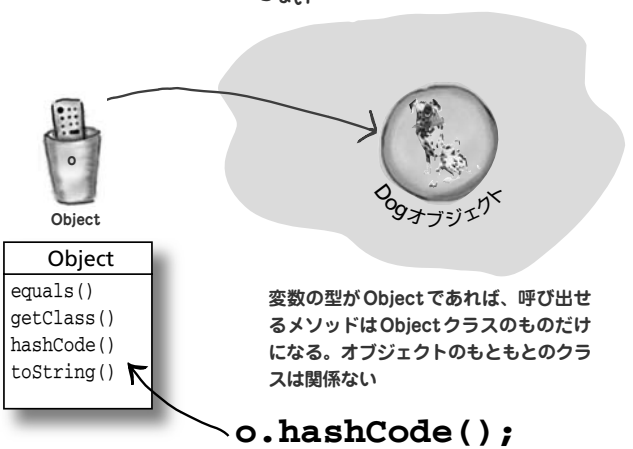
Objectクラスにはhashcode()メソッドがあるので、このコードは正しい。このメソッドは、オブジェクトのクラスが何であっても呼び出せる

コンパイルでエラーになる

このコードは誤り。Objectクラスにはbark()メソッドがないため。たとえプログラマに元のクラスがDogでDogクラスにこのメソッドが存在するということがわかっていても、コンパイラにはそれがわからない

コンパイラは、オブジェクトを使ってどのようなメソッドが呼び出せるかを、オブジェクトのクラスではなく、変数の型によって判断する。

たとえ人間の目から見て明らかにDogクラスのオブジェクトであっても、変数の型がObjectであれば、コンパイラはそれをObjectクラスのオブジェクトであると判断します。オブジェクトのクラスが実際に何であっても関係ないのです。あくまで変数の型で呼び出せるメソッドが判断されます。



変数の型がObjectであれば、呼び出せるメソッドはObjectクラスのものだけになる。オブジェクトのもともとのクラスは関係ない

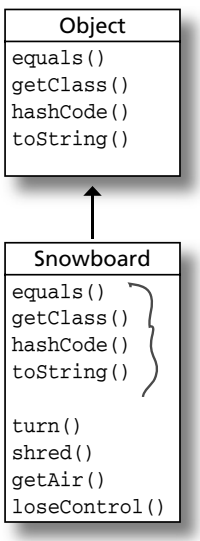
```
o.hashCode();
```

oの型はObjectなので、呼び出せるメソッドはObjectクラスのものだけ

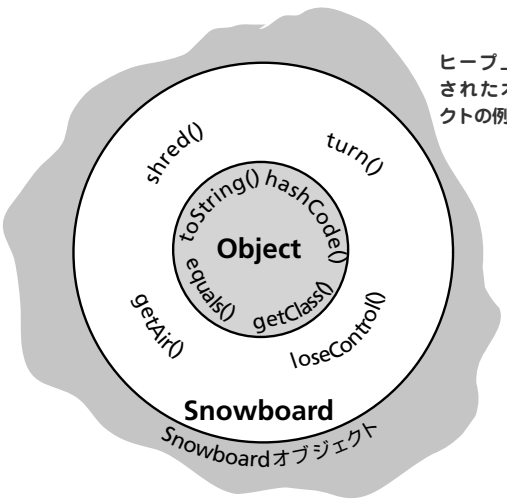
This PDF is prepared for tools@maruhouse.org, personal use only.

すべてのクラスはObjectクラスの特性、機能を受け継ぐ

クラスを継承するということは、そのクラスからすべてを受け継ぐということです。これは、あらゆるクラスがObjectクラスからすべてを受け継いでおり、あらゆるオブジェクトが、クラスに関わらず、Objectクラスのインスタンスとしての特性、機能を持っている、ということを意味します。Javaでは、すべてのオブジェクトをObjectクラスのオブジェクトとして扱ってよいのです。どのクラスのオブジェクトも、下の図のように内側がObjectクラスのオブジェクトになっていて、それにクラスごとの独自のカバーをかぶせているような状態になっている、と考えればいいでしょう。



Snowboardクラスは、スーパークラスであるObjectクラスからメソッドを引き継いでいる他、独自のメソッドも持っている



ヒープ上に作成されたオブジェクトの例

ヒープ上のSnowboardオブジェクトには、Snowboardオブジェクト固有の部分と、Objectオブジェクトとしての部分がある

This PDF is prepared for tools@maruhouse.org, personal use only.

「ポリモーフィズム」は元来「多くの形」を意味する

SnowboardクラスはSnowboardクラスとしてもObjectクラスとしても扱うことができる

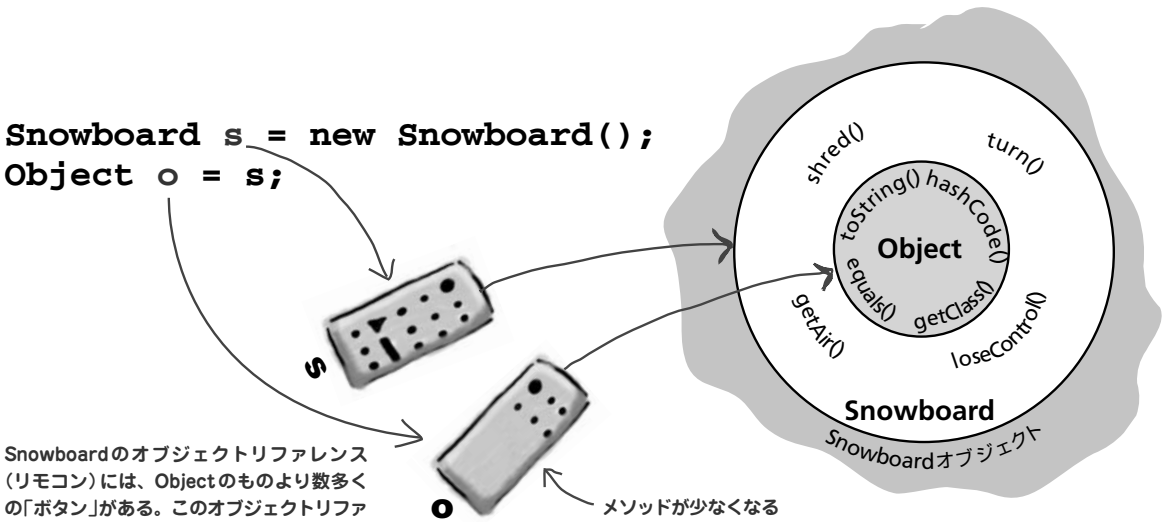
オブジェクトリファレンスは「リモコン」のようなものであるということはすでに述べました。このリモコンは、クラスが下位になるほどボタンが増えます。Objectクラスのオブジェクトリファレンスに比べると、Snowboardクラスのオブジェクトリファレンスの方がボタン（この場合はメソッドを指す）の数が多いのです。Snowboardクラスのオブジェクトリファレンスは、Objectクラスのオブジェクトリファレンスと共通のボタンに加え、独自のボタンも持つことになります。クラスが特殊（具体的）なものに対応するものであるほど、ボタンは増えるのです。

ただ、もちろん、例外はあります。サブクラスが独自のメソッドを持たず、スーパークラスのメソッドをオーバーライドするだけということもあるからです。ここで重要なのは、サブクラスのオブジェクトリファレンスをObject型の変数に代入すると、オーバーライドしたメソッドは呼び出せなくなるということです。呼び出せるのは、Objectクラスのメソッドです。

ArrayListに格納したオブジェクトリファレンスは、もともとのクラスに関係なく、すべてObjectクラスのオブジェクトリファレンスとして扱えます。

ArrayListから取り出す際には、すべてのオブジェクトリファレンスがObjectクラスのオブジェクトリファレンスになります。

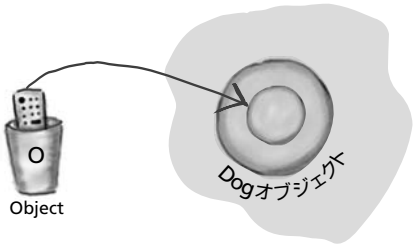
つまり、Objectオブジェクトを操作する「リモコン」が取り出されるということです。



This PDF is prepared for tools@maruhouse.org, personal use only.

ObjectのオブジェクトリファレンスのようになってしまったDogオブジェクトリファレンスを元に戻すには、「キャスト」という方法を使う

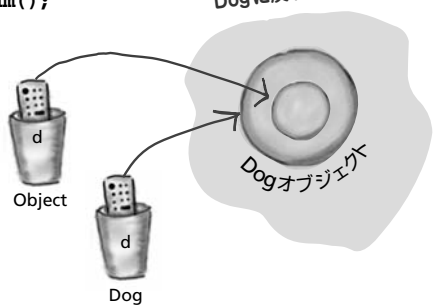
オブジェクトリファレンスの「型」を元に戻す



Object型の変数に代入したDogクラスのオブジェクトリファレンスを元に戻し、Dogクラスのメソッドも利用できるようにするためには、まず、Dog型の変数が必要になります*。Object型変数の値を、以下のように「(Dog) というコードを使って」Dog型の変数に代入すれば、オブジェクトリファレンスはDogクラスのものに戻ります。これを「キャスト」と呼びます。キャスト後は、再びDogクラスのメソッドも呼び出せます。

```
Object o = al.get(index);
Dog d = (Dog)o;
d.roam();
```

Objectをキャストして Dogに戻す



* Object型の変数に代入されているオブジェクトリファレンスのクラスがもともとDogであったかどうか確信が持てない場合は、以下のようにinstanceofという演算子を使って確認します。もし、誤ったクラスにキャストすると、プログラム実行時にClassCastExceptionという例外が発生し、処理が停止されてしまいます。

```
if (d instanceof Dog) {
  Dog d = (Dog)o;
}
```


This PDF is prepared for tools@maruhouse.org, personal use only.

これで、オブジェクトリファレンスを納める変数の型と呼び出せるメソッドの関係についてはわかったと思います。

変数の「型」となっているクラスに存在しないメソッドは呼び出すことはできません。

変数の「型」さえ合っていれば、クラス中のpublic宣言されたメソッドはすべて利用できます。

逆に、自分でクラスを作る時には、public宣言したものは、いつでも他人が自由に使うことができるということを意識しておくべきでしょう。

たとえば、“Simon's Surf Shop”という小さなお店のための会計プログラムを書くとします。開発作業を効率化するため、何か再利用できるクラスはないかと探してみたところ、Accountというクラスが見付かったとしましょう。ドキュメントを調べると、どうも自分の用途に合っているようです。Accountクラスのオブジェクトはそれぞれに「特定の顧客との取引内容」を表します。credit()、debit()、getBalance()などのメソッドがあるようだったので、売掛金の残高を知るためにgetBalance()メソッドを使うことにしました。

Account
debit(double amt) credit(double amt) double getBalance()

ところが、どうもそれがよくなかったようです。ドキュメントをよく読むと、getBalance()は確かに存在しているのですが、実は何もしないメソッドだったのです。プログラムは実行時にエラーを発生して止まってしまいました……。

と、実際にはこのようなことは起きません。オブジェクトリファレンスと「.」演算子を使ってメソッドを呼び出すコード（a.doStuff()など）は、すべてコンパイル時にチェックされるからです。メソッドが本当に呼び出せるものであるか、引数、戻り値の型が正しいかといったことが確認され、問題があればコンパイルの段階でエラーになります。

ここで注意すべきなのは、先述のとおり、コンパイラがチェックするのは、オブジェクトリファレンスを格納する変数の「型（クラス）」である、ということです。オブジェクト自体のもともとのクラスが何であったかということはチェックされません。

This PDF is prepared for tools@maruhouse.org, personal use only.

利用できるメソッドを変更する

Dogクラスのオブジェクトを使って呼び出せるメソッドは、Dogクラスのものだけではありません。それと同時に、Dogクラスのすべてのスーパークラスの（public宣言された）メソッドも呼び出すことができます。

Dogオブジェクトの場合、具体的にはどのようなメソッドが呼び出せるでしょうか。

それはスーパークラスにどのようなものがあるかを考えればわかります。

まず、Canineクラスのメソッドは呼び出せます。

また、Animalクラスのメソッドも呼び出せるでしょう。

そして当然、Objectクラスのメソッドも呼び出せます。

Dogクラスのオブジェクトは、Canineクラス、Animalクラス、Objectクラスのオブジェクトでもある、ということができるのです。

ここで例にあげているDogクラスは動物シミュレーションプログラムを作るためのものなので、メソッドもそれに合わせたものになっています。

しかし、同じクラスをペットショップのシミュレーションプログラムに再利用したい、という場合はどうなるでしょうか。

ペットショップのプログラムであれば、Dogクラスには、beFriendly()、play()といったメソッドも欲しいところですが、そういうメソッドはありません。

その場合は、「単純にメソッドを追加してしまえばいい」という気もします。既存のメソッドの内容を変えるわけではありませんから、Dogクラスを利用している他の人に迷惑がかかる、ということもなさそうです。

さて、本当にそれで問題はないのでしょうか。ここから先はしばらくそのことについて考えてみましょう。

頭の体操

ペットショッププログラムに対応するためのメソッドの追加をもし自分が担当するとしたら、あなたはどうしますか。もちろん、単純にDogクラスにメソッドを追加するという方法でも特に問題は生じないでしょう。すでにDogクラスを利用しているプログラムに影響を与えるわけではないからです。

ただ、ペットショップのプログラムということは犬以外の動物もいるはずなので、同様のメソッドを持つクラスを他にも作る必要がある、ということを考慮しなくてはなりません。また、同じDogクラスを、飼い犬としてではなく、野良犬として他のプログラムに利用したい、という人もいるかもしれません。そうしたことを考えた時、メソッドを追加するのに最も適切なのはどのような方法でしょうか。今の時点では具体的にJavaのコードをどう書けばよいか（あるいはJavaで実現可能な方法なのか）、ということは考えなくてかまわないので、ともかくAnimalクラスとそのサブクラスにペットショッププログラムに対応するための修正を加える妥当な方法を考えてみてください。

解答は次のページ以降に紹介されていますが、自分の力で十分に考えるまでは次のページに進まないようにしてください。

（考える前に答えを見てしまっては練習の意味がありません。それに、そんなことをすれば、脳でカロリーを消費するチャンスをみすみす逃すことになります）

This PDF is prepared for tools@maruhouse.org, personal use only.

ペットショッププログラムのための修正方法

これから何ページかにわたって、ペットショッププログラムに対応するためにAnimalクラスとそのサブクラスを修正する方法について考えてみましょう。取りあえずのところは、Javaプログラムに実際に採用できる方法であるかどうかということは考慮に入れていません。ただ、いくつか考えられる方法を示し、それぞれの長所、短所を示すだけにとどめています。

① **方法1**

最上位のAnimalクラスに、ペットショッププログラムで新たに必要となるメソッドを追加する（最も単純な方法）

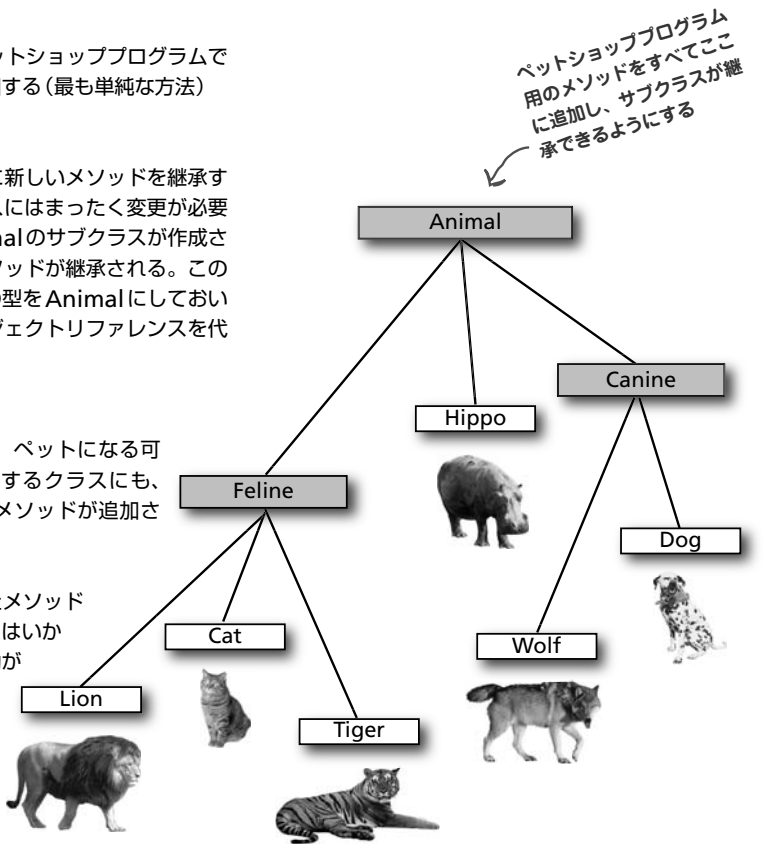
長所

Animalのサブクラスは自動的に新しいメソッドを継承することになる。既存のサブクラスにはまったく変更が必要ない。また、今後、新たにAnimalのサブクラスが作成された場合も、自動的に新しいメソッドが継承される。この方法だと、変数、引数、戻り値の型をAnimalにしておいて、実際にはサブクラスのオブジェクトリファレンスを代入しても何の問題も生じない。

短所

カバやライオン、オオカミなど、ペットになる可能性がほとんどない動物に対応するクラスにも、ペットショッププログラム用のメソッドが追加される。

また、Animalクラスに追加したメソッドは単に継承するだけというわけにはいかない。たとえば、犬と猫では行動が違うので、Dogクラス、Catクラスのそれぞれでメソッドに改造が必要になる。



This PDF is prepared for tools@maruhouse.org, personal use only.

② 方法 2

ペットショッププログラム関連のメソッドをAnimalクラスに追加する点は方法 1 と同じだが、方法 2 では、そのメソッドをすべて抽象メソッドにし、サブクラスで必ずオーバーライドするようにする。

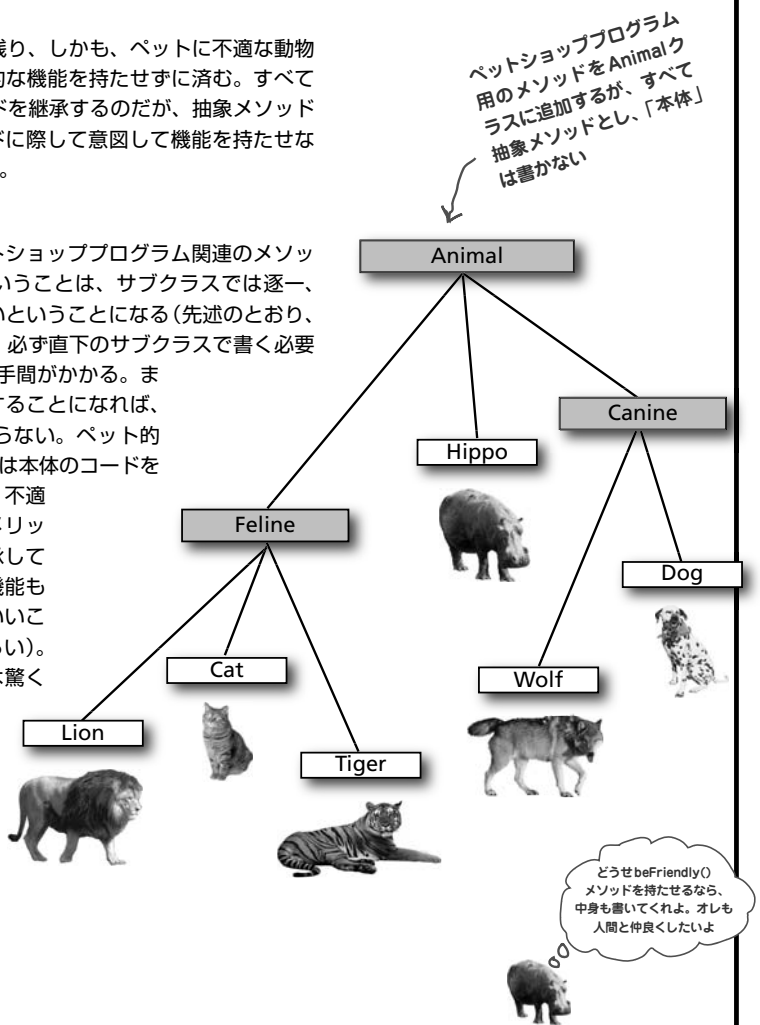
長所

方法 1 の長所は基本的にそのまま残り、しかも、ペットに不適な動物に対応するクラスに不当にペット的な機能を持たせずに済む。すべてのサブクラスが「ペット的」メソッドを継承するのだが、抽象メソッドになっているため、オーバーライドに際して意図して機能を持たせない限り、そのメソッドは何もしない。

短所

Animalクラスに追加されるペットショッププログラム関連のメソッドがすべて抽象メソッドであるということは、サブクラスでは逐一、「本体」のコードを書かねばならないということになる（先述のとおり、抽象メソッドの「本体」のコードは、必ず直下のサブクラスで書く必要がある）。これにはかなりの時間と手間がかかる。また、後で新たにサブクラスを追加することになれば、改めて本体のコードを書かねばならない。ペット的でない動物に対応するサブクラスでは本体のコードを実質的に何もしないものにすれば、不適切な機能を持たせずに済むというメリットはある。しかし、メソッドを継承しておいて、そのメソッドが実は何の機能も持っていない、というのはあまりいいことではない（外から見てわかりづらい）。知らずにメソッドを呼び出した人は驚くことになる。

全体的に見て、この方法はあまり良いとは言えない。特に、本質的に種類の異なる機能（Animalのすべてのサブクラスに持たせるべき機能とそうでない機能）を同時にAnimalクラスに持たせようとしている点に問題があるように思える。



This PDF is prepared for tools@maruhouse.org, personal use only.

③ **方法 3**

ペットショッププログラム用のメソッドを必要なクラスにのみ追加する。

長所

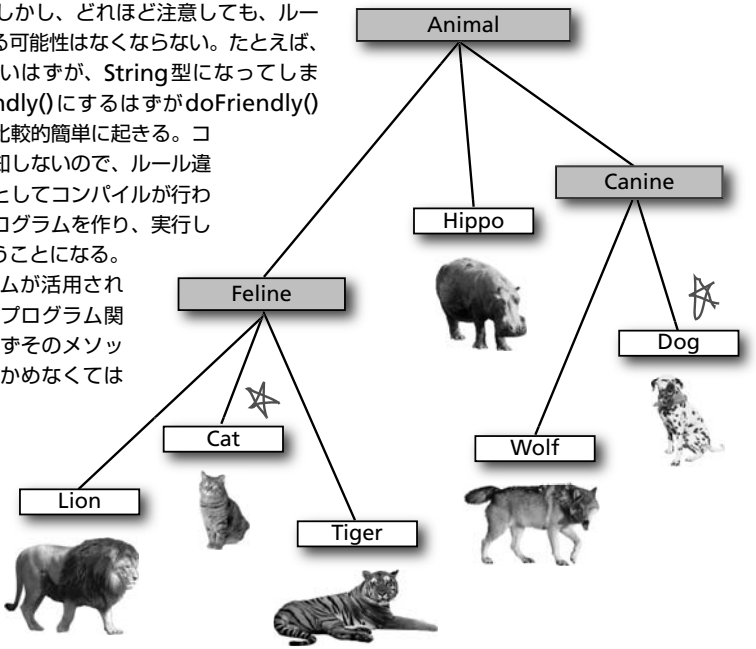
Hippoなど、不適切なクラスが「ペット的」な機能を持つ心配はない。あくまでペットとしてふさわしい動物に対応するクラスにだけ機能が追加される。具体的には、CatクラスとDogクラスだけにメソッドが追加され、他のクラスはそれにまったく影響を受けない。

短所

この方法には大きな問題が2つある。1つは、ペットショッププログラムのためにどのようなメソッドをどのように作らねばならないか、というルール（プロトコル）を明確に決め、それを関係するすべてのプログラマに周知しなければならないということ。Dog、Catだけでなく、今後、ペットとして適切な動物に対応するクラスを追加する場合には、必ずルールを守ってメソッドを作らねばならない。つまり、その作業はルールを知っているプログラマが行う必要がある。しかし、どれほど注意しても、ルールが守られずにメソッドが作られる可能性はなくならない。たとえば、引数をint型にしなければならないはずが、String型になってしまうという、メソッド名をbeFriendly()にするはずがdoFriendly()になってしまう、といったことが比較的簡単に起きる。コンパイラはこのルールについて関知しないので、ルール違反があっても何も問題がないものとしてコンパイルが行われる。誰かがクラスを利用してプログラムを作り、実行してみてはじめて誤りに気付くということになる。

2つ目の問題は、ポリモーフィズムが活用されないということ。ペットショッププログラム関連のメソッドを使いたい人は、必ずそのメソッドがどのクラスに属するのかを確かめなくてはならない。Animal型の変数や引数にDogやCatのオブジェクトリファレンスを代入すると、ペットショッププログラム関連のメソッドは呼び出せなくなってしまう（呼び出そうとすると、コンパイルでエラーになる）。Animalクラス自身にはそのメソッドがないため。

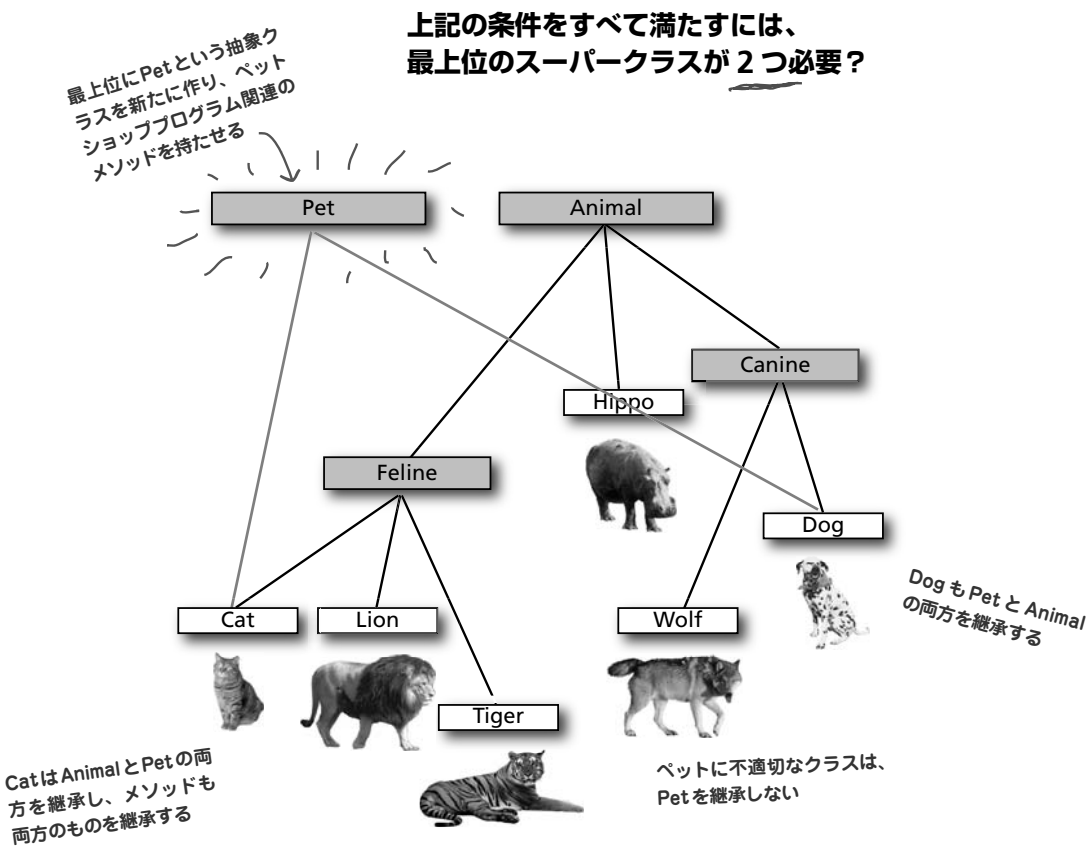
「ペット的」なメソッドを、Animalクラスではなく、必要なクラスにのみ追加する。



This PDF is prepared for tools@maruhouse.org, personal use only.

では、どのような方法が理想か？

- ペットショッププログラム関連のメソッドは、ペットにふさわしいクラスにのみ持たせる。
- ペットショッププログラム関連のメソッドが確実に同じルールに基づいて作られるようにする。メソッド名、引数、戻り値の型などは同じにする。また、必要なメソッドの欠落がないようにする。プログラマがルールを守ってくれることをただ祈る、ということではいけない。
- ポリモーフィズムが活かせるようにする。Animal型の変数や引数にDogやCatのオブジェクトリファレンスを代入して、ペットショッププログラム関連のメソッドを呼び出せるようにすべき。



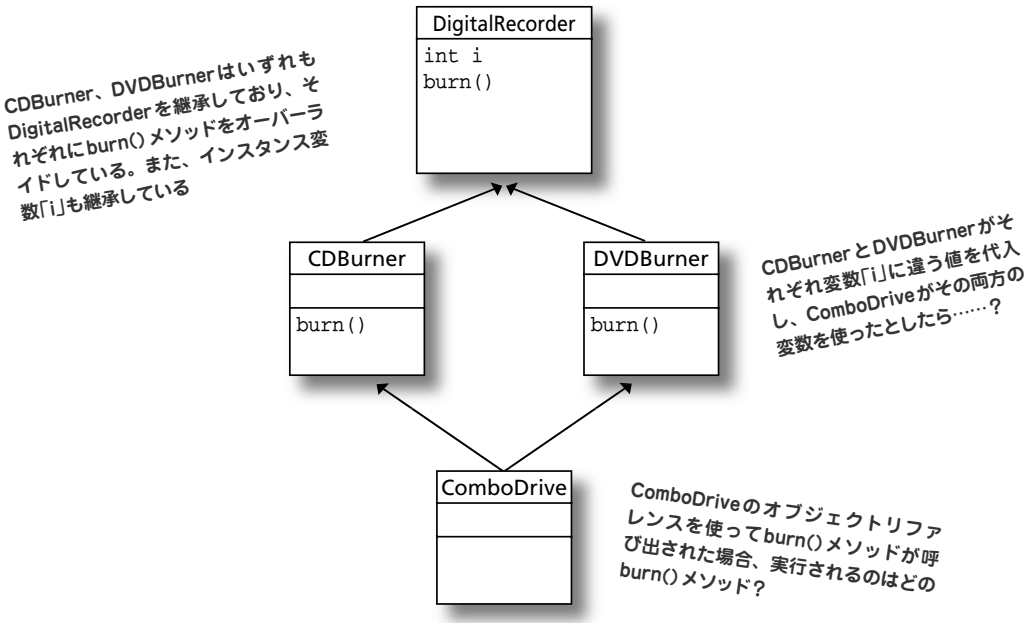
This PDF is prepared for tools@maruhouse.org, personal use only.

確かに前ページの方法ですべての条件を満たすことができますが、この方法には問題が……

このように2つ以上のクラスを継承することを「多重継承」と呼びますが、多重継承は重大な問題を引き起こすことがあるのです。

また、実はJavaでは多重継承は禁止されており、実際に行うことはできません。
多重継承に伴う「重大な問題」は、一般に「ダイアモンド継承」と呼ばれています。

ダイアモンド継承



プログラミング言語で上記のようなダイアモンド継承が許されていると、非常に面倒な事態になります。各所に生じる「曖昧さ」を排除するためにいくつものルールを作る必要があるからです。ルールが増えれば、それだけ学習が難しくなります。また、たとえルールが習得できたとしても、プログラミング作業には常に細心の注意が必要になるでしょう。Javaは常にシンプルなルールに基づいてプログラムが作れるよう、プログラムの種類によってルールが複雑になったり簡単になったりすることがないよう考えられています。Javaでは、ダイアモンド継承の問題は起きません（この点はC++と異なっている）。それは良いことなのですが、では一体、ここで例にあげているようなペットショッププログラムはどうやって作ればいいのでしょうか。

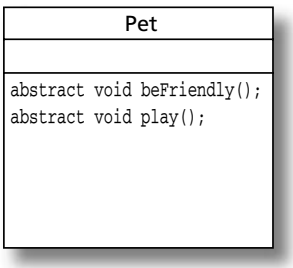
This PDF is prepared for tools@maruhouse.org, personal use only.

インターフェース

Javaでは、このような場合、「インターフェース」と呼ばれるものを使用します。この「インターフェース」は、「ユーザーインターフェース」などとはまったく無関係の言葉です。Javaのインターフェースには、それとは異なる特別の意味があるのです。

インターフェースを使えば、多重継承を行うことなく（ダイアモンド継承の問題も起こすことなく）、ポリモーフィズムを活用できます。

インターフェースは、先述のPetクラスのようなクラスに似ています。ただ、異なっているのは、すべてのメソッドが抽象メソッドになるという点です。つまり、サブクラスでは、そのメソッドの「本体」のコードを作る必要があるわけです（すでに述べたとおり、抽象メソッドは直下の具象クラスで「本体」を作らねばなりません）。インターフェースを使った場合には、同じ名前のメソッドが複数存在しても、実行時にJVMがどのクラスのメソッドを呼び出せばよいか混乱することはありません。



Javaのインターフェースは「完全な抽象クラス」とでも呼ぶべきもの

インターフェース中のメソッドはすべて抽象メソッド。つまり、Petを継承する具象クラスでは、そのメソッドの「本体」を作らなくてはならない（本体を作ることを「インプリメント」ともいう）。

インターフェースの作り方

```
public interface Pet {...}
```

通常は“class”と書くべきところに“interface”と書く

インターフェースを継承し、メソッドをインプリメントする方法

```
public class Dog extends Canine implements Pet {...}
```

インターフェースを継承し、メソッドをインプリメントする際は、このように、インターフェースの名の前に“implements”というキーワードを入れる。同時にクラスを継承できるという点に注目

This PDF is prepared for tools@maruhouse.org, personal use only.

Petインターフェースの作成とインプリメント

ここでは、「public abstract」と書いているが、インターフェースのメソッドには、わざわざこう書かなくても自動的に書いたのと同じ特性が持たされる（また、一般にはわざわざ書かない方が良いとされている。ここではわかりやすくするためあえて書いた）

"class" ではなく "interface" と書く

```
public interface Pet {
  public abstract void beFriendly();
  public abstract void play();
}
```

インターフェースのメソッドはすべて抽象メソッドなので、「本体」がなく、このようにセミコロンで終わる

Dog IS-A Animal
Dog IS-A Pet

インターフェース名の前に "implements" と書く

```
public class Dog extends Canine implements Pet {
  public void beFriendly() {...}
  public void play() {..}

  public void roam() {...}
  public void eat() {...}
}
```

ここにPetインターフェースのメソッドの「本体」を書く。セミコロンが{}に変わっている点に注意

これは単なるメソッドのオーバーライド

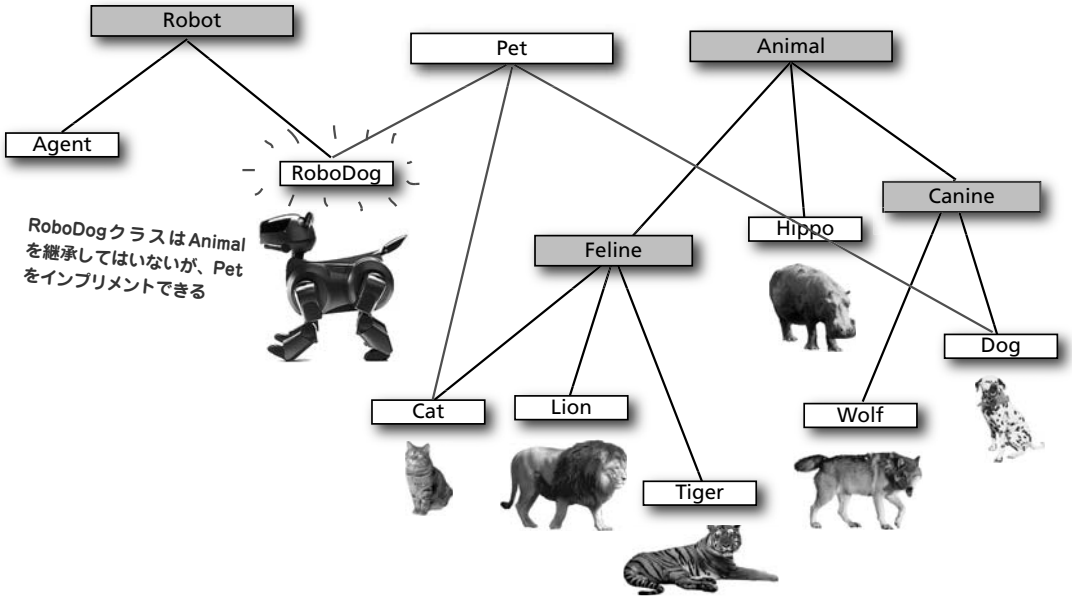
素朴な疑問に答えます

Q：ちょっと待ってください。メソッドに本体のコードがないインターフェースを使っても多重継承ができるわけではないのですよね。一体、インターフェースは何の役に立つのですか。

A：インターフェースは、「ポリモーフィズム」と大きく関係しています。たとえば、インターフェースを引数や戻り値の「型」として使用すると、そのインターフェースをインプリメントするあらゆるクラスのオブジェクトが引数や戻り値として使用できるようになります。また、重要なのは、スーパークラスの継承とインターフェースのインプリメントが1つのクラスで同時に行えることです。たとえば、まったく別のスーパークラスを継承する2つのクラスが、同じ1つのインターフェースをインプリメントする、ということも可能なのです。その他、インターフェースを使うことには、「どのクラスのインスタンスであるか」ということではなく、「プログラムの中でどのような役割を果たすべきものか」ということを考えてオブジェクトが使えるようになる、というメリットもあります。
同様の性質、機能を持つクラスを複数作る必要がある場合に、インターフェースを用意しておけば、すべてのクラスが継承する「最上位のスーパークラス」を作らずに済みます。クラスどうしの継承関係に注意せずに済むというのも利点でしょう。
インターフェースでは、メソッドをすべて抽象メソッドにします。これは、サブクラス（まだ存在していないものも含む）のことを考えてメソッドのコードを書かずに済むということを意味します。サブクラスがどのようなものになっても妥当性を失わないようなメソッドを書く、ということは非常に困難なので、抽象メソッドにし、各サブクラスで「本体」を書くようにすれば便利です。

This PDF is prepared for tools@maruhouse.org, personal use only.

インターフェースは、スーパークラスが何であるかに関係なく、あらゆるクラスでインプリメントできる



先述のとおり、AnimalやCanineといったクラスを変数、引数、戻り値の型に使った場合、実際に代入できるのは、そのサブクラスのオブジェクトリファレンス、ということになります。つまり、Canine型の変数や引数にWolfやDogのオブジェクトリファレンスを代入することはできても、CatやHippoのオブジェクトリファレンスは代入できないのです。

しかし、型としてPetなどのインターフェースを使った場合は事情が異なります。Petインターフェースをインプリメントするものでさえあれば、どのクラスのオブジェクトリファレンスであっても代入できるのです。この特性は、Java APIにとっては非常に重要なものです。継承関係のない複数のクラスに共通の機能を持たせるということが簡単にできるからです。たとえば、オブジェクトのシリアライズの機能を持たせたい時は、Serializableインターフェースを、マルチスレッド機能を持たせたい時はRunnableインターフェースをインプリメントするということをします。シリアライズやマルチスレッドの詳細については後述しますが、ともかく、こうした機能を必要とする場合には、そのクラスが何のサブクラスであるかということに関係なく、Serializableインターフェース、Runnableインターフェースをインプリメントするのだということだけ理解してください。実際、シリアライズやマルチスレッドの機能は、スーパークラスが何であるかに関係なく必要になるものです。

1つのクラスで複数のインターフェースをインプリメントすることもできる

Javaの場合、複数のクラスを同時に継承することはできません。DogはCanineを、CanineはAnimalを、AnimalはObjectというように、継承の道筋は常に一本になります。しかし、インターフェースは同時に複数をインプリメントできるのです。たとえば、以下のようなコードを書くこともあり得るということです。この場合、DogはPetだけでなく、他のインターフェースも同時にインプリメントすることになります。

```
public class Dog extends Animal implements
Pet, Saveable, Paintable { ... }
```


This PDF is prepared for tools@maruhouse.org, personal use only.

独立したクラス、サブクラス、抽象クラス、インターフェースのうちのどれを作るかの判断

* 既存のクラスのいずれとも「IS-A関係」にならないクラスは、独立したクラスにする（ただし、どのクラスも結局はObjectのサブクラスになる点に注意）。
* 既存のクラスをより特殊な（具体的な）ものにすれば済むような場合は、そのクラスを継承したサブクラスを作り、メソッドをオーバーライドしたり、新たなメソッドを追加したりする。
* 基本的な部分が共通したクラスを複数作らねばならないことがあらかじめわかっている場合は、抽象クラスを作ってそれを継承する。抽象クラスはいわば「テンプレート」のように使える。抽象クラスには、抽象メソッドと、本体を持つ通常のメソッドを混在させることができるが、最低1つは通常のメソッドにする。「このクラスはインスタンス化すべきではない」と判断したクラスは抽象クラスにすると良い。
* プログラムの中で同じ役割を果たすクラスを複数作る必要がある場合には、インターフェースを作る。インターフェースを使えば、クラスの継承関係を気にすることなく、同様の性質を持ったクラスを複数作れる。

This PDF is prepared for tools@maruhouse.org, personal use only.

スーパークラスバージョンのメソッドを利用する

Q：スーパークラスのメソッドをオーバーライドするのだが、基本的な機能はそのまま活かしたい、という場合はどうすればいいのですか。すでに存在している機能はそのまま残し、それに新たな機能を付け加えるだけで済ませたいということもあると思うのですが……。

A：クラスを継承する際に“extends”というキーワードを使用しますが、“extend”がもともと「拡張する」という意味である、という点に注目してください。オブジェクト指向プログラミングで重要なのは、オーバーライドすることがわかっているメソッドの本体をどのように作るかです。抽象クラスであっても、多くのサブクラスに共通して活かせるコードが書けるメソッドは抽象メソッドにせず、本体を持ったメソッドにします。しかし、あらゆるサブクラスにそのまま使えるような本体を書くことはまず不可能です。そこで、サブクラスでは、スーパークラスのコードをそのまま活かした上で、必要なコードを追加する、ということもできるようになっているのです。サブクラスのメソッドにスーパークラスのメソッドのコードをそのまま活かすには、“super”というキーワードを使用します。

```
abstract class Report {
  void runReport() {
    // メソッドのコード
  }
  void printReport() {
    // メソッドのコード
  }
}

class BuzzwordsReport extends Report {

  void runReport() {
    super.runReport();
    buzzwordCompliance();
    printReport();
  }
  void buzzwordCompliance(){...}
}
```

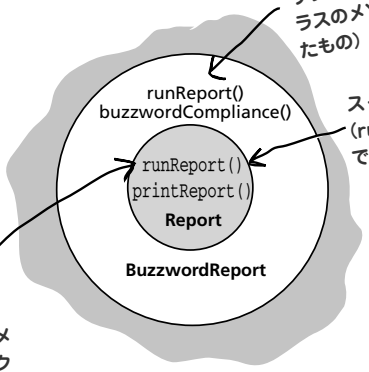
スーパークラスのメソッドには、多くのサブクラスが共通して使用できる機能を持たせる

このようにスーパークラスのメソッドを呼び出せば、そのコードをそのまま活かすことができる

以下のようなコードを書けば

```
super.runReport();
```

スーパークラスのrunReport()メソッドが実行される



サブクラスのメソッド（スーパークラスのメソッドをオーバーライドしたもの）

スーパークラスのメソッド（runReport()はサブクラスでオーバーライドされる）

super.runReport();

スーパークラスとサブクラスに同じ名前のメソッドがある場合、サブクラスのオブジェクトリファレンスを利用すれば、常にサブクラスの方のメソッドが呼び出されることになる。ただ、サブクラスのコードの中で“super”というキーワードを使用すれば、スーパークラスのメソッドを呼び出すことができる

“super”というキーワードは、「オブジェクト中のスーパークラスから継承した部分を参照せよ」という意味。サブクラスの中で「super.runReport();」のように使用すれば、スーパークラスバージョンのメソッドを呼び出せる

This PDF is prepared for tools@maruhouse.org, personal use only.

重要ポイント

- オブジェクト（インスタンス）を作ることが適切でないクラスは、“abstract”というキーワードを使って抽象クラスにする。
- 抽象クラスには、抽象メソッドと通常のメソッドを混在させることができる。
- 抽象メソッドを1つでも含むクラスは、抽象クラスにしなければならない。
- 抽象メソッドには「本体」のコードを書かず、末尾にはセミコロン（;）を入れる。{ }は使わない。
- 抽象メソッドは、必ず、直下のサブクラスで本体を作らなくてはならない。
- Javaのクラスはすべて、Object（java.lang.Object）クラスを直接、あるいは間接的に継承する。
- メソッドの引数、戻り値の型はObjectにすることができる。
- あるメソッドを呼び出すには、必ず、そのメソッドの属するクラス（あるいはインターフェース）が型になっている変数を使わなくてはならない。変数に代入されているオブジェクトの元々の型が何であろうと関係ない。したがって、Object型の変数を使って呼び出せるのは、Objectクラスのメソッドのみ。
- Object型になったオブジェクトは、キャストをしない限り、元の型の変数には代入できない。キャストによって、元の型の変数への代入が可能になるが、キャストした型と、元の型とが一致しないと、プログラム実行時にエラーとなる。

  例：Dog d = (Dog) x.getObject(aDog);

- ArrayList<Object>から取り出したオブジェクトは、すべてObject型になる（キャストをしない限り、Object型以外の変数には代入できない）。
- 「キャスト」というテクニックを使えば、オブジェクトのクラスを（Objectから）元に戻すことができる。
- オブジェクトリファレンスを使って呼び出せるメソッドは、リファレンスが格納されている変数の「型」となっているクラス（インターフェース）の持つメソッドのみ。オブジェクトリファレンスのもともとのクラスが何であるかは無関係。
- Javaでは、「ダイアモンド継承」と呼ばれる問題があるため多重継承はできないようになっている。つまり、同時に複数のクラスを継承することはできない（直接のスーパークラスは常に1つ）。
- Javaのインターフェースは、いわば「完全な抽象クラス」とでも呼ぶべきもの。インターフェースのメソッドはすべて抽象メソッド。
- インターフェースを作る場合には、“class”の代わりに“interface”というキーワードを使用する。
- インターフェースをインプリメントするには、"implements"というキーワードを使用する

  例：Dog implements Pet

- インターフェースは同時に複数インプリメントできる。
- インターフェースをインプリメントするクラスでは、インターフェースのすべてのメソッドの「本体」を作る必要がある。インターフェースのメソッドは何も指定しなくても、自動的に「public abstract」という指定があるものとみなされる。
- メソッドをオーバーライドしているサブクラスからスーパークラスのバージョンのメソッドを呼び出すには、"super"というキーワードを使用する。

  例：super.runReport();

素朴な疑問に答えます

Q：まだよくわからないのですが……。ArrayList<Dog>に保持したDogオブジェクトは、ArrayListから取り出した後にキャストしなくても、そのままDogオブジェクトとして使えるということですが、それはなぜですか。ArrayListクラスのメソッドの引数はObject型になっていて、DogやDotComといった他の型ではないはずですが。ArrayList<Dog>と書いた際に何か特別なことが起きるのですか。それを教えてください。

A：「特別なこと」は確かに起きます。おっしゃるとおり、ArrayListのメソッドの引数はObject型になっていますし、他の型、たとえばDog型のオブジェクトには対応できないように見えます。それでも、ArrayList<Dog>と書けば、保持したDogオブジェクトをそのままのかたちで取り出すことができます。キャストが必要ないというのも本当です。

その理由は、簡単に言えば「コンパイラがキャストをしてくれているから」ということになります。"ArrayList<Dog>"というコードを書けるからと言って、メソッドの引数、戻り値がDog型になったクラスが用意されているわけではありません。<Dog>は「ArrayListにはDogのみを入れる。他のオブジェクトはリストに追加できないようにせよ」とコンパイラに指示するためのコードと考えればいいでしょう。ArrayListにDog以外入ってこないとわかれば、コンパイラは「リストから取り出されるオブジェクトはすべてDogにキャストしてもよい」と判断します。そのため、プログラマが自分でキャストのためのコードを書く必要はなくなります。これには、単に手間が省けるという以上の意味があります。キャストのコードは実行時にエラーになる可能性があるからです。実行時にエラーになると、行っている処理が重要なものであった場合には深刻な問題を引き起こしかねません。それよりはコンパイル時にエラーになる方がずっと良いと言えるでしょう。

話はこれですべてというわけではありません。このことについてはまた後で触れます。

This PDF is prepared for tools@maruhouse.org, personal use only.

エクササイズ

下の 1 ～ 5 のクラス（インターフェース）間の関係を図に表してください。1 に関してはすでに作図してありますので、後はそれにならって作図してみてください（抽象クラスは、これまでの図の中にも出てきたように、具象クラスとは色を変えるようにします）。インプリメントは点線、継承は実践で表します。

クラス（インターフェース）

図

1)
```
public interface Foo { }
public class Bar implements Foo { }
```

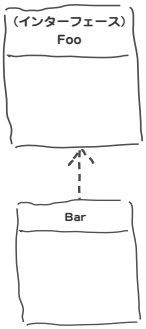
2)
```
public interface Vinn { }
public abstract class Vout implements Vinn { }
```

3)
```
public abstract class Muffie implements Whuffie { }
public class Fluffie extends Muffie { }
public interface Whuffie { }
```

4)
```
public class Zoop { }
public class Boop extends Zoop { }
public class Goop extends Boop { }
```

5)
```
public class Gamma extends Delta implements Epsilon { }
public interface Epsilon { }
public interface Beta { }
public class Alpha extends Gamma implements Beta { }
public class Delta { }
```

1)



2)

3)

4)

5)

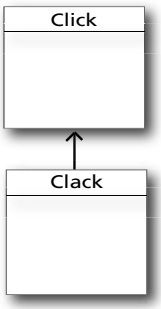
This PDF is prepared for tools@maruhouse.org, personal use only.

エクササイズ

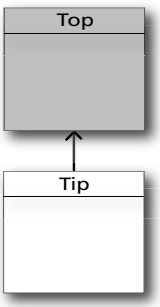
左に示した 1 ～ 5 の図を見て、それに対応するクラスの宣言を書いてみてください。1 についてはすでに書いてありますので、残りはそれを参考に書くようにしてください。

図

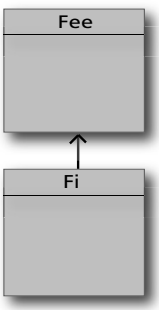
1)



2)

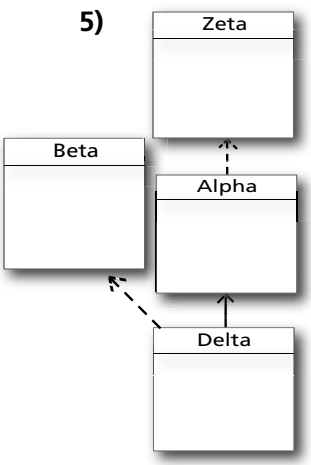


3)



4)

5)



宣言

1)
```
public class Click { }
public class Clack extends Click { }
```

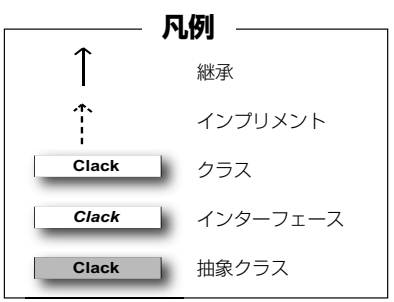
2)

3)

4)

5)

凡例



This PDF is prepared for tools@maruhouse.org, personal use only.

プールパズル

右のプログラムの欠落部分に、下のプールの中のコードを当てはめ、実行結果が以下に示すようなものになるプログラムを完成させてください。また、今回は実行結果の欠落部分も埋めてください。すべてのコードを使う必要はありません。同じコードを 2 回以上繰り返し使うこともできます。

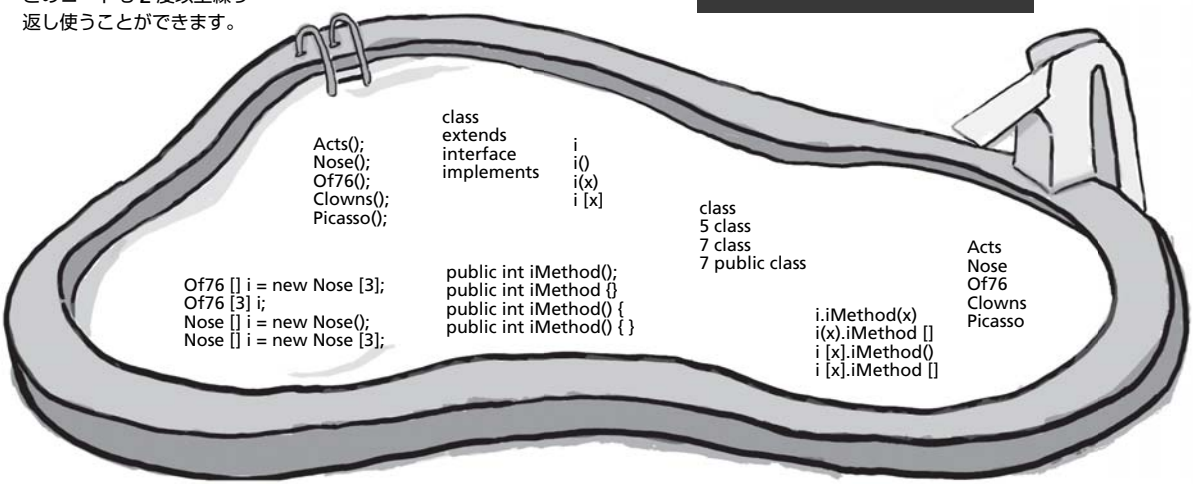
```
___________ Nose {
  ________________________
}

abstract class Picasso implements ______ {
  ________________________
    return 7;
  }
}

class _________ _________ _________ { }
class _________ _________ _________ {
  ________________________
    return 5;
  }
}
```

```
public ___________ ___________ extends Clowns {

  public static void main(String [] args) {
    ________________________
    i[0] = new __________
    i[1] = new __________
    i[2] = new __________
    for(int x = 0; x < 3; x++) {
      System.out.println(_________________
              + " " + ________.getClass( ) );
    }
  }
}
```

実行結果

```
File  Edit  Window  Help  BeAfraid
%java _____________
5 class Acts
7 class Clowns
________Of76
```

注意：
どのコードも 2 度以上繰り返し使うことができます。

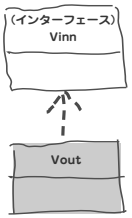


This PDF is prepared for tools@maruhouse.org, personal use only.

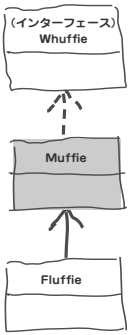
エクササイズの答え

図

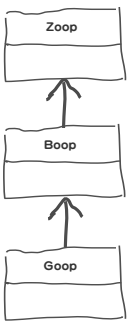
2)



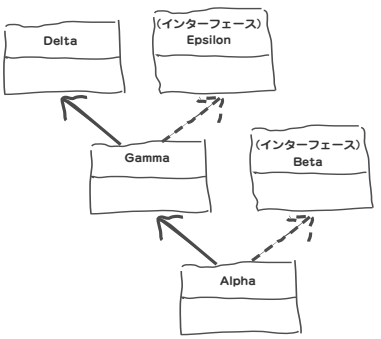
3)



4)



5)



宣言

2)
```
public abstract class Top { }
public class Tip extends Top { }
```

3)
```
public abstract class Fee { }
public abstract class Fi extends Fee { }
```

4)
```
public interface Foo { }
public class Bar implements Foo { }
public class Baz extends Bar { }
```

5)
```
public interface Zeta { }
public class Alpha implements Zeta { }
public interface Beta { }
public class Delta extends Alpha implements Beta { }
```


This PDF is prepared for tools@maruhouse.org, personal use only.

プールパズル

```
interface Nose {
  public int iMethod();
}
abstract class Picasso implements Nose {
  public int iMethod() {
    return 7;
  }
}
class Clowns extends Picasso { }

class Acts extends Picasso {
  public int iMethod() {
    return 5;
  }
}
```

```
public class Of76 extends Clowns {
  public static void main(String [] args) {
    Nose [] i = new Nose [3];
    i [0] = new Acts();
    i [1] = new Clowns();
    i [2] = new Of76();
    for(int x = 0; x < 3; x++) {
      System.out.println( i [x].iMethod()
        + " " + i [x].getClass());
    }
  }
}
```

実行結果

```
File  Edit  Window  Help  KillTheMime
% java Of76
5 class Acts
7 class Clowns
7 class Of76
```


This PDF is prepared for tools@maruhouse.org, personal use only.

9章　コンストラクタとガーベージコレクション

オブジェクトの生と死

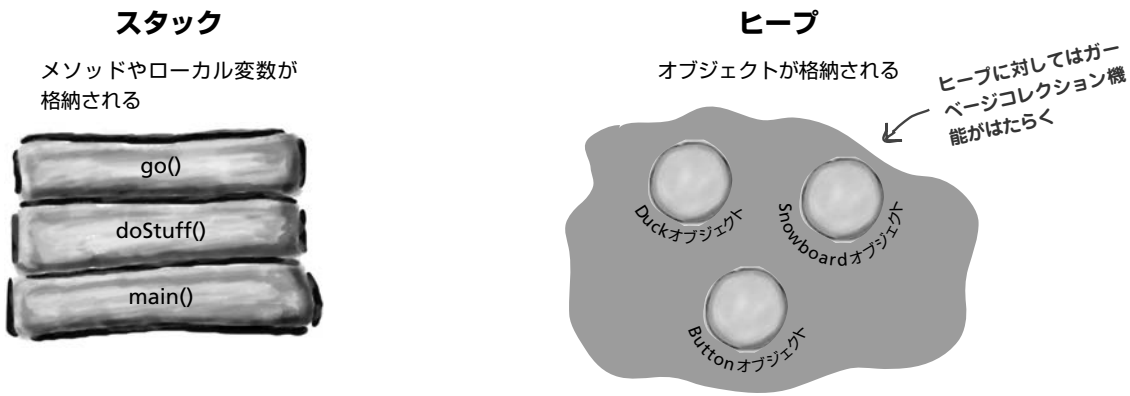
オブジェクトには、「生まれて、やがて死ぬ」というライフサイクルがあります。そのライフサイクルをどのようなものにするか決めるのはプログラマの仕事です。あるオブジェクトがいつ作られ、いつ削除されるか、ということをプログラマが決めるということです。ただ、Javaには、プログラマがオブジェクトを削除するためのコードを直接書くわけではないという特徴があります。プログラマが書くのは、厳密には、「オブジェクトを放棄するためのコード」です。放棄されたオブジェクトは、非情なガーベージコレクタによって消滅させられます。オブジェクトが消滅すると、そのオブジェクトに割り当てられていたメモリ領域は他の目的に使用できるようになります。一度生まれたオブジェクトが死ぬことなく、ずっとそのまま生き残っていれば、いずれメモリが足りなくなってしまいます。この章では、オブジェクトを作る方法、そして効率的に消滅させる方法について解説します。具体的には、ヒープ、スタック、スコープ、コンストラクタ、スーパーコンストラクタなどについて詳しく触れることになるでしょう。この章の内容は、読者の中には難しく感じる人がいるかもしれませんが、すぐに理解しようとあせる必要はありません。はじめは、ともかく一通り読んで次に進む、ということで十分です。

This PDF is prepared for tools@maruhouse.org, personal use only.

オブジェクトはどこに存在するか

まずは、Java言語でメモリがどのように使用されるか、ということについて解説しましょう。Javaを使う際には、メモリの2つの領域、ヒープとスタックについて知っておく必要があります。ヒープというのは、オブジェクトが格納される領域のことで、スタックというのは、メソッドやローカル変数が格納される領域のことです。JVMは起動すると、OSから一定の量のメモリを取得します。このメモリをJavaプログラムの実行に利用するのです。取得するメモリの量が具体的にどのくらいかは、JVMのバージョン(JVMの動作するプラットフォーム)によって異なります。また、JVMによっては、取得するメモリの量をプログラマが変更できる場合もあります(できない場合の方が多い)。ただし、正しくプログラムを作っていれば、まずその必要はありません。その点については後で詳述します。

オブジェクトがヒープに格納されるということについては本書ですでに触れました。しかし、変数がどこに格納されるのかということについてはまだ触れていません。変数の格納場所は、その変数の種類によって変わってきます。ただ、この場合の「種類」というのは、プリミティブ型、参照型といった種類のことを言っているのではありません。ここでの話に関係があるのは、「インスタンス変数」、「ローカル変数」という分類です。ローカル変数は別名「スタック変数」と呼ばれており、スタックに格納されるのですが、インスタンス変数はオブジェクトとともにヒープに格納されます。



インスタンス変数

インスタンス変数とは、メソッドの外で宣言される変数のことです。インスタンス変数には、個々のオブジェクト(インスタンス)ごとに異なる値が格納されます。インスタンス変数(の値)は、オブジェクトの一部としてメモリに格納されることになります。

```
public class Duck {
  int size;
}
```

size変数の値はDuckオブジェクトごとに異なる

ローカル変数

ローカル変数は、メソッドの中で宣言される変数です。メソッドのパラメータも、ローカル変数の一種です。ローカル変数(の値)は、メソッドがスタック上に存在する間(つまりメソッドの処理が終了するまでの間)だけスタックに格納されます。

```
public void foo(int x) {
  int i = x + 3;
  boolean b = true;
}
```

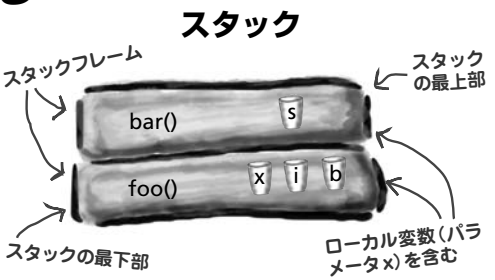
パラメータ「x」、変数「i」、「b」はローカル変数。

This PDF is prepared for tools@maruhouse.org, personal use only.

メソッドはスタックに格納される

メソッドは呼び出されると、メモリの「スタック（コールスタック）」と呼ばれる領域の最上部に格納されます（スタックに格納されることを「プッシュ」と呼びます）。スタックは、「スタックフレーム」という単位で構成されます。厳密には、個々のメソッドとそのローカル変数の値はスタックフレームに格納されることになります。

スタックの最上部には、常に、その時点で実行中のメソッドが置かれます（ここでは、スタックは1つだけ存在するものとして話をしていますが、14章でも触れるとおり、実際には複数にすることもできます）。また、メソッドは、その実行が終了するまでの間スタックに置かれることになります。たとえば、foo()メソッドからbar()というメソッドが呼び出されると、右の図のように、スタックは、foo()の上にbar()が置かれているという状態になります。



スタックの最上部には、常にその時点で実行中のメソッドが置かれる

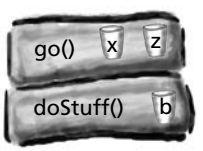
```
public void doStuff() {
  boolean b = true;
  go(4);
}
public void go(int x) {
  int z = x +24;
  crazy();
  // メソッドのコード
}
  public void crazy() {
  char c = 'a';
}
```

スタックの使用例

左のコードはdoStuff()、go()、crazy()という3つのメソッドから構成されています（クラス全体のコードがどのようになっているかは、ここでは考えなくてかまいません）。コードをよく見るとわかるとおり、doStuff()がgo()を、さらにgo()がcrazy()を呼び出すようになっています。各メソッドにはメソッドの本体で宣言されたローカル変数が1つずつあり、またgo()にはパラメータもあります（つまり、go()にはローカル変数が2つあるということ）。

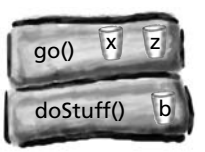
① まず、あるクラスのコードの中でdoStuff()が呼び出される。すると、doStuff()はスタック最上部のスタックフレームに格納される。同じスタックフレームには、bというboolean型のローカル変数も格納される。

② doStuff()がgo()を呼び出すと、go()はスタックの最上部にプッシュされる。変数x、zも同じスタックフレームに格納される。



③ go()がcrazy()を呼び出すと、crazy()がスタックの最上部に格納される。変数cも同じフレームに格納される。

④ crazy()の実行が完了すると、スタックフレームがスタックから出される（これを「ポップ」という）。そして、go()メソッドの実行が、crazy()メソッドを呼び出した後の行から再開される。

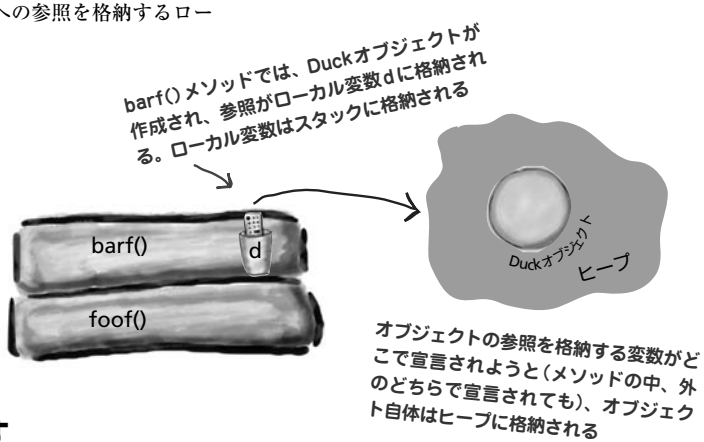


This PDF is prepared for tools@maruhouse.org, personal use only.

ローカル変数が参照型のものである場合は？

すでに何度か触れたとおり、変数にはプリミティブ型のもの以外に、オブジェクトへの参照を格納する参照型のものがあります。参照型変数はどこに格納されるのでしょうか。オブジェクトがヒープに格納されることはすでに述べました。それは参照が格納される変数がどこで宣言されようと（ローカル変数であろうと、インスタンス変数であろうと）、常に変わりません。しかし、そのオブジェクトへの参照を格納するローカル変数は、スタックに格納されます。

```
public class StackRef {
  public void foof() {
    barf();
  }

  public void barf() {
    Duck d = new Duck(24);
  }
}
```



素朴な疑問に答えます

Q：なぜスタックやヒープについて学ぶ必要があるのですか。知っていて何か役に立つことがあるのですか。知らなくて特に困らないような気がするのですが。

A：スタックやヒープについて知ることは、変数のスコープ、オブジェクトの作成、メモリ管理、スレッド、例外処理といったことがらについて理解する上で非常に重要です。スレッドや例外処理に関しては後の章で詳しく説明することになりますが、その他のことについてはすべてこの章で触れます。とはいえ、個々のJVM、プラットフォームにおいてスタックやヒープが具体的にどのように実現されているか、といったことまで知る必要はありません。前のページとこのページで解説したごく基本的なことさえわかっていれば十分です。それだけで、後で学ぶことがらについての理解がかなり容易になるでしょう。必ず、「何かピンとこなかったが、一応スタックとヒープについて知っておいてよかった」と思う時があるはずです。

重要ポイント

- Javaを使う際には、メモリのヒープとスタックという2つの領域について知っておく必要がある。
- インスタンス変数とはメソッドの外で宣言されている変数のこと。
- ローカル変数はメソッドの中で宣言されている変数、あるいはメソッドのパラメータのこと。
- ローカル変数は、メソッドとともにスタック（スタックフレーム）に格納される。
- オブジェクトリファレンスを格納する参照型変数も、プリミティブ型変数と同様、ローカル変数であればスタックに格納される。
- オブジェクトは、参照が格納される変数がローカル変数であるかインスタンス変数であるかに関わらずヒープに格納される。

This PDF is prepared for tools@maruhouse.org, personal use only.

インスタンス変数の格納場所

たとえば、「new CellPhone()」といったコードで新たにオブジェクトが作成された場合は、ヒープ上にそのオブジェクト（ここではCellPhoneオブジェクト）を格納するための領域が必要になります。この場合、厳密には、「オブジェクト」に、「オブジェクトのインスタンス変数」も含まれるという点に注意しなければなりません。つまり、インスタンス変数もすべて格納できるだけの領域が必要ということです。インスタンス変数は、その変数が属するオブジェクトとともにヒープに格納されるからです。

インスタンス変数がプリミティブ型の変数である場合は、型のサイズに合わせてヒープ上に領域を確保することになります。たとえば、int型の変数であれば32ビット、long型の変数であれば64ビットの領域が必要ということです。確保される領域が、変数の値が何であるかによって変化することはありません。変数の型がintであれば、値が32,000,000であろうと32であろうと確保される領域は32ビットで変わりないのです。

インスタンス変数が参照型である場合はどうでしょう。たとえばCellPhoneクラスに、Antennaオブジェクトへの参照を格納する変数があった場合はどうなるのでしょうか。

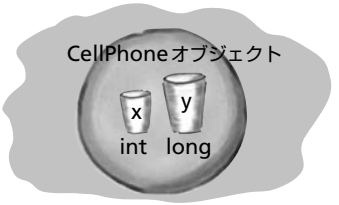
オブジェクトに参照型のインスタンス変数があった場合は、変数に対応するすべてのオブジェクトを格納できるだけのスペースがヒープに必要なのでしょうか。そんなことはありません。ヒープにインスタンス変数の値を格納するためのスペースが必要なのは確かですが、インスタンス変数に格納されるのはオブジェクトそのものではなく、「参照」です。つまり、たとえば、CellPhoneクラスに、Antenna型のインスタンス変数があったとしても、ヒープには、Antennaオブジェクトそのものではなく、オブジェクトへの参照を格納できるだけのスペースさえあればよい、ということになります。

この場合、Antennaオブジェクトそのもののためのスペースがヒープ上に確保されるのはいつでしょうか。簡単に言えば、それは「Antennaオブジェクトが作られた時」ということになります。いつAntennaオブジェクトが作られるかは、インスタンス変数の宣言の仕方によって変わります。以下のように、インスタンス変数の宣言のみが行われ、同時にオブジェクトの作成、参照の代入が行われなければ、その時点では、変数（参照）のためのスペースのみが確保されます。

```
private Antenna ant;
```

逆に、以下のようにオブジェクトの作成と参照の代入が同時に行われれば、Antennaオブジェクトそのもののためのスペースが確保されます。

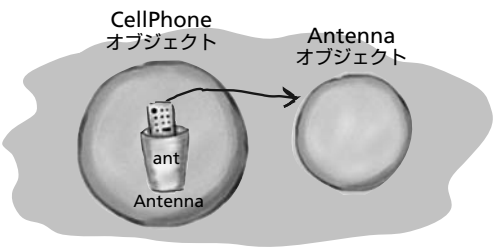
```
private Antenna ant = new Antenna();
```



プリミティブ型のインスタンス変数が2つあるオブジェクト。オブジェクトとともに変数がヒープに格納される

Antennaオブジェクトへの参照を格納する参照型の変数を持つオブジェクト。この場合、Antennaオブジェクトそのものでなく、参照がCellphoneオブジェクトとともにヒープに格納される。以下のように変数の宣言のみが行われると、Antennaオブジェクトは作成されない

```
public class CellPhone {
  private Antenna ant;
}
```



以下のように、変数の宣言とともにAntennaオブジェクトの作成、参照の代入が行われた場合、Antennaオブジェクトは、CellPhoneオブジェクトおよびそのインスタンス変数とは別に、ヒープに格納される

```
public class CellPhone {
  private Antenna ant = new Antenna();
}
```


This PDF is prepared for tools@maruhouse.org, personal use only.

オブジェクト誕生の神秘

変数やオブジェクトがどこに格納されるのか、ということは、ここまでの説明でわかったはずです。次は、「オブジェクトの誕生」という神秘の世界に迫ってみましょう。オブジェクトが作成される際には、参照型の変数の宣言と、その変数への参照の代入も行われますが、何と言っても不思議なのは、「オブジェクトの作成」という処理そのものです。

「生命誕生の神秘」といった類の（ともあれば、下品になりかねない）話が好きでない人もいるでしょうが、Javaを知る上ではとても重要なことですので、我慢してこの先を読み進めていってください。

変数の宣言、オブジェクトの作成、参照の代入

❶ 参照型の変数の宣言

オブジェクトへの参照を格納するための変数を宣言

```
Duck myDuck = new Duck();
```

Duckオブジェクトへの参照

❷ オブジェクトの作成

ここに神秘が……

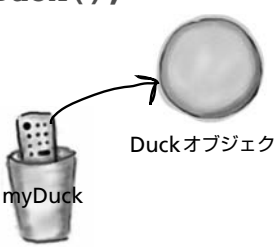
```
Duck myDuck = new Duck();
```

Duckオブジェクト

❸ オブジェクトの参照の代入

作成したオブジェクトへの参照を変数に代入する

```
Duck myDuck = new Duck();
```



Duckオブジェクト

Duckオブジェクトへの参照

This PDF is prepared for tools@maruhouse.org, personal use only.

Duck オブジェクトを作成する際には Duck() というメソッドを呼び出しているように見えますが、果たして本当にそうなのでしょうか。

```
Duck myDuck = new Duck();
```

Duck()はメソッドに似ていますがメソッドではなくコンストラクタと呼ばれるものです。

Duck() というメソッドを呼び出しているように見える。

コンストラクタはメソッドに似ていますが、メソッドではありません。またコンストラクタを使用する際には、必ず「new」というキーワードを使用します。「new」とコンストラクタを組み合わせて使用することでオブジェクト（インスタンス）が作れるのです。

コンストラクタの名前は、作成するオブジェクトのクラスと同じになります。「new」に続けてクラス名を書き、()を付ければコンストラクタが呼び出せると考えてもいいでしょう（この章で後述するとおり、コンストラクタの中で他のコンストラクタを呼び出す、という場合には、これとは異なる方法が使えることもありますが、一般のコードの中でコンストラクタを呼び出すには、この方法を使うしかありません）。

コンストラクタのコードは、「new Duck()」のようなコードによってクラスのインスタンス化が行われる際に実行されます。すべてのクラスはコンストラクタを持っています。プログラマが自ら作成しなかった場合は自動的に作られます

コンストラクタはどこにあるのでしょうか。クラスを作る際にはコンストラクタも作らねばならないのでしょうか。

コンストラクタは、クラスを作る際にプログラマが自ら作ることもできますが（その方法はすぐ後で説明します）、プログラマがコンストラクタを作らなかった場合には、コンパイラが自動的に作成してくれます。

コンパイラが作るコンストラクタのことを「デフォルトコンストラクタ」と呼びます。デフォルトコンストラクタのコードは次のようになります。

```
public Duck() {
}
```

メソッドとの違いがわかるでしょうか。よく見ると、メソッドには当然あるはずのものがない、ということがわかります。

名前は必ずクラスと同じになる。

```
public Duck() {
    // コンストラクタのコード
}
```

メソッドであれば、publicとDuck()の間に戻り値の型の指定が必要だが、コンストラクタには必要ない。

This PDF is prepared for tools@maruhouse.org, personal use only.

コンストラクタのコード

コンストラクタのコードは、オブジェクトの参照が変数に格納され前に実行されます。つまり、このコードには、作成したオブジェクトを利用可能な状態にするのに何か必要な処理があれば、それを行わせることができます。下に紹介したDuckクラスのコンストラクタのコードは、さほど重要な処理を行うものではありませんが、コンストラクタのコードがいつ実行され、何の役に立つかということは一応、わかるでしょう。

```
public class Duck {
  public Duck() {
    System.out.println("Quack");
  }
}
```

コンストラクタのコード

```
public class UseADuck {
  public static void main(String [] args) {
    Duck d = new Duck();
  }
}
```

Duckのコンストラクタを呼び出す

コンストラクタのコードは参照が変数に格納される前に実行される

```
File  Edit  Window  Help  Quack
% java UseADuck
Quack
```

自分で考えてみよう

コンストラクタのコードはnewキーワードによるオブジェクトの作成に際して実行されます。右の処理のうち、Carクラスのコンストラクタのコードに行わせるのにふさわしいものはどれですか。ふさわしいと思うものに✓を付けてください。このCarクラスは、レーシングゲームのプログラムに使うものと仮定します。

- ☐ Carクラスのオブジェクトがこれまでにいくつ作られたかを記録するためのカウンタをインクリメントする。
- ☐ レースの現状を表す変数に値を代入する。
- ☐ オブジェクトのインスタンス変数に値を代入する。
- ☐ コンストラクタを呼び出したクラス（new Car()というコードの含まれたクラス）のオブジェクトリファレンスを変数に代入する。
- ☐ CarオブジェクトをArrayListに追加する。
- ☐ Carオブジェクト以外のオブジェクト（インスタンス変数に参照を格納するオブジェクト）を作成する。
- ☐ _____（自分でも何かふさわしい処理を考えてください）

This PDF is prepared for tools@maruhouse.org, personal use only.

オブジェクトのインスタンス変数の設定

コンストラクタでは、多くの場合、以下のように、インスタンス変数の初期値の設定（オブジェクトのステートの初期設定、あるいは初期化などとも呼ばれる）が行われます。

```
public Duck() {
  size = 34;
}
```

ただ、たとえば、sizeというインスタンス変数の値としてはこのくらいが適切、ということがあらかじめわかっていればこれでいいのですが、状況によって適切な値が変わってしまう、というような場合はこのようなかたちで初期値を設定すべきではありません。

では、Duckオブジェクトを作成する度にsize変数に異なる値を設定したいという場合はどうすればいいのでしょうか。

すぐに思い付くのは、setSize()というメソッドを用意しておき、Duckオブジェクトを作成する度にこのメソッドを呼び出すようにする、という方法です。この方法では、Duckオブジェクトをいったんsize変数の値を設定せずに作成し*、後で変数の値を設定するということになります。つまり、Duckオブジェクトを作成する度に以下のようなコードを書くわけです。ただ、Duckオブジェクトを使うプログラマ全員がこの方法を利用してくれるという保証はありません。

```
public class Duck {
  int size;

  public Duck() {
    System.out.println("Quack");
  }

  public void setSize(int newSize) {
    size = newSize;
  }
}
```

インスタンス変数

コンストラクタ

セッターメソッド

```
public class UseADuck {
  public static void main (String [] args) {

    Duck d = new Duck();
    d.setSize(42);
  }
}
```

Duckオブジェクトはsize変数の値を設定せずに作成される。プログラマが確実にセッターメソッドを呼び出してsize変数の値を設定するという保証はない。必ずセッターメソッドを呼び出さねばならない、ということをプログラマが知らないこともあり得る

* インスタンス変数にはデフォルト値がある。プリミティブ型（数値型）の変数であればデフォルト値は0あるいは0.0、boolean型であればfalse、参照型であればnull。

素朴な疑問に答えます

Q：コンストラクタはコンパイラによって自動的に作られるということですが、それならなぜわざわざ自分でコンストラクタを作る必要があるのですか。

A：コンストラクタを自分で作るのは、通常、オブジェクト作成時に、インスタンス変数の初期値の設定など、必ず行っておくべきことがある場合です。初期値としてどのような値を設定するかを、コンストラクタを呼び出すプログラマに決めさせるということも可能です。また、「オブジェクト作成時に行うべきことは何もないが、それでもコンストラクタは作らねばならない」ということもあり得ます。それには「スーパークラスのコンストラクタ」が大きく関係するのですが、そのことについては後述します。

Q：コンストラクタとメソッドはどのようにして区別すればいいのですか。もし、クラスと同じ名前のメソッドがあった場合はどうすればいいのでしょうか。

A：確かにJavaではクラスと同じ名前のメソッドを作ることもできます。クラスと同じ名前でもコンストラクタとは限らないわけです。コンストラクタとメソッドは宣言の際に戻り値の型が指定されているかどうかで見分けます。メソッドには戻り値の型が指定されますが、コンストラクタには戻り値の型は指定されません。

Q：コンストラクタは継承されないのですか。コンストラクタを作らなかった場合、デフォルトコンストラクタが作られるのではなく、スーパークラスのコンストラクタがそのまま引き継がれるということはないのですか。

A：そういうことはありません。コンストラクタは継承されないのです。そのことについてはしばらく後に詳しく説明します。

This PDF is prepared for tools@maruhouse.org, personal use only.

コンストラクタを使用してオブジェクトのインスタンス変数の初期設定をする*

オブジェクトの中には、特定のインスタンス変数の初期値を設定しないうちは、誰にも使用させるべき(参照を変数に代入させるべき)ではない、というものもあります。たとえ「newによるインスタンス化の後は、必ずセッターメソッドを呼び出して初期値を設定する」というルールを作ったとしても、あらゆるプログラマにそれを徹底させることは不可能です。必ず、ルールを守らず(あるいは知らず)、必要な初期値を設定せずにオブジェクトリファレンスを変数に代入してしまうプログラマが現れます。やはりコンストラクタの中でどうしても初期値を設定しなければならない、という仕組みを作るべきでしょう。そのためには、以下に示すように、コンストラクタに引数(パラメータ)を設けることになります。

```
public class Duck {
  int size;

  public Duck(int duckSize) {
    System.out.println("Quack");

    size = duckSize;

    System.out.println("size is " + size);
  }
}
```

Duckのコンストラクタにint型のパラメータを追加

引数の値を使ってインスタンス変数sizeの初期値を指定する

```
public class UseADuck {
  public static void main (String [] args) {
    Duck d = new Duck(42);
  }
}
```

コンストラクタに値を渡す

セッターメソッドを使わなくても、1つのステートメントでDuckオブジェクトの作成と、size変数の初期値の設定ができる

```
File  Edit  Window  Help  Honk
% java UseADuck
Quack
size is 42
```

* 必ずすべてのインスタンス変数の初期値をオブジェクト作成時に設定しなければならないというわけではない

This PDF is prepared for tools@maruhouse.org, personal use only.

複数のコンストラクタを作る

インスタンス変数の初期値は、パラメータ付きのコンストラクタでオブジェクト作成時に同時に設定できると便利であるということはすでに説明しました。ただ、問題は引数としてどのような値が適切かをすべてのプログラマが知っているとは限らないということです。適切な値を知っているプログラマは引数を指定し、知らないプログラマは引数を指定しない（その場合は自動的に適切な値が指定される）、ということができると便利なのではないでしょうか。

つまり、プログラマには2つの選択肢が用意されるということです。自ら初期値を設定することも、自動的に初期値が設定されるのに任せることもできる……そういうことは不可能なのでしょうか。

こんなことはコンストラクタが1つでは当然できません。通常のメソッドと同様、コンストラクタも、パラメータがあればやはり引数を指定せずに呼び出すことはできないのです。引数の指定がなければデフォルト値が指定されるというわけにはいきません。引数を指定しなければコンパイルすらできなくなるでしょう。また、以下のようなコードを書けば選択肢を2つにすることはできそうですが、あまり良い方法とは言えません。

```
public class Duck {
  int size;

  public Duck(int newSize) {
    if (newSize == 0) {
      size = 27;
    } else {
      size = newSize;
    }
  }
}
```

パラメータの値が「0」であれば、sizeにデフォルト値「27」を指定し、さもなければ、パラメータの値をsizeの値として使用するというコード。これは良い方法とは言えない

このようなコンストラクタを作った場合は、プログラマが「sizeの値が自動的に指定されるようにするには引数に0を指定すればよい」ということを知っておくことが必要になります。プログラマ全員にそれを伝えるということは不可能です。また、sizeの値を本当に「0」にしたい人はどうすればいいのでしょうか（sizeの値が0にされるのを防ぐため、コンストラクタに値をチェックするためのコードを入れるということはあり得る）。プログラマが本当に「0」を設定したいのか、ただデフォルト値を得るための方便として「0」を指定しているのかを区別することはできません。

コンストラクタは複数作ることができる

```
public class Duck2 {
  int size;

  public Duck2() {
    // デフォルト値を設定
    size = 27;
  }

  public Duck2(int duckSize) {
    // パラメータの値をsizeに指定
   size = duckSize;
  }
}
```

引数を指定すれば下のコンストラクタが自動的に呼び出される

```
Duck2 d = new Duck2(15);
```

引数を指定しなければ上のコンストラクタが呼び出され、デフォルト値が設定される

```
Duck2 d2 = new Duck2();
```

このようにコンストラクタを2つ作れば、sizeの適切な値がわかる人は引数を指定し、わからない人は引数を指定せずにDuck2オブジェクトを作るということができる。

このように複数のコンストラクタを作ることを「コンストラクタのオーバーロード」という。

This PDF is prepared for tools@maruhouse.org, personal use only.

引数のないデフォルトコンストラクタはどのような場合でも必ず作られるのか No!

ええと……、「コンストラクタは自分で作ることができる」と。なるほど。「作らなければ、自動的に１つ作られる」か。これは覚えておかないと……。

前ページの例を見て、「引数のないコンストラクタはコンパイラによって自動的に作られるのだから、引数のあるコンストラクタだけ作ればよいのではないか」と思った人もいるでしょう。しかし、それは間違いなのです。コンパイラがデフォルトコンストラクタを作るのは、あくまでプログラマがまったくコンストラクタを作らなかった場合だけだからです。

引数のあるコンストラクタを作ったが、引数のないコンストラクタも同時に必要、という場合は引数のないコンストラクタを自分で作らねばなりません。

たとえどのようなものであろうと、プログラマが自らコンストラクタを作れば、コンパイラは「もうコンストラクタは存在するから改めて作る必要はないな」と判断します。

１つのクラスに複数のコンストラクタを作る場合は、必ず引数が異なるようにします。

「引数が異なる」というのは、引数の型が異なるということです。また、複数の引数がある時はその順序にも注意が必要です。引数が異なっている限り、コンストラクタは基本的にいくらでも増やすことができます。同じことはメソッドにも言えますが、その点については別の章で触れます。

This PDF is prepared for tools@maruhouse.org, personal use only.

クラスに複数のコンストラクタを持たせることを「コンストラクタのオーバーロード」という

コンストラクタが複数になる時は、それぞれに引数が異なっていなければコンパイルができない

「引数が異なる」というのがどういう意味なのか、例をあげて具体的に見てみましょう。以下に示したMushroomというクラスには4つのコンストラクタがあります。このようにコードを書けば、引数がそれぞれに異なるとみなされるので、コンパイルは可能です。しかし、たとえば、int型の引数を1つしか持たないコンストラクタが2つあったとしたら、引数が重複しているとみなされ、コンパイルはできないでしょう。引数（パラメータ）に付ける名前はこの際、関係ないのです。重要なのは、引数の型と順序です。順序さえ違っていれば、型はすべて同じでも、異なる引数とみなされます。つまり、Stringの引数の後にintの引数があるのと、intの引数の後にStringの引数があるのとでは違う、というわけです。

4つのコンストラクタがあるということは、Mushroomオブジェクトを作る手段が4つあるということ

```
public class Mushroom {
  public Mushroom(int size) { }
  public Mushroom( ) { }
  public Mushroom(boolean isMagic) { }
  public Mushroom(boolean isMagic, int size) { }
  public Mushroom(int size, boolean isMagic) { }
}
```

int型の引数のみ

引数なし

boolean型の引数のみ。int型の引数なし

boolean型の引数とint型の引数

型は同じだが、順序が違っているのでコンパイルできる

重要ポイント

- ■インスタンス変数は、オブジェクトとともにヒープに格納される。
- ■インスタンス変数が参照型のものであれば、変数も、対応するオブジェクトもヒープに格納される。
- ■コンストラクタのコードは、オブジェクトが作成される際に実行される。
- ■コンストラクタはクラスと同じ名前を持つ。戻り値の型は指定しない。
- ■コンストラクタには、オブジェクト作成時にインスタンス変数の初期値を設定するためのコードを持たせることができる。
- ■コンストラクタは、プログラマが自ら作成しなければ、コンパイラによって自動的に作成される。自動的に作成されるコンストラクタを「デフォルトコンストラクタ」と呼ぶ。
- ■デフォルトコンストラクタは必ず引数のないコンストラクタになる。
- ■どのようなものであれ、プログラマが自らコンストラクタを1つでも作れば、デフォルトコンストラクタが作られることはない。
- ■引数のあるコンストラクタと引数のないコンストラクタの両方が必要な場合は、どちらもプログラマが自ら作らねばならない。
- ■オブジェクトがインスタンス変数の値の初期設定を必要とするものである場合、引数のあるコンストラクタとともに、デフォルト値を設定する機能を備えた引数のないコンストラクタも用意しておくと、プログラマにとってはオブジェクトの作成が容易になり、便利である。
- ■1つのクラスに複数のコンストラクタを作ることを「コンストラクタのオーバーロード」という。
- ■コンストラクタのオーバーロードを行う場合は、引数をコンストラクタごとに異なるものにしなければならない。
- ■1つのクラスの中に、引数の型と順序がまったく同じコンストラクタが複数存在してはならない。
- ■インスタンス変数にはデフォルト値がある。デフォルト値は、プリミティブ型（数値型）の変数の場合は0か0.0、boolean型の変数の場合はfalse、参照型の変数の場合はnullである。

This PDF is prepared for tools@maruhouse.org personal use only.

自分で考えてみよう

左に示したプログラムには、コンストラクタを呼び出すためのコードがいくつかありますが、それぞれ右側のどのコンストラクタに対応するでしょうか。対応するものどうしを例にならって矢印で結んでください。

```
public class TestDuck {

  public static void main(String [] args) {

    int weight = 8;
    float density = 2.3F;
    String name = "Donald";
    long [] feathers = {1, 2, 3, 4, 5, 6};
    boolean canFly = true;
    int airspeed = 22;

    Duck [] d = new Duck [7];

    d [0] = new Duck();

    d [1] = new Duck(density, weight);

    d [2] = new Duck(name, feathers);

    d [3] = new Duck(canFly);

    d [4] = new Duck(3.3F, airspeed);

    d [5] = new Duck(false);

    d [6] = new Duck(airspeed, density);
  }
}
```

```
class Duck {

  int pounds = 6;
  float floatability = 2.1F;
  String name = "Generic";
  long [] feathers = {1, 2, 3, 4, 5, 6, 7};
  boolean canFly = true;
  int maxSpeed = 25;

  public Duck() {
    System.out.println("type 1 duck");
  }

  public Duck(boolean fly) {
    canFly = fly;
    System.out.println("type 2 duck");
  }

  public Duck(String n,long [] f) {
    name = n;
    feathers = f;
    System.out.println("type 3 duck");
  }

  public Duck(int w, float f) {
    pounds = w;
    floatability = f;
    System.out.println("type 4 duck");
  }

  public Duck(float density, int max) {
    floatability = density;
    maxSpeed = max;
    System.out.println("type 5 duck");
  }
}
```

素朴な疑問に答えます

Q：「オブジェクトがインスタンス変数の値の初期設定を必要とするものである場合でも、デフォルト値を設定する機能を備えた引数のないコンストラクタを用意しておけば、オブジェクトの作成が容易になる」というのはわかりました。ただ、オブジェクトの種類によっては、この種のコンストラクタを用意しない方がいいものもあるのではないでしょうか。適切なデフォルト値があらかじめわかるインスタンス変数ばかりではないと思います。

A：そのとおりです。引数のないコンストラクタを持たせるべきではないクラスは少なくありません。Java APIを見ても、引数のないコンストラクタを持たないクラスがいくつも存在することがわかります。その例としてはまず、Colorクラスがあげられます。Colorクラスは、スクリーンフォントやGUIボタンの色の設定、変更などに使用されます。Colorクラスのインスタンスは必ず特定の色に対応します。そのため、Colorオブジェクトを作成する際には以下のように、対応する色をコンストラクタの引数として指定するようになっているのです。

```
Color c = new Color(3, 45, 200);
```

（上記のコードでは、int型の数値を使ってRGBの各成分を指定する、という方法を使用しています。詳細については後の章で説明します）では、引数を指定せずにコンストラクタを呼び出したらどうなるのでしょうか。Java APIの開発者の趣味で、デフォルトで「藤色」が指定される、などということはないのでしょうか。

引数なしでColorのコンストラクタを呼び出すには、以下のようなコードを書くことになります。

```
Color c = new Color();
```

このようなコードを書くと、Colorクラスに引数を持たないコンストラクタが存在しないため、コンパイル時にエラーが発生します。

```
File  Edit  Window  Help  StopBeingStupid
cannot resolve symbol
:constructor Color()
location: class
java.awt.Color
Color c = new Color();
          ^

1 error
```


This PDF is prepared for tools@maruhouse.org, personal use only.

コンストラクタの復習

① **以下のようなコードでオブジェクトが作成される時には、コンストラクタが実行される。**

```
Duck d = new Duck();
```

② **コンストラクタの名前はクラスと同じにする。戻り値の型は指定しない。**

```
public Duck(int size) { }
```

③ **プログラマが自らコンストラクタを作成しなければ、コンパイラが「デフォルトコンストラクタ」を自動的に作成する。デフォルトコンストラクタは必ず以下のように引数のないコンストラクタになる。**

```
public Duck() { }
```

④ **1つのクラスに複数のコンストラクタを持たせることもできる。その場合は、コンストラクタごとに引数を変える。複数のコンストラクタを持たせることを「コンストラクタのオーバーロード」という。**

```
public Duck() { }
public Duck(int size) { }
public Duck(String name) { }
public Duck(String name, int size) { }
```

次は「頭の体操」です。頑張って脳のニューロンを成長させてください。

頭の体操

スーパークラスのコンストラクタはサブクラスのオブジェクトが作成されると同時に実行されるのでしょうか。たとえば、何度か例にあげたDogクラスのオブジェクトが作られる際には、Canineクラスのコンストラクタも実行されるのでしょうか。
また、スーパークラスが抽象クラスである場合でも、コンストラクタを持つ必要はあるのでしょうか。

こうした事柄についてはこの後、詳しく説明していきますが、次のページに進む前に一度自分でも考えてみてください。

素朴な疑問に答えます

Q：コンストラクタは必ずpublic宣言しなければならないのですか。

A：そんなことはありません。コンストラクタのアクセスレベルはpublicでもprivateでも、default（defaultの場合はアクセス修飾子は付けない）でもかまいません。“default”というアクセスレベルに関しては、16章と付録Bで詳しく説明します。

Q：private宣言したコンストラクタは何の役に立つのですか。誰もコンストラクタを呼び出せなければ、誰もオブジェクトを作成できないということになりませんか。

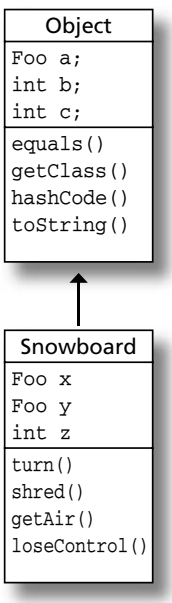
A：private宣言したからと言って「誰も」呼び出せないということはありません。そのクラスの外のコードからは呼び出せない、というだけのことです。ただ、こう言うと、小説『キャッチ=22』を思い出す人もいるかもしれません。private宣言されたコンストラクタが同じクラス内のコードからのみ呼び出せるとすると、すでにそのクラスのオブジェクトが存在しない限り、コンストラクタを呼び出すことができない、ということになるからです。では、最初のオブジェクトはどのようにして作ったのでしょうか。そのクラスの機能はどのようにして利用すればいいのでしょうか。このことについては次章で解説しますので、しばらく待ってください。

This PDF is prepared for tools@maruhouse.org, personal use only.

スーパークラスのコンストラクタはサブクラスのコンストラクタと同時に実行されるのか

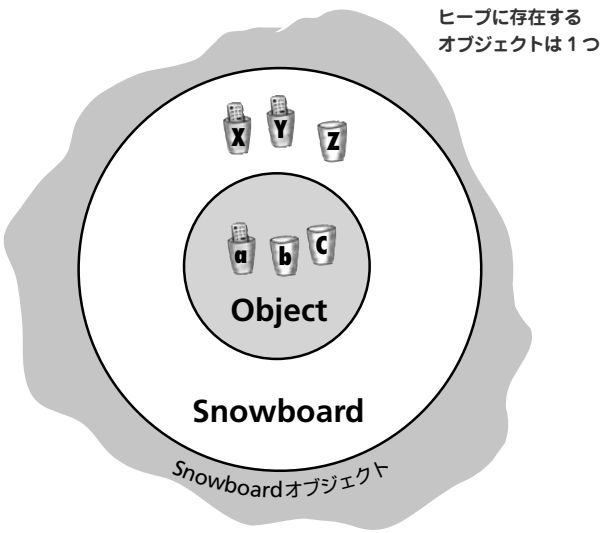
他のクラスを継承しているクラスのオブジェクトは、そのクラスに固有の部分と、スーパークラスのオブジェクトとしての部分を持つことになります。また、あらゆるクラスはObjectのサブクラスなので、どのオブジェクトも、少なくとも、そのクラスに固有の部分以外に、Objectオブジェクトとしての部分を持っています。インスタンス変数に関しても、そのクラスに独自のもの以外に、スーパークラスのものを引き継ぐのです(つまり、どのクラスのオブジェクトも最低限Objectクラスのインスタンス変数は引き継ぐということ)。

したがって、誰かが「new」というキーワードを使ってオブジェクトを作ろうとすれば、その時には上位にあるすべてのクラス(スーパークラス、さらにそのスーパークラス、など)のインスタンス変数を格納する領域がメモリに必要になります。スーパークラスの中には、インスタンス変数をprivate宣言し、セッターメソッドを設けるという「カプセル化」を行っているものもあるかもしれません。しかし、たとえprivate宣言されたインスタンス変数であってもそれをどこかに格納しなければならないのです。あるオブジェクトが作られるというのは、実はそのスーパークラスのオブジェクト、さらにそのスーパークラスのオブジェクト……という具合にいくつものオブジェクトが同時に作られるようなものです。概念的には、すべてのオブジェクトが下の絵のようにいくつかの層を成すと考えてもいいでしょう。



Objectクラスには、カプセル化されたインスタンス変数がある。このインスタンスは、サブクラスがインスタンス化される際にメモリに格納される(Objectオブジェクトそのものが実際に作成されるわけではない。カプセル化されているインスタンス変数については意識しなくてよい)

Snowboardクラスにも独自のインスタンス変数がある。Snowboardオブジェクトが作成される際には、Snowboardクラス、Objectクラス双方のインスタンス変数のための領域が必要になる



ヒープに存在するオブジェクトはSnowboardオブジェクトのみ。しかし、Snowboardオブジェクトには、Objectオブジェクトとしての部分も存在する。両方のクラスのインスタンス変数がメモリに格納される

This PDF is prepared for tools@maruhouse.org, personal use only.

オブジェクトのライフサイクルにおける スーパークラスのコンストラクタの役割

あるクラスがインスタンス化される際には、そのクラスが直接的、間接的に継承するすべてのクラスのコンストラクタが実行されます。

その点について詳しく見てみましょう。

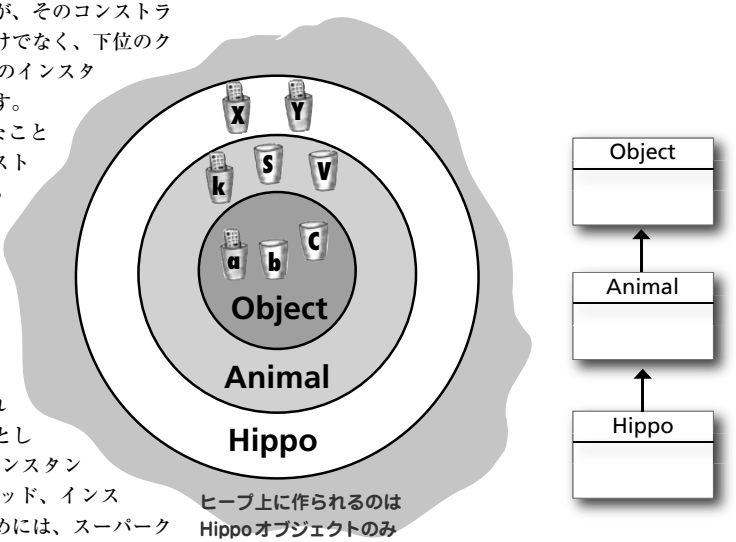
どのクラスにもコンストラクタはあるのですが、そのコンストラクタは、自らのインスタンスが作成される時だけでなく、下位のクラス（サブクラス、さらにサブクラス……など）のインスタンスが作成される際にも同時に実行されるのです。

プログラム中に「new」と書くのは実は大変なことです。「new」と書いただけで、いくつものコンストラクタが連鎖反応的に実行される可能性があるからです。注意すべきなのは「抽象クラスにもコンストラクタがある」という点です。このコンストラクタをプログラマが「new」を使って直接実行することはないのですが、下位のコンクリートクラスのコンストラクタが実行された時には同時に実行されることになります。

スーパークラスのコンストラクタが実行されるのは、オブジェクトの中の、スーパークラスとしての部分を作成するためです。サブクラスのインスタンスが作成される際には、スーパークラスのメソッド、インスタンス変数もメモリ上に格納されます。そのためには、スーパークラスのコンストラクタを実行する必要があるのです。サブクラスが継承していないインスタンス変数も、初期化され、メモリに格納されます。たとえば、Animalクラスの中に、Hippoクラスが継承していないインスタンス変数（private宣言されたものなど）があったとしても、Hippoクラスがインスタンス化される時には、そのインスタンス変数がメモリに格納されるのです。これは、Hippoが、そのインスタンス変数を利用するメソッドを必要とするためです。

コンストラクタが実行される際には、そのスーパークラスのコンストラクタ、さらにそのスーパークラスのコンストラクタ、という具合に、最終的にObjectクラスのコンストラクタに行き着くまで、順に上位のクラスのコンストラクタが実行されていきます。

ここからしばらくは、スーパークラスのコンストラクタがどのように呼び出されるのか、また自らの意志でスーパークラスのコンストラクタを呼び出すにはどうしたらよいのか、といったことを解説していきます。スーパークラスのコンストラクタに引数がある場合にはどうするのか、といったこともわかるでしょう。



Hippoオブジェクトには、Animalオブジェクトとしての部分、Objectオブジェクトとしての部分がある。Hippoオブジェクトを作る際には、そうした部分も同時に作らねばならない

コンストラクタが連鎖反応的に実行されるのはそのため

This PDF is prepared for tools@maruhouse.org, personal use only.

Hippoオブジェクトを作るには、Animalオブジェクト、Objectオブジェクトとしての部分も必要……

```
public class Animal {
  public Animal() {
    System.out.println("Making an Animal");
  }
}
```

```
public class Hippo extends Animal {
  public Hippo() {
    System.out.println("Making a Hippo");
  }
}
```

```
public class TestHippo {
  public static void main (String [] args) {
    System.out.println("Starting...");
    Hippo h = new Hippo();
  }
}
```

自分で考えてみよう

左のTestHippoを実行した場合の実行結果はA、Bどちらになるでしょうか。（答えはこのページの中にあります）

A

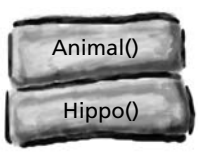
```
File  Edit  Window  Help  Swear
% java TestHippo
Starting...
Making an Animal
Making a Hippo
```

B

```
File  Edit  Window  Help  Swear
% java TestHippo
Starting...
Making a Hippo
Making an Animal
```

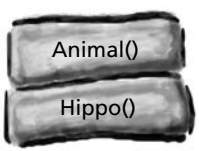
① いずれかのプログラムで「new Hippo()」というコードが実行されると、Hippo()コンストラクタがスタックの最上部のフレームに格納される。

② Hippo()はスーパークラスのコンストラクタAnimal()を呼び出す。すると、Animal()がスタックの最上部に格納（プッシュ）される。



③ Animal()は、スーパークラスのコンストラクタObject()を呼び出す。すると、Object()がスタックの最上部にプッシュされる。

④ Object()の実行が完了すると、Object()はスタックからポップされ、Animal()の実行が、Object()呼び出しの直後のコードから再開される。



実行結果はAになる。Hippo()コンストラクタが先に呼び出されるが、実行が完了するのはAnimal()が先

This PDF is prepared for tools@maruhouse.org, personal use only.

スーパークラスのコンストラクタを意図的に呼び出す場合は？

時には、コンストラクタの中でスーパークラスのコンストラクタを意図的に呼び出さねばならない、という場合もあります。そういう場合、たとえば以下のようなコードを書くことはできません。

```
public class Duck extends Animal {
  int size;

  public Duck(int newSize) {
    Animal();
    size = newSize;
  }
}
```

誤り　こういうコードは書けない

スーパークラスのコンストラクタを呼び出す場合には、必ず「super()」と書きます。
多少、奇異に感じるかもしれませんが、これで良いのです。

```
public class Duck extends Animal {
  int size;

  public Duck(int newSize) {
    super();
    size = newSize;
  }
}
```

super()と書く

コンストラクタの中でsuper()が呼び出されると、スタックの最上部にスーパークラスのコンストラクタが格納されます。そして、このスーパークラスのコンストラクタは、さらに自分の上位にあるスーパークラスのコンストラクタを呼び出します。同じことが、Objectクラスのコンストラクタがスタックに格納されるまで続けられるのです。Object()の処理が終了すれば、スタックからポップされ、すぐ下位のクラスのコンストラクタがスタックの最上部に置かれることになります。そして、そのコンストラクタの処理が完了すれば、すぐ下位にあるコンストラクタに処理が引き継がれる……という具合に、スタックの最上部に置かれるのが最初に呼び出されたコンストラクタになるまで同じことが繰り返されるのです。

super()と書かなくてもスーパークラスのコンストラクタが呼び出されるのはなぜか

その理由は簡単です。

プログラマがsuper()と書かなかった場合は、コンパイラがコンストラクタにsuper()というコードを入れます。

つまり、コンパイラは以下のふたとおりの方法でコンストラクタ作成に介入するということになります。

① プログラマがコンストラクタが作らなかった場合

この場合、コンパイラは以下に示すようなコンストラクタを自動的に作成します。

```
public ClassName() {
  super();
}
```

② プログラマがコンストラクタを作ったが、super()というコードを書かなかった場合

この場合、コンパイラは、super()というコードをコンストラクタに入れます。コンストラクタがオーバーロードされている場合は、複数のコンストラクタすべてにsuper()と入れます*。コンパイラが挿入するのは、常に、以下のように引数のないsuper()です。

```
super();
```

スーパークラスのコンストラクタがオーバーロードされている場合は、引数のないものだけが呼び出されることになります。

* ただし、コンストラクタの中で同じクラスの他のコンストラクタが呼び出されている場合には、super()は入れない（詳細については後述）。

This PDF is prepared for tools@maruhouse.org, personal use only.

親より先に子供は産まれるか？

スーパークラスを「親」、サブクラスを「子」であると考えれば、どちらが先に生まれるかはわかると思います。オブジェクトが作られる際には、まずスーパークラスに対応する部分が完全にできてから、サブクラスに対応する部分ができます。サブクラスのオブジェクトは、スーパークラスから継承する部分があってはじめて存在できるのです。できる順序が逆ということはあり得ません。つまり、スーパークラスのコンストラクタの処理は、必ずサブクラスのコンストラクタの処理より先に終わるということになります。

2ページ前の図を見るとそのことがよくわかります。最初に呼び出されるのはHippoのコンストラクタですが（このコンストラクタが最初にスタックに格納される）、実行が完了するのはこのコンストラクタが最後です。コンストラクタはそれぞれ、呼び出されると即、直接のスーパークラスのコンストラクタを呼び出します。この連鎖は、Objectクラスのコンストラクタがスタックに格納されるまで続きます。その後は、Objectクラスのコンストラクタの処理が完了すればAnimalクラスのコンストラクタ、Animalクラスのコンストラクタの処理が完了すればHippoクラスのコンストラクタ、という具合に順に実行されていきます。Hippoコンストラクタの中の、Animalコンストラクタを呼び出した後のコードが実行されるのは、Animalコンストラクタの実行が終わった後です。ここで覚えておくべきことは、

super()は必ずコンストラクタのコードの1行目に置かれる*ということです。

正しいコンストラクタ、正しくないコンストラクタ

```
☑ public Boop() {
    super();
  }

  public Boop(int i) {
☑   super();
    size =i;
  }
```

この2つのコンストラクタは、super()を明示的に1行目に配置しているのでいずれも正しい

```
☑ public Boop() {
  }

☑ public Boop(int i)
  {
    size =i;
  }

⊘ public Boop(int i)
  {
    size =i;
    super();
  }
```

この場合は、コンパイラが自動的にsuper()を1行目に配置するのでいずれも正しい

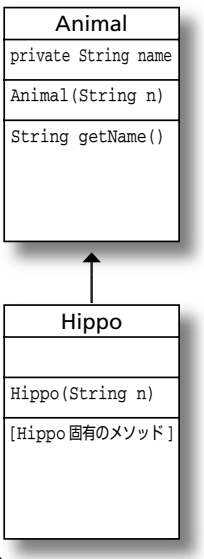
これは誤り。コンパイルができない。super()は必ず1行目に置く

* ただし、このルールには例外がある。詳しくは256ページを参照。

This PDF is prepared for tools@maruhouse.org, personal use only.

スーパークラスのコンストラクタに引数がある場合

スーパークラスのコンストラクタに引数がある場合にはどうなるのでしょうか。super() に引数を指定するということができるのでしょうか。もちろんできます。さもなければ、引数なしのコンストラクタを持たないクラスは継承できないということになってしまいます。たとえば、Animal クラスに name というインスタンス変数があるとしましょう。そして、Animal クラスの getName() というメソッドをつかって、name 変数の値が得られると仮定します。name 変数がたとえ private 宣言されていたとしても、Hippo などのサブクラスは getName() メソッドは継承します。するとここで、「Hippo クラスは getName() を持っているのに name 変数は持っていない」という矛盾が生じてしまうのです。getName() が呼び出された時に問題なく実行されるようにするためには、Animal オブジェクトとしての部分を利用するしかありません。それには、super() を使うことになります。super() を使って、自らの Animal オブジェクトとして部分に存在する name 変数に値を渡すのです。この方なら、たとえ private 宣言されたインスタンス変数であっても値を渡すことが可能なのです。




```
public abstract class Animal {
  private String name;

  public String getName() {
    return name;
  }

  public Animal(String theName) {
    name = theName;
  }
}
```

インスタンス変数 name

Hippo が継承するゲッターメソッド

name 変数に引数に指定された値を設定するコンストラクタ

```
public class Hippo extends Animal {

  public Hippo(String name) {
    super(name);
  }
}
```

Hippo のコンストラクタに引数 name を持たせる

name の値が Animal のコンストラクタとともにスタックに格納される

```
public class MakeHippo {
  public static void main(String [] args) {
    Hippo h =new Hippo("Buffy");
    System.out.println(h.getName());
  }
}
```

"Buffy" という引数を指定してコンストラクタを呼び出し、Hippo オブジェクトを作成する。

```
File  Edit  Window  Help  Hide
% java MakeHippo
Buffy
```


This PDF is prepared for tools@maruhouse.org, personal use only.

コンストラクタの中でコンストラクタを呼び出すには？

引数の型だけが異なるが、その他の処理内容はまったく同じ、というコンストラクタをいくつも作らねばならない、という場合はどうすればいいのでしょうか。個々のコンストラクタについて同じコードを繰り返し書くのは非効率ですし、保守も難しくなります。その場合は、複数のコンストラクタで必要になるコードを（super()なども含め）一度書くだけで済む方法があります。一度だけいずれかのコンストラクタにコードを書いておけば、後は他のコンストラクタからそれを呼び出すということができるのです。呼び出しには、this()というコードを使用します。「this」というのは「コンストラクタが属する、このクラス（オブジェクト）」という意味です。

this()というコードが使えるのはコンストラクタの中だけです。また、this()は必ずコンストラクタの1行目に置かねばなりません。

ただ、ここで問題になるのは、super()もコンストラクタの1行目に置かねばならないということです。これは、使用できるのは常にどちらか一方のみであるということを意味します。

this()は、コンストラクタから、同じクラスの他のコンストラクタを呼び出すのに使用します。

this()が使用できるのはコンストラクタの中だけです。また、必ず1行目に置く必要があります。

super()とthis()を同時に使用することはできません。

コンストラクタには、super()、this()のいずれかが使えるが、両方同時には使えない

```
class Mini extends Car {

  Color color;

  public Mini() {
    this(Color.Red);
  }

  public Mini(Color c) {
    super("Mini");
    color = c;
    // その他の初期化
  }

  public Mini(int size) {
    this(Color.Red);
    super(size);
  }
}
```

引数のないコンストラクタでは、colorのデフォルト値を指定して、別のコンストラクタ（super()を呼び出すコンストラクタ）を呼び出す

このコンストラクタに、super()の呼び出しを含む共通のコードを持たせる

これは誤り。super()とthis()を同じコンストラクタに書くことはできない。いずれも1行目に書く必要があるため

```
File  Edit  Window  Help  Drive
javac Mini.java
Mini.java:16: call to super must
be first statement in constructor
      super();
          ^
```


This PDF is prepared for tools@maruhouse.org, personal use only.

自分で考えてみよう

以下に示したSonOfBooクラスには、コンパイルができない誤ったコンストラクタが含まれています。コードをよく見て、どのコンストラクタが誤っているかを考えてみてください。また、右側に示したエラーメッセージから、誤ったコンストラクタに対応するものを選び、対応するエラーメッセージとコンストラクタを線で結んでください。

```
public class Boo {
  public Boo(int i) { }
  public Boo(String s) {}
  public Boo(String s, int i) {}
}
```

```
class SonOfBoo extends Boo {

  public SonOfBoo() {
    super("boo");
  }

  public SonOfBoo(int i) {
    super("Fred");
  }

  public SonOfBoo(String s) {
    super(42);
  }

  public SonOfBoo(int i, String s) {
  }

  public SonOfBoo(String a, String b, String c) {
    super(a, b);
  }

  public SonOfBoo(int i,int j) {
    super("man", j);
  }

  public SonOfBoo(int i, int x, int y) {
    super(i, "star");
  }
}
```

覚えておこう

親は子より必ず先に産まれる

オブジェクトが作成される時は、上位クラスに対応する部分から順に作られていく。これは親が子より先に産まれるようなもの

```
File  Edit  Window  Help  Blahblahblah
% javac SonOfBoo.java
cannot resolve symbol
symbol : constructor Boo
(java.lang.String,java.la
ng.String)
```

```
File  Edit  Window  Help  Yadayadayada
% javac SonOfBoo.java
cannot resolve symbol
symbol : constructor Boo
(int, java.lang.String)
```

```
File  Edit  Window  Help  ImNotListening
% javac SonOfBoo.java
cannot resolve symbol
symbol : constructor Boo()
```


This PDF is prepared for tools@maruhouse.org, personal use only.

オブジェクトの「寿命」はどのくらい？

オブジェクトの寿命は、そのオブジェクトへの「参照」の寿命によって決まります。参照が「生きている」間は、オブジェクトもヒープ上で生き続けることになります。参照が死ねば（この場合「死ぬ」というのが何を意味するかについては後述）、オブジェクトも死ぬのです。

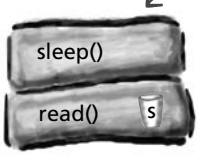
では、参照の寿命はどのくらいなのでしょうか？

それは、参照を格納する変数が、ローカル変数なのか、インスタンス変数なのかによって違ってきます。以下にローカル変数の寿命についてまとめておきます。以下で例にあげているのはプリミティブ型の変数ですが、寿命に関しては参照型の変数にも同じことが言えます。

```
public class TestLifeOne {

  public void read() {
    int s = 42;
    sleep();
  }

  public void sleep() {
    s = 7;
  }
}
```

変数sが有効なのは、read()メソッドの中でのみ。他の場所でこのsを利用することはできない

これは誤り。変数sはここでは使えない



sleep()メソッドからは変数sは使えない。sleep()が格納されているスタックフレームにsは置かれないため。sleep()は変数sには関知できない

変数sが生きるのはread()メソッドの中でのみ。sleep()の実行が完了すれば、read()がスタックの最上部に配置され、変数sが有効になる。read()の実行が完了すると、read()はスタックからポップされ、変数sは死ぬ

① ローカル変数は、宣言されたメソッド内でのみ有効

```
public void read() {
  int s = 42;
  // 変数sは
  // このメソッド内でのみ有効
  // メソッドの実行が完了すれば
  // sは消滅する
}
```

変数sは、read()メソッド内でのみ使用できる。ローカル変数は宣言されたメソッド内でのみ有効。その他のコードではsは使用できない。

② インスタンス変数は、オブジェクトが生きている限り生き続ける。オブジェクトが存在する限りはインスタンス変数も有効

```
public class Life {
  int size;

  public void setSize(int s) {
    size = s;
    // 変数sはsetSize()メソッドの中でのみ
    // 利用できる
    // しかし変数sizeは、Lifeクラスの中であれば
    // どこででも利用できる
  }
}
```

変数s（メソッドのパラメータ）は、setSize()メソッドの中でのみ有効。しかし、インスタンス変数sizeは、オブジェクト中のどこでも有効。

This PDF is prepared for tools@maruhouse.org, personal use only.

ローカル変数の「寿命」と「スコープ（有効範囲）」の違い

寿命

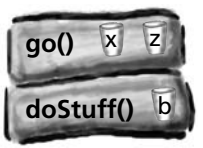
ローカル変数は、スタック上のフレームに格納されている限り生き続ける。つまり、メソッドの実行が完了するまで生き続けるということになる。

スコープ（有効範囲）

ローカル変数は、宣言されたメソッドの中でのみ有効。つまり、スコープ（有効範囲）はメソッド内に限定される。メソッドの外からは、メソッドを呼び出してローカル変数を「産まれさせる」ことはできても、利用することはできない。利用できるのはあくまで同じメソッド内でのみ。

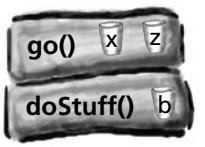
```
public void doStuff() {
  boolean b = true;
  go(4);
}

public void go(int x) {
  int z = x + 24;
  crazy();
  // その他のコード
}

public void crazy() {
 char c = 'a';
}
```

❶ doStuff()がスタックに格納される。変数bが産まれ、有効になる



❷ go()がスタックにプッシュされる。変数x、yが産まれ、有効になる。bが引き続き生きているが、有効ではない

❸ crazy()がスタックにプッシュされる。変数cが産まれ、有効になる。他の3つの変数は生き続けているが有効ではなくなっている



❹ crazy()の実行が完了し、スタックからポップされる。変数cは有効ではなくなり、死ぬ。その後、go()の実行が再開され、生き続けていた変数x、yが再び有効になる。変数bは生き続けているが、（go()の実行が完了するまでの間は）有効ではない

ローカル変数が生きている間はその値も保存されます。たとえば、doStuff()メソッドがスタック上に存在する間は、変数bの値は保存されるということです。変数bが有効になり、利用できるのは、doStuff()のフレームがスタックの最上部にある時のみです。メソッドが実行中以外はローカル変数は利用できないということです（他のメソッドの実行の完了を待っている間は利用できない）。

This PDF is prepared for tools@maruhouse.org, personal use only.

参照型変数の場合

寿命とスコープに関するルールは、プリミティブ型の変数でも、参照型の変数でも同じです。ただ、参照型の変数が利用できない、ということは、対応するオブジェクトが利用できない、ということを意味するので注意が必要です。参照を格納した変数が生きていて、有効になっていない限り、オブジェクトは利用できないのです。

変数の寿命とオブジェクトの寿命との関係は？

オブジェクトは、参照を格納する変数が生きている限りは生き続けます。たとえ、変数が有効ではなくなったとしても、生き続けてさえいれば、オブジェクトが死ぬことはありません。では、メソッドの実行が完了し、参照を格納している変数がスタックからポップされたら、どのようなことが起きるのでしょうか。

もしオブジェクトへの参照が1つしかなかった場合、その参照がなくなれば、オブジェクトはいったんヒープ上で放置された状態になります。スタックフレームとの結び付きがなくなったオブジェクトはそれ以後、「不要なもの」として扱われます。そして、不要なものになったオブジェクトは、ガーベージコレクションができる状態となるのです。

不要になったオブジェクトが使っていたメモリ領域の解放などについては、プログラマが関知する必要はありません。メモリが足りなくなれば、ガーベージコレクションによって自動的に不要なオブジェクトが破壊され、メモリが解放されるからです。もちろん、それでもメモリ不足の危険性は残りますが、少なくとも不要なオブジェクトの破棄が遅れたという理由でメモリ不足に陥るということはないのです。メモリ不足を回避するためにプログラマとして注意すべきことは、使い終わったオブジェクトはできるだけ早くガーベージコレクション可能な状態にするということでしょう。すでに使い終わったオブジェクトへの参照を残すようなことをすると、ガーベージコレクションができなくなってしまい、メモリが不足する可能性が高くなります。

参照がまったく存在しなければ、オブジェクトに存在する価値はまったくない
参照の存在しないオブジェクトは、何の仕事もすることもなく、ただメモリを消費しているだけ
Javaの場合、参照が存在しなくなると、ガーベージコレクタがそれを察知して、オブジェクトを廃棄する

オブジェクトは、参照がすべてなくなると、ガーベージコレクションが可能な状態になる

オブジェクトへの参照は、以下のような場合に消滅する

① 参照が格納された変数が有効でなくなり、しかも以後有効になる可能性がなくなる

```
void go() {
  Life z = new Life();
}
```

zに格納されていた参照がメソッド終了とともに消滅

② 変数に別の参照が代入される

```
Life z = new Life();
z = new Life();
```

zに元々格納されていた参照は、新たな参照が格納されることで消滅する

③ 変数に明示的にnullが代入される

```
Life z = new Life();
z = null;
```

zに元々格納されていた参照は、nullが格納されることで消滅する

This PDF is prepared for tools@maruhouse.org, personal use only.

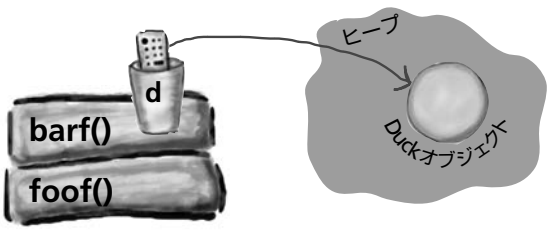
オブジェクトへの参照の消滅 パターン1

参照が格納された変数が有効でなくなり、しかも以後有効になる可能性がなくなる。

```
public class StackRef {
  public void foof() {
    barf();
  }

  public void barf() {
    Duck d = new Duck();
  }
}
```

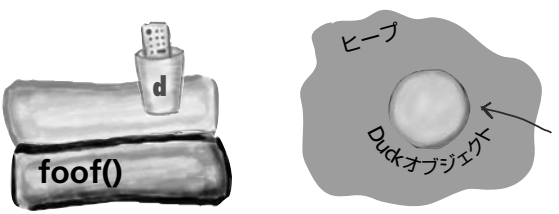
1 何の変数も宣言されていないfoof()メソッドがスタックにプッシュされる。

2 参照型の変数が宣言されたbarf()メソッドがスタックにプッシュされる。変数には、新たに作成されたオブジェクトへの参照が格納される。オブジェクトはヒープ上に配置され、変数は有効な状態になる。



新しいDuckオブジェクトがヒープに格納される。barf()メソッドが実行されている間は、変数dが生きており、有効になっているので、Duckオブジェクトも「生きた」状態になる

3 barf()メソッドの処理が完了し、スタックからポップされる。変数dに格納されていた参照は消滅する。その後、foof()の実行が継続されるが、foof()では変数dは利用できない。



barf()メソッドがスタックからポップされると、変数dと格納された参照が消滅する。Duckオブジェクトは放置された状態になり、いつでもガーベージコレクションできるようになる

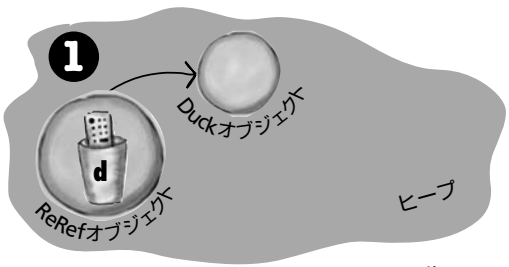
This PDF is prepared for tools@maruhouse.org, personal use only.

オブジェクトへの参照の消滅 パターン 2

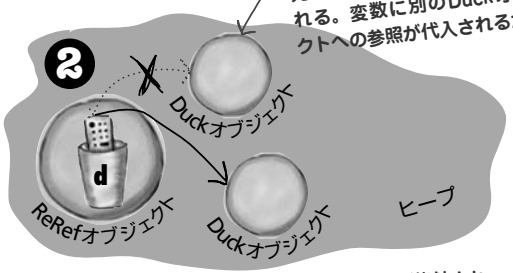
変数に別の参照が代入される

```
public class ReRef {

  Duck d = new Duck();

  public void go() {
    d = new Duck();
  }
}
```

なるほど。参照を「リセット」すればいいわけだな。これでメモリのことは気にしなくていい、と……



新しく作成されたDuckオブジェクトがヒープに格納され、変数dに参照が代入される。dはインスタンス変数なので、DuckオブジェクトはReRefオブジェクトが生きている限り生き続けることになる。ただし……



go()メソッドが呼び出されると、元のDuckオブジェクトは放棄される。変数に別のDuckオブジェクトへの参照が代入されるため

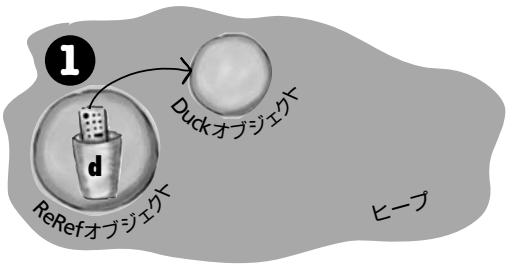
変数dに新たなDuckオブジェクトへの参照が格納される。元のDuckオブジェクトは参照を失い、事実上、死んだ状態になる

This PDF is prepared for tools@maruhouse.org, personal use only.

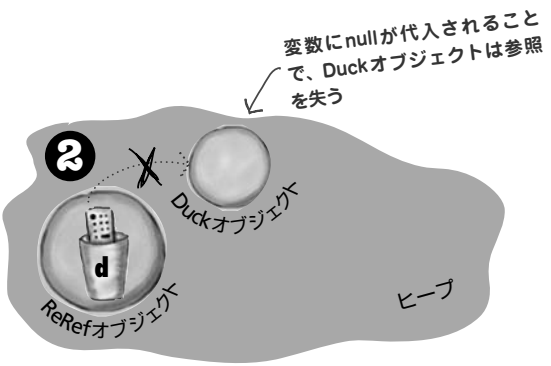
オブジェクトへの参照の消滅 パターン3

変数に明示的にnullが代入される

```
public class ReRef {

  Duck d = new Duck();

  public void go() {
    d = null;
  }
}
```



新しく作成されたDuckオブジェクトがヒープに格納され、変数dに参照が代入される。dはインスタンス変数なので、DuckオブジェクトはReRefオブジェクトが生きている限り生き続けることになる。ただし……



変数dにnullが代入されると、対応するオブジェクトがなくなる。変数dは何らかのオブジェクトへの参照が改めて代入されない限り、利用できない

nullの意味

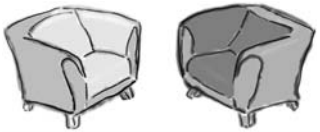
参照型の変数にnullを代入すると、対応するオブジェクトがなくなります。これは、リモコンはあってもそれによって操作できるテレビがなくなるようなものです。ただ、nullも、対応するオブジェクトがないだけで、「値」であることに違いはない、という点には注意が必要です（具体的にどのような値かは、プログラマが知る必要はなく、JVMさえ「どの値がnullか」を認識できればよい）。
リモコンに対応するテレビがなければ、リモコンのボタンを押しても何もできないということになります。Javaの場合は、参照型の変数にnullが代入されると、「.」演算子を使っても何もできません。たとえば、メソッドを呼び出すことなどもできないのです（ただし、nullの代入された変数を使ってメソッドを呼び出すコードを書いても、コンパイル段階ではエラーにはならず、実行時に問題が起きる）。

nullの代入された参照型変数を使うと、実行時にNullPointerExceptionという例外が発生します。

例外の詳細については11章で触れます。

This PDF is prepared for tools@maruhouse.org, personal use only.

特別座談会



今夜は、インスタンス変数さんとローカル変数さんにお越しいただき、生と死についてじっくり語っていただきました。

インスタンス変数：まず私から話しましょうか。プログラムにおいては、どうしてもローカル変数さんより私の方が重要ですからね。オブジェクトが生きている限り、そばにいてサポートし続けるわけですから。私たちインスタンス変数に格納される値がないオブジェクトなんて意味がないですよ。

ローカル変数：あなたのおっしゃることも、まあわかります。インスタンス変数さんは本当に重要なものですから。しかし、その言い方は誤解を生むと思いますよ。ローカル変数だって負けず劣らず重要なんです。あなたの言い方を借りれば、「メソッドのないオブジェクトなんて意味がない」ということになりますが、そのメソッドのアルゴリズムを機能させるのが他ならぬ私たちローカル変数なんですよ。

インスタンス変数：それは失礼しました。悪く思わないでください。私もローカル変数さんのメソッドの中での役割は十分にわかっています。ただ、ローカル変数さんの寿命はあまりに短い。はかない存在です。「一時変数」なんて呼ばれることもあるみたいですね。

ローカル変数：ローカル変数にとって、その「一時変数」という呼び方は屈辱的なものです。せめて「スタック変数」あるいは「自動変数」と呼んでもらいたいですね。

インスタンス変数：申し訳ありません。了解しました。

ローカル変数：とはいえ、我々の寿命が長くないのは確かです。それに、多くの場合、「いい人生」は送れないですね。産まれてすぐに他のローカル変数と同じスタックフレームに詰め込まれてしまいますし、他のメソッドが呼ばれると、新たなフレームが上からのしかかってきます。メソッドが呼び出される度に上からの圧力は大きくなるんです。上に乗ったフレーム中のメソッドの実行が完了し、すべてがポップされるまで永遠とも思える時間をじっと動かずに待ち続けなければならないこともありますね。

インスタンス変数：そんなことは思ってもみませんでした。他のメソッドが実行されている間、スタックの最上部に戻れるまでの間は何をしているんですか。

ローカル変数：何も。まったく何もしません。ただ、じっとしているんです。SF映画で長い宇宙旅行などをする時に乗組員が椅子に固定されるシーンがあるでしょう。そんな感じですよ。ただ、フレームが存在する限りは我々も我々の保持する値も安全は保証されているんです。ですから、フレームがスタックの最上部に戻り、メソッドが再び動きだすと複雑な気持ちですね。一方では、活動できるというううれしさもあるんですが、もう一方では、我々の短い

This PDF is prepared for tools@maruhouse.org, personal use only.

一生の終わりに向かってのカウントダウンが始まるとも言えるからです。動いている時間が長くなるほど、我々の属するメソッドの実行の完了が近づくのです。メソッドの実行が完了すれば、自分がどうなるかはわかっているわけですし。

インスタンス変数：そのことについては私も知っています。ひどい話ですよね。メソッドの実行が完了すれば、フレームはスタックから追い出されるんですから。つらいです。

ローカル変数：おっしゃるとおりです。コンピュータサイエンスの世界では「ポップ」なんて楽しげな言葉を使うようですが、とんでもないです。あなたがこの事実を知っていてくださったのはうれしいですね。あなたのことについてもお話くださいませんか。我々は小さいスタックフレームに閉じこめられるわけですが、あなたはどこに？

インスタンス変数：私はオブジェクトとともにヒープに置かれます。まあ、正確に言えば、オブジェクトの「中」にいるということになります。私はオブジェクトに関する情報を保持します。こう言ってはなんですが、ヒープの中での生活はかなり贅沢で、我々の多くは罪悪感を持っているんですよ。特に休日には罪悪感が大きくなりますね。

ローカル変数：でも、あなたもオブジェクトほどは長生きできないんですよね。たとえば、DogクラスにCollarオブジェクトへの参照を格納するインスタンス変数が、さらにCollarクラスにBuckleオブジェクトへの参照を格納するインスタンス変数があったとします。普通に参照が格納されているうちは平和ですが、Dogオブジェクトが新たなCollarオブジェクトを必要とした場合や、Collarオブジェクトを必要としなくなった場合には、確かCollarオブジェクトへの参照を格納するインスタンス変数はガーベージコレクションの対象になってしまうはずです。それに、Collarオブジェクトが不要になれば、Collarオブジェクトのインスタンス変数もガーベージコレクションによって処分されてしまいます。

インスタンス変数：そのとおりです。私がもしCollarオブジェクトのインスタンス変数なら、Collarオブジェクトがガーベージコレクションの対象になれば、私もピザの箱のように捨てられるんです。しかし、そういうことはめったに起きないと聞いています。

ローカル変数：本当に信じているんですか。やる気をなくさせないため、生産性を下げないための方便だと思いますよ。それに、あなたは1つ忘れていますよ。自らの属するオブジェクトへの参照がローカル変数に格納された場合はどうなるんですか。そうなれば、あなたは否応なしにローカル変数と運命を共にすることになります。まあ、もうこの話はやめましょう。どうです？：これから一杯やって、RAMの愚痴でも……。

インスタンス変数：RAMに見つかったら大変ですね。

This PDF is prepared for tools@maruhouse.org, personal use only.

エクササイズ

ガーベージコレクタになってみよう

左のソースコードを、ガーベージコレクタになったつもりで見てみましょう。そして、右側の 1 ～ 9 のコードの中から、A の部分に当てはめた時に、オブジェクトを 1 つだけ新たにガーベージコレクションの対象にできるものを選んでください。選ぶのは 1 つとは限りません。

```
public class GC {
  public static GC doStuff() {
    GC newGC = new GC();
    doStuff2(newGC);
    return newGC;
  }

  public static void main(String [] args) {
    GC gc1;
    GC gc2 = new GC();
    GC gc3 = new GC();
    GC gc4 = gc3;
    gc1 = doStuff();
```

A

```
    // その他のメソッド呼び出し
  }

  public static void doStuff2(GC copyGC) {
    GC localGC = copyGC;
  }
}
```

1 `copyGC = null;`

2 `gc2 = null;`

3 `newGC = gc3;`

4 `gc1 = null;`

5 `newGC = null;`

6 `gc4 = null;`

7 `gc3 = gc2;`

8 `gc1 = gc4;`

9 `gc3 = null;`

This PDF is prepared for tools@maruhouse.org, personal use only.

エクササイズ

人気者をさがせ

以下のコードの中ではいくつかのオブジェクトが作られます。そのオブジェクトのうち、最も人気があるもの、つまり最も「参照」の数が多いものはどれでしょうか。そのオブジェクトにいくつの参照があるか、またその参照はどの変数に格納されているかをリストにしてみてください。コードには、ヒントも書いておきました。

```
class Bees {
  Honey [] beeHA;
}

class Raccoon {
  Kit k;
  Honey rh;
}

class Kit {
  Honey kh;
}

class Bear {
  Honey hunny;
}

public class Honey {
  public static void main(String [] args) {
    Honey honeyPot = new Honey();
    Honey [] ha = {honeyPot, honeyPot, honeyPot, honeyPot};
    Bees b1 = new Bees();
    b1.beeHA = ha;
    Bear [] ba = new Bear [5];
    for (int x=0; x < 5; x++) {
      ba [x] = new Bear();
      ba [x].hunny = honeyPot;
    }
    Kit k = new Kit();
    k.kh = honeyPot;
    Raccoon r = new Raccoon();

    r.rh = honeyPot;
    r.k = k;
    k = null;
  } // mainの終了
}
```

Raccoonオブジェクトが作成されている

Raccoonオブジェクトへの参照が変数rに格納されている

This PDF is prepared for tools@maruhouse.org, personal use only.

「もう 4 回もシミュレーションしているんですが、どうしてもメインモジュールの温度が予定より低くなってしまうんです」Sarahがイライラした様子で言った。「先週、プログラムを新しくしたんです。居住区域を冷やすラジエータプログラムは仕様どおりになっているようだったんで、逆に居住区域を温める保温プログラムの方を集中的に調べていたんですが……」Tomはため息をついた。当初は彼らの技術力を持ってすれば前倒しで進むと考えられていたプロジェクトだった。しかし、打ち上げまで 5 週間と迫った今になっても、主要なライフサポートシステムがシミュレーションテストを通過していない。

「V3Radiator、V2Radiator、RetentionBotがどのくらいの比率で動くと想定してシミュレーションをしているんですか」とTomは尋ねた。

5 分間ミステリー

「その点については十分に検討したので間違いはないと思います」とSarahは言った。「管制センターの承認も得ていますし。我々は、V3Radiatorが 2 に対し、V2Radiatorが 1 の比率で動くものと考えています。V2Radiator、V3Radiatorと、RetentionBotを対比すれば、だいたい 4 対 3 ということになるでしょう」

「電力消費量はどうですか」とTomが尋ねると、Sarahはしばらく間を置いて答えた。「そういえば、電力消費量は予想を上回っていますね。その点についても調査してみたんですが、何しろ、冷却のための電力消費量と保温のための電力消費量の切り分けが非常に困難なので……、仕様では、冷却のための電力消費量の方が 3 対 2 で多くなるはずなのですが」

Sarahの話を聞いていたTomは言った。「なるほど。じゃあともかくシミュレーションのためのコードを一緒に確認してみましょう。急いで問題点を見つけ出さなくては」

```
import java.util.*;
class V2Radiator {
  V2Radiator(ArrayList list) {
    for(int x=0; x<5; x++) {
      list.add(new SimUnit("V2Radiator"));
    }
  }
}

class V3Radiator extends V2Radiator {
  V3Radiator(ArrayList lglist) {
    super(lglist);
    for(int g=0; g<10; g++) {
      lglist.add(new SimUnit("V3Radiator"));
    }
  }
}

class RetentionBot {
  RetentionBot(ArrayList rlist) {
    rlist.add(new SimUnit("Retention"));
  }
}
```


This PDF is prepared for tools@maruhouse.org, personal use only.

```
public class TestLifeSupportSim {
  public static void main(String [] args) {
    ArrayList aList = new ArrayList();
    V2Radiator v2 = new V2Radiator(aList);
    V3Radiator v3 = new V3Radiator(aList);
    for(int z=0; z<20; z++) {
      RetentionBot ret = new RetentionBot(aList);
    }
  }
}

class SimUnit {
  String botType;
  SimUnit(String type) {
    botType = type;
  }
  int powerUse() {
    if ("Retention".equals(botType)) {
      return 2;
    } else {
      return 4;
    }
  }
}
```

Tomはコードを少し見ると、口元にかすかに笑いを浮かべて言った。「何が問題なのかだいたいわかりましたよ。このコードだと、確かに電力消費量が予想を上回ることになるでしょうね。予想を何パーセント上回るかもわかりますよ」

Tomはどのようなことに気付いたのだろうか。なぜ、電力消費量が予想をどのくらい上回るのか、ということまでわかるのだろうか。また、上記のプログラムに少しコードを追加すれば、デバッグが容易になるが、それはどのようなコードだろうか。

This PDF is prepared for tools@maruhouse.org, personal use only.

エクササイズの答え

ガーベージコレクタになってみよう

1 `copyGC = null;` × – copyGCという変数は有効なものではない。

2 `gc2 = null;` ○ – このコードによって参照が存在しないオブジェクトができる。

3 `newGC = gc3;` × – newGCという変数は元々有効なものではない。

4 `gc1 = null;` ○ – newGCは有効ではないため、gc1 に格納されているのはオブジェクトにとって唯一の参照。

5 `newGC = null;` × – newGCは有効ではない。

6 `gc4 = null;` × – gc3 にまだ同じオブジェクトへの参照が残っている。

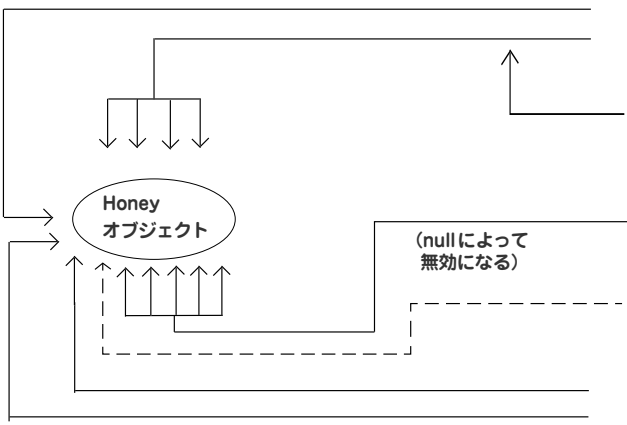
7 `gc3 = gc2;` × – gc4 にまだ同じオブジェクトへの参照が残っている。

8 `gc1 = gc4;` ○ – gc4 に格納されていた参照をgc1 に代入することで、gc1 に元々対応していたオブジェクトは参照を失う。

9 `gc3 =null;` × – gc4 にまだ同じオブジェクトへの参照が残っている。

人気者をさがせ

Honeyオブジェクトへの参照（honeyPot変数などに格納されているもの）が圧倒的に多いということはおそらくすぐにわかるでしょう。ただ、同じHoneyオブジェクトへの参照が正確にいくつあるかは少しわかりにくいかもしれません。慎重に数えるとわかりますが、main()メソッドの完了時点では、Honeyオブジェクトへの有効な参照の数は合計で 12 になります。注意すべきなのは、k.khに格納される参照が最終的にはnullによって無効になってしまうという点です。また、明示的にHoneyオブジェクトへの参照が格納されるわけではないのでわかりにくいですが、r.k.khはHoneyオブジェクトを参照しています。




```
public class Honey {
  public static void main(String [] args) {
    Honey honeyPot = new Honey();
    Honey [] ha = {honeyPot, honeyPot,
                   honeyPot, honeyPot};
    Bees b1 = new Bees();
    b1.beeHA = ha;
    Bear [] ba = new Bear [5];
    for (int x=0; x < 5; x++) {
      ba [x] = new Bear();
      ba [x].hunny = honeyPot;
    }
    Kit k = new Kit();
    k.kh = honeyPot;
    Raccoon r = new Raccoon();

    r.rh = honeyPot;
    r.k = k;
    k = null;
  } //mainの終了
}
```


This PDF is prepared for tools@maruhouse.org, personal use only.

5分間ミステリー

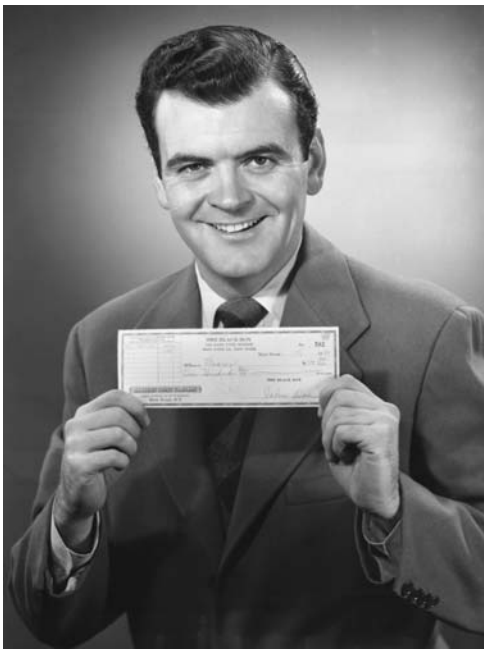
Tomは、V2Radiatorクラスのコンストラクタの引数がArrayListになっていることに気付いたのである。これは、V3Radiatorクラスのコンストラクタが呼び出されるたびに、super()を通じてV2Radiatorクラスのコンストラクタに ArrayListが渡されることを意味する。本来は、V3Radiatorオブジェクトが作られる際に、インスタンス変数botTypeに“V3Radiator”が指定されたSimUnitオブジェクトだけを作りたいのだが、これでは、botTypeに“V2Radiator”が指定された余分なSimUnitオブジェクトも同時にできてしまう。これによって、電力が予想より多く消費されているかのような実行結果が得られることになる。

V2Radiator、V3Radiator、RetentionBotといったクラスがいずれも SimUnitオブジェクトを作るようになっているため、SimUnitクラスのコンストラクタに、オブジェクトが作られるたびにその旨を画面に表示するコードを入れておけば、問題が容易に発見できる。

This PDF is prepared for tools@maruhouse.org, personal use only.

This PDF is prepared for tools@maruhouse.org, personal use only.

10章　数値の扱いとスタティック変数、メソッド
数値の扱い



この章ではJavaにおける数値の扱いについて解説します。「数値の扱い」には、具体的には、絶対値を求める、丸め（概算）を行う、2つの数値の大小を比較する、などさまざまなことがらが含まれます。小数点以下の桁数を正確に定めて出力するなども必要になるかもしれませんし、桁数の大きい数値には、読みやすいよう、途中にカンマを入れて出力することなども必要になるでしょう。日付の扱いも問題です。日付と一口に言ってもその表示方法はさまざまだからです。また、「現在の日付に3週間足すと何日になるか」と言ったこともわかれば便利です。その他、String（文字列）から数値、あるいは数値からStringへの変換についても知っておく必要があるでしょう。幸い、Java APIにはそうしたことを容易に行えるメソッドが用意されています。ただ、問題は、そうしたメソッドの多くが「スタティック」なものであるということです。まずはこの「スタティック」という言葉の意味から明らかにしていくことにしましょう。Javaにおいて変数（定数も含む）やメソッドがスタティックであるとはどういうことであるか、がわかると思います。

This PDF is prepared for tools@maruhouse.org, personal use only.

Mathクラスのメソッド：「グローバル的」なメソッド

すでに述べたとおり、Javaには「グローバル」な変数、メソッドはありません。しかし、メソッドの中には、オブジェクトのインスタンス変数の値が機能に影響しないというものもあり得ます。Mathクラスのメソッド、round()などはそうしたメソッドの例で、インスタンス変数の値に関係なく、常に同じ処理（メソッドの引数として指定された浮動小数点数を最も近い整数に変える）を行います。本当に常に同じなのです。たとえば、Mathオブジェクトが1万個あったとして、そのそれぞれを利用してround(42.2)というメソッドを行えば、すべてで「42」という同じ実行結果が得られます。このメソッドは、単に引数として指定された値を操作するだけで、インスタンス変数とは何の関係もないのです。round()メソッドの動作を変えるには、引数を変えるしかありません。

round()メソッドを実行するためだけに逐一Mathクラスのインスタンスを作るのはヒープ領域の大変な無駄遣いではないでしょうか。同じことは、min()、max()、abs()といったMathクラスの他のメソッドにも言えます（min()は2つの引数のうち小さい方を戻すメソッド。max()はその逆。abs()は引数として指定された数値の絶対値を戻すメソッド）。

こうしたメソッドは決してインスタンス変数の値を利用しないのです。そもそもMathクラスにはインスタンス変数がありません。つまり、Mathクラスのインスタンスを作成しても何のメリットもないのです。では、どうすればいいのでしょうか。実はMathクラスのインスタンスは作る必要がありません。また、作ろうとしても作ることができないようになっています。

以下のようなコードでMathクラスのインスタンスを作ろうとすると、

```
Math mathObject = new Math();
```

下に示すようなエラーが発生する。

```
File  Edit  Window  Help  IwasToldThereWouldBeNoMath
% java TestMath
TestMath.java : 3: Math() has private
access in java.lang.Math
     Math mathObject = new Math()
                       ^
1 error
```

これは、「Mathのコンストラクタがprivate宣言されている」というエラー。つまり、「new Math()」と書いてもMathオブジェクトは作れないということ

Mathクラスのメソッドはインスタンス変数を利用しない。また、Mathクラスのメソッドは「スタティック」なものなので、メソッドを使うのにMathクラスのインスタンスを作る必要はない。ただ、以下のようなコードを書けばメソッドが使える。

```
int x = Math.round(42.2);
int y = Math.min(56,12);
int z = Math.abs(-343);
```

どのメソッドもインスタンス変数を利用しないのでオブジェクトも必要としない

This PDF is prepared for tools@maruhouse.org, personal use only.

通常のメソッドと スタティックメソッドの違い

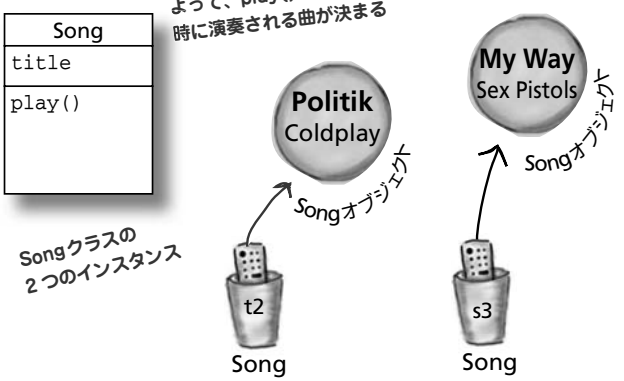
Javaは「オブジェクト指向言語」なのですが、時には、クラスのインスタンス（オブジェクト）が必要ではない、という場合があります。たとえば、前ページのMathクラスのメソッドのような「ユーティリティメソッド」を使用する場合などがそうです。メソッドを、インスタンス（オブジェクト）を作らずに実行できるようにするには、「static」というキーワードを使用します。そのようなメソッドを「スタティックメソッド」と呼びます。スタティックメソッドの特徴は、インスタンスを必要としないということです。

通常の（スタティックでない）メソッド

```
public class Song {
  String title;
  public Song(String t) {
    title = t;
  }
  public void play() {
    SoundPlayer player = new SoundPlayer();
    player.playSound(title);
  }
}
```

インスタンス変数の値がplay()メソッドの動きに影響を与える

インスタンス変数titleの値によって、play()が呼び出された時に演奏される曲が決まる




```
t2.play();
```

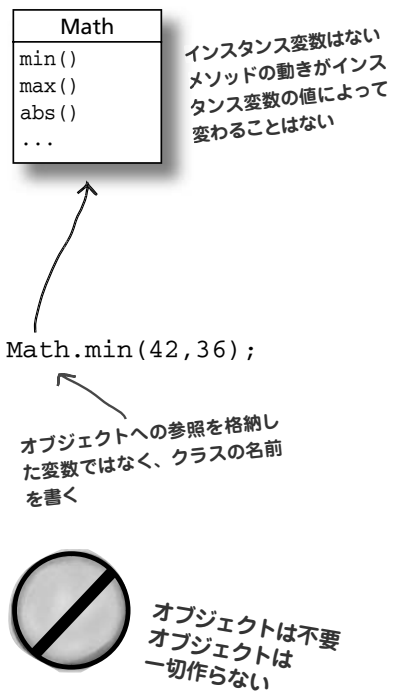
オブジェクトへの参照を使ってplay()を呼び出し、"Politik"を演奏する

```
t3.play();
```

オブジェクトへの参照を使ってplay()を呼び出し、"My Way"を演奏する

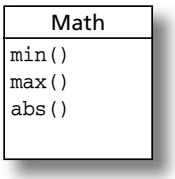
スタティックメソッド

```
public static int min(int a, int b) {
  // a、bのうち小さい方の値を戻すためのコード
}
```



This PDF is prepared for tools@maruhouse.org, personal use only.

**スタティックメソッドは
クラス名で呼び出す**




```
Math.min(88,86);
```

**通常のメソッドはオブジェクトへの
参照を格納した変数名で呼び出す**

```
Song t2 = new Song();
t2.play();
```

スタティックメソッドを持つクラスを作る

注意すべきなのは、スタティックメソッドを持つクラスのすべてがインスタンス化できないというわけではないことです。8 章では抽象クラスについて触れました。「abstract」という修飾子を使うと、そのクラスのインスタンスを作ることはできません。つまり、抽象クラスのインスタンスは作れないということです。

しかし、クラスのインスタンスを作れなくする方法は他にもあります。コンストラクタを private 宣言するという方法です。「private」というキーワードを使えば、コンストラクタは同じクラス内でのみ呼び出せるものになります。他のクラスのコードではそのコンストラクタを呼び出せないのです。Math クラスのインスタンスが作れないのは、コンストラクタが private 宣言されているためです。「private」と書かれていれば、コンパイラが、そのコンストラクタには外部からアクセスできないものと判断します。

クラスにスタティックメソッドがある、というだけではインスタンス化できないとは限りません。そのことは、main() メソッドはすべてスタティックメソッドであるということを考えただけでわかると思います。

通常、main() メソッドは、他のクラスをインスタンス化し、そのメソッドを呼び出すために使用されます。他のクラスのテストをする目的で使用されることもあります。

1 つのクラスの中には通常のメソッドとスタティックメソッドを混在させることができます。通常のメソッドが 1 つでもあれば、そのクラスは何らかの方法でインスタンス化できなくてはなりません。クラスをインスタンス化するには、「new」を使うか、デシリアライゼーション（詳しくは後述）を行うしかありません。（Java Reflection API も使えるが、本書では詳述しない）ただ、後でわかりますが、「new」を使う場合は、それを「どこで使うか」が問題になります。

This PDF is prepared for tools@maruhouse.org, personal use only.

スタティックメソッドでは スタティックでない変数（インスタンス変数）は利用できない

スタティックメソッドは、クラスのインスタンス変数の値には一切関知せずに動きます。また、すでに見てきたとおり、スタティックメソッドが実行される際には、そのメソッドの属するクラスのインスタンスがまったく存在しないことも多いのです。スタティックメソッドを呼び出す際には、オブジェクトリファレンスを格納した変数名ではなく、クラス名を使用します（t2 .play() というようなコードは使わず、Math.random() などのようなコードを使うということ）。スタティックメソッドの中でインスタンス変数を使用すると、その変数がどのインスタンスに属するものなのかがわからないため、コンパイル時にエラーが発生します。

スタティックメソッドでインスタンス変数を利用しようとすると、その変数がどのインスタンスに属するものかがわからないため、コンパイルでエラーになる。たとえヒープ上に 10 個の Duck オブジェクトが存在したとしてもスタティックメソッドでは利用できない

以下のコードをコンパイルすると……

```
public class Duck {

  private int size;

  public static void main (String [] args) {
    System.out.println("Size of duck is " + size);
  }

  public void setSize(int s) {
    size = s;
  }
  public int getSize() {
    return size;
  }
}
```

どのDuckオブジェクトのsize変数かがわからない

ヒープ上にDuckオブジェクトが存在したとしても、このコードでは利用できない

```
File  Edit  Window  Help  Quack
% javac Duck.java
Duck.java: 6: non-static variable
size cannot be referenced from a
static context
      System.out.println("Size
of duck is " + size);
                ^
```


This PDF is prepared for tools@maruhouse.org, personal use only.

スタティックメソッドの中で通常のメソッドを呼び出すことはできない

スタティックメソッドでない通常のメソッドは、多くの場合、インスタンス変数の値によって動きが変わります。たとえば、getName()というメソッドは、nameという変数の値を戻すメソッドなので、どのオブジェクトを利用して呼び出すかによってnameの値は変わります。つまりメソッドの戻す値が変わるということです。

コンパイルできない誤ったコード……

getSize()を呼び出すとインスタンス変数sizeを使うことになる

```
public class Duck {

  private int size;

  public static void main (String [] args) {
    System.out.println("Size is" + getSize());
  }

  public void setSize(int s) {
    size = s;
  }
  public int getSize() {
    return size;
  }
}
```

結局、どのインスタンスのsize変数かわからない

```
File  Edit  Window  Help  Jack-in
% javac Duck.java
Duck.java : 6 : non-static method
getSize() cannot be referenced
from a static context
        System.out.println("Size
of duck is " + getSize());
                ^
```

覚えておこう

スタティックメソッドではインスタンス変数は利用できない
スタティックでないメソッドも利用できない

素朴な疑問に答えます

Q：スタティックメソッドから通常のメソッドを呼び出すことは、通常のメソッドでインスタンス変数をまったく利用しない場合でも不可能ですか。やはりコンパイルエラーになってしまうのでしょうか。

A：不可能です。人間の目から見れば、メソッドでインスタンス変数が利用されているかどうかはすぐにわかりますが、コンパイラにはそれはわかりません。もし、インスタンス変数をたまたま利用していないからという理由で通常のメソッドの呼び出しを認めてしまうと、万一、将来メソッドにインスタンス変数を利用するような改造を行った場合に困ったことになってしまいます。また、そのメソッドがサブクラスでオーバーライドされ、インスタンス変数を利用するものになる恐れもあります。

Q：以前、スタティックメソッドをオブジェクトへの参照を利用して呼び出してみたことがあります。特に問題なく呼び出せたようですが、いけないんでしょうか。

A：実を言えば、参照を利用してスタティックメソッドを呼び出すことは絶対に不可能とは言えません。ただ、できるからといって、そうすることが望ましいかというと、それはまた別の話になります。たとえ参照（インスタンス）を利用して呼び出せたとしても、コードが誤解されやすく、わかりくいものになってしまう恐れがあるからです。たとえば、

```
Duck d = new Duck();
String [] s = {};
d.main(s);
```

このコードは誤りとは言えません。dにはDuckオブジェクトへの参照が格納さえており、そのDuckオブジェクトのmain()メソッドが呼び出されているということは、コンパイラにも理解できるからです。ただ、このような方法でmain()メソッドを呼び出したからといって、main()メソッドはやはりスタティックメソッドには変わりなく、dに対応するオブジェクトに関する情報を持つことはないのです。ただ呼び出し方を普通と変えたというだけのことで、スタティックメソッドはあくまでスタティックメソッドなのです。

This PDF is prepared for tools@maruhouse.org, personal use only.

スタティック変数：クラスのすべてのインスタンスで値が同じになる変数

たとえば、プログラム実行中にDuckクラスのインスタンスがいくつ作成されたかをカウントしなければならないとします。どうすれば可能になるでしょうか。以下のように、コンストラクタの中で値がインクリメントされるインスタンス変数を作ればいいのではないか、と思う人がいるかもしれません。

```
class Duck {
  int duckCount = 0;
  public Duck() {
    duckCount++;
  }
}
```

Duckオブジェクトが作成されるたびにduckCountの値が1増える？

しかし、これではうまくいきません。duckCountはインスタンス変数なので、オブジェクト作成のたびに初期化され、いくつDuckオブジェクトが作成されようと、値は結局1になるだけです。オブジェクトが作成されるたびに加算されていくわけではないのです。この場合、必要なのは、すべてのインスタンスによって共有できる変数です。

このようにすべてのインスタンスによって共有できる変数を「スタティック変数」と呼びます。スタティック変数の値は、インスタンスごとではなく、クラスごとに設定されます。

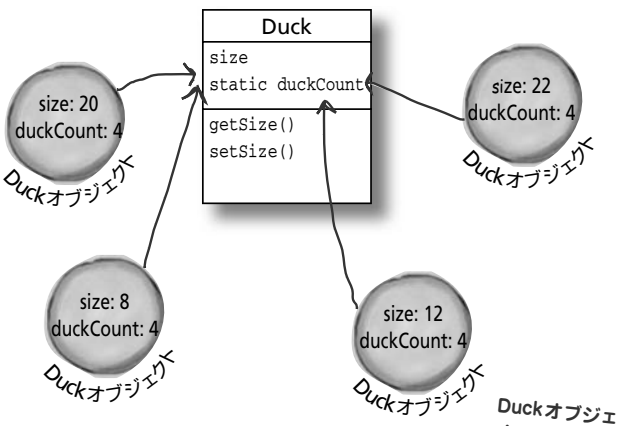
duckCountをスタティック変数にすれば、初期化は、最初にインスタンスが作成される時にのみ行われる。その後はいくつインスタンスが作成されても初期化されない

```
public class Duck {

  private int size;
  private static int duckCount = 0;

  public Duck() {
    duckCount++;
  }

  public void setSize(int s) {
    size =s;
  }
  public int getSize() {
    return size;
  }
}
```

duckCountはスタティック変数なので、値が0に戻ることはなく、Duckコンストラクタが実行されるたびに値がインクリメントされていく

個々のDuckオブジェクトが専用のduckCount変数を持つことはない

duckCountはスタティック変数なので、すべてのDuckオブジェクトが1つのduckCount変数を共有する。スタティック変数は、オブジェクトではなく、クラスが保持する変数と考えることもできる



Duckオブジェクトは専用のsize変数を持つことはできるが、専用のduckCount変数は持てない。クラスのオブジェクトすべてで共有する

This PDF is prepared for tools@maruhouse.org, personal use only.

**スタティック変数は
すべてのインスタンスによって
共有される**

**同じクラスのインスタンスすべてが
１つのスタティック変数を
共有する**

**インスタンス変数：インスタンスにつき１つ
スタティック変数：クラスにつき１つ**

頭の体操

先述のとおり、コンストラクタをprivate宣言すると、他のクラスのコードでそのクラスをインスタンス化することはできなくなります。private宣言されたコンストラクタは、同じクラス内のコードでのみ呼び出せるのです（ニワトリが先かタマゴが先か、という話に似ていますが……）。話は変わりますが、あるクラスのインスタンスを１つだけ作れるようにする、ということは可能でしょうか。そのクラスのインスタンスは常に１つしか存在せず、あらゆる人が同じ１つのインスタンスを利用するようにする、というようなことはできるでしょうか。考えてみてください。

This PDF is prepared for tools@maruhouse.org, personal use only.

スタティック変数の初期化

スタティック変数は、クラスがロードされる際に初期化されます。クラスがロードされるのは、JVMが「ロードすべき」と判断した時です。通常、JVMがクラスをロードすべきと判断するのは、あるクラスのインスタンスが最初に作成される時、あるいは、あるクラスのスタティックメソッド、スタティック変数が利用される時です。プログラマが自らJVMにクラスをロードするよう指示することもできますが、まずそういうことは必要ありません。ほとんどの場合、JVMに任せておいた方が無難でしょう。

スタティック変数の初期化に関しては右のようなことが言えます。

- **スタティック変数は、クラスのあらゆるインスタンスが作成される前に初期化される**
- **スタティック変数は、クラスのあらゆるスタティックメソッドが実行される前に初期化される**

```
class Player {
  static int playerCount = 0;
  private String name;
  public Player(String n) {
    name = n;
    playerCount++;
  }
}

public class PlayerTestDrive {
  public static void main(String [] args) {
    System.out.println(Player.playerCount);
    Player one = new Player("Tiger Woods");
    System.out.println(Player.playerCount);
  }
}
```

playerCountは、クラスのロード時に初期化される。ここではその際に0を設定するコードを入れているが、intのデフォルト値は0なので、このコードはなくてもよい。スタティック変数の初期化のためのコードをプログラマが書かなかった場合にはインスタンス変数の場合と同じようにデフォルト値が設定される

デフォルト値はスタティック変数でもインスタンス変数でも同じ。整数型（long、short、intなど）では0、浮動小数点型（float、double）では0.0、boolean型ではfalse、参照型ではnullになる
スタティック変数へのアクセスはスタティックメソッドの場合と同じようにクラス名を使って行う

スタティック変数は、クラスのあらゆるインスタンスが作成される前に初期化される

スタティック変数が、クラスがロードされる際に初期化されるというのはすでに述べたとおりです。プログラムに、初期化のコードが入っていない場合には、自動的にデフォルト値が設定されることになります。デフォルト値は、変数がたとえばint型であれば0になります。つまり、"playerCount = 0"のようなコードはあえて書く必要がないということです。スタティック変数を宣言し、初期化をしなければ、その変数には必ずデフォルト値が設定されます。この点はインスタンス変数と変わりありません。

```
File  Edit  Window  Help  What?
% java PlayerTestDrive
0
1
```

インスタンス作成前

インスタンス作成後

This PDF is prepared for tools@maruhouse.org, personal use only.

static finalと宣言した変数は定数のようになる

「final」というのは、「初期化後は変更できない」という意味です。これは、「static final」と宣言された変数の値が、クラスがロードされている限り一定に保たれるということを意味します。たとえば、Mathクラスには、PIという変数があり、宣言、初期化のコードは以下のようになっています。

```
public static final double PI =3.141592653589793;
```

publicと宣言されているため、この変数にはどのコードからでもアクセスが可能です。また、staticという宣言もあるので、Mathのインスタンスも作る必要はありません（Mathのインスタンスはすでに述べたとおり、元々作ることはできないのですが）。「final」は、PIの値が変更できないことを（少なくともJavaにおいては）意味します。

この方法で変数を定数のようなものにすることができます。また、「この変数は定数のように扱う」ということがすぐわかるようにするため、慣例では、名前をすべて大文字にすることになっています。

static finalと宣言した変数（定数）の初期化を、以下に示すような「スタティックイニシャライザ」と呼ばれるコードブロックを使って行うこともあります。スタティックイニシャライザは、クラスのロード時に、他のコードからそのコードが利用できるようになる前に実行されます。

```
class Foo {
  final static int x;
  static {
    x = 42;
  }
}
```

static final 変数の初期化の方法

① 宣言と同時に初期化する

```
public class Foo {
  public static final int FOO_X = 25;
}
```

名前の付け方に注意。static final変数は事実上、定数なので、名前はすべて大文字にする。複数の単語から成る名前の場合は単語と単語の間に下線を入れる

または

② スタティックイニシャライザを使用する

```
public class Bar {
  public static final double BAR_SIGN;

  static {
    BAR_SIGN = (double)Math.random();
  }
}
```

このコードは、クラスがロードされるとすぐに実行される。それより前にスタティックメソッドが呼び出されたり、スタティック変数が使用されることはない

いずれかの方法で初期値を指定しないと、

```
public class Bar {
  public static final double BAR_SIGN;
}
```

初期化していない！

コンパイラがエラーを発生させる

```
File  Edit  Window  Help  Jack-in
% javac Bar.java
Bar.java : 1: variable BAR_SIGN
might not have been initialized
1 error
```


This PDF is prepared for tools@maruhouse.org, personal use only.

finalは必ずしもstaticと同時に使用しなくてよい

キーワード「final」は、必ずしも「static」とともに使用しなくてかまいません。また、インスタンス変数、ローカル変数、メソッドのパラメータのいずれにも使用することができます。どのように使用した場合でも「この変数の値は変更できない」という意味になることは変わりません。ただ、メソッドやクラスに「final」を使用した場合には、「このメソッドはオーバーライドできない」、「このクラスは継承できない」という意味になります。

変数にfinalを使った場合は値が変更できなくなる

メソッドにfinalを使った場合にはオーバーライドができなくなる

クラスにfinalを使った場合には継承ができなくなる（サブクラスができないということ）

スタティックでない変数にfinalを使う

```
class Foof {
  final int size = 3;  ← sizeは変更できない
  final int whuffie;

  Foof() {
    whuffie = 42;  ← whuffieは変更できない
  }

  void doStuff(final int x) {
    // xは変更できない
  }

  void doMore() {
    final int z = 7;
    // zは変更できない
  }
}
```

finalメソッド

```
class Poof {
  final void calcWhuffie() {
    // このコードは
    // オーバーライドできない
  }
}
```

finalクラス

```
final class MyMostPerfectClass {
  // このクラスは継承できない
}
```

This PDF is prepared for tools@maruhouse.org personal use only.

素朴な疑問に答えます

Q：スタティックメソッドではスタティックでない変数は利用できないということですが、逆にスタティックでないメソッドでスタティック変数を利用することはできるのですか。

A：もちろんです。スタティックでないメソッドでは、スタティックメソッドを呼び出すことも、スタティック変数を利用することも可能です。

Q：クラスをfinal宣言する必要があるのはなぜですか。オブジェクト指向の考え方とfinal宣言とは矛盾するのではないですか。

A：そうとも言えますが、そうでないとも言えます。クラスをfinal宣言するのは1つにはセキュリティのためです。たとえば、Stringクラスのサブクラスは作れないようになっています。Stringクラスのサブクラスが知らない間に作られ、Stringクラスの代わりに使用されたとしたら、どれほど大変なことになるか想像してみてください。このように、あるクラスが常に「サブクラスが代替として使われることがない」という前提で使われ、勝手に継承されては困るような場合にはfinal宣言する必要があるのです。

Q：final宣言したクラスのメソッドをfinal宣言するのは余計ではありませんか。

A：そのとおりです。クラスをfinal宣言した場合には、メソッドをfinal宣言する必要はありません。クラスをfinal宣言すれば、サブクラスが作れないので、必然的にメソッドをオーバーライドすることはできなくなるからです。
反対に、クラスの継承は許可するが、一部のメソッドはオーバーライドされると困る、という場合は、クラスをfinal宣言せず、オーバーライドを禁止するメソッドのみfinal宣言すればいいでしょう。

重要ポイント

- スタティックメソッドは、オブジェクトへの参照を納めた変数名ではなく、クラス名を使って呼び出す。

  例：`Math.random()`

- スタティックメソッドは、そのメソッドが属するクラスのインスタンスがヒープ上になくても呼び出せる。
- スタティックメソッドは、インスタンス変数の値によって動作が変化せず、ユーティリティメソッドに適している。
- スタティックメソッドは、特定のインスタンスと結び付くことがなく、インスタンス変数の値を得ることなどはできない。スタティックメソッドの中にインスタンス変数にアクセスするコードを入れても、その変数がどのインスタンスに属するものかが判別できず、コンパイルでエラーになる。
- スタティックメソッドでは、スタティックでないメソッドを利用することはできない。スタティックでないメソッドは通常、インスタンス変数の値によって動作が変化するため。
- スタティックメソッドのみを持つクラスがインスタンス化されないようにするには、コンストラクタをprivate宣言する。
- スタティック変数とは、あるクラスのすべてのメンバによって共有される変数のこと。インスタンス変数のようにオブジェクトごとに存在するのではなく、1つを同じクラスのすべてのオブジェクトで利用することになる。
- スタティックメソッドでは、スタティック変数にアクセスすることができる。
- Javaで定数を作る時は、変数をstatic final宣言する。
- static final変数の初期化は、必ず、宣言と同時に行うか、以下のようなスタティックイニシャライザを使用して行う。

```
static {
 DOG_CODE = 420;
}
```

- static final変数（定数）の名前は慣例上、すべて大文字にする。
- final宣言した変数の値は、初期化後は変更できない。
- final宣言したインスタンス変数の初期化は、必ず、宣言と同時に行うか、コンストラクタの中で行う。
- final宣言したメソッドはオーバーライドできない。
- final宣言したクラスは継承できない（サブクラスが作れない）。

This PDF is prepared for tools@maruhouse.org, personal use only.

自分で考えてみよう

正しいコードはどれ？

これまでに学んだことをもとに、以下のコードの中からコンパイル可能なものを選んでください。

1
```
public class Foo {
  static int x;

  public void go() {
    System.out.println(x);
  }
}
```

2
```
public class Foo2 {
  int x;

  public static void go() {
    System.out.println(x);
  }
}
```

3
```
public class Foo3 {
  final int x;

  public void go() {
    System.out.println(x);
  }
}
```

4
```
public class Foo4 {
  static final int x = 12;

  public void go() {
    System.out.println(x);
  }
}
```

5
```
public class Foo5 {
  static final int x = 12;

  public void go(final int x) {
    System.out.println(x);
  }
}
```

6
```
public class Foo6 {
  int x = 12;

  public static void go(final int x) {
    System.out.println(x);
  }
}
```


This PDF is prepared for tools@maruhouse.org, personal use only.

Mathクラスの メソッドの概要

スタティックメソッドがどのような性質のものかということは、これまでの説明でわかったと思います。ここでは、Mathクラスのスタティックメソッドの代表例をいくつか紹介しましょう。この他には、sqrt()、tan()、ceil()、floor()、asin()といったメソッドもあります。

Math.random()

0.0 ～ 1.0 の範囲の無作為の数値を戻す（ただし、0.0、1.0 は含まない）。

```
double r1 = Math.random();
int r2 = (int)(Math.random() * 5);
```

Math.abs()

引数として指定された数値の絶対値を戻す。このメソッドはオーバーロードされており、引数にint型の値を指定した場合にはint型の値が戻され、double型の値を指定した場合にはdouble型の値が戻される。

```
int x = Math.abs(-240); // 240を戻す
double d = Math.abs(240.45); // 240.45を戻す
```

Math.round()

引数として指定された数値に最も近い整数（引数に指定された値がfloat型であればint型の値を、double型であればlong型の値）を戻す。

```
int x = Math.round(-24.8f); // -25を戻す
int y = Math.round(24.45f); // 24を戻す
```

浮動小数点数をそのままのかたちで指定する際は「f」と書かない限り型がdoubleになる

Math.min()

2 つの引数のうちの小さい方を戻す。このメソッドはオーバーロードされており、int型、long型、float型、double型の引数を指定できる。

```
int x = Math.min(24, 240); // 24を戻す
double y = Math.min(90876.5, 90876.49); // 90876.49を戻す
```

Math.max()

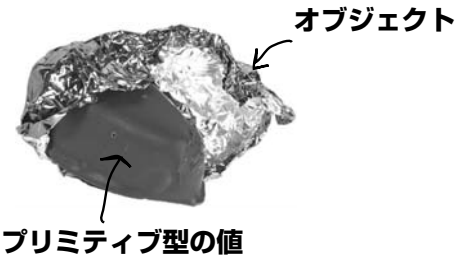
2 つの引数のうちの大きい方を戻す。このメソッドはオーバーロードされており、int型、long型、float型、double型の引数を指定できる。

```
int x =Math.max(24, 240); // 240を戻す
double y = Math.max(90876.5, 90876.49); // 90876.5を戻す
```


This PDF is prepared for tools@maruhouse.org, personal use only.

プリミティブのラッピング

プリミティブ型の値は、時にはそのままのかたちでは扱えない場合があります。たとえば、5.0より前のJavaでは、ArrayListやHashMapといったコレクションに直接プリミティブ型の値を格納することはできません。




```
int x = 32;
ArrayList list = new ArrayList();
list.add(x);
```

Java 5.0を使っていなければ、このコードは誤り。5.0より古いJavaの場合、ArrayListにint型などプリミティブ型の引数を指定できるadd()メソッドはない（存在するのは、オブジェクトリファレンスを引数とするadd()メソッドのみ）

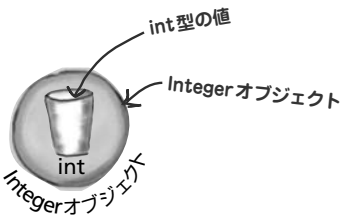
このような場合は、プリミティブ型の値をいったんオブジェクトに格納するという方法をとります。これを「ラッピング」と呼びます。ラッピングは、ArrayListの他、HashMapにプリミティブ型の値を格納したい時に使用できます。

ラッピングに使用するオブジェクト（クラス）はjava.langパッケージにあらかじめ用意されています（この種のクラスを「ラッパ」と呼ぶ）。使用するのにインポートの必要はありません。個々のデータ型について、それぞれ対応するクラスがあるのですが、Javaの名前付け規則に従い先頭が大文字になっている点に注意してください。データ型名とクラス名は**必ず同じとは限りません**。ラッピングのためのクラスには以下のようなものがあります。

Java 5.0より前のバージョンではArrayList、HashMapに格納する場合などのように、プリミティブ型の値をそのまま扱えない時は、ラッピングする

Boolean
Character
Byte
Short
Integer
Long
Float
Double

名前がデータ型と完全には一致しない点に注意。Integer、Characterは省略されていない



ラッパのコンストラクタの引数としてプリミティブ型の値を指定する

値のラッピング

```
int i = 288;
Integer iWrap = new Integer(i);
```

すべてのラッパに同様のメソッドが用意されている。BooleanのbooleanValue()、CharacterのcharValue()など

ラッピングしたデータの取り出し

```
int unWrapped = iWrap.intValue();
```

メモ：ラッパは、上に載せた写真のようにチョコレートの包み紙のようなものと言えます。中にはイモの皮ようなものと思う人もいるかもしれません。

This PDF is prepared for tools@maruhouse.org, personal use only.

バカげてる。それじゃ int 型の値を
直接入れる ArrayList は作れないってこと？
いちいち Integer オブジェクトを作って
ラッピングして、値にアクセスする時にはいちいち
アンラッピングしなくちゃいけないなんて。
時間の無駄だし、ミスもしやすいと
思うんだけど。

Java 5.0 より前は確かにそのとおり……

彼女の言うとおりです。Java 5.0 までは、どのバージョンでも、プリミティブはプリミティブ、オブジェクトはオブジェクトであり、両者を同じように扱うことはできませんでした。同じように扱おうとすれば、プログラマがラッピング、アンラッピングをする必要がありました。オブジェクトを渡すはずの引数にプリミティブ型の値を渡すことはできないし、戻り値として得られたオブジェクトをプリミティブ型の変数に代入することも直接にはできなかったのです。たとえ戻り値が Integer オブジェクトであっても、それを int 型の変数に代入することはできません。Integer と int の間には何の関係もないからです。ただ Integer に int 型のインスタンス変数（この変数にラッピング対象の int 型の値を保持する）があるというだけの話です。ラッピング、アンラッピングの操作をプログラマが行わない限り、両者が関係することはありません。

ArrayList に int 型の値を収める

Java 5.0 より前のバージョンの場合（ラッピング、アンラッピングは自動的には行われない）

```
public void doNumsOldWay() {

        ArrayList listOfNumbers = new ArrayList() ;

        listOfNumbers.add(new Integer(3));

        Integer one = (Integer) listOfNumbers.get(0);

        int intOne = one.intValue():
}
```

ArrayList を作成（5.0 より前のバージョンでは型の指定はできない。ArrayList は Object 型オブジェクトのリストとなる）

プリミティブ型の「3」リストに追加することはできないので、まず Integer にラッピングする

取り出されたものは Object 型のオブジェクトになるので、Integer にキャストする必要がある

これでようやく Integer オブジェクトからプリミティブ型の値が取り出せる

This PDF is prepared for tools@maruhouse.org, personal use only.

プリミティブとオブジェクトの境界線を曖昧にする機能

Java 5.0からは、プリミティブ、ラッパオブジェクト間の自動変換機能が追加されています。ArrayListにint型の値を保持する場合を例に、この機能が具体的にどのようなものかを見てみましょう。

Java 5.0以降のプリミティブ型とArrayList（ラッピング、アンラッピングが自動化されている）

```
public void doNumsNewWay() {

        ArrayList<Integer> listOfNumbers = new Array List<Integer>() ;

        listOfNumbers.add(3) ;

        int num = listOfNumber.get(0) ;
}
```

Integer型のArrayListの作成

このようにして要素を追加できる

コンパイラはIntegerオブジェクトのアンラッピングも自動的に行う。intValue()メソッドを呼び出さなくても、int型の変数にget()の戻り値を直接代入できる

ArrayListには、add(int)というメソッドはないのだが、コンパイラがラッピングを自動的に行うためにadd(3)というコードが書ける。ArrayListには実際にはIntegerオブジェクトが保持されているのだが、int型の数値を直接追加しているかのようなコードを書くことができる（ArrayList<Integer>には、見かけ上、intとIntegerの両方が追加できるということ）

Q：int型の値を保持するのならArrayList<int>と宣言すればいいのではないですか。

A：それはできません。Java 5.0のジェネリック型の規則では、指定できるのはクラス、あるいはインターフェースのみとなっており、プリミティブ型は指定できません。ArrayList<int>というコードを書いてもコンパイルできないということです。しかし、上記のコードを見てわかるとおり、それでも特に問題はないでしょう。ArrayList<Integer>に直接、int型の値を書き込むことが可能だからです。Java 5.0では、ArrayListの型となっているクラスがそのラッパとなれるのであれば、プリミティブ型の値は自由に追加できます。ラッピングの処理がコンパイラによって自動的に行われるからです。たとえば、ArrayList<Boolean>には、boolean型の値が、ArrayList<Character>にはchar型の値が追加できます。

This PDF is prepared for tools@maruhouse.org, personal use only.

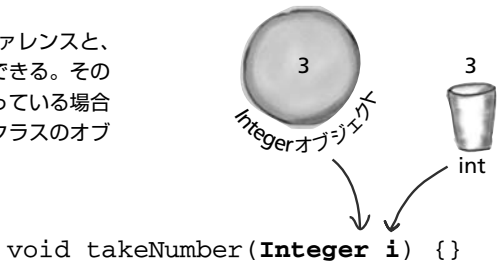
さまざまなケースで行われる自動ラッピング、アンラッピング

ラッピング、アンラッピングが自動的に行われるというのは、単にコレクションにプリミティブ型を直接格納できるという以上のことを意味します。この機能があるために、プリミティブ型の値を、以前なら使えなかった数多くの場面で使えるようになるからです。次に示す例を見てください。

自動ラッピング、アンラッピングの実例

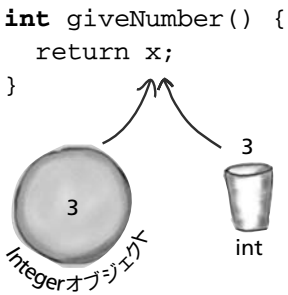
メソッドの引数

メソッドの型がラッパになっていれば、オブジェクトリファレンスと、ラッパに対応するプリミティブ型の値の両方を渡すことができる。その逆のことも言える。メソッドの引数がプリミティブ型になっている場合は、互換性のあるプリミティブ型の値と、対応するラッパクラスのオブジェクトの両方を渡せる。



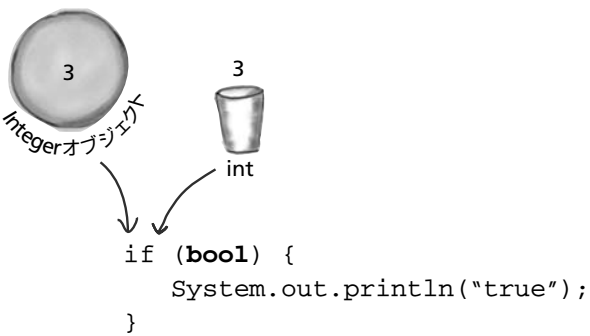
戻り値

メソッドの戻り値の宣言がプリミティブ型になっている場合は、互換性のあるプリミティブ型の値と、対応するラッパクラスのオブジェクトの両方を戻せる。反対に、戻り値の宣言がラッパになっている場合も、ラッパクラスのオブジェクトと、対応するプリミティブ型の値の両方を戻せる。



ブール値

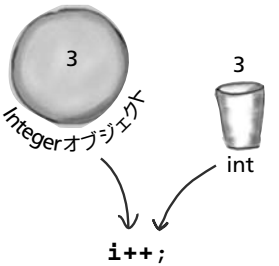
ブール値の入るべき箇所には、演算結果がブール値になる式（例：4 > 2）、boolean型（プリミティブ型）の値、Booleanクラスのラッパオブジェクトを入れることができる。



This PDF is prepared for tools@maruhouse.org, personal use only.

式

奇妙に思うかもしれないが、Java 5.0では、通常であればプリミティブ型の値を使うはずの式のオペランドとしてもラッパオブジェクトが使える。たとえば、下の例のように、Integerオブジェクトのインクリメントなどもできるということ。
と言っても厳密には、本当にこうした演算子がオブジェクトに対して機能するよう改造されたわけではない。コンパイラが演算のまえに自動的にラッパオブジェクトをプリミティブ型の値に変換しているというだけである。それがわかっていても、奇妙に見えるのは確か。



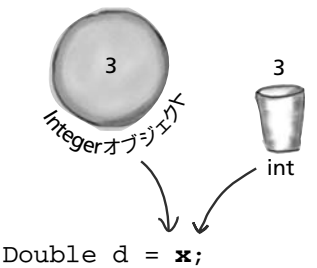

```
Integer i = new Integer(42);
i++;
```

以下のようなコードも書ける。

```
Integer j = new Integer(5);
Integer k = j + 3;
```

代入

変数の型がラッパであっても、プリミティブであっても、ラッパオブジェクト、プリミティブ型の値の両方が代入できる。たとえば、int型（プリミティブ型）の変数にIntegerオブジェクトへの参照を代入することも、逆に、Integer型の変数にint型の値を代入することも可能。



自分で考えてみよう

右のコードはコンパイルできるでしょうか。また、実行できるでしょうか。実行できるとしたら、どのような動きをするでしょうか。しばらく時間をとって考えてみてください。コードには、ここでは触れなかった自動ラッピング機能の応用例も含まれています。
実際にコードをコンパイルしてみた方がわかりやすいのは間違いありません（その方が学習の効果も高いでしょう）。

```
public class TestBox {

  Integer i ;
  int j ;

  public static void main (string[] args) {
       TestBox t = new TestBox() ;
       t.go() ;
  }

  public void go() {
       j=i ;
       System.out.printIn(j) ;
       System.out.printIn(i) ;
  }
}
```


This PDF is prepared for tools@maruhouse.org, personal use only.

ラッパに用意されている ユーティリティメソッド

ラッパは他の通常のクラスと同様の機能を果たす他、便利なユーティリティメソッド（スタティックメソッド）もいくつか備えています。本書ですでに使用したInteger.parseInt()はそうしたユーティリティメソッドのうちの1つです。

parseInt()は、引数として指定された数字（文字列）を、それに対応する数値に変換するという機能を備えています。parseDouble()など名前が“parse”で始まるメソッドはどれもほぼ同様の機能を備えています。

Stringをプリミティブ型の値に変換する

```
String s = "2";
int x = Integer.parseInt(s);
double d = Double.parseDouble("420.24");

boolean b = new Boolean("true").booleanValue();
```

数字の“2”が数値の“2”に変換される

Boolean.parseBoolean()というメソッドもありそうに思えるが、実際には存在しない。ただ、幸い、Booleanクラスには、Stringを引数として指定できるコンストラクタが用意されている。そのコンストラクタを使って作ったBooleanオブジェクトのラッピングを解いてやれば、文字列“true”、“false”をboolean型の値“true”、“false”に変換できる

誤った例

```
String t = "two";
int y = Integer.parseInt(t);
```

このコードはコンパイルはできるが実行時にエラーになる。指定した文字列が数字と解釈されなかった場合にはNumberFormatExceptionという例外が発生する

上記のコードは実行時にエラーを発生させる

```
File  Edit  Window  Help  Clue
% java Wrappers
Exception in thread "main"
java.lang.NumberFormatException: two
at java.lang.Integer.parseInt(Integer.java:409)
at java.lang.Integer.parseInt(Integer.java:458)
at Wrapers.main(Wrapers.java:9)
```

Stringを処理するメソッドやコンストラクタはNumberFormatExceptionを発生させる恐れがある。この例外は「実行時例外」と呼ばれる種類のもので、プログラマが宣言や処理のためのコードを書く必要はない（書いた方がよい場合もある）。

（例外については次章で詳述）

This PDF is prepared for tools@maruhouse.org, personal use only.

数値を数字に変換する

次に、逆に数値を数字（String）に変換する方法について見てみましょう。方法はいくつかありますが、最も簡単なのは、他の文字列に連結するというものです。

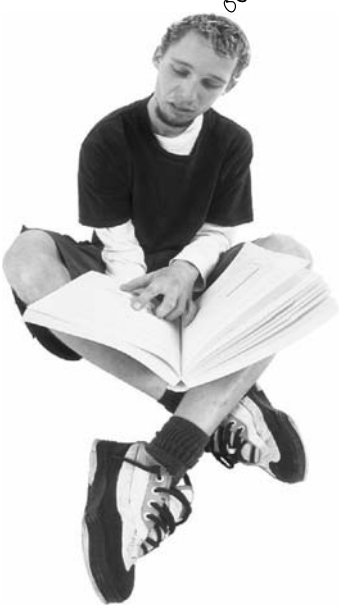
```
double d = 42.5;
String doubleString = "" + d;
```

「+」演算子はJavaではオーバーロードされており（唯一のオーバーロード演算子）、文字列（String）の連結にも使用できる。Stringに「+」を使って連結されるものは、何であろうと自動的にStringに変換される

```
double d = 42.5;
String doubleString = Double.toString(d);
```

ラッパのスタティックメソッドを使用する方法もある

一応、わかったな。でも、金額の表示なんかはどうするんだろう。ドルの場合だと、「$56.87」という感じに「$」マークを付けて、小数点以下を 2 桁にしなくちゃならないし。「45,687,890」ていう具合にカンマを付けるには どうすればいいのか……。



printfを使うんじゃないのかな。C言語だと、数値のフォーマッティングにはprintf文を使うんだけど……。I/Oに関係するんだよね。たぶん。

This PDF is prepared for tools@maruhouse.org, personal use only.

数値のフォーマッティング

Javaでは、C言語のように、数値や日付のフォーマッティングが常にI/Oと結び付いているなどということはありません。これは現状から考えれば、望ましいことであると言えるでしょう。数値を画面に表示する場合、現在ではGUIを使用するのが一般的です。つまり、Stringをスクロール可能なテキスト領域やテーブルなどに配置するわけですが、フォーマッティングを常にprint文で行わねばならないとしたら、GUIに合わせたフォーマッティングなどとても不可能でしょう。以前は、フォーマッティングの大半をjava.textパッケージのクラスで扱うようになっていましたが、Java 5.0から状況は変わっています（本書の前の第1版ではjava.textパッケージについて少し触れましたが、第2版では触れません）。

Java 5.0からは、java.utilパッケージにFormatterクラスが加わり、より高度で自由なフォーマッティングができるようになりました。ただし、フォーマッティングに際してFormatterクラスのインスタンスを作成してメソッドを呼び出すということは必要ありません。I/OクラスやStringクラスに便利なメソッドが用意されているからです（I/Oクラスにはprintf()なども加わっている）。フォーマッティングの対象となるものと、フォーマッティングに関する指示を引数に指定して、String.format()などのメソッドを呼び出せば良いだけです。

とはいえ、フォーマッティングに関する指示の出し方は学ぶ必要があります。CやC++のprintf()関数に慣れていない人は、はじめは少し苦労するかもしれません。幸い、ごく基本的な部分だけなら、printf()を知らない人でも比較的簡単に覚えることができます。ここでは、使い方の「レシピ」のようなものを提示することにします。複雑な指示をして、自分の思いどおりのフォーマッティングができるようになるまでには訓練が必要になるでしょう。

以下に簡単なフォーマッティングの例を示し、解説を加えます（フォーマッティングに関しては、I/Oについて解説する際に再度触れます）。

数字にカンマを入れる

```
public class TestFormats {
  public static void main (string[] args) {

      String s = String.format("%, d", 1000000000) ;
      System.out.printIn(s) ;
  }
}
```

フォーマッティング対象の数値（これにカンマを入れる）

第2引数に指定された値（ここではint型の値）をどのようにフォーマッティングするかをここに指定。このメソッドの引数はあくまで2つであることに注意。最初のカンマは、Stringのリテラルの一部であり、引数の区切りではない

```
1,000,000,000
```

数字にカンマが挿入されている

This PDF is prepared for tools@maruhouse.org, personal use only.

フォーマッティングについての指示の仕方

フォーマッティングについて指示をするためのコードは、基本的には 2 つの部分から構成されます（もっと多くの部分から構成されるコードも書けますが、わかりやすいようここでは最も基本的な 2 つに絞って話を進めます）。

① フォーマッティングの内容を書く部分

どのようなフォーマッティングをするか、その内容を特殊な形式のコードで記述します。

② フォーマッティング対象を指定する部分

フォーマッティング対象は複数を指定することもできるのですが、まずは 1 つ指定するケースに限定して説明をします。フォーマッティング対象としては、どんな型の値でも指定できるというわけではありません。1 の引数のコードに従ってフォーマッティングができる値に限られます。たとえば、1 のコードで浮動小数点数を指定しているのに、2 の引数で Dog 型のオブジェクトや、String 型の「数字」を指定するといったことは不可能です（数値と「数字」は見かけ上同じように見えるが別のもの）。

注意：以降の何ページかは、すでにprintfについての知識がある人は、軽く目を通すだけでいいが、そうでない人は熟読すること

フォーマッティングの内容　フォーマッティングの対象

```
format("%, d", 1000000000) ;
```

① ②

第 1 引数に指定されたフォーマッティングを　第 2 引数に指定された値に対して行う

上記のコードの意味は？

上記のコードは、「第 2 引数に指定された数値（1000000000）を 10 進整数として扱ってフォーマッティングを行い、カンマを挿入する」という意味になります。

フォーマッティングの内容を記述するコードの形式

フォーマッティングの内容を記述するコードには、"%" などの記号や "d" などの文字を使いますが、詳細については次のページで解説します。はじめは、"%" という記号を見たら「フォーマッティングの内容を記述しているな」とわかる、というくらいで十分でしょう（format() メソッドの引数の先頭は常に "%" になる）。"%" を変数のようなものと考えてもわかりやすいかもしれません。変数の値は、もう 1 つの引数に指定する、というわけです。"%" の後にどのような文字を指定するかによって、もう 1 つの引数に指定された値をどうフォーマッティングするかが決まります。

This PDF is prepared for tools@maruhouse.org, personal use only.

"%"は「この部分にフォーマッティング対象の値が入る」ということを意味する

format()メソッドの第1引数は「フォーマット文字列」と呼ばれています。"%"の前後には、そのままのかたちでフォーマッティングの処理を行わずに出力する文字を付け加えることもできます。"%"は、フォーマッティング対象の値を保持する変数、と考えてもいいでしょう。

この文字列はそのままのかたちでformat()メソッドの戻り値になる

第2引数(数値)に対しどのようなフォーマッティングをするかをここに指定する

この文字列もそのままのかたちでformat()メソッドの戻り値になり、フォーマッティング対象の数値の後に入る

第2引数にはフォーマッティング対象の値を指定する

```
format("I have %.2f bugs to fix.", 476578.09876);
```

出力

```
I have 476578.10 bugs to fix.
```

小数点以下が一部なくなっている点に注意。これで".2f"の意味がわかる人がいるかもしれない

"%"は「この部分にフォーマット対象の値(ここではformat()メソッドの第2引数の数値)が入り、フォーマッティングされる」ということを意味します。具体的にどのようなフォーマティングを行うかは"%"の後の".2f"の部分に指定されます。その後の"bugs to fix."はそのままのかたちで出力されます。

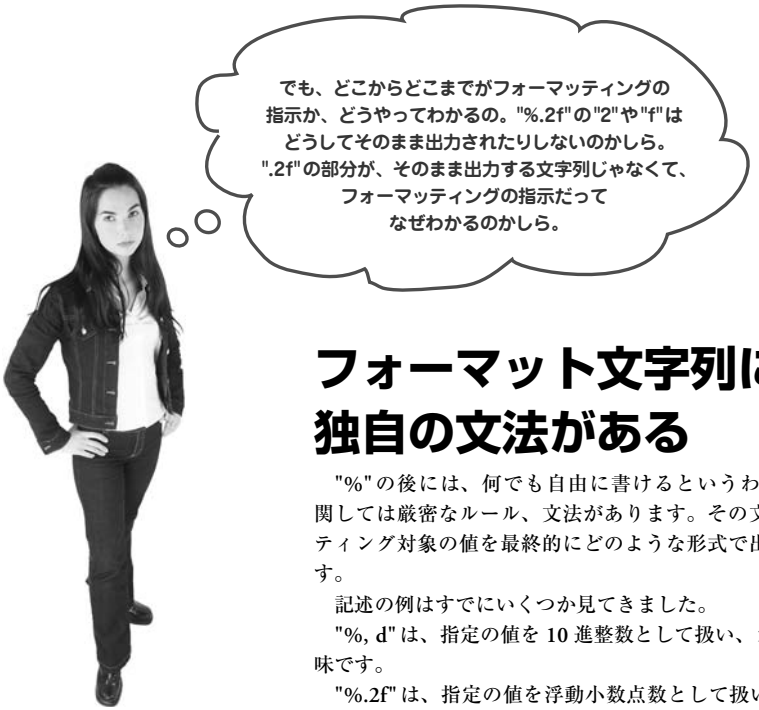
カンマの追加

```
format("I have %,.2f bugs to fix.", 476578.09876);
```

```
I have 476,578.10 bugs to fix.
```

フォーマッティングの内容を指示する"%.2f"の部分を"%,.2f"に変えると、出力される数字にカンマを入れることができる

This PDF is prepared for tools@maruhouse.org, personal use only.



フォーマット文字列には独自の文法がある

"%"の後には、何でも自由に書けるというわけではありません。記述に関しては厳密なルール、文法があります。その文法に基づいて、フォーマッティング対象の値を最終的にどのような形式で出力するかを指定するわけです。

記述の例はすでにいくつか見てきました。

"%, d"は、指定の値を 10 進整数として扱い、カンマを挿入する、という意味です。

"%.2f"は、指定の値を浮動小数点数として扱い、精度は小数第 2 位までとする、という意味です。

"%,.2f"は、指定の値を浮動小数点数として扱い、精度は小数第 2 位までとし、カンマを挿入する、という意味です。

ここで重要なのは、"%"の後の記述をどのようにすれば、自分の思いどおりのフォーマッティングができるかということです。10 進整数を意味する"d"、浮動小数点数を意味する"f"といった文字、カンマなどの記号を、それぞれどのような場合に、どの位置に指定すればよいのかを知る必要があります。たとえば、"%,d"ではなく、"%d,"のように、"d"の後にカンマを書くとどうなるでしょうか。

以下のコードは正しく機能するでしょうか。機能するとしたら、実行結果はどのようになるでしょうか。

```
String.format("I have %2f, bugs to fix.", 476578.09876) ;
```

（答えは次のページ）

This PDF is prepared for tools@maruhouse.org, personal use only.

フォーマッティング指示のコード

フォーマッティングの内容を指定するためのコードは、"%"で始まり、"d"や"f"といった「型」を指定する文字で終わります。"d"や"f"などの後に何か書くと、新たにもう1つ"%"が指定されない限り、「そのままのかたちで出力すべきもの」と解釈されます。ただ、ここで問題なのは、"%"の後にもう1つ"%"を入れることなど可能なのか、ということです。それができるとすると、フォーマッティング対象の値も、もう1つ指定できることになりますが、その点については後で触れます。ここではとりあえず、"%"は1つであると仮定して、フォーマッティング指示のコードを見てみましょう。

フォーマッティングの内容を指定するコードは、最大で、以下の5つの部分から構成されます(%を除いて5つ)。[]の中の部分はいずれもオプションなので、最低限、"%"と、型を指定する文字だけを指定すればよいということになります。ただし、指定の順序を変えることはできません。5つの部分はすべて、以下に示す順序で指定しなくてはならないということです。

```
% [argument number] [flags] [width] [.precision] type
```

この部分については後で触れる。フォーマッティング対象の引数が複数ある場合、ここに引数の番号を指定する(今のところはわからなくてよい)

カンマを挿入する、負の数をカッコに入れる、数字を左揃えにする、など特殊な処理をするためのオプションを指定する

フォーマッティング後の桁数を最低いくつにするかを指定する。「合計いくつ」ではなく「最低いくつ」である点に注意。フォーマッティング対象の値の桁数がたとえ[width]の指定より多くても、すべての桁が出力される。フォーマッティング対象の値の桁数が指定より少ない場合は、足りない部分に自動的に0が入る

すでに少し触れたが、この部分では精度、つまり小数第何位まで対応するかを指定する。"."を忘れないよう注意する

型の指定は必須(次ページも参照)。"d"は10進整数を、"f"は浮動小数点数を意味する

```
% [argument number] [flags] [width] [.precision] type

format("%,6.1f", 42.000) ;
```

上記のコードでは[argument number]以外のすべての部分が指定されている

This PDF is prepared for tools@maruhouse.org, personal use only.

必須なのは型の指定だけ

フォーマッティングの内容を指定するためのコードに絶対に必要なのは型の指定だけです。ただし、注意すべきなのは、他の指定を加える場合には、型の指定を必ず最後にするということです。型の指定には数多くの種類がありますが、実際に使用するのはほとんどの場合、"d(10 進整数）"か"f(浮動小数点数）"です（日付や時刻に使用するコードもまた別に存在する）。"f"を使用する場合には、同時に"."を使用して、精度（小数第何位まで対応するか）を指定することが多い。

型は必須、他はすべてオプション

%d　　10 進整数

```
format("%d", 42);
```

```
42
```

"42.25"のような値は指定できない。double型の値をint型の変数に代入するのと同じことになる

フォーマッティング対象となる値は、10 進整数として扱えるものでなくてはならない。つまり、byte、short、int、charのうちのいずれか（あるいは対応するラッパ）でなくてはならないということ。

%f　　浮動小数点数

```
format("%.3f", 42.000000);
```

```
42.000
```

"f"とともに".3"を使って精度を指定しているので、出力結果の小数点以下に 0 が 3 つ並んでいる

フォーマッティング対象となる値は、浮動小数点数として扱えるものでなくてはならない。つまり、floatかdouble（あるいは対応するラッパ）、もしくはBigDecimal（本書では詳しく触れない）でなくてはならないということ。

%x　　16 進数

```
format("%x", 42);
```

```
2a
```

フォーマッティング対象となる値は、byte、short、int、long（あるいは対応するラッパ）、もしくは BigInteger でなくてはならない。

%c　　文字

```
format("%c", 42);
```

```
*
```

"42"は文字"*"に対応する

フォーマッティング対象となる値は、byte、short、char、int（あるいは対応するラッパ）でなくてはならない。

フォーマッティングの内容を指定するためのコードには、必ず型の指定を入れる。その他のコードを入れる場合には、型の指定を最後に置く。フォーマッティングの対象となるのは通常、数値で、多くの場合、型の指定には "d（10 進整数）"、"f（浮動小数点数）" を使う。

This PDF is prepared for tools@maruhouse.org, personal use only.

フォーマッティング対象を複数指定する

たとえば、フォーマッティングによって以下の出力結果を得るには、どのようなコードを書けばいいでしょうか。

"The rank is 20,456,654 out of 100,567,890.24."

しかも、フォーマッティング対象は変数を使って指定しなくてはならないとします。この場合は、フォーマッティングの内容を指定するためのコードの後に、引数を 2 つ指定することになります。したがって、format() メソッドを 2 つではなく、3 つの引数を指定して呼び出すわけです。さらに、最初の引数（フォーマット文字列）には、"%" を含むコードを 2 つ指定します。1 つ目のコードが format() の 2 つ目の引数に、2 つ目のコードが format() の 3 つ目の引数に対応します。つまり、フォーマッティング対象の値は、指定された順序で使用されるということになります。

```
int one = 20456654 ;
double two = 100567890.248907 ;
String s = String.format("The rank is %,d out of %,.2f", one, two) ;
```

```
The rank is 20,456,654 out of 100,567,890.25
```

フォーマッティング対象の値を複数指定した場合は、指定の順序に従って、"%" を含むコードに対応する

2 つの値の両方にカンマを挿入し、浮動小数点数（2 番目に指定された値）の精度は小数第 2 位までにしている

この後、日付のフォーマッティングについて学ぶとわかりますが、時には、同じ値に複数種類のフォーマッティングを行わなくてはならない、という場合もあり得ます。それがどういう状況なのかは、これまでのような数値のフォーマッティングだけを見ていても想像はしにくいでしょう。日付の扱いについて知れば、フォーマッティング内容を指定するコードとフォーマッティング対象値との間の対応関係について、より明確に理解することができるはずです。

Q：何か変ではないですか。フォーマッティング対象の値はいったいいくつまで指定できるのですか。疑問なのは、String クラスの format() メソッドにはいくつのオーバーロードバージョンがあるのかということです。仮に、フォーマッティング対象の値を一度に 10 個指定したらどうなりますか。

A：鋭いですね。確かにここで説明したことは（少なくともこれまでに学んできたことからすれば）奇妙に見えると思います。実は、フォーマッティング対象の値の数を何通りにも変えられるのは、format() メソッドにたくさんのオーバーロードバージョンがあるからではありません。新たな（C 言語で言う printf 的な機能を持った）フォーマッティング API をサポートするため、Java 言語には「可変引数リスト」と呼ばれる新しい機能が追加されているのです。可変引数リストについて詳しくは触れません。フォーマッティングそのものとは直接関係がない上、可変引数リストを使うことがデザイン上あまり好ましいとは言えないからです。

This PDF is prepared for tools@maruhouse.org, personal use only.

日付のフォーマッティング

たとえば、"Sunday, Nov 28 2004" という文字列を出力しなくてはならないとします。「特に何も難しくはない」と思うかもしれません。では、Date型の変数を基に、この文字列を出力するとしたらどうでしょうか。Dateは、タイムスタンプに対応するJavaクラスです。format() メソッドに、数値ではなくDateオブジェクトを指定して、この文字列が出力されるようにするわけです。

数値のフォーマッティングと日付のフォーマッティングの最大の違いは、日付のフォーマッティングでは、型の指定に "t" で始まる 2 文字のコード（"f" や "d" のような 1 文字のコードではない）を使用する点です。この型指定コードの使用例を以下に示します。

日付と時刻を完全なかたちで出力する　%tc

```
String.format("%tc, new date()) ;
```

```
Sun Nov 28 14:52:41 MST 2004
```

時刻だけを出力する　%tr

```
String.format("%tr, new date()) ;
```

```
03:01:47 PM
```

曜日、月、日を出力する　%tA　%tB　%td

このケースでは、1 つのコードだけで必要な出力結果を得ることはできず、"%tA（曜日に対応）"、"%tB（月に対応）"、"%td（日に対応）" という 3 つのコードを組み合わせることになります。

```
Date today = new Date() ;
String.format("%tA, %tB %td, today, today, today) ;
```

このカンマはフォーマッティング指示のコードには含まれない。出力結果の中で、曜日の直後に置かれる

この場合、3 つのコードに対応して、Dateオブジェクトも 3 つ渡す必要がある。曜日、月、日のそれぞれについて同様の処理を行うということ

```
Sunday, November 28
```

同じ引数を繰り返し指定せずに曜日、月、日を出力する　%tA　%tB　%td

```
Date today = new Date() ;
String.format("%tA, %<tB %<td, today) ;
```

このコードは、Dateオブジェクトを使用して 3 種類のゲッターメソッドを呼び出しているようなものと考えることもできる。それによって、Dateオブジェクトから 3 種類のデータを取り出す

"<"は、フォーマッティング指示の中で使用するフラグの一種で「同じ引数を繰り返し使う」という意味。このフラグを使うと、同じ引数を繰り返さずに済み、同じ引数に対し複数とおりの処理ができる

This PDF is prepared for tools@maruhouse.org, personal use only.

日付の活用

日付を扱う場合、ただ今日の日付が出力できれば十分とは限りません。日付を変更する、ある時点から別の時点までの経過時間を算出する、スケジュールを立てる、スケジュールに優先順位を付けるといったことを行うプログラムを作成しなくてはならない場合もあるでしょう。Javaには、そうしたケースに役立つ機能も用意されています。

プログラムは、もちろん、自分でゼロから作ることも可能です。ただし、その場合には、うるう年やうるう秒などについても考慮しなくてはならないので、煩雑な作業が必要になり、相当な困難を伴います。Java APIには、日付操作のためのクラスが多数用意されているので、それを利用すれば、プログラミングの負担はかなり軽くなるはずです。クラスの数が豊富過ぎるのが難点かもしれませんが……。

This PDF is prepared for tools@maruhouse.org, personal use only.

時間を進める、さかのぼる

月曜日から金曜日まで、週に5日出勤する会社があるとします。その会社に勤めていて、ある年のカレンダーで、各月の最終出勤日がいつになるかを知ることのできるプログラムを作れと命じられたらどうすればいいでしょうか。

java.util.Dateクラスは「時代遅れ」……

今日の日付を知るのにjava.util.Dateクラスが使えることはすでに見てきました。したがって、上記のような日付に関連するプログラムを作るには、java.util.Dateクラスを基礎にすると良さそうにも思えます。しかし、APIのドキュメントを見るとわかるとおり、Dateのメソッドは、その多くが廃止されることになっており、使用は推奨されていません。

Dateクラスはいまだに、「タイムスタンプ（現在の日付と時刻についての情報を保持するオブジェクト）」を得るには非常に便利なので、今日が何日で、今が何時なのかを知る必要がある場合には使えばいいでしょう。

ただ、その他の場合には、java.util.Calendarというクラスを使う方が便利です。次に、java.util.Calendarについて見てみましょう。

タイムスタンプを得るには Date クラスがまだ使えるが、他の目的には Calendar クラスを使うべき

日付操作にはjava.util.Calendarを使う

Calendar APIは、グローバルな日付、時刻に対応できるよう作られており、またサブクラス、メソッドなどは、名前を見てすぐに機能がわかるよう作られています。プログラム中で日付、時刻を扱う場合には、常にCalendarクラス（のスタティックメソッド）を使う、と単純に考えていても基本的には問題ないでしょう（Calendarクラスのスタティックメソッドについては次ページ参照）。Calendarクラスを使う際、具体的にはまず、Calendarのサブクラスのインスタンスを得ることになります（Calendarは抽象クラスなので、実際に使うのは具象クラスであるサブクラスということになる）。

面白いのは、Calendar APIを使うと、プログラマがそれぞれ自分の目的に合ったカレンダーを得られるという点です。現在、世界の大部分では「グレゴリオ暦」のカレンダーを使っていますが、Calendar APIは仏教暦、イスラム暦、日本暦のカレンダーなどにも対応できるので、グレゴリオ暦以外の暦を使っている地域にいても困ることはありません。

ここでの説明では、java.util.GregorianCalendarクラスを使用することになりますが、これは標準のJava APIに含まれているクラスです。通常、Calendar APIを使用する際、自分がCalendarのどのサブクラスを使用しているかを意識する必要はありません。ただ、Calendarクラスのメソッドを使用していると思っているだけで、特に問題は起きないのです。

This PDF is prepared for tools@maruhouse.org, personal use only.

Calendarを継承するオブジェクトの取得

Calendar自体は抽象クラスなので、インスタンスを取得するということはもちろんできません。次のようなコードは書けないということです。

誤り：

```
Calender cal = new Calender() ;
```

このコードは、コンパイラによって誤りとみなされる

スタティックメソッドgetInstance()を使用してインスタンスを取得する：

```
Calender cal = Calender.getInstance() ;
```

スタティックメソッドを使ったコードはすでに本書でも紹介している

Calendarクラス（抽象クラス）のインスタンスは取得できないので、代わりにそのサブクラス（具象クラス）のインスタンスを取得する

Calendarが抽象クラスで、インスタンスが作成できないと言っても、そのスタティックメソッドを呼び出すことはできます。スタティックメソッドは、特定のインスタンス名ではなく、クラス名を使用して呼び出します。クラス名"Calendar"を使用してスタティックメソッド"getInstance()"を呼び出すと、サブクラス（具象クラス）のインスタンスが得られます。Calendarのサブクラスのインスタンスということは、Calendar型の変数に代入できるということです。また、Calendarクラスのメソッドはすべて備えています。

世界の大部分ではグレゴリオ暦が使用されているため、多くの場合はjava.util.GregorianCalendarクラスのインスタンスを取得することになります（通常、このクラスのインスタンスがデフォルトで得られるようになっている）。

This PDF is prepared for tools@maruhouse.org, personal use only.

Calendarオブジェクトを使う

Calendarのサブクラスのインスタンスを使う際は、まず以下のことを覚えておく必要があります。

- フィールドに「ステート」が保持される —— Calendarオブジェクトには、オブジェクトにとって特に重要な、日付や時刻に関連する情報（ステート）を保持するフィールドが数多くあります。このフィールドを利用することで、たとえば、年、月についての情報を得る、年、月を設定するといったことができます。
- 日付、時刻はインクリメントできる —— Calendarクラスには、フィールドに保持された、日付、時刻に関する情報をインクリメント、デクリメントするためのメソッドが用意されています。たとえば、時間を 1 ヶ月進める、3 年さかのぼるといったことができます。
- 日付、時刻はミリ秒単位にできる —— Calendarクラスには、日、時間をミリ秒に換算する（またはその逆の換算をする）機能があります（1970 年 1 月 1 日からある時点までの経過時間をミリ秒で表すことなどもできる）。この機能により、2 つの時点の間の経過時間を正確に計算することができる他、「ある時点に 63 時間 23 分 13 秒を加える」といった計算もできます。

Calendarオブジェクトの使用例：

```
Calender c = Calender.getInstance();
c.set(2004,0,7,15,40);
long day1 = c.getTimeInMillis();
day1 += 1000 * 60 * 60;
c.setTimeInMillis(day1);
System.out.printIn("new hour " + c.get(c.HOUR_OF_DAY));
c.add(c.DATE, 35) ;
System.out.printIn("add 35 days " + c.getTime());
c.roll(c.DATE, 35) ;
System.out.printIn("roll 35 days " + c.getTime()) ;
c.set(c.DATE, 1) ;
System.out.printIn("set to 1 " + c.getTime()) ;
```

日付、時刻を 2004 年 1 月 7 日 15 時 40 分に設定する（月が 0 から始まっている点に注意）

時刻の値をミリ秒に換算する

day1 の値に 1 時間（ミリ秒換算で）を加え、時刻を更新する（"+="は、"day1 = day1 +..."と同様の意味）

日付に 35 を加えて 2 月にする

同じく日付に 35 を加えるが、roll() を使った場合は月が変わらず、2 月のまま「回転」する

インクリメントするのではなく、日付を「1」に設定する

```
File  Edit  Window  Help  Time-Flies
new hour 16
add 35 days Wed Feb 11 16:40 MST 2004
roll 35 days Tue Feb 17 16:40:41 MST 2004
set to 1 Sun Feb 01 16:40:41 MST 2004
```

実行結果で各メソッドの機能をよく確認する

This PDF is prepared for tools@maruhouse.org, personal use only.

Calendar APIの主要機能

Calendarクラスには、すでに少し触れたものに加え、以下のようなフィールドやメソッドがあります。このAPIは規模が大きく、フィールドやメソッドが数多く存在するため、ここでは特に使用頻度の高いものに絞って紹介しておきます。ただ、いくつか使い方を理解すれば、残りについても比較的容易に使えるようになるはずです。

Calendarクラスの主要メソッド

add(int field, int amount)
指定のフィールドの値をインクリメント、デクリメントする。

get(int field)
指定のフィールドの値を戻す。

getInstance()
Calendarのサブクラスのインスタンスを戻す（どの暦、ロケールに対応するサブクラスのインスタンスを戻すかは指定できる）

getTimeInMillis()
時刻をミリ秒単位で（long型の値として）戻す。

roll(int field, boolean up)
上位のフィールドの値に変更を加えずに、指定のフィールドの値をインクリメント、デクリメントする

set(int field, int value)
指定のフィールドの値を設定する

set(year, month, day, hour, minute)（引数の型はすべてint）
年、月、日、時、分のフィールドの値を設定する。set()の引数には他にもバリエーションがあるが、これが比較的使用頻度が高い。

setTimeInMillis(long millis)
時刻をミリ秒単位で（long型の値として）設定。

//その他多数……

Calendarクラスの主要フィールド

DATE / DAY_OF_MONTH
「日」の値の取得、設定に使用する

HOUR / HOUR_OF_DAY
「時（12時制、あるいは24時制）」の値の取得、設定に使用する

MILLISECOND
ミリ秒単位で日付、時刻の値を取得、設定するのに使用する

MINUTE
「分」の値の取得、設定に使用する

MONTH
「月」の値の取得、設定に使用する

YEAR
「年」の値の取得、設定に使用する

ZONE_OFFSET
GMT（グリニッジ標準時）から直接計算したオフセットをミリ秒単位で取得、設定するのに使用する

//その他多数……

This PDF is prepared for tools@maruhouse.org, personal use only.

スタティックインポート

「スタティックインポート」はJava 5.0から導入された機能ですが、一概に「良い」とは言い切れない機能です。気に入って使う人もいるでしょうが、嫌う人も多いはずです。スタティックインポートの長所は、タイピングの手間が省ける、ということだけでしょう。タイピングが面倒という人にはいいかもしれません。ただし、注意しなければ、コードが非常に読みにくくなってしまいます。

スタティックインポートを使用するとコードが読みにくくなる恐れがあるので注意

スタティックインポートを使用しないコード：

```
import java.lang.Math;
class NoStatic {
  public static void main (String [] args) {
      System.out.printIn("sqrt " + Math.sqrt(2.0));
      System.out.printIn("tan " + Math.tan(60));
  }
}
```

スタティックインポートを使用したコード：

スタティックインポートの宣言に使用する構文

```
import static java.lang.System.out;
import static java.lang.Math.*;
class WithStatic {
  public static void main(String [] args) {
      out.printIn("sqrt " + sqrt (2.0));
      out.printIn("tan " + tan (2.0));
  }
}
```

スタティックインポートの使用例

重要ポイント

- 使用するスタティックメンバが1つだけで、しかも数回使うだけ、という場合には、コードを読みにくくするだけなので、スタティックインポートを使うべきではない。
- スタティックメンバを多数使用する場合（数学計算を多数行う場合など）であれば、おそらくスタティックインポートを使っても問題にはならない。
- スタティックインポートの宣言の中では、ワイルドカード（.*）が使用できることに注意。
- スタティックインポートを使用した場合、大きな問題となるのは、名前の重複が起きやすいということ。たとえば、2つのクラスに"add()"というメソッドがあると、コンパイラにはそれがどちらのものなのか区別ができなくなる。

This PDF is prepared for tools@maruhouse.org, personal use only.

特別座談会

今夜は、インスタンス変数さんとスタティック変数さんがゲストですが、どうもインスタンス変数さんの態度には問題があるようです ……

インスタンス変数：どうも座談会を開くほどの意味があるかどうか疑問ですね。スタティック変数さんはせいぜい定数に使われるくらいでしょう。そもそも使用頻度がさほど高いとは思えない。APIの中でも使われているのは、4つくらいのものでしょう。ともかく、皆が使うようなものではないと思いますよ。

スタティック変数：ちゃんと確かめてものを言って欲しいですね。あなたがAPIを最後に確認したのはいつなんですか。APIは今やスタティック変数だらけですよ。定数を保持するだけのために作られているクラスもあるくらいです。たとえば、SwingConstantsなどがそうです。

インスタンス変数：「スタティック変数だらけ」ですか。そうなのかもしれません。Swingライブラリにいくつかあるのは知っていますよ。でも、Swingをプログラミングに使うのはむしろ特殊なケースでしょう。

スタティック変数：特殊なケースという言えばそうですが、Swingライブラリの役割が重要であることも確かですよ。Colorクラスのことも忘れて欲しくないですね。自分でRGB値を設定して色を作るのがどれほど大変かは知っているでしょう。でも、実際には、青、白、赤といった一般的な色に対応する定数があらかじめ用意されています。これは非常に便利ですよね。

インスタンス変数：それはまあわかります。でも、GUI以外でスタティック変数が役立っている例を1つでもあげることができますか。本当に多くの人に役立っているという例ですよ。

スタティック変数：System.outはどうですか。outはSystemクラスのスタティック変数です。outへのアクセスの際は、Systemクラスのインスタンスは作成せず、クラス名である「System」を使用します。

インスタンス変数：System.outも「多くの人に役立っている」とは言えないと思いますね。使うのはデバッグの時くらいでしょう。

スタティック変数：デバッグが重要でないとお考えですか。驚きですね。
あなたのような頭の固い方には思いもよらないでしょうが、スタティック変数には大きなメリットがあるんですよ。まず、効率的です。インスタンスごとではなく、クラスに1つだけ存在するので、メモリが大幅に節約できます。

This PDF is prepared for tools@maruhouse.org, personal use only.

インスタンス変数：1つ忘れていませんか。

スタティック変数：何ですか。

インスタンス変数：スタティック変数はオブジェクト指向の考え方に反するということです。オブジェクト指向のはずのJava言語で、なぜわざわざ、手続き指向的なスタティック変数を使うんですか。時代を逆戻りさせることになるのではありませんか。

スタティック変数：「オブジェクト指向の考え方に反する」とはどういうことですか。

インスタンス変数：スタティック変数はグローバル変数のようなものでしょう。どれほど初歩のプログラマでもグローバル変数の弊害は知っていますよ。

スタティック変数：スタティック変数はグローバル変数とは違いますよ。まったく違います。特定のクラスに属する変数なんです。「オブジェクト指向的でない」という指摘はまったく当てはまらないと思いますね。オブジェクトを構成する要素には違いないんです。オブジェクトの外に存在するわけではありません。インスタンス変数と異なるのは、同じクラスのすべてのインスタンスに共有されるという点だけです。このことが効率向上に役立つんです。

インスタンス変数：オブジェクトではなく「クラス」に属するわけですよね。では、「オブジェクト指向」ではなく「クラス指向」とでも呼ぶんですか。バカげてますよ。所詮、スタティック変数など過去の遺物ということです。頭の古い人がJavaに馴染みやすいように用意されているだけです。

スタティック変数：もういいでしょう。そのへんでやめておいた方がいいですよ。あなたの言うことはまったくのデタラメです。スタティック変数の中にはシステムにとって非常に重要なものもあります。また、さほど重要でないものでも、便利であることは間違いありません。

インスタンス変数：それは、時にはスタティック変数を使うと良い場合もあるでしょう。でも、スタティック変数やスタティックメソッドを濫用するのは、オブジェクト指向プログラマとしてはまだ未熟である証拠だと思いますよ。できる限り使わないよう考えるべきじゃないでしょうか。

スタティック変数：何を根拠にそんなことを言うんですか。スタティックメソッドの何が問題なんですか。

インスタンス変数：スタティックメソッドこそ最悪です。スタティックメソッドを使うというのは、多くの場合、「手続き指向的」に考えている証拠だからです。オブジェクトごとのインスタンス変数の値に左右されないメソッド、というのは、オブジェクト指向言語のプログラムに使うものとしてはあまりにおかしいですよ。

スタティック変数：もちろん、オブジェクト指向プログラミングでは、オブジェクトが中心に置かれるべきとは思います。ただ、それにとらわれ過ぎると、大事なことを忘れてしまう危険もありますよ。現にそういう困ったプログラマがたくさんいます。スタティックメソッドやスタティック変数をどうしても使わねばならない場合もあるということは覚えておかねば。

インスタンス変数：なるほどね。そうまでおっしゃるなら、そうかもしれませんが……

This PDF is prepared for tools@maruhouse.org, personal use only.

エクササイズ

コンパイラになってみよう

左のソースコードを、コンパイラになったつもりで見てみましょう。そして、コンパイルが正しく行えるかどうかを考えてください。コンパイルできると判断した場合は、実行結果が右側に示したもののうちどちらになるかも答えてください。コンパイルできないと判断した場合は、コンパイルできるようコードを修正してください。

```
class StaticSuper {

  static {
    System.out.println("super static block");
  }

  StaticSuper {
    System.out.println("super constructor");
  }
}

public class StaticTests extends StaticSuper {
  static int rand;

  static {
    rand = (int) (Math.random() * 6);
    System.out.println("static block " + rand);
  }

  StaticTests() {
    System.out.println("constructor");
  }

  public static void main(String [] args) {
    System.out.println("in main");
    StaticTests st = new StaticTests();
  }
}
```

コンパイルできた場合の実行結果はどちらになるでしょうか

実行結果 1

```
File  Edit  Window  Help  Cling
% java StaticTests
static block 4
in main
super static block
super constructor
constructor
```

実行結果 2

```
File  Edit  Window  Help  Electricity
% java StaticTests
super static block
static block 3
in main
super constructor
constructor
```


This PDF is prepared for tools@maruhouse.org, personal use only.

エクササイズ

○か×か

次の文は、いずれもこの章で学んだことがらについて述べたものです。正しいものに○を、誤っているものに×をつけてください。

1. Mathクラスを使用する際にはまず、Mathクラスのインスタンスを作成する。
2. コンストラクタには「static」というキーワードを使うことができる。
3. スタティックメソッドは、特定のオブジェクトのインスタンス変数にアクセスできない。
4. スタティックメソッドは、オブジェクトへの参照を納めた変数を利用して呼び出すのが望ましい。
5. あるクラスのインスタンスがいくつ作られたかを記録するにはスタティック変数を使用する。
6. コンストラクタはスタティック変数が初期化される前に呼び出される。
7. “MAX_SIZE”はstatic final変数の名前としてふさわしい。
8. スタティックイニシャライザは、クラスのコンストラクタより前に実行される。
9. クラスをfinal宣言した場合には、そのクラスのメソッドもすべてfinal宣言しなければならない。
10. final宣言されたメソッドは、クラスを継承した場合にはオーバーライドできる。
11. boolean型の値に対応するラッパは存在しない。
12. ラッパは、プリミティブ型の値をオブジェクトとして使用したい場合に使う。
13. “parseXxx”という名前のメソッドは必ずStringを戻す。
14. フォーマッティングに関するクラスはjava.formatに用意されている。

This PDF is prepared for tools@maruhouse.org, personal use only.

プログラミングマグネット

この章では日付、時刻の扱いについて学びましたが、それを応用したパズルに挑戦してみましょう。以下に示したのは、満月の日付、時刻を知らせるプログラムです。図のように、月が満月になる日付、時刻を出力するようになっています。順序がバラバラになっているので、並べ替えて正しいJavaプログラムにしてください。ここでは、満月になる周期が29.52日で、2004年1月7日15時40分が満月であると仮定しています（満月の日付、時刻はタイムゾーンによって変わってきます）。{、}などが抜けていれば付け加えてもかまいません。中には不要なコードも含まれている可能性があるので注意してください。

```
long day1 = c.getTimeInMillis();
c.set(2004,1,7,15,40);
import static java.lang.System.out;
static int DAY_IM = 60 * 60 * 24;
("full moon on %tc", c));
(c.format
Calendar c = new Calendar();
class FullMoons {
public static void main(String [] args) {
day1 += (DAY_IM * 29.52);
for (int x = 0; x < 60; x++) {
static int DAY_IM = 1000 * 60 * 60 * 24;
println
import java.io.*;
("full moon on %t", c));
import java.util.*;
static import java.lang.System.out;
c.set(2004,0,7,15,40);
out.println
c.setTimeInMillis(day1);
(String.format
Calendar c = Calendar.getInstance();
```

```
File  Edit  Window  Help  Howl
% java FullMoons
full moon on Fri Feb 06 04:09:35 MST 2004
full moon on Sat Mar 06 16:38:23 MST 2004
full moon on Mon Apr 05 06:07:11 MDT 2004
```


This PDF is prepared for tools@maruhouse.org, personal use only.

エクササイズの答え

○か×か

1. Mathクラスを使用する際にはまず、Mathクラスのインスタンスを作成する　×
2. コンストラクタには「static」というキーワードを使うことができる　×
3. スタティックメソッドは、特定のオブジェクトのインスタンス変数にアクセスできない　○
4. スタティックメソッドは、オブジェクトへの参照を納めた変数を利用して呼び出すのが望ましい　×
5. あるクラスのインスタンスがいくつ作られたかを記録するにはスタティック変数を使用する　○
6. コンストラクタはスタティック変数が初期化される前に呼び出される　×
7. “MAX_SIZE” はstatic final変数の名前としてふさわしい　○
8. スタティックイニシャライザは、クラスのコンストラクタより前に実行される　○
9. クラスをfinal宣言した場合には、そのクラスのメソッドもすべてfinal宣言しなければならない　×
10. final宣言されたメソッドは、クラスを継承した場合にはオーバーライドできる　×
11. boolean型の値に対応するラッパは存在しない　×
12. ラッパは、プリミティブ型の値をオブジェクトとして使用したい場合に使う　○
13. “parseXxx” という名前のメソッドは必ずStringを戻す　×
14. フォーマッティングに関するクラスはjava.formatに用意されている　×

コンパイラになってみよう

```
StaticSuper () {
  System.out.println(
    "super constructor");
}
```

StaticSuperはコンストラクタなので、()が必要。この点を修正した場合の実行結果は以下のようになる。スタティックイニシャライザの部分がコンストラクタよりも先に実行されている点に注意

実行結果

```
File  Edit  Window  Help  Cling
% java StaticTests
super static block
static block 3
in main
super constructor
constructor
```


This PDF is prepared for tools@maruhouse.org, personal use only.

エクササイズの答え

プールパズル

```
import java.util.*;
import static java.lang.System.out;
class FullMoons {
  static int DAY_IM = 1000 * 60 * 60 * 24;
  public static void main(String [] args) {
    Calendar c = Calendar.getInstance();
    c.set(2004,0,7,15,40);
    long day1 = c.getTimeInMillis();
    for (int x = 0; x < 60; x++) {
      day1 += (DAY_IM * 29.52)
      c.setTimeInMillis(day1);
      out.println(String.format("full moon on %tc" , c));
    }
  }
}
```

実際の満月の日付、時刻は、このプログラムの結果とは 1 日くらいずれることもあります。満月の周期は、様々な要因で変化するので、完全なプログラムを作るのは容易ではありません。ここでのエクササイズではそこまでは必要ないでしょう。

タイムゾーンの違いに対応できるようプログラムを改造するのも練習になるでしょう。どこを変えればいいか考えてみてください。

実行結果

```
File  Edit  Window  Help  Howl
% java FullMoons
full moon on Fri Feb 06 04:09:35 MST 2004
full moon on Sat Mar 06 16:38:23 MST 2004
full moon on Mon Apr 05 06:07:11 MDT 2004
```


This PDF is prepared for tools@maruhouse.org, personal use only.

11章　例外処理

危険をともなうプログラムの作成

プログラムにはトラブルはつきものです。原因は、必要とされているファイルが実は存在しない、サーバーがダウンしている……など、さまざまです。プログラマがいかに優秀でも、あらゆる物事を完全にコントロールすることはできません。どれほど工夫しようと、ダメな時はダメなのです。トラブルの起きそうな処理を行うプログラムを書く時には、あらかじめそのトラブルに対処するコードを組み込んでおく必要があります。ただ、問題は、トラブルが起きそうかどうかをどのように判断するか、ということです。トラブルに対処するコードをどこに入れるかということも知っておかねばなりません。本書ではこれまでのところ、そうした点についてはまだ触れていません。実行するとエラーになるプログラムはいくつか紹介しましたが、それはプログラムそのものにはじめから欠陥（バグ）があったからです。その場合には、バグを修正しさえすればエラーは起きなくなります。それとは異なり、どれほど慎重に正しいプログラムを作っても、確実に動作する保証がないという場合もあるのです。たとえば、あるファイルが特定のディレクトリに存在するという前提で動くプログラム、特定のサーバが動作しているという前提で動くプログラム、あるスレッドがスリープ状態にあるという前提で動くプログラムなどが例としてあげられるでしょう。この章では、その種のプログラムの作り方を、JavaSound APIを利用した音楽演奏プログラムを例に解説します。

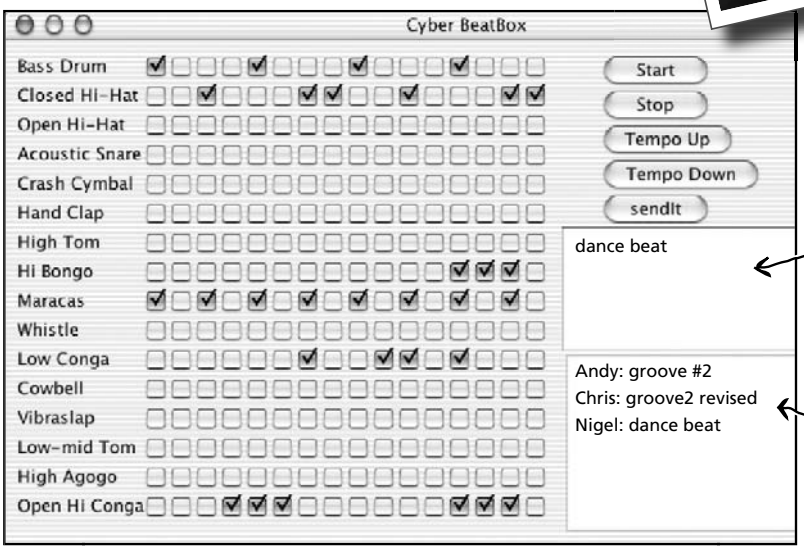
This PDF is prepared for tools@maruhouse.org, personal use only.

音楽演奏プログラムを作る

この後は、いくつかの章にわたって、ドラムマシンプログラム「BeatBox」をはじめ、何種類かの音楽演奏プログラムを作っていきます。本書を読み終わる頃には、チャットルームに文章を送るのと同じ要領で、作ったドラムパターンをサーバに送る、というようなこともできるようになるはずです。基本的にはどのプログラムも著者があらかじめ用意したコードを利用して作ることができるのですが、ゼロから自分で作るための方法も紹介します。もちろん、現実にはビジネスの世界などで音楽演奏プログラムのニーズが強い、というようなことはありません。この種のプログラムを作るのはあくまでJavaをより深く、そして楽しく学ぶためです。

BeatBoxプログラムの実行画面は以下のようになります。

チェックボックスに「✓」マークを付けることで、ドラムパターン（16ビットのパターン）を作ることができる



作成したドラムパターンは「sendit」ボタンでサーバーに送れるが、送信する際には、ここに「このパターンはどのようなものであるか」というメッセージを書き込む

他のプログラムからドラムパターンとともにサーバに送られてきたメッセージがここに表示される。メッセージをクリックすればパターンがロードされる。「Start」ボタンをクリックすればパターンが再生される

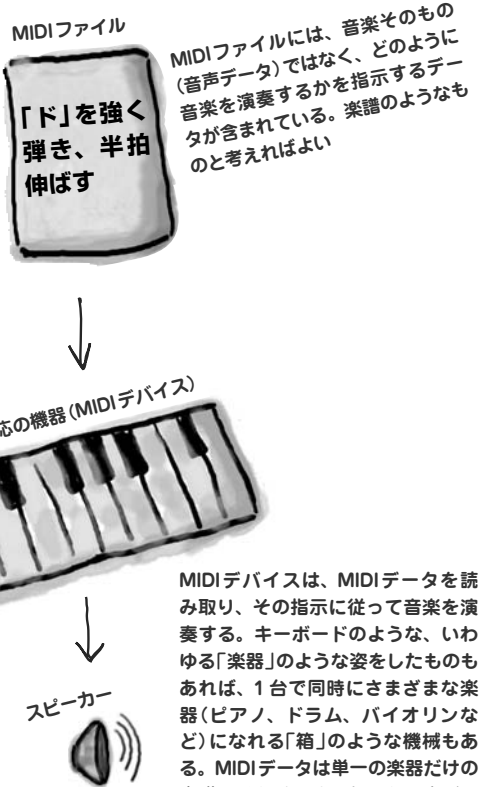
画面上のチェックボックスに「✓」マークを付けていけば、16ビットのドラムパターンが自由に作れます。上の例では、まずベースドラムとマラカスが鳴った後、一瞬無音になり、続いてマラカスと閉じた状態のハイハットシンバルの音が鳴る……というようなパターンになっています。「Start」ボタンをクリックすればパターンは再生され、「Stop」ボタンがクリックされるまで繰り返されます。作成したパターンはいつでもBeatBoxサーバに送れます（サーバに送ると、他の人のBeatBoxプログラムでパターンが利用できるようになる）。また、パターンに添付されたメッセージをクリックすることで、BeatBoxサーバからパターンをロードすることも可能です。

This PDF is prepared for tools@maruhouse.org, personal use only.

基礎的なことがらについて学ぶ

プログラムの具体的な作り方の前に、前提となる基礎的なことがらについて解説しておきましょう。そうした「基礎的なことがら」の中には、Swing GUIの作り方、マシン間のネットワーク接続の方法、マシン間でデータを送受信するためのI/Oなどが含まれます。

そして、忘れてはならないのがJavaSound APIです。この章ではまずこのJavaSound APIから話を始めることになります。今のところはひとまずGUIやネットワーキング、I/Oなどについては考えなくてかまいません。MIDIサウンドをコンピュータ上で取り扱うことについてのみ理解できればいいでしょう。また、MIDIについてよく知らないという人や、音楽に関する知識があまりない（楽譜が読めない、など）という人でも心配はいりません。知っておくべきことがらについてはすべてこの章の中で説明します。「ほとんど何も知らなかった」という人でもかなり本格的なことができるはずです。



JavaSound API

JavaSound APIは、バージョン1.3からJavaに追加されたAPIです。J2SEの標準クラスライブラリに含まれているので、さほど特殊なものではありません。JavaSound APIは大きく、MIDIパッケージ、Sampledパッケージの2つに分けることができます。本書ではもっぱらMIDIパッケージを使用します。MIDI（Musical Instrument Digital Interface）は、さまざまな電子楽器やコンピュータなどの間でデータをやりとりするための規格です。今のところは、「MIDIデータは電子楽器やコンピュータが音楽を演奏するための楽譜のようなものである」という程度の認識でかまわないでしょう。MIDIデータ（MIDIファイル）には「音楽そのもの」ではなく、「どのような音楽を演奏すればよいか」という命令が含まれているからです。MIDIデータをHTMLデータのようなものと考えてもわかりやすいかもしれません。電子楽器やコンピュータはWebブラウザがHTMLデータを読み取るのと同じようなかたちでMIDIデータを読み取るからです。

MIDIデータに「音楽そのもの」が含まれておらず、どのような音楽を演奏すればよいかの命令だけが含まれているとすれば、逆に言えば、MIDIデータは、演奏のための機器（MIDIデバイス）を必要とするということになります。MIDIデータだけでは、「ド」の音を強く鳴らし、半拍伸ばす、といったことがわかるだけで、実際の音はまったく聞こえてこないのです。注意すべきなのは、MIDIデバイスには、キーボードなどのいわゆる「楽器」の姿をしたものもあれば、1台でバンドやオーケストラにもなれる「箱」のような機械もあるということです。また、コンピュータ上のソフトウェアがMIDIデバイスとして機能することもあります。

BeatBoxプログラムで使用するのは、Javaソフトウェアの MIDIデバイスです。自在にさまざまな音色を作り出せることから「シンセサイザー」のようなもの、と考えれば間違いはないでしょう。

This PDF is prepared for tools@maruhouse.org, personal use only.

まず必要なのは Sequencerオブジェクト

音楽演奏のためには、まずSequencerというオブジェクトが必要になります。Sequencerオブジェクトは、MIDIデータを適切なMIDIデバイスに送るという役割を果たすオブジェクトです。事実上、音楽を演奏するオブジェクトと言ってもいいでしょう。Sequencerオブジェクトにはさまざまな機能がありますが、本書では、音楽演奏（再生）の機能だけを利用します。音楽を再生するだけ、という点ではCDプレーヤーに似ているかもしれません（もう少し機能は多いですが）。Sequencerはjavax.sound.midiパッケージ（Javaのバージョン1.3からは標準のライブラリに含まれる）のクラスです。では、Sequencerオブジェクトの作り方から見ていきましょう。

```
import javax.sound.midi.*;
public class MusicTest1 {

  public void play() {
    Sequencer sequencer = MidiSystem.getSequencer();

    System.out.println("We got a sequencer");
  } // playメソッドの終了

  public static void main(String [] args) {
    MusicTest1 mt = new MusicTest1();
    mt.play();
  } // mainメソッドの終了
} // クラスの終了
```

javax.sound.midiパッケージをインポートする

Sequencerオブジェクトは、MIDIデバイス（楽器）にとって重要な役割を果たすオブジェクト。事実上、MIDIデータを基に音楽を演奏するオブジェクトとも言える。Sequencerオブジェクトはプログラマが自ら「new」を使って作ることはできない。このコードのように、MidiSystemクラスを利用して取得する

このコードには誤りが……

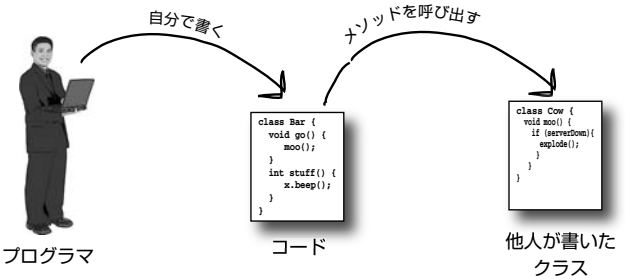
上記のコードをコンパイルしようとすると、以下のようなエラーメッセージが表示される。これは、"MidiUnavailableException"という例外が発生する恐れがあるのに、それに対処するコードがない、という意味のメッセージ

```
File Edit Window Help SayWhat?
% javac MusicTest1.java
MusicTest1.java: 13:unreported exception javax.sound.midi.
MidiUnavailableException; must be caught or declared to be
thrown

    Sequencer sequencer = MidiSystem.getSequencer();
                                     ^
1 errors
```

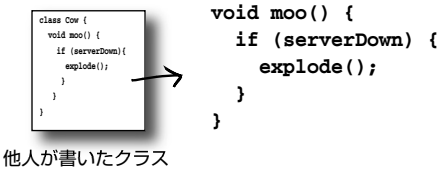

This PDF is prepared for tools@maruhouse.org, personal use only.

正しく動作しない恐れがある（他人が書いた）メソッドを呼び出す

① 自分以外の人が書いたクラスのメソッドを呼び出す。

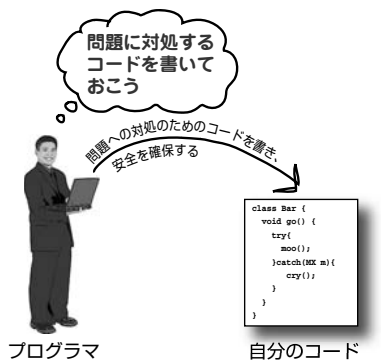


② 呼び出したメソッドがプログラム実行時に正しく動作するとは限らない、という場合もある。



③ 呼び出すメソッドが確実に正しく動作しない恐れがある場合は、そのことを事前に知っておく必要がある。

④ 万が一にでも、正しく動作しない恐れがあることがわかれば、発生し得る問題に対処するコードをあらかじめ書いておく。



* explode = 爆発

This PDF is prepared for tools@maruhouse.org, personal use only.

Javaのメソッドでは、実行中に何か問題が起きたことを呼び出し側のコードに知らせるために「例外」を発生させる

Javaでは、この「例外」が発生した場合、その状況に対処するコード（例外処理のコード）を非常に簡単に、しかも書いた本人以外にもわかりやすいかたちで書くことができます。ただ、そうした例外処理のコードを書くためには、呼び出そうとしているメソッドが例外を発生させる可能性があるかを知る必要があります。例外が発生する可能性があることが事前にわかっていれば、適切な例外処理のコードを書き、致命的な状況に陥ることを防止したり、プログラムの動きを正常に戻したり、といったことができます。

あるメソッドが例外を発生させる可能性があるかどうかはメソッドの宣言でわかります。例外を発生させる可能性がある場合には宣言に「throws」というキーワードが使われるからです。

getSequencer()メソッドは実行時に例外を発生させる可能性がある。そのことは宣言を見ればわかる。

この画面はAPIのドキュメント。APIのドキュメントでgetSequencer()の宣言を見ると、「MidiUnavailableException」という例外を発生させる可能性があることがわかる

ここに、「MidiUnavailableException」という例外がどのような条件で発生するかが書かれている。MidiUnavailableExceptionは、リソースの制限によってSequencerオブジェクトが利用できない場合（つまり、すでに他のプログラムでSequencerオブジェクトが利用されている場合）に発生する

This PDF is prepared for tools@maruhouse.org, personal use only.

前略　コンパイラ様

自分が呼び出しているメソッドが例外を発生させる可能性があることはわかっています。でも、そのことをどうやってあなたに知らせたらよいですか。

名もないプログラマより

ワイキキにて

名もないプログラマ様

try/catchブロックを使ってください。tryブロックでメソッドを呼び出し、catchブロックで例外を処理するのです。そうすればあなたが危険を覚悟の上でメソッドを呼び出していることが私にわかります。

try/catchブロック

例外を発生させる可能性があるメソッドを呼び出す際には、try/catchブロックと呼ばれる形式のコードを使用します。これにより、「例外を発生させる可能性があるメソッドを、そうとわかって呼び出している」ということがコンパイラに知らされるのです。

例外処理のためのコードも、このtry/catchブロックの中に記述します。例外処理をどのように行うかということにはコンパイラは関知しません。ただ、try/catchブロックが使われていれば、「例外に対処するための準備はできている」とみなすのです。

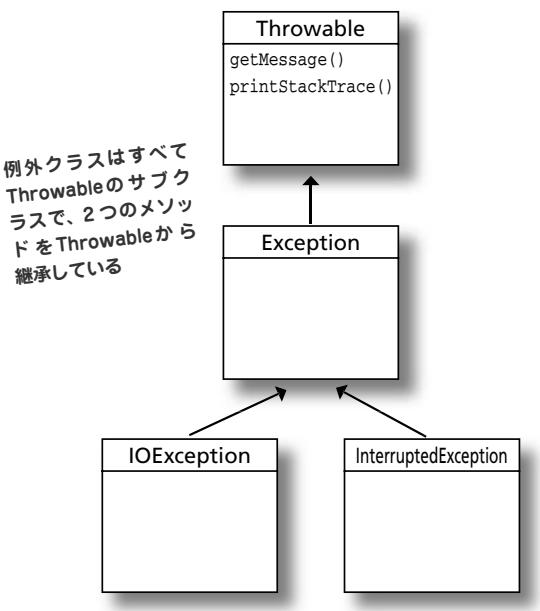
```
import javax.sound.midi.*;

public class MusicTest1 {
  public void play() {

    try {
      Sequencer sequencer = MidiSystem.getSequencer();
      System.out.println("Successfully got a sequencer");
    } catch(MidiUnavailableException ex) {
      System.out.println("Bummer");
    }
  } // playメソッドの終了

  public static void main(String [] args) {
    MusicTest1 mt = new MusicTest1();
    mt.play();
  } // mainメソッドの終了
} // クラスの終了
```

例外を発生させる可能性のあるメソッドの呼び出しはtryブロックに入れる

例外処理のためのコードはcatchブロックに入れる。例外「MidiUnavailableException」はgetSequencer()が発生させる

This PDF is prepared for tools@maruhouse.org, personal use only.

絶対にマネしないでください



例外クラスはすべてThrowableのサブクラスで、2つのメソッドをThrowableから継承している

例外はすべてException型を継承するクラスオブジェクト

Javaでは、例外もやはりオブジェクトとなります。例外のクラスは数多く存在するのですが、いずれもExceptionクラスを継承します。「Exception」というのは「例外」という意味の単語ですから覚えやすいと思います。

ここまで読んできた人にはわかると思いますが、例外がすべてExceptionのサブクラスのオブジェクトであるということは、Exception型の変数で、あらゆる例外に対応できるということを意味します(ポリモーフィズム)。

catchブロックはいわば、この例外オブジェクトを「捕まえる」ためのものです。以下のコードでは、catchブロックの引数exの型がExceptionとなっていますが、この変数であらゆる例外を「捕まえる」ことができるということになります。

```
try {
   // 例外を発生させる恐れの
   // あるメソッドを呼び出す

} catch (Exception ex) {
   // 例外処理を行う
}
```

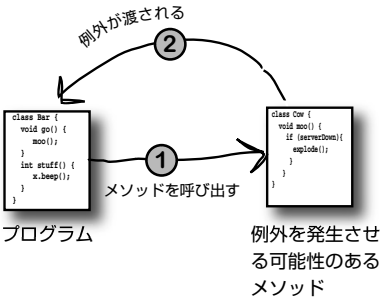
メソッドの引数と同様に宣言する

このコードは例外発生時にのみ実行される

catchブロックにどのようなコードを書くべきかは、発生する例外によって変わってきます。たとえば、メソッドを呼び出した際にサーバがダウンしている恐れがあるという場合には、おそらくcatchブロックに書くコードは他のサーバにアクセスを試みるようなものになるでしょう。必要なファイルが存在していない恐れがある場合には、ファイルを見つけるためのコードを書くといいかもしれません。

This PDF is prepared for tools@maruhouse.org, personal use only.

例外を発生させるメソッドを作る



プログラマは、例外処理のコードを書くだけではなく、自ら例外を発生させるメソッドを作ることもあります。ただ、Javaでは、例外処理のコードを書く機会の方がはるかに多いと言えます。今のところは、宣言に「throws」というキーワードが使われているメソッドは例外を発生させる可能性があるのだ、ということさえわかっていればあまり困ることはないでしょう。

いずれ例外を発生させるメソッドを自らも書くことがあるでしょうが、誰がメソッドを作ったにしても、呼び出す際には例外処理のコードが必要になるという点は同じです。

例外を発生させるメソッドの書き方と呼び出し方の基本を以下にまとめておきます。

① 例外を発生させるメソッドを書く

「throws」というキーワードによって、BadExceptionという例外が発生する可能性があることを示す

```
public void takeRisk() throws BadException {
  if (abandonAllHope) {
    throw new BadException();
  }
}
```

例外オブジェクトを作成し、「throw」というキーワードで例外を発生させる

メソッドが発生させた例外は、必ず呼び出し側のメソッドに渡される。呼び出し側のメソッドは例外を「捕まえる」
例外を発生させるメソッドの宣言には「throws」キーワードを使ってそのことが明記される

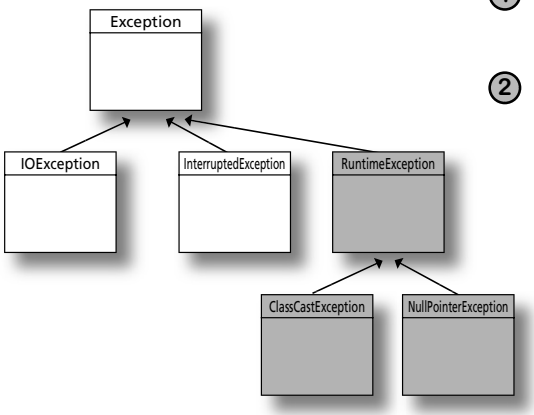
② 例外を発生させる可能性のあるメソッドを呼び出す

```
public void crossFingers() {
  try {
    anObject.takeRisk();
  } catch (BadException ex) {
    System.out.println("Aaargh!");
    ex.printStackTrace();
}
```

このように書いておくと、例外から復旧はできなくても、printStackTrace()メソッド（あらゆる例外クラスが継承しているメソッド）によって最低限スタックトレース（どのような経緯で例外が発生したかの情報）は得られる

This PDF is prepared for tools@maruhouse.org, personal use only.

RuntimeExceptionを継承する例外オブジェクトを除き、例外オブジェクトはそれを処理するためのコードを必要とする。さもなければコンパイルエラーとなる



通常の例外オブジェクトに関してはコンパイルエラーを起こさないため以下の2点に注意

① 例外を発生させるメソッドを作る時は、宣言で「throws」キーワードを使って例外が発生する可能性があることを明記する。

② 例外が発生する可能性があるメソッド（宣言で「throws」キーワードが使われているメソッド）を呼び出す時は、try/catchブロックを使用する。これで、例外発生の可能性を認識していることがコンパイラに知らされる（try/catchブロック以外にも方法はあるが、後述）。RuntimeExceptionを継承するものを除き、すべての例外オブジェクトについて、コンパイラは宣言と処理のコードが正しく書かれていることをチェックする。このように、コンパイラのチェックの対象となる例外のことを「チェック例外」と呼ぶ。

コンパイラは、RuntimeExceptionを継承する例外オブジェクトに関してはチェックを行わない。そこで、この種の例外は「非チェック例外」と呼ばれている。非チェック例外に関しても、発生、処理のコードを明示的に書くことはできるが、コンパイラが書かれているかどうかをチェックしないので、書かなくても問題は起きない

素朴な疑問に答えます

Q：これまで紹介されてきたプログラム例に例外処理のコードがまったくなかったのはなぜですか。例外が発生する恐れのあるメソッドも中にはあったと思うのですが。現に、私は、Integer.parseInt()メソッドがNumberFormatExceptionという例外を発生させるのを見たことがあります。しかし、それを処理するためのコードは書かれていませんでした。

A：RuntimeExceptionを継承する例外に関しては、コンパイラは例外処理のコードがあるかどうかをチェックしません。この種の例外は明示的な処理コードを必要としないのです。実は、RuntimeExceptionを継承する例外は、宣言やtry/catchブロックの有無を問わず、いつでも、どのような状況でも発生する可能性があります。そのため、コンパイラがチェックをしても意味がないのです。

Q：それでもやはり例外処理のコードがないのは問題ではないですか。それによって、例外が致命的な問題に発展し、システムが完全に停止してしまう、というようなことはないのでしょうか。

A：RuntimeExceptionを継承する例外の多くはプログラムのロジックに問題があるために発生します。不測の事態あるいは防ぎようのない事故などを原因として発生することはむしろ少ないと言えます。もちろん、あるはずのないファイルが知らない間になくなる、サーバが突然停止してしまう、といったことは不測の事態と言っていいでしょうが、配列にサイズを超える数の要素が追加される、というようなことは、事前に十分に考えておけば防げるはずです（そのために.lengthという属性も用意されています）。
開発、テスト時にはRuntimeExceptionを継承する例外が頻繁に発生することも考えられますが、それでもtry/catchブロックを書く必要はありません。ただオーバーヘッドを増やすだけだからです。それよりもできるだけ例外が発生しないよう考える方が得策です。
try/catchブロックは不測の事態に備えるためのものであって、プログラムの欠陥を補うためのものではありません。やむを得ず予想どおり進まない恐れのある処理を行う場合に使用するのです。catchブロックには最低限、例外が発生したことを知らせるメッセージを表示するためのコード、スタックトレース（どのような経緯で例外が発生したかの情報）を得るためのコードは入れるようにします。

This PDF is prepared for tools@maruhouse.org, personal use only.

重要ポイント

- メソッドの中には実行に際して問題が起きると例外を発生させるものがある。
- 例外は常にExceptionを継承するクラスのオブジェクト（つまり、Exception型の変数であらゆる例外に対応できる）。
- コンパイラは、RuntimeException を継承する例外には関知しない。この種の例外に関しては、発生する可能性があることをメソッドの宣言で示す必要はなく、例外処理のためのtry/catchブロックを用意する必要もない（ただし、そうしてはいけない、ということではない）。
- コンパイラが関知する種類の例外のことを「チェック例外（コンパイラがチェックする例外という意味）」と呼ぶ。それに対し、Runtime Exceptionを継承する例外を「非チェック例外」と呼ぶ。チェック例外を発生させる可能性のあるメソッドを呼び出す時は、try/catchブロックを使用する必要がある。これにより、確かにそれを認識しているということをコンパイラに知らせることができる。
- メソッド中のどこで例外を発生させるかを明確にさせるには、以下のように「throw」というキーワードに続けて、例外オブジェクトを作成するコードを書く。

```
throw new NoCaffeineException();
```

- チェック例外を発生させるメソッドを作る時は、宣言で「throws」というキーワードを使って、「例外を発生する可能性がある」ということを明記する。
- チェック例外を発生させる可能性のあるメソッドを呼び出す際には、「例外が発生する可能性があることをわかってメソッドを呼び出している」ということをコンパイラに知らせる必要がある。
- try/catchブロックを使用してメソッドを呼び出し、catchブロックに例外処理のコードを書けば、「例外が発生する可能性があることをわかってメソッドを呼び出している」ということをコンパイラに知らせることができる。
- try/catchブロックを使わなくてもコンパイルエラーが起きないようにする方法もあるが、それについては後述。

「認知科学的」アドバイス

何か新しいことを学ぶ時は一日の最後がいいようです。ですから、今日、この後、本書を（不本意にも）閉じることになったら、後は寝るまで読むものはシリアルのパッケージに書かれている説明書きくらいにとどめ、決して他に難しい本などは読まないようにしてください。脳は読んだことや学んだことを処理するのにある程度時間を必要とするのです。だいたい数時間は必要だと考えておいた方がいいでしょう。せっかくJavaを勉強しても、寝る前に何か他のことを頭に入れてしまうと、学んだことが頭から押し出されてしまいます。

ただし、同じ学ぶのでも、体を使うことは例外です。寝る前にキックボクシングの練習をしたからといって、Javaの学習に悪影響を与えるということはありません。

ともかく、学習効果を高めるため、寝る前には本書を読む（挿し絵を見るだけでもよい）習慣をつけるようにしてください。

自分で考えてみよう

プログラムの中で左の文に書かれたような処理を行う場合、それぞれどのような問題が発生すると処理が正しく行われなくなると考えられますか。1つずつ答えてください（最初の文に関してはすでに解答が書いてあります）

処理	発生する問題
✓ リモートサーバーへの接続を行う	サーバーがダウンする
＿＿ 配列の要素にアクセスする	
＿＿ 画面上にウィンドウを表示する	
＿＿ データベースからデータを抽出する	
＿＿ 特定のディレクトリに存在するはずのテキストファイルを利用する	
＿＿ 新しいファイルを作成する	
＿＿ ユーザーの入力した文字を読み取る	

This PDF is prepared for tools@maruhouse.org, personal use only.

try/catch ブロックでの処理の流れ

例外を発生させる可能性のあるメソッドを呼び出した場合、その結果は 2 とおり考えられます。1 つは例外が発生せず、処理が正しく終了するというもの、そしてもう 1 つは処理が正しく行われず、例外が発生するというものです。前者の場合は、try/catch ブロックのうち try ブロックだけが実行されます。そして、後者の場合は catch ブロックも実行されます。

メソッド（ここでは doRiskyThing() メソッド）が正しく実行され、例外が発生しなかった場合

```
try {
  ① Foo f = x.doRiskyThing();
     int b = f.getNum();

  } catch(Exception ex) {
     System.out.println("failed");
  }
  ② System.out.println("We made it!");
```

tryブロックが実行された後、catchブロックは飛ばしてその下のコードが実行される

catchブロックのコードは実行されない

```
File Edit Window Help RiskAll
% java Tester
We made it!
```

メソッド（ここでは doRiskyThing() メソッド）が正しく実行されず、例外が発生した場合

```
try {
  ① Foo f = x.doRiskyThing();
     int b = f.getNum();

  } catch (Exception ex) {
  ② System.out.println("failed");
  }
  ③ System.out.println("We made it!");
```

まずtryブロックが実行されるが、doRiskyThing() が 例 外 を発生すると、tryブロックのその後のコードは実行されずcatchブロックが実行される。catchブロック終了後はその後のコードが実行される

tryブロックの残りの部分は飛ばされる。この部分はdoRiskyThing()メソッドの処理が正しく行われる前提で書かれているが、飛ばされるので問題は起きない

```
File Edit Window Help RiskAll
% java Tester
failed
We made it!
```


This PDF is prepared for tools@maruhouse.org, personal use only.

finallyブロック：例外が発生した場合もしなかった場合も行うべき処理

オーブンで料理をする時には、誰でもまずオーブンのスイッチを入れます。そして、料理が失敗に終わった時は、オーブンのスイッチを切ります。また、その一方で料理が見事に成功した場合にもやはりオーブンのスイッチは切ることになります。つまり、結果はどうであれ、オーブンのスイッチを切ることには変わりはないわけです。

例外が発生する、しないに関わらず行うべき処理は、以下のように「finallyブロック」に記述します。

```
try {
  turnOvenOn();
  x.bake();
} catch (BakingException ex) {
    ex.printStackTrace();
} finally {
    turnOvenOff();
}
```

finallyブロックを使わない場合には、以下に示すように、tryブロックとcatchブロックの両方にturnOvenOff()と書かねばなりません。finallyブロックを使うことによってこうした重複を避け、すっきりとしたコードが書けるのです。

```
try {
  turnOvenOn();
  x.bake();
  turnOvenOff();
} catch (BakingException ex) {
    ex.printStackTrace();
    turnOvenOff();
}
```

tryブロックの処理が成功しなかった（例外が発生した）場合は、ただちにcatchブロックが実行される。catchブロックの処理が終了したら、次にfinallyブロックが実行される。finallyブロック終了後は、その後のコードが実行される。

tryブロックの処理が成功した（例外が発生しなかった）場合、catchブロックは飛ばされ、すぐにfinallyブロックが実行される。finallyブロック終了後は、その後のコードが実行される。

tryブロックやcatchブロックにreturn文があったとしても、finallyブロックはやはり実行される。順序としては、finallyブロック実行後にreturn文が実行される、ということになる。

This PDF is prepared for tools@maruhouse.org, personal use only.

自分で考えてみよう

処理の流れを見極める

左のコードでは、3行目に変数testの値が"no"に設定されています。まず、このままのかたちでコードの実行結果がどのようになるかを考えてください。
そして、3行目が「String test = "yes";」に変わった場合に実行結果がどのように変わるかも考えてください。
Scary ExceptionはExceptionを継承するものとします。

```
public class TestExceptions {

  public static void main(String [] args) {

    String test = "no";
    try {
      System.out.println("start try");
      doRisky(test);
      System.out.println("end try");
    } catch(ScaryException se) {
      System.out.println("scary exception");
    } finally {
      System.out.println("finally");
    }
    System.out.println("end of main");
  }

  static void doRisky(String test) throws ScaryException {
    System.out.println("start risky");
    if ("yes".equals(test)) {

      throw new ScaryException();
    }
    System.out.println("end risky");
    return;
  }
}
```

test = "no" の場合の実行結果

test = "yes" の場合の実行結果

test = "no" の場合の処理順序：try ブロックの開始 -doRisky() メソッドの開始 -doRisky() メソッドの終了 -try ブロックの終了 -finally ブロック -main の終了
test = "yes" の場合の処理順序：try ブロックの開始 -doRisky() メソッドの開始 -catch ブロック -finally ブロック -main の終了

This PDF is prepared for tools@maruhouse.org, personal use only.

1 つのメソッドが複数種の例外を発生させることもある

メソッドの中には、複数種の例外を発生させる可能性のあるものも存在します。その場合は、宣言の中で「throws」キーワードに続けて、発生する可能性のある例外がすべて明記されます（例外がすべて共通のクラスを継承するものであれば、スーパークラスのみを宣言に書くこともできる）。

複数の例外に対応する

メソッドの発生し得る例外（チェック例外）が複数の場合、呼び出すコードがすべての例外に対応できるものになっていなければ、コンパイル時にエラーになります。複数の例外に対応するためには、1 つの try ブロックの後に続けて複数の catch ブロックを書くことになります。場合によっては、catch ブロックの並び順にも注意しなければならないことがありますが、その点については後で説明します。

```
public class Laundry {
  public void doLaundry() throws PantsException, LingerieException {
    // 2 つの例外のうちのいずれかを発生させる可能性のあるコード
  }
}
```

これによって、2 つの例外を発生させる可能性のあることが示される

```
public class Foo {
  public void go() {
    Laundry laundry = new Laundry();
    try {
      laundry.doLaundry();
    } catch(PantsException pex) {
      // 例外処理コード

    } catch(LingerieException lex) {
      / / 例外処理コード
    }
  }
}
```

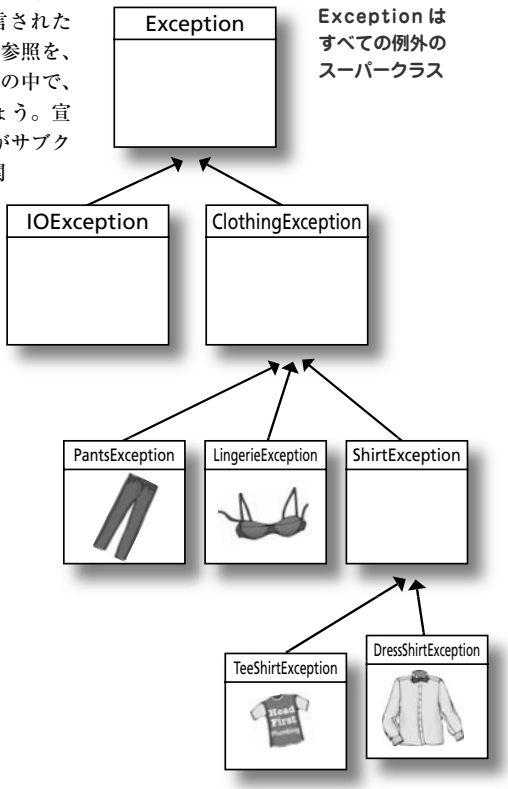
doLaundry() が PantsException を発生させた場合には、上の catch ブロックが実行される

doLaundry() が LingerieException を発生させた場合には、下の catch ブロックが実行される

This PDF is prepared for tools@maruhouse.org, personal use only.

例外とポリモーフィズム

すでに述べたとおり、Javaにおいては例外もやはりオブジェクトです。問題発生時にメソッドによって呼び出し側のコードに渡される、という点を除いては、他のオブジェクトと何ら変わることはありません。例外は、他のオブジェクトが持つ「ポリモーフィズム」という性質も持っています。たとえば、右の図のLingerieExceptionオブジェクトへの参照は、スーパークラスであるClothingExceptionオブジェクトへの参照を納めることができる、と宣言された変数に代入することができます。また、PantsExceptionオブジェクトへの参照を、Exception型の変数に格納することも可能です。その他、メソッドの宣言の中で、発生する例外を完全に特定せずに済む、という点もメリットの1つでしょう。宣言で指定した例外クラスにサブクラスが存在すれば、実際に発生したのがサブクラスの方の例外であっても同じように扱われるのです。catchブロックに関しても同様のことが言えます。「このcatchブロックはどの例外に対処するものか」を完全に明確にする必要はありません。



① 発生する可能性のある例外が共通のスーパークラスを持つ場合は、宣言にスーパークラスの名前を書けばよい

```
public void doLaundry() throws ClothingException {
```

ここにClothingExceptionと書くと、自動的に、「doLaundry()はClothingExceptionを継承する例外が発生することもある」という意味になる。つまり、PantsException、LingerieException、TeeShirtException、DressShirtExceptionなどの例外が発生することもあるということ

② catchブロックの引数にもスーパークラスの名前だけを指定することができる

```
try {
  laundry.doLaundry();
} catch(ClothingException cex) {
  // 例外処理コード
}
```

ClothingExceptionを継承するあらゆる例外に対応できる

```
try {
  laundry.doLaundry();
} catch(ShirtException sex) {
  // 例外処理コード
}
```

TeeShirtException、DressShirtExceptionにのみ対応できる

This PDF is prepared for tools@maruhouse.org, personal use only.

スーパークラスの名前を指定することで あらゆるサブクラスに対応する、という方法を 使うべきでない場合もある。

catchブロックの引数に、以下のように「Exception」と書けば、あらゆる種類の例外に対応するコードが作れます。すべての例外がExceptionのサブクラスだからです。

```
try {
  laundry.doLaundry();
} catch(Exception ex) {
    // 例外処理コード
}
```

これで確かにあらゆる例外に対応できるが、実際にどのような例外に対応すべきかが事前にわからないため、例外処理のコードをどのように書いてよいかわからない

独自の対処を必要とする例外に関しては、 個々に処理コードを書く。

たとえば、TeeShirtExceptionとLingerieExceptionとでは、発生した問題に関し、違う対処が必要であるとします。その場合は以下のように個別に処理コード（catchブロック）を書くのです。また、ClothingExceptionを継承するその他の例外がすべて同じように処理できるのであれば、catchブロックは共通にします。

```
try {
  laundry.doLaundry();

} catch(TeeShirtException tex) {
  //TeeShirtExceptionに対処するコード

} catch(LingerieException lex) {
  //LingerieExceptionに対処するコード

} catch(ClothingException cex) {
  // その他の例外に対処するコード

}
```

TeeShirtException、LingerieExceptionには独自の対処が必要なので、専用のcatchブロックが必要。

ClothingExceptionを継承するその他のすべての例外については、共通のcatchブロックを用意する

This PDF is prepared for tools@maruhouse.org, personal use only.

catchブロックを複数作る場合は、対応する例外クラスが下位のものから順に並べる

TeeShirtException のみに対応するcatch ブロック

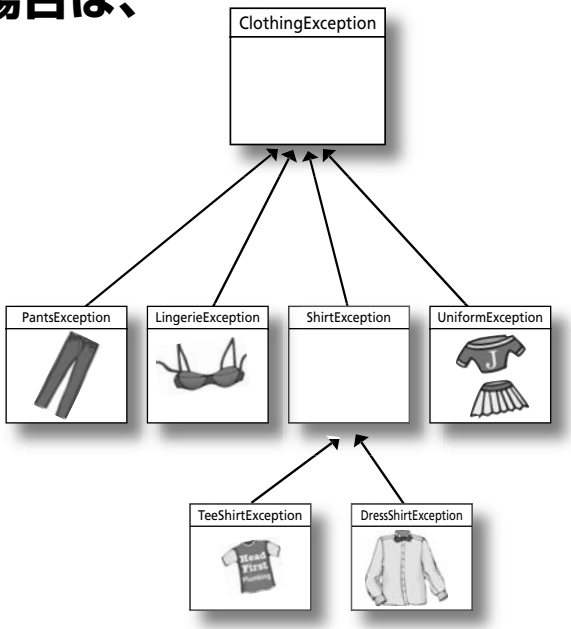
```
catch(TeeShirtException tex)
```

TeeShirtExceptionを除く、ShirtExceptionを継承するあらゆる例外に対応するcatchブロック

```
catch(ShirtException sex)
```

TeeShirtException、ShirtExceptionを除く、ClothingExceptionを継承するあらゆる例外に対応するブロック

```
catch(ClothingException cex)
```



catchブロックの対応する例外クラスが上位のものであればあるほど、ブロックは多様な例外を受け取れるようになります。逆に、対応する例外クラスが下位のものになれば、それだけ受け取れる例外の種類は少なくなります。上位のクラスに対応するよう作っておけば、そのすべてのサブクラスに同時に対応できるからです。

たとえば、ShirtExceptionに対応するよう作られたcatchブロックは、同時にTeeShirtException、DressShirtExceptionなどにも対応できます(その他、後でShirtExceptionのサブクラスが追加された場合もそれに対応できる)。ClothingExceptionに対応するcatchブロックはさらに多くの種類の例外を受け取れます(ClothingExceptionの方がサブクラスが多いため)。ClothingExceptionと同時に、PantsException、UniformException、LingerieException、ShirtExceptionなどにも対応できるのです。catchブロックが受け取れる例外の種類は、引数がException型になった場合に最も大きくなります。引数をException型にした場合、catchブロックは、非チェック例外も含めたあらゆる例外に対応できます。ただし、引数をException型にしたcatchブロックはテストの場合以外にはほとんど使用することはありません。

This PDF is prepared for tools@maruhouse.org, personal use only.

上位の例外クラスに対応するcatchブロックを先にすることはできない

もし、上位の例外クラスに対応するcatchブロックを先にすると、コンパイル時にエラーになります。catchブロックは、オーバーロードしたメソッドとは違い、適切なものが自動的に探されるわけではないのです。catchブロックの場合は、プログラム実行時にJVMによって上から（下位の例外クラスに対応するものから）順に検索され、発生した例外を受け取れるものが見付かったところで検索が終了します。つまり、最も上に置かれたcatchブロックの引数がException型であれば、その下にどのようなcatchブロックを置いても使われることはないのです。

catchブロックがたくさんある時は対応できる例外の種類の多さが重要なのよ。たくさんの例外に対応できるものほど下に置かないと。決まった例外にしか対応しないものを下に置いても意味がないわ

悪い例

```
try {
  laundry.doLaundry();
```

```
} catch(ClothingException cex) {
  // ClothingException に対応するコード
```

```
} catch(LingerieException lex) {
  // LingerieException に対応するコード
```

```
} catch(ShirtException sex) {
  // ShirtException に対応するコード
}
```

同じ階層の例外に対応するcatchブロックの並び順には特にきまりはない。対応する例外が重複することはないため

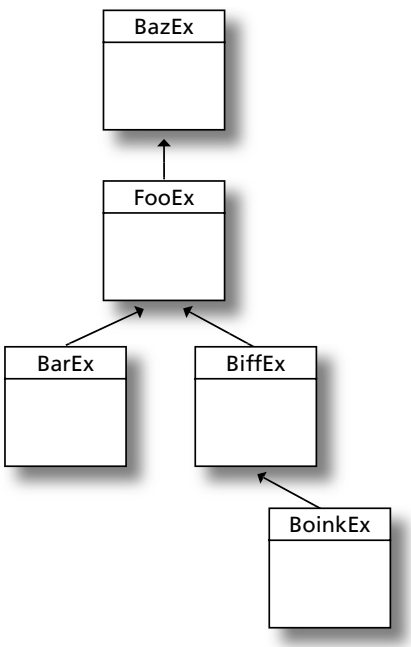
ShirtExceptionに対応するcatchブロックとLingerieExceptionに対応するcatchブロックなら、どちらを上にしてもかまいません。ShirtExceptionにはサブクラスがあるため、受け取れる例外の数自体は多いのですが、両者が受け取るべき例外に重複がないので、別にどちらが上でも問題は起きないのです。

This PDF is prepared for tools@maruhouse.org, personal use only.

自分で考えてみよう

左に示したコードが正しいものだとすると、catch ブロックに対応する例外クラスどうしの関係はどのようになるでしょうか。2 とおり考え、図に表してみてください。

```
try {
  x.doRisky();
} catch(AlphaEx a) {
  // AlphaEx に対応するコード
} catch(BetaEx b) {
  // BetaEx に対応するコード
} catch(GammaEx c) {
  // GammaEx に対応するコード
} catch(DeltaEx d) {
  // DeltaEx に対応するコード
}
```



例外クラスどうしが左の図に示したような関係にある場合、どのような try/catch ブロックを書けば正しく対応できるでしょうか。2 とおり考えてみてください。図に示した例外は、すべて try ブロックで呼び出す 1 つのメソッドによって発生すると仮定します。

This PDF is prepared for tools@maruhouse.org, personal use only.

例外処理をしたくない時は……

「回避！」

先述のとおり、例外を発生させる可能性のあるメソッドを呼び出す際は、コンパイラに「例外が発生するかもしれないことを認識してメソッドを呼び出している」ということを知らせる必要があります。知らせる手段としては、多くの場合、try/catchブロックを使用します。しかし、実はtry/catchブロックを使わない方法もあるのです。それは、例外を発生させるメソッドを呼び出すメソッドの宣言で「throws」キーワードを使う、という方法です。

この「throws」キーワードの後に、本来はtry/catchブロックを使って処理すべき例外名を書くわけです。「throws」キーワードを使ったからといって、本当にそのメソッドで例外を発生させるわけではないのですが、ひとまずこれでコンパイルエラーは回避できます。

しかし、この方法を使った場合、発生した例外はどうなってしまうのでしょうか。try/catchブロックがなくて問題は起きないのでしょうか。

メソッドは、例外を発生させると、即座にスタックからポップされます。そして例外は、スタックですぐ下のフレームに格納されていたメソッド（呼び出し側のメソッド）に渡されます。ただ、このメソッドも、try/catchブロックがない場合には、即、ポップされ、例外はさらに下のメソッドに渡されることになります。try/catchブロックが見付からない限り、これが繰り返されるのですが、結局まったくtry/catchブロックが見付からなければどうなるのでしょうか。その点については少し後で触れます。

```
public void foo() throws ReallyBadException {
    // try/catch ブロックを使わずに例外を発生させるメソッドを呼び出す
    laundry.doLaundry();
}
```

このメソッドが例外を発生させるわけではないが、try/catchブロックを使わなければこの記述が必要になる。これにより、foo()メソッドを呼び出す際には、例外処理が必要になる

This PDF is prepared for tools@maruhouse.org, personal use only.

例外処理の回避は実は「先送り」

例外処理は遅かれ早かれどこかで行う必要があります。しかし、main()メソッドで処理が回避されてしまったらどうなるのでしょうか。

```
public class Washer {
  Laundry laundry = new Laundry();

  public void foo() throws ClothingException {
    laundry.doLaundry();
  }

  public static void main(String [] args) throws ClothingException {
    Washer a = new Washer();
    a.foo();
  }
}
```

いずれのメソッドも「throws」キーワードを使っており、例外処理のためのコードはどこにもない。ただし、このプログラムは正しくコンパイルできる

❶ dolaundry()がClothingExceptionを発生させる。

main()がfoo()を呼び出す。foo()はdolaundry()を呼び出す。dolaundry()は実行中にClothingExceptionを発生させる。

❷ foo()が例外処理を回避する。

dolaundry()は即座にスタックからポップされ、例外はfoo()に渡される。

しかし、foo()にはtry/catchブロックがない……。

❸ main()が例外処理を回避する。

foo()が即座にスタックからポップされるが、例外を渡す相手はJVM以外には存在しない。しかし、JVMに例外を処理する機能はない。

❹ JVMが停止する。

ClothingExceptionを表現するのにここではTシャツを使いましたが、これは単なる著者の好みです。特に理由はありません。

This PDF is prepared for tools@maruhouse.org, personal use only.

例外についてのまとめ

ここまで見てきたとおり、例外を発生する可能性のあるメソッドを呼び出す際には、コンパイルエラーにならないよう、try/catchブロックを使用して例外処理のコードを書くか、throwsキーワードを使用して例外処理を回避する必要があります。

① 例外処理のコードを書く

try/catchブロックを使ってメソッドを呼び出し、例外処理のコードも書く

```
try {
  laundry.doLaundry();
} catch(ClothingException cex) {
  // 例外処理のコード
}
```

この引数の型は、doLaundry()メソッドの発生するあらゆる例外に対応できるよう指定する。あまり下位のクラスを指定するとコンパイルできない恐れがある

② 例外処理を回避する

呼び出し側のメソッドでthrowsキーワードを使用すれば、例外処理を回避できる。

```
void foo() throws ClothingException {
  laundry.doLaundry();
}
```

doLaundry()メソッドはClothingExceptionを発生させるが、呼び出し側のメソッドfoo()でこのように宣言すると例外処理が回避できる。try/catchブロックが必要なくなる

このように書くと、foo()を呼び出すメソッドに例外処理のコード（あるいは例外処理回避のコード）が必要になる。以下のように、foo()で例外処理を回避しているにもかかわらず、それを呼び出すmain()で例外処理も回避も行わなければ、コンパイルができなくなる。

```
public class Washer {
  Laundry laundry = new Laundry();

  public void foo() throws ClothingException {
    laundry.doLaundry();
  }

  public static void main(String [] args) {
    Washer a = new Washer();
    a.foo();
  }
}
```

誤り!!
main()をこのように書くと、コンパイルができない。例外発生の可能性を認識していないとみなされるため。foo()で例外処理が回避されると、コンパイラにとってfoo()は例外を発生させるメソッドということになる

foo()メソッドで、doLaundry()の発生するClothingExceptionの処理を回避しているので、main()では、foo()をtry/catchブロックで呼び出すか、宣言に「throws ClothingException」というコードを入れるかしなければならない

This PDF is prepared for tools@maruhouse.org, personal use only.

音楽演奏プログラム

例外の説明も一段落したので、そろそろJavaSound APIを使った音楽演奏プログラムの話（すっかり忘れている人も多いでしょうが）に戻りましょう。Sequencerオブジェクトを作成するコードを書くと、コンパイルがエラーになるというところまで話したはずです。エラーになるのは、Midi.getSequencer()メソッドがチェック例外（MidiUnavailableException）を発生させる恐れがあるのに、それに対応するコードを書いていないためです。以下に示すようにtry/catchブロックを使用して修正すれば、エラーは起きなくなります。

```
public void play() {
  try {
    Sequencer sequencer = MidiSystem.getSequencer();
    System.out.println("Successfully got a sequencer");
  } catch(MidiUnavailableException ex) {
    System.out.println("Bummer");
  }
} // playメソッドの終了
```

getSequencer()をtry/catchブロックの中で呼び出している。これでコンパイルができる

catchの引数は、発生する例外に対応するものでなくてはならない。この場合、catch(FileNotFoundException f)などと書くとコンパイルができなくなる。MidiUnavailableExceptionとFileNotFoundExceptionは継承関係にはないため発生する例外そのもの、あるいはそのスーパークラスを指定しなければならない

例外に関するルール

① **tryブロックなしでcatchブロック、finallyブロックを作ることはできない。**

```
void go() {
  Foo f = new Foo();
  f.foof();
  catch(FooException ex) { }
}
```

誤り！ tryブロックがない

② **tryブロックとcatchブロックの間に余分なコードを入れることはできない。**

```
try {
  x.doStuff();
}
  int y = 43;
} catch(Exception ex) { }
```

誤り！ tryブロックとcatchブロックの間にコードは入れない

③ **tryブロックの後にはcatchブロック、あるいはfinallyブロックを置く。**

```
try {
  x.doStuff();
} finally {
  // cleanup
}
```

catchブロックはないがfinallyブロックがあるので問題ない。tryブロックだけを作るということはできない

④ **tryブロックとfinallyブロックだけを作り、catchブロックを作らなかった場合は、メソッドの宣言に例外処理回避のためのコードを入れる。**

```
void go() throws FooException {
  try {
    x.doStuff();
  } finally { }
}
```

catchブロックがなければ例外処理はできないので、例外処理回避のコードが必要

This PDF is prepared for tools@maruhouse.org, personal use only.

プログラム・キッチン

「絶対にそうしなければならない」というわけではありません。でも、プログラムは自分で作る方がはるかに楽しいのは確かです。

ここから先は、音楽演奏のプログラムをゼロから自分で作る方法について説明します。著者があらかじめ用意したコードを利用してプログラムを作ることもできますので、興味がない人は飛ばして読んでもかまいません。

ただ、飛ばさずに読めば、Java Sound APIについてより深く知ることができるでしょう。

This PDF is prepared for tools@maruhouse.org, personal use only.

音楽演奏プログラムの基本的な作り方

章の冒頭近くですでに述べたとおり、MIDIデータには、「どのように音楽を演奏すべきか」という指示だけが納められており、音楽（音声データなど）そのものは納められていません。スピーカーから実際に音を出すためには、MIDIデータをMIDIデバイスに送る必要があります。このMIDIデバイスがMIDIデータの指示に基づき、実際の演奏をするわけです。MIDIデバイスの中には、いわゆる「楽器」の形態をしたハードウェアもあれば、コンピュータの中で動くソフトウェアもあります。本書ではもっぱら、ソフトウェアのMIDIデバイスを使用します。音楽演奏のためのプログラムには、以下のような4つの要素が必要になります。

4つの要素

① **演奏そのものを行うSequencerオブジェクト**

② **Sequencerオブジェクトによる演奏の対象となるSequenceオブジェクト**

③ **演奏データを保持し、Sequenceオブジェクトの構成要素となるTrackオブジェクト**

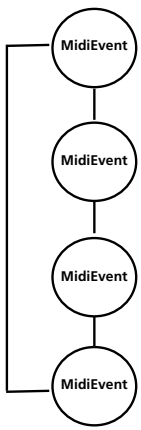
④ **Trackオブジェクトに保持される演奏データ（MidiデータにMidiEventオブジェクト**

Sequencer —演奏→ **Sequence** —構成要素→ **Track** —保持→

Sequencerオブジェクトは、演奏を実行するオブジェクト。ちょうどステレオで言えばCDプレーヤーのような役割をする。

Sequenceオブジェクトは、Sequencerオブジェクトによる演奏の対象となる。ちょうどCDプレーヤーに対するCD（ただし収録される曲は1曲のみ）のようなもの。

Sequenceオブジェクトの構成要素となるTrackオブジェクトだが、本書で使用するSequenceオブジェクトはすべて1つのTrackオブジェクトのみから構成されるもの。これは、CDに1曲しか入っていないようなものと考えればわかりやすい。演奏のデータ（MIDIデータ）はすべてこのTrackオブジェクトが保持する。



MidiEventはMIDIデータを表すオブジェクト。MIDIデータには、「『ド』の音を強く鳴らし、半拍伸ばす」といった指示が納められる。
また、「楽器をピアノからフルートに変更する」といった指示もMIDIデータに納められる。

本書では、Sequenceオブジェクトは、1曲だけを収録したCDのようなものになる（1つだけのTrackオブジェクトから構成される）。「音楽をどのように演奏すればよいか」という情報はTrackオブジェクトに保持される。TrackオブジェクトはSequenceオブジェクトの構成要素

This PDF is prepared for tools@maruhouse.org, personal use only.

音楽演奏プログラムでは以下の 5 つの処理を行う必要がある

① **Sequencerオブジェクトの取得、オープン**

```
Sequencer player = MidiSystem.getSequencer();
player.open();
```

② **Sequenceオブジェクトの作成**

```
Sequence seq = new Sequence(timing, 4);
```

③ **Sequenceオブジェクトを構成するTrackオブジェクトの取得**

```
Track t = seq.createTrack();
```

④ **TrackオブジェクトへのMidiEventオブジェクトの追加、Sequencerオブジェクトによる演奏の対象となるSequenceオブジェクトの設定**

```
t.add(myMidiEvent1);
player.setSequence(seq);
```

This PDF is prepared for tools@maruhouse.org, personal use only.

簡単な音楽演奏プログラム

まずは下のプログラムを入力し、実行してみてください。単音ですが、ピアノの音が聞こえてくるはずです。

```
import javax.sound.midi.*;

public class MiniMiniMusicApp {

    public static void main(String [] args) {
      MiniMiniMusicApp mini = new MiniMiniMusicApp();
      mini.play();
    } // mainの終了

    public void play() {
      try {
①     Sequencer player = MidiSystem.getSequencer();
      player.open();

②     Sequence seq = new Sequence(Sequence.PPQ, 4);

③     Track track = seq.createTrack();

④     ShortMessage a = new ShortMessage();
      a.setMessage(144, 1, 44, 100);
      MidiEvent noteOn = new MidiEvent(a, 1);
      track.add(noteOn);

      ShortMessage b = new ShortMessage();
      b.setMessage(128, 1, 44, 100);
      MidiEvent noteOff = new MidiEvent(b, 16);
      track.add(noteOff);

      player.setSequence(seq);

      player.start();

      Thread.sleep(60*20) //20秒間スリープ（この間に音が出る）
      player.close() //sequencerオブジェクトのクローズ

    } catch(Exception ex) {
      ex.printStackTrace();
    }
  } // playメソッドの終了
} // クラスの終了
```

midiパッケージのインポートを忘れずに

Sequencerオブジェクトの取得、オープン（これでSequencerオブジェクトが使えるようになる。オープンという手続きを経なければSequencerオブジェクトは利用できない）

Sequenceのコンストラクタの引数については今のところ何も考えなくてよい。ただ、このまま入力する

Trackオブジェクトを取得する。Trackオブジェクトは MIDIデータを保持する

TrackオブジェクトにMidiEventオブジェクトをいくつか追加する。この部分のコードに関してはあまり意味がわからなくても、このまま入力すればよい。現時点で唯一知っておくべきなのは、setMessage()メソッドの引数、MidiEventコンストラクタの引数の意味。それについては次ページで触れる

Sequencerオブジェクトによる演奏の対象となるSequenceオブジェクトの設定（CDプレーヤーにCDを入れるのに似た処理）

演奏のスタート（プレーヤーの再生ボタンを押すのに似ている）

This PDF is prepared for tools@maruhouse.org, personal use only.

MidiEventオブジェクト（MIDIデータ）の作成

MidiEventオブジェクトはそれぞれ、Midiデバイスへの指示に対応します。「どのタイミングでどのくらいの長さ、強さ、高さの音を鳴らすのか」といった指示が、このオブジェクトに納められるのです。MidiEventオブジェクトに納められた指示に順に従って音を出していけば、音楽ができていきます。音楽を演奏するのですから、特に、タイミングや音の長さなどは重要と言えるでしょう。MidiEventオブジェクトに納める指示はプログラマが設定することができるのですが、設定の仕方は少し複雑です。たとえば、音の長さを決めるには、音の出し始めのタイミングと音を切るタイミングを指定する、という方法を使います。音の出し始めにはNOTE ONイベント、音を切る時にはNOTE OFFイベントを発生させるのですが、NOTE OFFイベントをNOTE ONイベントより先に発生させると誤りになってしまいます。

音の高さ、強さ、使用する楽器の種類などの設定は、ShortMessageオブジェクトを使用して行います。MidiEventオブジェクトには、ShortMessageオブジェクトの情報に加え、その情報をどのタイミングで使用するか（いつ音を鳴らすか、あるいは止めるか）という情報が格納されます。ShortMessageオブジェクトには、NOTE ONイベントを発生するためのものもあれば、NOTE OFFイベントを発生するためのものもあり、どちらになるかはsetMessage()メソッドの引数によって決まります。前者のShortMessageオブジェクトを格納したMidiEventオブジェクトを使用すると音が出て、後者のShortMessageオブジェクトを格納したMidiEventオブジェクトを使用すると音が止まることになります。

MidiEventオブジェクトは常にShortMessageオブジェクトと組み合わせて使用されるという点に注意してください。

簡単に言うと、ShortMessageオブジェクトには「何を行うか」という情報が、MidiEventオブジェクトには「それをいつ行うか」という情報が格納されるのです。

MidiEventには何をどのタイミングで行うかという情報が保持される

音楽の演奏においてはタイミングの指示が重要な意味を持つ

つまり、音をいつ出し始め、いつ止めるかが重要ということ

① **ShortMessageオブジェクトを作成する。**

```
ShortMessage a = new ShortMessage();
```

② **ShortMessageオブジェクトの情報を指定する。**

```
a.setMessage(144, 1, 44, 100);
```

このメソッドの引数に音程、音色、音の強さなどを指定する（詳しくは次のページを参照）

③ **ShortMessageオブジェクトを使用してMidiEventオブジェクトを作成する。**

```
MidiEvent noteOn = new MidiEvent(a, 1);
```

MidiEventオブジェクトには、ShortMessageオブジェクトに格納された指示をどのタイミングで実行するか、という情報を格納する。「1」はタイミングを表す数値

④ **MidiEventオブジェクトをTrackオブジェクトに追加する。**

```
track.add(noteOn);
```

MidiEventオブジェクトはすべてTrackオブジェクトに格納される。Sequenceオブジェクトは、MidiEventオブジェクトを使用する順序に従って並べ、最終的にはSequencerオブジェクトがその順序でMidiEventオブジェクトを使用し、指示に従って演奏を行うことになる。複数の音を同時に鳴らしたり、複数の楽器を同時に演奏したり、といったことも可能

This PDF is prepared for tools@maruhouse.org, personal use only.

ShortMessageオブジェクトの内容

ShortMessageオブジェクトには、すでに述べたとおり、Sequencerオブジェクトに対する、「何をすべきか」という指示が格納されます。指示の指定にはsetMessage()メソッドを使用します。このメソッドの最初の引数に指定するのは、発生するイベントのタイプです。残り3つの引数の意味は、最初の引数の値によって変わってきます。たとえば、最初の引数の値が「144」であれば、発生されるイベントは「NOTE ON」になります。この場合、残り3つの引数では、鳴らす楽器（あるいは音色）の種類、音程、音の強さを指定することになります。強い「ド」をピアノの音で鳴らす、といったことが指定できるわけです。

注意すべきなのは、ShortMessageオブジェクトの情報だけでは演奏ができないということです。これに加えて、指示されたことを「いつ行うか」という情報も必要です。

setMessage()メソッドの引数

setMessage()メソッドの最初の引数に指定するのは、発生するイベントのタイプ。残り3つの引数の意味は、最初の引数の値によって変わる。

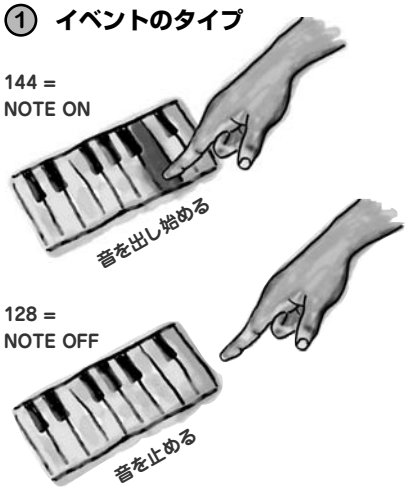
イベントのタイプ　チャネル　音程　音の強さ

```
a.setMessage(144, 1, 44, 100);
```

最初の引数の値によって残り3つの引数の意味は変わる。ここでは、最初の引数の値が「NOTE ON」を意味するものになっているので、残りの引数では、演奏する音についての指示を行う

ShortMessageオブジェクトには「何を行うか」という情報が、MidiEventオブジェクトには「それをいつ行うか」という情報が格納される

① イベントのタイプ

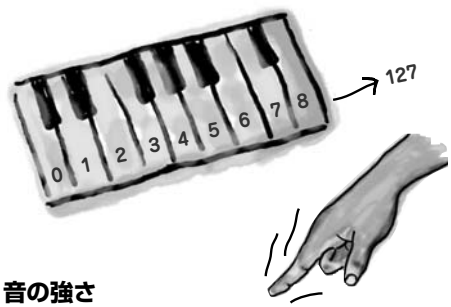


② チャネル

チャネルというのは、ここでは楽器（音色）の種類のこと。チャネル1はキーボード、チャネル9はドラムという具合に対応付けがなされる。

③ 音程

0～127の範囲の数字が各音程に割り当てられる。数字が大きいほど音程が高い。



④ 音の強さ

0が最も弱く（ほとんど何も聞こえない）、数字が大きいほど強くなる。通常は100程度がちょうどよい。

This PDF is prepared for tools@maruhouse.org, personal use only.

setMessage() メソッド

setMessage() メソッドの使い方をもう少し詳しく見てみましょう。setMessage() メソッドを使うことで音程の設定、音の長さの設定、音の追加、楽器（音色）の変更などが行えます。

① 音程の設定

0 ～ 127 の範囲の値を指定する。値をさまざまに変えて音程がどのように変わるか確かめるとよい。

```
a.setMessage(144, 1, 20, 100);
```

② 音の長さの設定

音の長さの設定は、setMessage() メソッドと MidiEvent のコンストラクタを組み合わせて使う。NOTE OFF イベントの発行のタイミングを様々に変えれば、音の長さは変わる。

```
b.setMessage(128, 1, 44, 100);
MidiEvent noteOff = new MidiEvent(b, 3);
```

③ チャネルに対応する楽器（音色）の変更

setMessage() メソッドでは、チャネルに対応する楽器（音色）の変更もできる。その後の setMessage() メソッドには変更が反映される。以下に示した例では、チャネル 1 の楽器をデフォルトの「ピアノ」から他のものに変更している。楽器の変更の際は最初の引数の値を「192」にする。変更後の楽器を何にするかの設定に使うのは第 3 引数（0 ～ 127 の範囲の値を設定してどのように楽器が変わるか確かめるとよい）。

```
first.setMessage(192, 1, 102, 0);
```

楽器の変更に対応する値
変更対象となるチャネルの番号
どの楽器に変更するかをここで指定

This PDF is prepared for tools@maruhouse.org, personal use only.

楽器と音程を変更できるプログラム

次に示したのは、実行時に使用する楽器と音程を数字で指定できる音楽演奏プログラムのコードです。0 ～ 127 の範囲の様々な数字を指定して、どのように音色と音程が変わっていくかを試してみてください。最初に指定する数字が楽器（音色）、次に指定する数字が音程に対応します。

```
import javax.sound.midi.*;

public class MiniMusicCmdLine { // 音楽演奏プログラム

  public static void main(String [] args) {
    MiniMiniMusicCmdLine mini = new MiniMiniMusicCmdLine();
    if (args.length < 2) {
      System.out.println("Don't forget the instrument and note args");
    } else {
      int instrument = Integer.parseInt(args[0]);
      int note = Integer.parseInt(args[1]);
      mini.play(instrument, note);
    }
  } // main の終了

  public void play(int instrument, int note) {

    try {

      Sequencer player = MidiSystem.getSequencer();
      player.open();
      Sequence seq = new Sequence(Sequence.PPQ, 4);
      Track track = seq.createTrack();

      ShortMessage first = new ShortMessage();
      first.setMessage(192, 1, instrument, 0);
      MidiEvent changeInstrument = new MidiEvent(first, 1);
      track.add(changeInstrument);

      ShortMessage a = new ShortMessage();
      a.setMessage(144, 1, note, 100);
      MidiEvent noteOn = new MidiEvent(a, 1);
      track.add(noteOn);

      ShortMessage b = new ShortMessage();
      b.setMessage(128, 1, note, 100);
      MidiEvent noteOff = new MidiEvent(b, 16);
      track.add(noteOff);
      player.setSequence(seq);
      player.start();

    } catch(Exception ex) {ex.printStackTrace();}
  } // play の終了
} // クラスの終了
```

実行時に以下のように 0 ～ 127 の範囲の数字を 2 つ指定する。

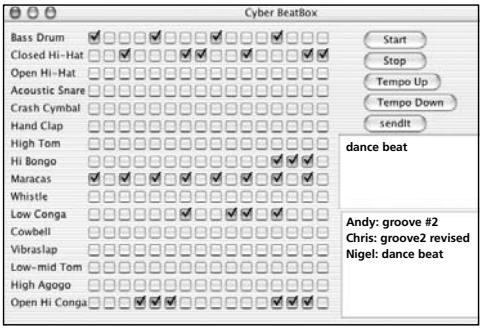
```
File  Edit  Window  Help  Attenuate
% java MiniMiniMusicCmdLine 102 30
% java MiniMiniMusicCmdLine 80 20
% java MiniMiniMusicCmdLine 40 70
```


This PDF is prepared for tools@maruhouse.org, personal use only.

12 章以降で作成するプログラムの概要

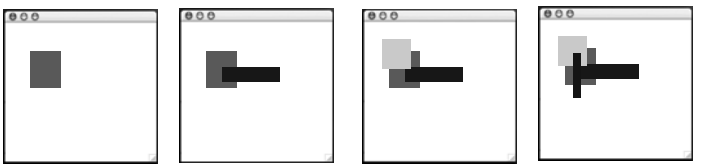
15 章：最終目標

この章では、先述のBeatBoxプログラムを完成させます。このプログラムはすでに述べたとおり、「ドラムチャット」のクライアントとしても機能します。BeatBoxプログラムの完成のためには、GUI(イベント処理を含む)、I/O、ネットワーキング、スレッドといったことがらについて学ぶ必要があります。12 ～ 15 章はそのために存在していると言ってもいいでしょう。



12 章：ミュージッククリップ

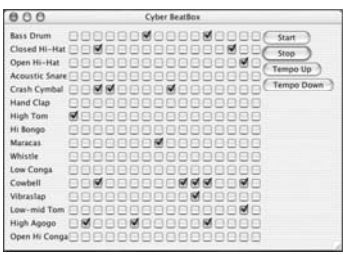
この章では、音楽に合わせてさまざまな長方形を画面に表示させる、「ミュージックビデオプログラム(というのはちょっと大げさかもしれませんが……)」を作成します。そのために、MidiEventオブジェクトについてさらに詳しく学ぶことになります。



1　2　3　4……

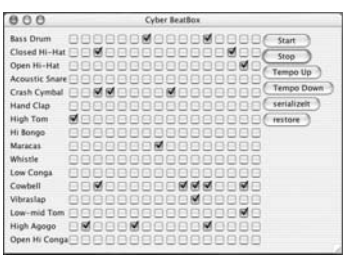
13 章：スタンドアロンBeatBox

この章では、GUIも備えたBeatBoxプログラムを作るのですが、このBeatBoxプログラムは機能の限定されたものです。まず、作成したドラムパターンは保存できません。また、ネットワークによる通信機能も持っていません。単にドラムパターンを作って演奏できるというだけです。



14 章：ドラムパターンの保存

この章では、作成したドラムパターンをファイルに保存できるようBeatBoxプログラムを改良します。保存したパターンはもちろん、プログラムに再度読み込むことができるようにします。ファイルに保存するためのテクニックは、15 章でネットワーク経由でのデータ送信機能を追加する際に役立ちます。



This PDF is prepared for tools@maruhouse.org, personal use only.

エクササイズ

○か×か

次の文は、いずれもこの章で学んだことがらについて述べたものです。正しいものに○を、誤っているものに×をつけてください。

1. tryブロックの後にはcatchブロックとfinallyブロックの両方が必要。
2. チェック例外を発生させる可能性のあるメソッドを呼び出す際は、必ずtry/catchブロックを使用しなければならない。
3. catchブロックでは、引数に名前の指定された例外以外に、その例外のサブクラスにも対応できる。
4. catchブロックで対応できるのはチェック例外のみ。
5. try/catchブロックにはfinallyブロックを作ることもできるが、必須ではない。
6. tryブロックは、catchブロック、finallyブロックのうちのいずれか、あるいはその両方と組み合わせることができる。
7. 例外を発生するメソッドを作る際には、宣言で「throws」キーワードを使い、発生する例外の名前を指定するとともに、例外発生のためのコードをtry/catchブロックを使って記述する必要がある。
8. 例外処理は一応、回避できるが、最終的にmain()メソッドでは必ず例外処理を行わなくてはならない。
9. 1つのtryブロックに複数のcatchブロックを対応させることもできる。
10. 1つのメソッドが複数種の例外を発生させるよう指定することはできない。
11. finallyブロックのコードは、例外が発生する、しないを問わず実行される。
12. 対応するtryブロックなしで、finallyブロックだけを作ることもできる。
13. tryブロックは、catchブロック、finallyブロックなしで単独で存在できる。
14. 例外処理の回避には「throws」キーワードを使用する。
15. catchブロックを複数作る時の並び順には特にルールはない。
16. tryブロックとfinallyブロックを作れば、例外処理回避のためのコードは必要ない。
17. 非チェック例外にも、チェック例外と同様に、try/catchブロック、あるいはthrowsキーワードを使用して対処する必要がある。

This PDF is prepared for tools@maruhouse.org, personal use only.

エクササイズ

プログラミングマグネット

バラバラになったJavaプログラムを並べ替え、下のような実行結果が得られるようにしてください。もし、{ 、} などが抜けているところがあれば付け加えてもかまいません。

```
System.out.print("r");
```

```
try {
```

```
doRisky(test);
```

```
System.out.print("t");
```

```
System.out.println("s");
```

```
} finally {
```

```
System.out.print("o");
```

```
class MyEx extends Exception {
}

public class ExTestDrive {
```

```
System.out.print("w");
```

```
if ("yes".equals(t)) {
```

```
System.out.print("a");
```

```
throw new MyEx();
```

```
} catch (MyEx e) {
```

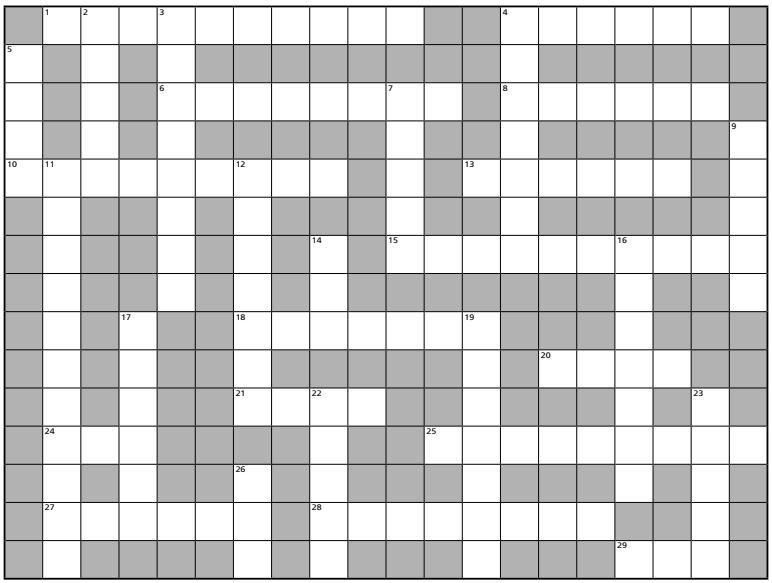
```
static void doRisky(String t) throws MyEx {
  System.out.print("h");
```

```
public static void main(String [] args) {
  String test = args[0];
```

File Edit Window Help ThrowUp

```
% java ExTestDrive yes
thaws

% java ExTestDrive no
throws
```


This PDF is prepared for tools@maruhouse.org, personal use only.



Javaに関する用語のクロスワードです（注：このクロスワードには英語で、すべて大文字で答えてください）。

ヨコのカギ

1. 変数に値を与える
4. スタックから取り出す
6. 継承して作るクラス
8. 実行開始
10. クラスの継承によってできる構造
13. tryとcatch
15. 問題発生を知らせるオブジェクト
18. 特別な意味を持つ言葉
20. クラス階層
21. 例外を処理しない
24. 整数
25. 処理の手法
27. 例外の発生
28. 抽象的でない
29. コンストラクタの起動

タテのカギ

2. 有効範囲
3. オブジェクト
4. 外から利用できない
5. 数字に関係するクラス
7. ステート
9. オブジェクトの設計図
11. オブジェクトを作る
12. 正しく扱わなければコンパイルエラーに
14. catchと……
16. 受け継ぐ
17. 値の設定に使う
19. メソッドの……
22. 例外処理のブロック
23. 例外発生箇所の特定
26. Industry Standard Architecture

ヒント

ヨコのカギ
6. 子供
8. eで終わる
13. 回避しない
20. 上下関係
21. アヒル
27. 投げる
28. 具体的

タテのカギ
2. sc……
3. for～（たとえば）
5. 数学は英語で……
9. 学級
16. tで終わる
17. ゲッターではない

This PDF is prepared for tools@maruhouse.org, personal use only.

エクササイズの答え

○か×か

1. ×　どちらか一方でよい
2. ×　メソッドの宣言で例外の名前を指定するだけでもよい
3. ○
4. ×　非チェック例外にも対応できる
5. ○
6. ○
7. ×　宣言でthrowsキーワードを使って名前を指定するだけでよい
8. ×　ただし、main()で例外処理を回避すると例外発生時にJVMが停止する
9. ○
10. ×
11. ○　finallyブロックは特定の処理の終了に使用されることが多い
12. ×
13. ×
14. ○
15. ×　受け取れる例外の種類が多いものほど下に置く
16. ×　catchブロックを作らない場合は、例外処理回避のためのコードが必要
17. ×

プログラミングマグネット

```
class MyEx extends Exception { }

public class ExTestDrive {

  public static void main(String [] args) {
    String test = args [0];
    try {

      System.out.print("t");

      doRisky(test);

      System.out.print("o");

    } catch(MyEx e) {

      System.out.print("a");

    } finally {

      System.out.print("w");

    }
    System.out.println("s");
   }

    static void doRisky(String t) throws MyEx {
      System.out.print("h");

      if ("yes".equals(t)) {

        throw new MyEx();
    }

    System.out.print("r");

  }
}
```

```
File  Edit  Window  Help  Chill
% java ExTestDrive yes
thaws

% java ExTestDrive no
throws
```


This PDF is prepared for tools@maruhouse.org, personal use only.

Java クロスワード

	1 A	2 S	S	3 I	G	N	M	E	N	T			4 P	O	P	P	E	D	
5 M		C		N									R						
A		O		6 S	U	B	C	L	A	7 S	S		8 I	N	V	O	K	E	
T		P		T						T			V						9 C
10 H	11 I	E	R	A	R	12 C	H	Y		A		13 H	A	N	D	L	E		L
	N			N		H				T			T						A
	S			C		E		14 T		15 E	X	C	E	P	T	16 I	O	N	S
	T			E		C		R								N			S
	A		17 S			18 K	E	Y	W	O	R	19 D				H			
	N		E			E						E		20 T	R	E	E		
	T		T			21 D	U	22 C	K			C				R		23 T	
	24 I	N	T					A			25 A	L	G	O	R	I	T	H	M
	A		E			26 I		T				A				T		R	
	27 T	H	R	O	W	S		28 C	O	N	C	R	E	T	E			O	
	E					A		H				E				29 N	E	W	

代入 = assignment
呼び出し = invoke
階層構造 = hierarchy
例外処理 = handle
例外 = exceptions
インスタンス化 = instantiate
例外処理の回避 = duck
チェック例外 = checked
継承 = inherit
宣言 = declare

This PDF is prepared for tools@maruhouse.org, personal use only.

12章　GUIの基礎

グラフィカルなインターフェース

GUI(Graphical User Interface)は今や、プログラム(アプリケーション)にとって不可欠なものと言えるでしょう。それが不特定多数の人に向けて作るものであろうと、自分専用に作るものであろうと、GUIは必要です。また、プログラム開発に利用するツールにも、GUIは必要になります。たとえ、作るのがWebブラウザによって利用されるサーバサイドのプログラムであったとしてもその点に変わりはありません。世の中には、古いことが価値になるものも存在しますが、コマンドラインにそういう意味での価値はないでしょう。不便で使いづらく、融通が利かないなど、デメリットの方が多いからです。12、13章では、このGUIについて、そしてイベント処理、内部クラス、といったJava言語にとって重要な概念について解説していきます。12章ではまず、画面上にボタンを作り、クリックに反応して何らかの処理を行うようにする方法を学ぶことになります。その他、画面上に絵を描く方法、JPEG画像を表示する方法、アニメーションを作る方法なども学んでいきます。

This PDF is prepared for tools@maruhouse.org, personal use only.

まずウィンドウを作る

画面上にウィンドウを作るには、ウィンドウに対応するオブジェクトであるJFrameを使用します。ボタン、チェックボックス、テキストフィールドといった「インターフェース部品」は、JFrameでウィンドウに配置していくことになります。ウィンドウには、当然のことながらメニューバー（とメニュー項目）を持たせることができます。その他、ウィンドウを最大化、最小化するためのボタンやウィンドウを閉じるためのボタンは、はじめから付いています。

JFrameのウィンドウの外観は、使用するプラットフォームによって異なってきます。たとえば、Mac OS Xであればの外観は以下のようになります。

これ以上
コマンドライン
アプリケーションを
作るなら、クビだ！

メニューバーとボタンと
ラジオボタンを備えた
JFrameのウィンドウ

ウィンドウにインターフェース部品を追加する

JFrameのウィンドウにインターフェース部品（「ウィジェット」と呼ばれることがある）を追加するには、Swingコンポーネントと呼ばれるオブジェクトを使用します。JFrame自身もSwingコンポーネントの一種です。Swingコンポーネントには、多数の種類があり、javax.swingというパッケージに納められています。Swingコンポーネントの中でも特によく使用されるのは、JButton、JRadioButton、JCheckBox、JLabel、JList、JScrollPane、JSlider、JTextArea、JTextField、JTableなどです。コンポーネントの多くは簡単に使用できるものですが、中には（JTableなどのように）多少わかりにくいものもあります。

GUIの作成手順

① ウィンドウ（フレーム）を作成する

```
JFrame frame = new JFrame();
```

② ウィジェット（ボタン、テキストフィールドなど）を作成する

```
JButton button = new JButton( “click me” );
```

③ ウィジェットをウィンドウに組み込む

```
frame.getContentPane().add(button);
```

ウィジェットはウィンドウ（フレーム）に直に組み込むわけではない。ウィジェットを追加する際は、まずウィンドウから「コンテンツペイン」と呼ばれる種類のオブジェクトを取り出し、そのコンテンツペインにウィジェットを組み込む。フレームはいわば「窓枠」のようなもので、ペインはそれに対して、窓に取り付けられた「ガラス」のようなものであると言える。

④ 作成したGUIのサイズを指定し、画面に表示する

```
frame.setSize(300, 300);
frame.setVisible(true);
```


This PDF is prepared for tools@maruhouse.org, personal use only.

ボタンを備えたウィンドウ（フレーム）を作る

```
import javax.swing.*;

public class SimpleGui1 {
  public static void main (String [] args) {
    JFrame frame = new JFrame();
    JButton button = new JButton("click me");

    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    frame.getContentPane().add(button);

    frame.setSize(300, 300);

    frame.setVisible(true);
  }
}
```

swingパッケージをインポートする

フレームとボタンを作成する

（JButtonオブジェクトのコンストラクタの引数には、ボタン上に表示するテキストを指定できる）

このように書くと、ウィンドウを閉じると同時にプログラムが終了するようになる（これを省略するとプログラムは永遠に終了しない）

ボタンをフレームのコンテンツペインに組み込む

フレームのサイズを指定する（単位：ピクセル）

これでGUIが画面に表示される（このコードがないとプログラムを実行しても画面に何も表示されない）

上記のコードの実行結果は以下のようになります。

```
% java SimpleGui1
```

これではあまりにボタンが大きすぎる……

このプログラムでは、ボタンがフレームいっぱいに表示されてしまいます。ボタンの大きさ、表示位置の設定の仕方については後述します。

This PDF is prepared for tools@maruhouse.org, personal use only.

前ページのプログラムの場合、ボタンをクリックしても何も起きない……

実を言えば、「何も起きない」というのは正確ではありません。ボタンをクリックすると、押されたことが見てわかるようなかたちに変わる（具体的にどのように変わるかはプラットフォームごとに異なるが、ともかく変わることは確か）からです。

とはいえ、ボタンの形が変わるだけではやはり意味がありません。ボタンをクリックした時には、それに対応して、何か意味のある処理が行われるようにしなければならないのです。

そのためには、次の 2 つのものが必要です。

① ユーザーがボタンをクリックした時に自動的に呼び出されるメソッド（このメソッドがボタンクリックに対応する処理を行う）。

② ユーザーがボタンをクリックしたことを検知し、メソッドを呼び出すための手段。

ユーザーがボタンをクリックした時には、それを検知する必要がある

これは、「ユーザーによるボタンのクリック」というイベントを検知するということ

素朴な疑問に答えます

Q：JButtonオブジェクトを使ったボタンの外観は、Windowsでプログラムを実行した場合、Windowsのボタンと同じようなものになるのですか。

A：「Windowsのボタンのようなものにしたい」と思えばそうすることもできます。インターフェースのルック＆フィールを決めるためのクラスがコアライブラリに用意されているのでそれを使用すればよいのです。通常、ルック＆フィールとしては、「Metal」と呼ばれるJavaの標準のものか、プラットフォームのネイティブルック＆フィール（プラットフォームがWindowsであればWindowsのルック＆フィール）かを選ぶことになります。本書では、「Aqua」と呼ばれるMac OS Xのルック＆フィールを使用していますが、Metalを選ぶことも可能です。

Q：どのプラットフォームでもルック＆フィールが「Aqua」になるプログラムを作ることは可能ですか。たとえばWindows上でもMac OS Xのような外観で動くようにしたいのですが。

A：それはできません。ルック＆フィールの中には、プラットフォームによっては利用できない、というものもあるのです。プラットフォームがどのように変わっても問題が起きないようにするためには、ルック＆フィールを明示的に「Metal」に指定するか、まったくルック＆フィールを指定しない（常にデフォルトを使用するようにする）のが無難でしょう。

Q：「Swingは遅いので誰も使わない」と聞いたことがあるのですが……

A：以前は確かにそうでしたが、今ではそんなことはありません。マシンの性能が低ければ、遅くて困ることもあるかもしれませんが、最近のデスクトップマシンを使い、Javaのバージョン 1.3 以降を使えば、Swing GUIとプラットフォームのネイティブGUIとの違いはほとんど感じないはずです。Swingは現在、あらゆる種類のアプリケーションに広く使用されています。

This PDF is prepared for tools@maruhouse.org, personal use only.

イベント処理

たとえば、ユーザーがクリックした時に表示されるテキストが「click me」から「I've been clicked」に変化するようなボタンを作りたい、という場合にはどうすればいいのでしょうか。それにはまず、ボタン上のテキストを変更させるメソッドを作ることになるでしょう。

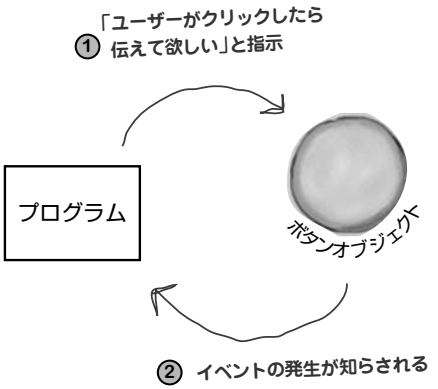
```
public void changeIt() {
  button.setText("I've been clicked!");
}
```

ただ、これだけでは不十分です。このメソッドはボタンがクリックされた時に実行されなければならないのですが、このままではボタンがクリックされたことを知りようがないからです。

「ボタンがクリックされた」など、何かが起こったり、行われたりしたこと（これを「イベント」という）をきっかけに何らかの処理を実行するためには、「イベント処理」と呼ばれる特殊な手続きが必要になります。Javaのイベントには多数の種類がありますが、その多くがGUIに対するユーザーの操作に関係のあるものです。「ユーザーによるボタンクリック」もその1つというわけです。当然ですが、ユーザーがボタンをクリックするということは、そのボタンに対応する処理が行われることを期待していることを意味します。たとえば、「テンポを落とす」と書かれたボタンをクリックしたユーザーはテンポが落ちることを、「送信」ボタンをクリックしたユーザーは、データが送られることを期待しているのです。プログラムの側では、「クリック」というイベントが発生した時に、適切な処理を行うようにすればいいわけです。これはイベントへの対応としては、最も単純な部類に入ります。

ボタンの場合、重要なのは通常、「クリックされた」というイベントだけです。「ボタンの上にマウスポインタが置かれた」、「ユーザーの手がマウスボタンから離れた」といった「細かい」イベントは重要性を持ちません。マウスポインタがボタンの上に何秒置かれていようと、「ユーザーがクリックしかけて途中で気が変わってやめた」というようなことが何度あろうと関係はないのです。ともかく、ユーザーによるクリックというイベントだけがボタンからプログラムに伝えられ、それに応じた処理が行えれば何も問題はありません。

ボタンから伝えられるべきなのは通常、「ユーザーがクリックした」というイベントのみ



ボタンには、イベント発生を知らせる手段が必要

頭の体操

1) 「ユーザーがクリックした」というイベントの発生だけを知らせて欲しいとボタンオブジェクトに指示するにはどうすればいいと思いますか。

2) ボタンがイベント発生を知らせるにはどのような手段があると思いますか。イベントに対応するメソッドの名前（changeIt()など）をボタンに伝えることはできないと仮定してください。どうすれば、イベント発生時に適切なメソッドが確実に実行されるようになるでしょうか。

This PDF is prepared for tools@maruhouse.org, personal use only.

イベント発生を検知し、イベントに対応した処理を行うプログラムを書くには、リスナーインターフェースと呼ばれるインターフェースをインプリメントする

リスナーインターフェースは、プログラムとイベント発生オブジェクト(イベントソース)との間の架け橋の役割を果たす。

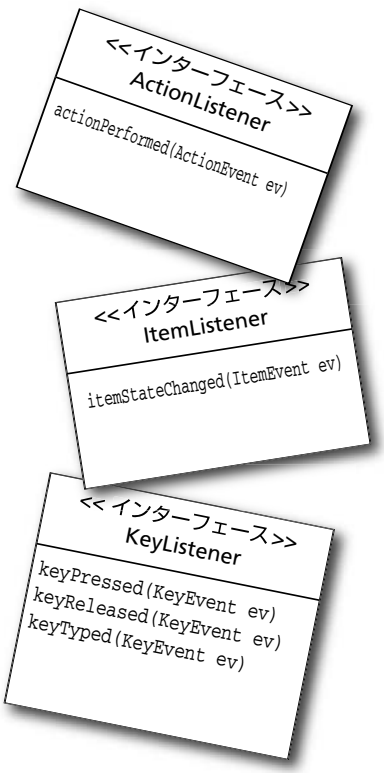
Swingコンポーネントは、イベントソースとしての機能を備えたオブジェクトです。イベントソースとしての機能とは、Javaでは、マウスをクリックする、キー入力をする、ウィンドウを閉じるといったユーザーの操作をイベントにすることができる機能のことを指します。また、Javaでは、イベントもやはり一種の「オブジェクト(イベントクラスのオブジェクト)」となります。APIのjava.awt.eventパッケージを見ると、イベントクラスが数多く用意されているのがわかります(イベントクラスは名前に「Event」が入るのでわかりやすい)。イベントクラスの例としては、MouseEvent、KeyEvent、WindowEvent、ActionEventなどがあげられます。

ボタンなどのイベントソースでは、ユーザーのクリックなどが行われた時にイベントオブジェクトが作成されます。プログラマが実際に書くのは、ほとんどの場合、イベントオブジェクトを作成するコードではなく、イベントオブジェクトを受け取る(つまりイベントを検知する)コードです。イベントオブジェクトを受け取るコードを「イベントリスナー」と呼びますが、本書で取りあげるのも、もっぱらイベントリスナーの方です。言い換えれば、プログラマはイベントソースではなくイベントリスナーを作ることが多い、ということになるでしょう。

イベントオブジェクトにはそれぞれ対応するリスナーインターフェースがあります。たとえば、MouseEventに対応するイベントリスナーを作る際には、MouseListenerインターフェースを、WindowEventに対応するイベントリスナーを作る際には、WindowListenerインターフェースをインプリメントしますMouseEventに対応するイベントリスナーを作る際には、MouseListenerインターフェースをインプリメントします。インターフェースをインプリメントするには、先述のとおり、「Dog implements Pet」という具合に宣言し、インターフェースに含まれるメソッドの「本体」のコードをすべて書くことになります。

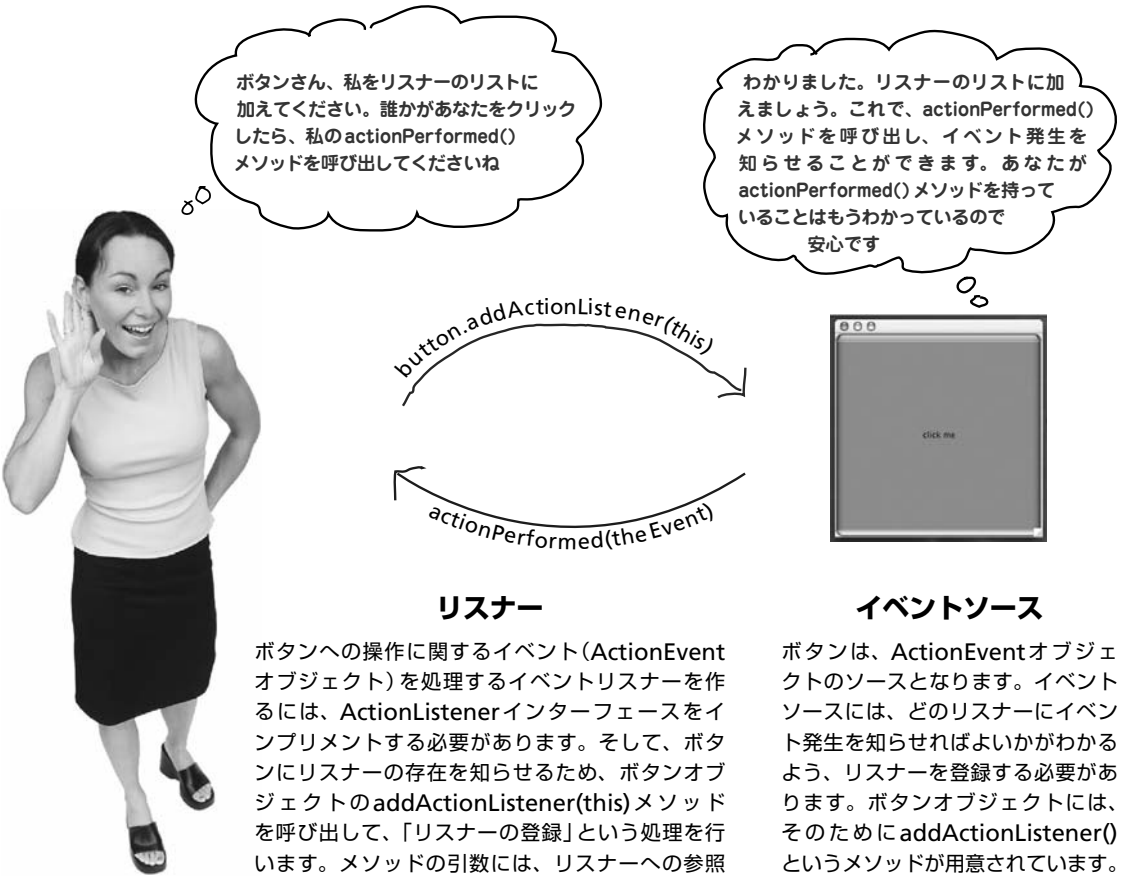
リスナーインターフェースの中には複数のメソッドを持っているものもありますが、これは、1つのイベントオブジェクトが複数のイベントに対応することがあるからです。MouseListenerをインプリメントする場合は、mousePressed、mouseReleased、といったメソッド(いずれもMouseEventオブジェクトを引数とするメソッド)をインプリメントしなくてはなりません。mousePressedはユーザーがマウスボタンを押した時のイベント、mouseReleasedはマウスボタンから手を離した時のイベントです。

リスナーインターフェースをインプリメントしてイベントリスナーを作れば、イベントソースからイベントオブジェクトを受け取ることができる



This PDF is prepared for tools@maruhouse.org, personal use only.

イベントリスナーとイベントソースのコミュニケーション



リスナー

ボタンへの操作に関するイベント（ActionEventオブジェクト）を処理するイベントリスナーを作るには、ActionListenerインターフェースをインプリメントする必要があります。そして、ボタンにリスナーの存在を知らせるため、ボタンオブジェクトのaddActionListener(this)メソッドを呼び出して、「リスナーの登録」という処理を行います。メソッドの引数には、リスナーへの参照（ここでは、リスナー自身のコードの中でメソッドを呼び出しているので、引数はthisになっている）を指定します。ボタンオブジェクトは、イベント発生時にリスナーのメソッドを呼び出すことになります。ActionListenerインターフェースをインプリメントしたリスナーの場合、呼び出されるメソッドはactionPerformed()だけです。actionPerformed()がリスナーでインプリメントされていないと、コンパイルでエラーになります。

イベントソース

ボタンは、ActionEventオブジェクトのソースとなります。イベントソースには、どのリスナーにイベント発生を知らせればよいかがわかるよう、リスナーを登録する必要があります。ボタンオブジェクトには、そのためにaddActionListener()というメソッドが用意されています。addActionListener()メソッドの引数には、リスナーオブジェクトへの参照が指定され、その参照がリストに登録されるのです。ユーザーがボタンをクリックすると、ボタンはリストに登録された各リスナーのactionPerformed()メソッドを呼び出し、イベント発生を知らせます。

This PDF is prepared for tools@maruhouse.org, personal use only.

ActionEventの扱い

① ActionListenerインターフェースをインプリメントする。

② リスナーをボタンに登録する（これで「イベントを受け取りたい」という意志がイベントソースに伝わる）。

③ イベント処理のためのメソッド（ActionListenerインターフェースを使う場合はactionPerformed()メソッド）をインプリメントする。

```
import javax.swing.*;
import java.awt.event.*;

public class SimpleGui1B implements ActionListener {
  JButton button;

  public static void main(String [] args) {
    SimpleGui1B gui = new SimpleGui1B();
    gui.go();
  }

  public void go() {
    JFrame frame = new JFrame();
    button = new JButton("click me");

    button.addActionListener(this);

    frame.getContentPane().add(button);
    frame.setDefaultCloseOperation(Jframe.EXIT_ON_CLOSE);
    frame.setSize(300, 300);
    frame.setVisible(true);
  }

  public void actionPerformed(ActionEvent event) {
    button.setText("I've been clicked!");
  }
}
```

ActionListener、ActionEventを含むパッケージのインポート

①

インターフェースをインプリメントする。これによって、SimpleGui1BはActionListenerのインプリメンテーションになる（ボタンで発生するイベントはActionListenerのインプリメンテーションにのみ渡される）

②

イベントリスナーを登録する。これによって、ボタンのリスナーリストにリスナーが追加される。引数には、ActionListenerインターフェースをインプリメントするオブジェクトへの参照を指定する

③

ActionListenerインターフェースのactionPerformed()メソッドをインプリメントする。実際のイベント処理はこのメソッドで行われる

ボタンは、イベント発生時にこのメソッドを呼び出し、それを知らせる。引数としてActionEventオブジェクトが渡される。ただし、このオブジェクトは、受け取ってもここでは使用しない。このプログラムは、イベント発生が検知できれば機能するため

This PDF is prepared for tools@maruhouse.org, personal use only.

リスナー、ソース、イベント

Javaプログラミングにおいてイベントソースを作るということはほとんどありません。通常、作るのはイベントリスナーです。（どれほど優秀なプログラマであってもイベントソースを作るということはめったにありません）

つまり、リスナーさえ正しく作ることができればまず問題は生じないということです。（それだけでプログラミングがかなりスムーズにできるようになります）

私はイベントリスナーだから、リスナーインターフェースをインプリメントするわ。ボタンに自分を登録して、イベントの処理もするのよ

リスナーは
イベントオブジェクトを
受け取る

僕はイベントソースだから、リスナーを登録してもらって、ユーザーの操作（クリック）によってイベントが発生したら、リスナーのイベント処理メソッドを呼び出すよ

ソースは
イベントオブジェクトを
送る

僕がいることも忘れないで。僕はイベントオブジェクト。リスナーのイベント処理メソッドが呼び出される時に引数に指定されるんだ。僕には、リスナーにイベントに関する情報を伝えるという役目があるよ

イベントオブジェクトは
イベントに関する情報を
保持する

This PDF is prepared for tools@maruhouse.org, personal use only.

素朴な疑問に答えます

Q：イベントソースを自分で作ってはいけないのですか。

A：作ってはいけないというわけではありません。ただ、ソースを作るよりリスナーを作ることの方がはるかに多い、というだけのことです（特に初心者のうちはそうです）。また、作成するイベントリスナーは、多くの場合、Java APIのクラスによって作られるイベントを受け取るものになります。もちろん、独自のイベントソースを作ることが絶対にないということはありません。たとえば、株式市場を監視し、何か重要なことがおきたら通知するというプログラムを作るとします。この場合は、通知のために独自のイベントを作成できるようにするということも必要でしょう。つまり、イベントソース、イベントクラスを自ら作るということになるわけです。このような独自のイベントであっても、扱い方はボタンで発生するイベントと基本的に同じです。やはりリスナーインターフェースをインプリメントしてイベントリスナーを作り、その中の特定のメソッド（このメソッドも自分で作る）を呼び出すことでイベント発生が通知されるようにするのです。その他、イベントソースにはリスナーの登録のためのメソッドも用意しなくてはなりません。イベントソースが独自のものである場合には、それに対応するリスナーインターフェースも自分で用意しなくてはならない、という点にも注意してください。

Q：リスナーのメソッドを呼び出す際になぜ引数としてイベントオブジェクトを渡す必要があるのか、私にはわかりません。メソッドが呼び出されたことでイベントが発生したことはわかるのですから、それで十分ではないですか。

A：確かにイベントオブジェクトが本当に必要になることというのはさほど多くはありません。イベントオブジェクトは、イベントに関する若干の情報を運ぶものに過ぎないからです。ただ、時によってはこの情報が重要な役割を果たすことがあるのです。たとえば、「マウスボタンが押された」というイベントが発生した時には、MouseListenerのmousePressed()というメソッドが呼び出されます。これでイベントが発生したという事実は通知されるのですが、もし、「マウスボタンは、マウスポインタがどこに置かれている時にクリックされたのか」ということを知る必要があったとしたらどうでしょうか。この場合は、画面上でのマウスポインタのx、y座標という情報が伝達されなくてはならないのです。
また、1つのリスナーを複数のイベントソースに同時に登録しなくてはならない、という場合はどうでしょうか。現に、電卓のプログラムなどではそういうことが必要になります。電卓プログラムでは、数字キーを10個作るわけですが、どれも基本的な機能は同じなので、キーごとにリスナーを作るのはあまりに非効率です。そこで1つのリスナーを、10個のキーオブジェクト（それぞれがイベントソース）に登録することになります。どのキーが押されても同じリスナーの同じメソッドが呼び出されるのですが、「押された（イベントを発生させた）キーがどれなのか」ということはわからないと困ります。そこで、その情報をイベントオブジェクトが保持するのです。

自分で考えてみよう

左に列挙したウィジェット（ユーザーインターフェースオブジェクト）は、いずれもイベントソースとなるものです。それぞれのウィジェットでイベントが発生した際に呼び出される可能性のあるメソッドを右から選んでください。1つのウィジェットが複数種のイベントを発生させることもあるので注意が必要です。また、複数のウィジェットによって呼び出されるメソッドもあります。

ウィジェット	メソッド
チェックボックス	windowClosing()
テキストフィールド	actionPerformed()
スクローリングリスト	itemStateChanged()
ボタン	mousePressed()
ダイアログボックス	keyTyped()
ラジオボタン	mouseExited()
メニュー項目	focusGained()

オブジェクトがイベントソースかどうかを見分ける方法は？

APIのドキュメントでメソッドに関する記述を見て、名前が「add」で始まり「Listener」で終わるメソッド、引数の型がリスナーインターフェースになっているメソッドがあるかどうかを確かめます。たとえば、addKeyListener(KeyListener k)というようなメソッドがあれば、そのクラスはKeyEventというイベントのソースと考えられます。このように、基本的には「名前のパターン」を手掛かりにイベントソースを探すことができるのです。

This PDF is prepared for tools@maruhouse.org, personal use only.

再びGUIの話……

イベント処理についてはひととおり理解できたと思いますので（後でさらに詳しく説明します）、再びGUIの話に戻りましょう。まずはGUI部品に何かを組み込んだり、GUI部品上に何かを描いたりという操作から見ていくことにします。

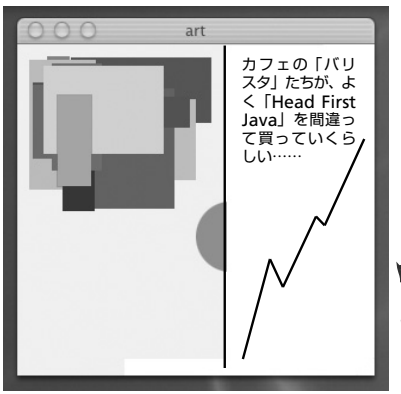
Javaでは、GUI部品に関して以下のような操作が可能です。

① **フレーム（ウィンドウ）にウィジェットを組み込む**

ボタン、メニュー、ラジオボタンなどのウィジェットをフレームに組み込むには以下のようなコードを書きます。

```
frame.getContentPane().add(myButton);
```

javax.swingパッケージには、多数のウィジェットクラスが用意されています。



② **ウィジェット上に2Dグラフィックスを描く**

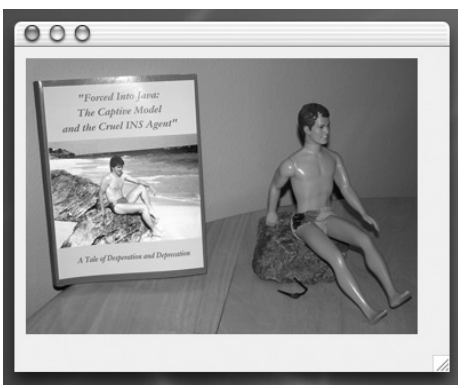
graphicsオブジェクトのメソッドを使用すると2次元の図形を描くことができます。

```
graphics.fillOval(70, 70, 100, 100);
```

Java2D APIを利用すれば、単純な長方形や円だけでなく、もっと複雑な図形を描くことが可能です。

ゲーム、シミュレーションなどの他、お絵かきソフトにも使える

ビジネス文書などに使うグラフも作成できる



③ **ウィジェットにJPEG画像を組み込む**

ウィジェットには自分で作った画像を組み込むこともできます。

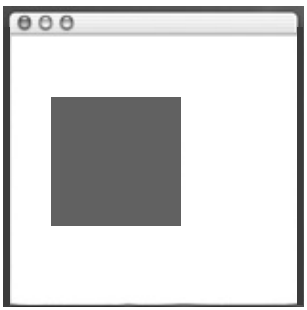
```
graphics.drawImage(myPic, 10, 10, this);
```


This PDF is prepared for tools@maruhouse.org, personal use only.

描画パネルをフレームに組み込む

フレームには、ボタンやチェックボックス以外に「描画パネル」と呼ばれるウィジェットを組み込むこともできます。この描画パネルには、プログラマが自ら図形を描くことができます。この図形はアニメーションにして動かすことなども可能です。また、「ボタンのクリックによって色が変化する」といった機能を持たせることもできます。

描画パネルの作り方は非常に簡単です。



それにはまずJPanelというクラスを継承し、paintComponent()というメソッドをオーバーライドすることになります。

paintComponent()メソッドはグラフィックスを表示する際には必ず使用されるメソッドです。システムがウィジェットなどのグラフィックスを画面に表示する際に呼び出すメソッド、と考えればほぼ間違いないでしょう。たとえば円を描画パネルに表示するには、paintComponent()メソッドに円を描画するためのコードが必要になります。描画パネルの組み込まれたフレームが画面に表示される際には、このpaintComponent()メソッドが呼び出されることで円が描かれるわけです。paintComponent()メソッドは、ウィンドウがアイコン化（最小化）され、再度元に戻されるような場合にも呼び出されます。図形を再び描く必要があるからです。それ以外にも、図形を描き直す必要がある場合には、その都度paintComponent()メソッドが呼び出されるのです。

その他、重要なのは、このメソッドを呼び出すコードをプログラマが書くことはない、という点です。ただ、後述するように、画面を更新するためのrepaint()メソッドを呼び出すことで、間接的にpaintComponent()メソッドを呼び出すことは可能です。paintComponent()メソッドの引数となっているGraphicsオブジェクトは、描画のための「キャンバス」のような役目を果たすオブジェクトです。このオブジェクトを作るためのコードもプログラマが書くことはできません。このオブジェクトはシステムによって自動的に作成されます。

```
import java.awt.*;
import javax.swing.*;

class MyDrawPanel extends JPanel {

  public void paintComponent(Graphics g) {

    g.setColor(Color.orange);

    g.fillRect(20, 50, 100, 100);
  }
}
```

どちらのコードも必要

JPanelを継承する。このウィジェットは、他のウィジェットと同様にフレームに組み込むことができる。ただ、このウィジェットには、他のウィジェットとは異なり、プログラマのオリジナルの図形が表示される

グラフィックスの表示には不可欠のメソッド。プログラマが自ら呼び出すことはないという点に注意。このメソッドはシステムによって自動的に呼び出される。このメソッドが呼び出されると、Graphicsオブジェクトの「キャンバス」に、図形が描画される

「g」は描画を行う機械のようなもの、と考えてもよい。ここに示したようなコードで、描画に使用する色や描画すべき図形の形状、配置（座標）、大きさなどが指定できる

This PDF is prepared for tools@maruhouse.org, personal use only.

paintComponent()メソッドの例

次にpaintComponent()メソッドのオーバーライド例を2つほど紹介しましょう。2つの例について理解できたら、その後は自分でもオーバーライドをしてみてください。Graphicsオブジェクトの使い方は、APIのドキュメントで確認してください。

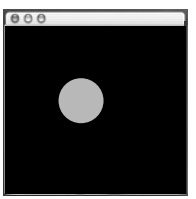
JPEG画像を表示する

```
public void paintComponent(Graphics g) {

  Image image = new ImageIcon("catzilla.jpg").getImage();

  g.drawImage(image, 3, 4, this);

}
```

JPEGファイルの名前はここに指定する

「3、4」は、画像の左上角の座標。「3」は、「パネルの左端から数えて3番目のピクセル」を、「4」は「パネルの上端から数えて4番目のピクセル」を意味する。この座標は常に、ウィジェット（ここではJPanelのサブクラスのオブジェクト）上での位置を表し、フレーム全体における位置を表すものではない

黒の背景に様々な色の円を描画する（色はランダムに決定する）



```
public void paintComponent(Graphics g) {

  g.fillRect(0, 0, this.getWidth(), this.getHeight());

  int red = (int)(Math.random() * 255);
  int green = (int)(Math.random() * 255);
  int blue = (int)(Math.random() * 255);

  Color randomColor = new Color(red, green, blue);
  g.setColor(randomColor);
  g.fillOval(70, 70, 100, 100);
}
```

パネル全体を黒で塗りつぶす（黒はデフォルトカラー）

最初の2つの引数で、描画を開始する地点の座標を指定する。座標はパネルの左上角を基点（0, 0）とする。残り2つの引数は、描画する長方形の縦、横の長さをパネルと同じにするためのもの

3つの整数値（RGB値）を指定することで色を作る

描画の開始点の座標を（70, 70）とし、描画する図形のサイズを縦100ピクセル、横100ピクセルにする

This PDF is prepared for tools@maruhouse.org, personal use only.

Graphics2DをGraphicsの代わりに使用する

paintComponent()の引数の型はGraphics(java.awt.Graphics)になっています。

```
public void paintComponent(Graphics g) { }
```

つまり、Graphicsのサブクラスのオブジェクトリファレンスも引数として使えるということになります。

現に、前ページにも出てきた「g」という変数には、「Graphics2D」というGraphicsのサブクラスのオブジェクトリファレンスが格納されます。

Graphics2DクラスはGraphicsクラスにない機能をいくつも備えているため、Graphics2Dクラスのオブジェクトリファレンスを使う方が高度なプログラムが作れます。

すでに見てきたとおり、コンパイラは、呼び出されたメソッドがどれであるかを、オブジェクトではなく参照の格納された変数の型によって判断します。たとえば、以下のように、Animal型の変数にDogオブジェクトへの参照を格納した場合は、

```
Animal a = new Dog();
```

以下のようなコードを書くことはできません(bark()がDogクラスのメソッドである場合)

```
a.bark();
```

人間の目から見れば、このコードでDogクラスのbark()メソッドを呼び出そうとしているということがすぐにわかるのですが、コンパイラは、変数の型のみで判断するため、Animalクラスにないbark()を呼び出せば誤りとみなされるのです。どうしてもbark()メソッドを呼び出す必要がある場合は、以下のように変数の型をDogに変えます。

```
Dog d = (Dog) a;
d.bark();
```

同じことは変数「g」にも言えます。

paintComponent()メソッドのパラメータであるgには、すでに述べたとおり、Graphics2Dクラスのオブジェクトリファレンスが格納されるのですが、そのままでは、Graphics2Dクラス独自のメソッドを呼び出すことはできません。そのためには以下のように、変数のキャストを行う必要があるのです。

```
Graphics2d g2d = (Graphics2d) g;
```

Graphics型の変数を使って呼び出せるメソッド

```
drawImage()
drawLine()
drawPolygon()
drawRect()
drawOval()
fillRect()
fillRoundRect()
setColor()
```

Graphics型の変数をGraphics2D型にキャストする

```
Graphics2D g2d = (Graphics2D) g;
```

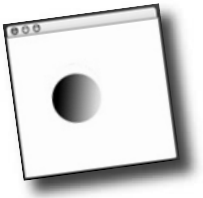
Graphics2D型の変数を使って呼び出せるメソッド

```
fill3DRect()
draw3DRect()
rotate()
scale()
shear()
transform()
setRenderingHints()
```

(メソッドはここにあげたものですべてではありません。詳しくはAPIのドキュメントを参照してください)

This PDF is prepared for tools@maruhouse.org, personal use only.

描画する円の色をグラデーションにするためのコード



```
public void paintComponent(Graphics g) {
  Graphics2D g2d = (Graphics2D) g;

  GradientPaint gradient = new GradientPaint(70, 70, Color.blue, 150, 150, Color.orange);

  g2d.setPaint(gradient);
  g2d.fillOval(70, 70, 100, 100);
}
```

実際にはGraphics2Dオブジェクトへの参照だが、Graphicsオブジェクトへの参照に見える

Graphics2Dクラス独自のメソッドを呼び出すため、変数をキャスト

グラデーション開始地点の座標

グラデーションを開始する色

終了地点の座標

終了する色

この設定によって、単色ではなく、グラデーションで彩色が行われるようになる

fillOval()は、楕円を塗りつぶすためのメソッド。setPaint()の引数にgradientが指定されると塗りつぶしの色がグラデーションになる

```
public void paintComponent(Graphics g) {
  Graphics2D g2d = (Graphics2D) g;

  int red = (int) (Math.random() * 255);
  int green = (int) (Math.random() * 255);
  int blue =(int) (Math.random() * 255);
  Color startColor = new Color(red, green, blue);

  red = (int) (Math.random() * 255);
  green = (int) (Math.random() * 255);
  blue = (int) (Math.random() * 255);
  Color endColor = new Color(red, green, blue);

  GradientPaint gradient = new GradientPaint(70, 70, startColor, 150, 150, endColor);
  g2d.setPaint(gradient);
  g2d.fillOval(70, 70, 100, 100);
}
```

このコードも上のものとほぼ同じだが、グラデーションを開始する色と終了する色が無作為に決められるという点が異なる

This PDF is prepared for tools@maruhouse.org, personal use only.

重要ポイント

イベント

- GUIを作成するには、通常、まずJFrameオブジェクト(ウィンドウ)を作ることになる。

```
JFrame frame = new JFrame();
```

- ウィンドウ(フレーム)にウィジェット(ボタン、テキストフィールドなど)を組み込むには、以下のようなコードを書く。

```
frame.getContentPane().add(button);
```

- 他のコンポーネントとは異なり、JFrameには、他のコンポーネントを直接組み込むことができない。まず「コンテンツペイン」と呼ばれるコンポーネントを組み込む必要がある。
- ウィンドウのサイズを画面に表示するには、以下のようなコードを使用する。まずサイズを設定している点に注意。

```
frame.setSize(300, 300);
frame.setVisible(true);
```

- 「ボタンがクリックされた」といったユーザーの操作を検知するには、イベントリスナーが必要。
- イベントリスナーは対応するイベントソースに登録が必要。イベントソースとはユーザーの操作などに応じてイベントを発生させるオブジェクト(ボタン、チェックボックスなど)。
- イベントリスナーを作るにはリスナーインターフェースとそのメソッド(イベント処理メソッド)をインプリメントする。メソッドは、イベント発生時にイベントソースによって呼び出される。
- イベントリスナーを登録するには、イベントソースに用意された登録用のメソッドを呼び出す。このメソッドの名前は常に「add<EventType>Listener」という形式になる。たとえば、ボタンのActionEventに対応するリスナーの登録は以下のようなコードを使って行う。

```
button.addActionListener(this);
```

- リスナーインターフェースのメソッドのインプリメントの際は、イベント処理のコードを書く。たとえば、ActionEventに対応するメソッドのコードは以下のようになる。

```
public void actionPerformed(ActionEvent
                                    event) {
  button.setText("you clicked!");
}
```

- イベント処理メソッドには、引数としてイベントオブジェクトが渡される。イベントオブジェクトはイベントに関する情報(イベントがどこで発生したか、など)を保持する。

グラフィックス

- ウィジェット上には2次元の図形を直接描くということもできる。
- ウィジェット上にGIF画像、JPEG画像を表示することもできる。
- 図形を描いたり、GIF、JPEG画像を表示したりするには、JPanelクラスを継承し、paintComponent()メソッドをオーバーライドする必要がある。
- paintComponent()はシステムによって自動的に呼び出されるメソッド。このメソッドを呼び出すコードをプログラマが書くことはない。paintComponent()の引数となるGraphicsオブジェクトは、描画のための「キャンバス」のような役目を果たすオブジェクト。このオブジェクトを作成するためのコードをプログラマが書くことはできない。
- Graphicsオブジェクト(paintComponent()の引数)を使用して呼び出されるメソッドの例としては以下のようなものがあげられる。

```
graphics.setColor(Color.blue);
g.fillRect(20, 50, 100, 120);
```

- JPEG画像を表示するにはまず以下のようなコードで画像データを取得する。

```
Image image = new ImageIcon("catzilla.
jpg").getImage();
```

そして以下のようなコードで描画を行う。

```
g.drawImage(image, 3, 4, this);
```

- paintComponent()の引数は「Graphics型」になっているが、実際に格納されるのは、正確にはGraphics2Dオブジェクトへの参照。Graphics2Dクラスには以下に示すような独自のメソッドがある。

```
fill3DRect()、draw3DRect()、rotate()、
scale()、shear()、transform()
```

- Graphics2Dのメソッドを呼び出すには、gの型をキャストする必要がある。

```
Graphics2d g2d = (Graphics2d) g;
```

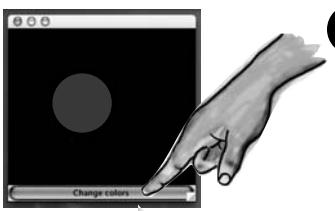

This PDF is prepared for tools@maruhouse.org, personal use only.

イベントと図形の描画

それでは次に、イベントが発生するとそれに応じて描画パネル上の表示を変化させる、というプログラムを作ってみましょう。ユーザーがボタンをクリックすると画面上の円の色が変わる、というプログラムです。このプログラムでの処理の流れは以下のようになります。

プログラムの起動

1 描画パネルとボタンという 2 つのウィジェットを備えたフレームが作成され、画面に表示される。ボタンには、イベントリスナーが登録され、ユーザーのクリックに対応できる状態になる。



2 ユーザーがボタンをクリックする。すると、イベントオブジェクトが作成され、イベントリスナーのメソッド（イベント処理メソッド）が呼び出される。

3 イベント処理メソッド（イベントハンドラとも呼ばれる）がフレームオブジェクトのrepaint()メソッドを呼び出す。すると、システムによって描画パネルオブジェクトのpaintComponent()メソッドが呼び出される。

4 paintComponent()によって円の色が変更される。色は無作為に選ばれる。

This PDF is prepared for tools@maruhouse.org, personal use only.

フレームに複数のウィジェットを組み込む

GUIのレイアウトについては次の章で詳しく解説しますが、ここでも簡単に触れておくことにします。フレームはデフォルトでは5つの「領域」に分かれており、各領域に1つずつウィジェットを組み込めるようになっています。ただ、1つの領域に描画パネルを組み込んだとすると、その描画パネルにはさらに複数のウィジェットを組み込むことができます。そして、組み込んだウィジェットも描画パネルであれば、さらにその中に複数のウィジェットが組み込める……といった具合なので、実際にはフレームに5つ以上のウィジェットを組み込むことは可能なのです。ここでは、ボタンと描画パネルを別の領域に組み込むことにします。

```
frame.getContentPane().add(button);
```

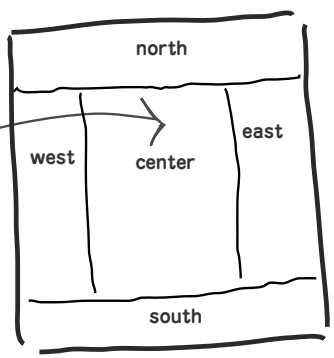
ボタンをこのようなかたちで組み込むことはしない（引数が1つのadd()メソッドは使わない）

```
frame.getContentPane().add(BorderLayout.CENTER,button);
```

ボタンを組み込む領域を指定する

引数が1つのadd()メソッドを使用すると、ウィジェットは自動的に「center」領域に組み込まれる。引数が2つのadd()メソッドを使用すれば、CENTERなどの定数を使用してウィジェットを組み込む領域を指定できる

デフォルト領域は「center」



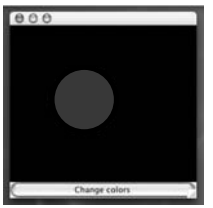
自分で考えてみよう

プログラムの画面が前ページの図のようになるよう、ボタンと描画パネルをフレームに組み込むコードを書いてみてください。

This PDF is prepared for tools@maruhouse.org, personal use only.

ボタンのクリックによって円の色を変化させる

描画パネル（独自に作成したMyDrawPanelクラスのオブジェクト）をフレームの「CENTER」領域に組み込む



ボタンは「SOUTH」領域に組み込む

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class SimpleGui3C implements ActionListener {

  JFrame frame;

  public static void main (String [] args) {
    SimpleGui3C gui = new SimpleGui3C();
    gui.go();
  }

  public void go() {
    frame = new JFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    JButton button = new JButton("Change colors");
    button.addActionListener(this);

    MyDrawPanel drawPanel = new MyDrawPanel();

    frame.getContentPane().add(BorderLayout.SOUTH, button);
    frame.getContentPane().add(BorderLayout.CENTER, drawPanel);
    frame.setSize(300, 300);
    frame.setVisible(true);
  }

  public void actionPerformed(ActionEvent event) {
    frame.repaint();
  }
}
```

リスナー（this）をボタンに登録する

2つのウィジェット（ボタンと描画パネル）をフレームの2つの領域に組み込む

ユーザーがクリックをすると、このメソッドによりフレームが再描画される。その際には、フレーム中のすべてのウィジェットオブジェクトのpaintComponent()メソッドが呼び出される

```
class MyDrawPanel extends JPanel {
  public void paintComponent(Graphics g) {
    // 楕円を無作為に選択した色で塗りつぶすためのコード
    // このコードはすでに紹介済み
  }
}
```

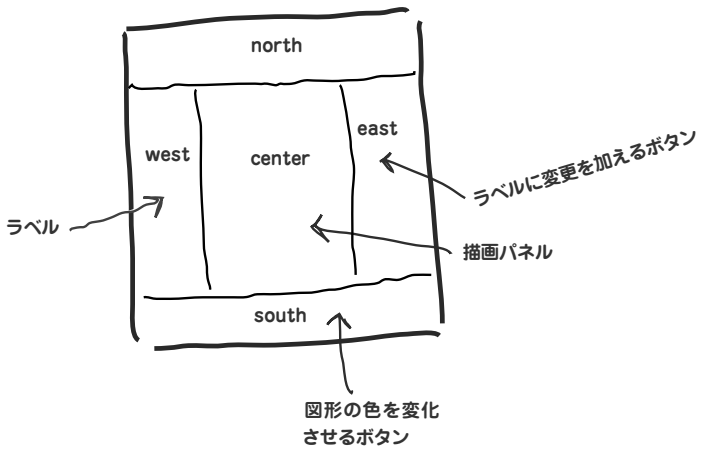
ユーザーがクリックをするごとに、描画パネルオブジェクトのpaintComponent()メソッドが呼び出される

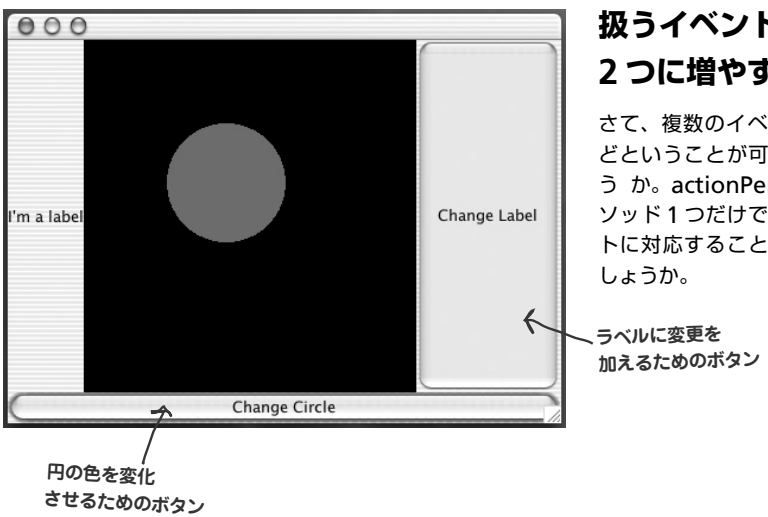
This PDF is prepared for tools@maruhouse.org, personal use only.

ボタンを 2 つにする

描画パネルの図形に変更を加えるボタンに加え、今度はeast領域に、ラベル（画面上のテキストのこと）に変更を加えるためのボタンを組み込んでみましょう。

ウィジェットを 4 つに増やす





扱うイベントは 2 つに増やす

さて、複数のイベントを扱うなどということが可能なのでしょう か。actionPerformed() メソッド 1 つだけで 2 つのイベントに対応することはできるのでしょうか。

This PDF is prepared for tools@maruhouse.org, personal use only.

ボタンが 2 つあり、それぞれに異なる機能を持たせる場合、イベント処理をどのように行うべきか

① 方法 1：actionPerformed()メソッドを 2 つ作る

```
class MyGui implements ActionListener {
    // ここにさまざまなコードが入る

    public void actionPerformed(ActionEvent event) {
      frame.repaint();
    }

    public void actionPerformed(ActionEvent event) {
      label.setLabel("That hurt!");
    }
  }
```

こんなことは不可能！

問題点：このようなコードは実際には書けない。1 つのクラスに同じメソッドのインプリメンテーションを複数通り作ることは不可能。上記のようなコードはコンパイルできない。また、たとえコンパイルできたとしても、イベントソースが 2 つのメソッドを区別できないので正しく動作しない。

② 方法 2：2 つのボタンに同じリスナーを登録する

```
class MyGui implements ActionListener {
  // ここにインスタンス変数の宣言がいくつか入る

  public void go() {
    // GUI の作成
    colorButton = new JButton();
    labelButton = new JButton();
    colorButton.addActionListener(this);
    labelButton.addActionListener(this);
    // さらに GUI 関連のコードが続く ...
  }

  public void actionPerformed(ActionEvent event) {
    if (event.getSource() == colorButton) {
      frame.repaint();
    } else {
      label.setLabel("That hurt!");
    }
  }
}
```

ここで 2 つのボタンに同じリスナーを登録する

イベントオブジェクトを利用してイベントソースがどちらのボタンなのかを確認し、その結果によって処理を変える

問題点：このコードは一応正しく機能するが、「オブジェクト指向的」とは言えない。1 つのイベントハンドラにいくつもの機能を持たせると、後で修正することが難しくなり、保守性、拡張性の面で問題が出てくる。どのイベントソースとどのイベントハンドラが対応するのかがわかりにくくなるため。

This PDF is prepared for tools@maruhouse.org, personal use only.

ボタンが 2 つあり、それぞれに異なる機能を持たせる場合、イベント処理をどのように行うべきか（続き）

③ 方法 3：イベントリスナーを 2 つ作る

```
class MyGui {
  JFrame frame;
  JLabel label;
  void gui() {
    //2 つのリスナーをインスタンス化し、一方を色変更のためのボタン、
    // もう一方をラベル変更のためのボタンに登録するコード
  }
} // クラスの終了
```

```
class ColorButtonListener implements ActionListener {
  public void actionPerformed(ActionEvent event) {
    frame.repaint();
  }
}
```

誤り。このクラスには、frame という変数はない。
これは MyGui クラスの変数

```
class LabelButtonListener implements ActionListener {
  public void actionPerformed(ActionEvent event) {
    label.setLabel("That hurt!");
  }
}
```

誤り。このクラスには、label という変数はない。

問題点：この場合、リスナーオブジェクトは frame、label といった変数にはアクセスできない。各リスナーで MyGui クラスのインスタンスを作成すれば、actionPerformed() メソッドで変数 frame、label にアクセスするということは可能だが、その場合、「new」を使ってインスタンスを作成するのは安全の面で問題があるのでゲッターメソッド（getFrame()、getLabel() など）を使うことになる。また、リスナークラスには、インスタンス化の際に MyGui クラスオブジェクトリファレンスを引数に指定できるコンストラクタの追加も必要。これではあまりにプログラムが複雑になってします。

もっと良い方法があるのでは？

This PDF is prepared for tools@maruhouse.org, personal use only.

This PDF is prepared for tools@maruhouse.org, personal use only.

内部クラス

Javaでは、クラスの中にクラスを作るということが簡単にできます。この種のクラスを「内部クラス」と呼びます。内部クラスを作る際には、そのコードを、完全に所属するクラスの{ }の中に入れるよう注意します。

簡単な内部クラスの例

```
class MyOuterClass {

    class MyInnerClass {
      void go() {
      }
    }

}
```

内部クラスは完全に外部クラスの中に入れる

内部クラスでは、外部クラスのメンバを、たとえprivate宣言されていても自らのものと同様に利用できる
内部クラスで外部クラスの変数、メソッドを利用する際には、内部クラスの変数、メソッドと区別する必要はない

内部クラスでは、所属するクラス（外部クラス）の変数やメソッドを自らのものと同様に利用できます。その変数やメソッドがprivate宣言されていてもその点は同じです。また、その他の点は通常のクラスとほぼ同じです。

内部クラスで外部クラスの変数を利用する

```
class MyOuterClass {

  private int x;

  class MyInnerClass {
    void go() {
      x = 42;
    }
  } // 内部クラスの終了
} // 外部クラスの終了
```

変数xは内部クラスの変数と同じように使用できる

This PDF is prepared for tools@maruhouse.org, personal use only.

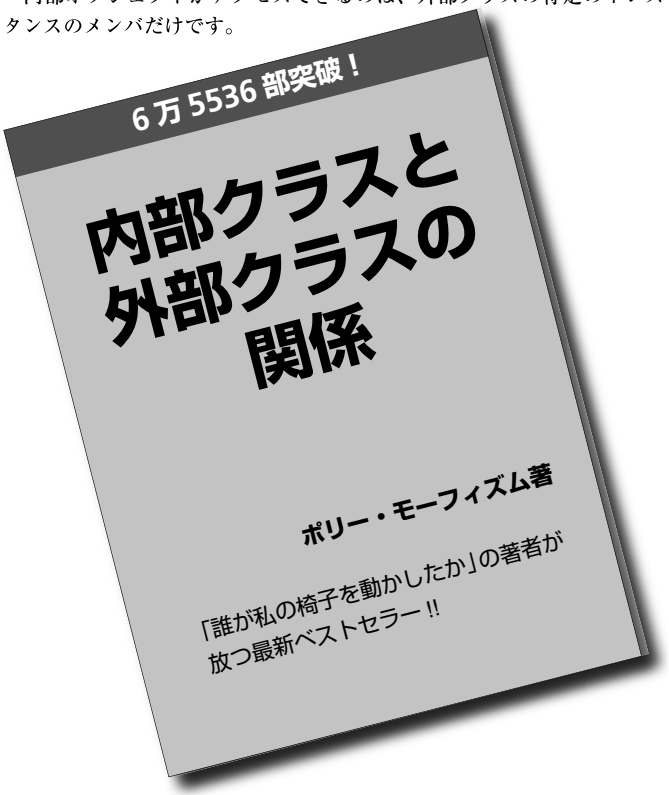
内部クラスのインスタンス（内部オブジェクト）は、ヒープ上で、常に特定の外部クラスのインスタンス（外部オブジェクト）と緊密に結び付く *

内部クラスが外部クラスのメンバにアクセスするというのは、正確には内部オブジェクトが外部オブジェクトのメンバにアクセスするということです。ただ、1 つ注意しておかねばならないことがあります。

内部オブジェクトがアクセスするのは、外部クラスのどのインスタンスのメンバでもいい、というわけではないということです。

内部オブジェクトがアクセスできるのは、外部クラスの特定のインスタンスのメンバだけです。

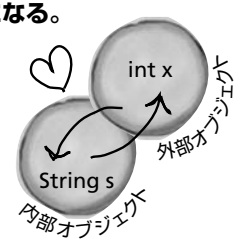
内部オブジェクトと外部オブジェクトは特別な絆で結ばれている



① **外部クラスのインスタンスを作成する。**

② **内部オブジェクトを、すでに作成した外部オブジェクトを使って作成する。**

③ **これにより、外部オブジェクトと内部オブジェクトは緊密に結び付くことになる。**



2 つのオブジェクトは、ヒープ上で緊密に結び付き、どちらもお互いのメンバを利用できる

* ただ、非常に特殊なケースですが、例外もあります。それは、内部クラスのコードが、スタティックメソッドのコードの中に書かれている場合です。ここではそのことに詳しくは触れません。Java を使っていて一度もそのような内部クラスを見ることのない人も多いからです。

This PDF is prepared for tools@maruhouse.org, personal use only.

内部オブジェクトを作る

内部オブジェクトは、外部クラスのコードによって作成されれば、その外部クラスのオブジェクトと結び付くことになります。インスタンス作成のコードがメソッドの中にあれば、そのメソッドの属する外部クラスのインスタンスと結び付きます。

内部クラスのインスタンスは、外部クラスのコードの中で作る場合には、通常のクラスのインスタンスと同様に「new」キーワードを使って作成できます。「new MyInner()」のように書けるわけです。

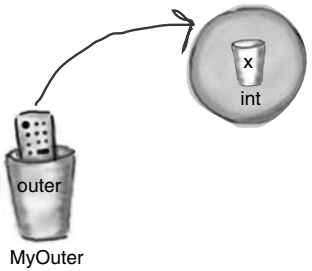
```
class MyOuter {

  private int x;

  MyInner inner = new MyInner();

  public void doStuff() {
    inner.go();
  }

  class MyInner {
    void go() {
      x = 42;
    }
  } // 内部クラスの終了

} // 外部クラスの終了
```

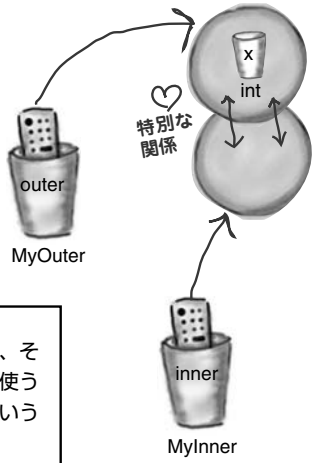
外部クラスのprivate宣言されたインスタンス変数x

内部クラスのインスタンスを作成

内部クラスのメソッドを呼び出すコード

内部クラスのメソッドで、内部クラスの変数と同じように、外部クラスのインスタンス変数xを利用できる





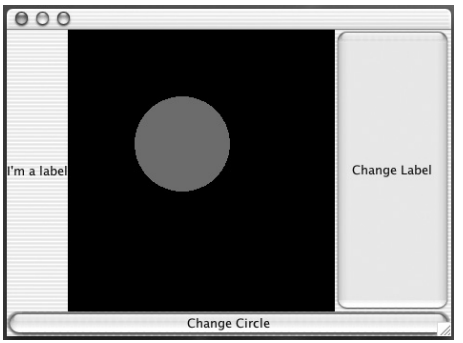
メモ

内部クラスのインスタンスは、外部クラス以外のクラスで作成することもできます。ただ、その場合には、単に「new MyInner()」のように書けばいいわけではなく、特殊なコードを使う必要があります。外部クラス以外のクラスで内部クラスのインスタンスを作成するなどということはめったにありませんが、参考までに書き方を見ておいてください。

```
class Foo {
  public static void main(String [] args) {
    MyOuter outerObj = new MyOuter();
    MyOuter.MyInner innerObj = outerObj.new Inner();
  }
}
```


This PDF is prepared for tools@maruhouse.org, personal use only.

2 つのボタンにそれぞれ異なる機能を持たせるコードは内部クラスを使って以下のように書ける

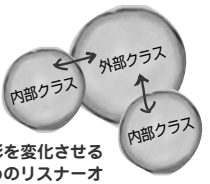



```
public class TwoButtons {

  JFrame frame;
  JLabel label;

  public static void main(String [] args) {
    TwoButtons gui = new TwoButtons ();
    gui.go();
  }

  public void go() {
    frame = new JFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    JButton labelButton = new JButton("Change Label");
    labelButton.addActionListener(new LabelListener());

    JButton colorButton = new JButton("Change Circle");
    colorButton.addActionListener(new ColorListener());

    label = new JLabel("I'm a label");
    MyDrawPanel drawPanel = new MyDrawPanel();

    frame.getContentPane().add(BorderLayout.SOUTH, colorButton);
    frame.getContentPane().add(BorderLayout.CENTER, drawPanel);
    frame.getContentPane().add(BorderLayout.EAST, labelButton);
    frame.getContentPane().add(BorderLayout.WEST, label);

    frame.setSize(300, 300);
    frame.setVisible(true);
  }

  class LabelListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
      label.setText("Ouch!");
    }
  } // 内部クラスの終了

  class ColorListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
      frame.repaint();
    }
  } // 内部クラスの終了
}
```

外部クラス。ActionListenerはインプリメントしない

ボタンのリスナーを登録するメソッドの引数として、「this」ではなく、内部クラス(いずれもリスナークラス)のインスタンスを指定している

2 つのボタンオブジェクト



図形を変化させるためのリスナーオブジェクト

ラベルを変化させるためのリスナーオブジェクト

2つのイベントリスナーが 1 つのクラスの中に納められている

内部クラスなのでインスタンス変数labelが利用できる

外部クラスのインスタンスへの参照がなくても、frame変数が利用できる

This PDF is prepared for tools@maruhouse.org, personal use only.

Javaの真実

ゲスト: 内部オブジェクト

著者: 内部クラスの価値はどういうところにあるのですか。

内部オブジェクト(以下、内部): そうですね、何からお話すればいいか……。まず、1つのクラスの中に、同じインターフェースのインプリメンテーションを複数通り作れるというところでしょうか。通常、そういうことはできませんから。ところが、内部クラスを使えば、それが可能になるんです。名前、引数が同じメソッドを同じクラスの中に複数個作ることもできます。

著者: 同じメソッドを複数個作ることがなぜ必要なのですか。

内部: たとえば、ボタンが3つあって、個々に違ったイベント処理を必要としているとします。そんな場合には、同じクラスの中にActionListenerをインプリメントする内部クラスを3つ作れば、それぞれのactionPerformed()メソッドに違う機能を持たせるということができます。

著者: では、内部クラスはイベント処理のために作るものだ、ということですか。

内部: いえ、それだけではありません。イベント処理は応用例の一つにすぎません。独立しているようでもあり、他のクラスの一部のようでもある、というクラスが必要な場合には、内部クラスを作ることが最良の(おそらく唯一の)選択肢となるでしょう。

著者: どうもまだよくわからないのですが……。内部クラスが所属するクラスの一部でもあり、同時に独立したクラスでもある、というのはどういうことでしょうか。そもそもなぜ独立したクラスのかたちをとる必要があるのですか。普通のコードではいけないのでしょうか。

内部: それはすでにお話したとおり、普通のコードでは、1つのクラスの中に同じインターフェースのインプリメンテーションを複数通り作るということができないからです。また、インターフェースを使用しない場合でも内部クラスを作ることはあります。その方がより「オブジェクト指向的」になることがあるからです。

著者: 待ってください。「オブジェクト指向的」とおっしゃいましたが、オブジェクト指向で重要なのは、再利用と保守ですよね。本質的に異なるコードを1つのクラスにまとめてしまうより、クラスを分けた方がそれぞれを独立して使うことができ、修正なども容易になるというのはわかります。しかし、内部クラスはどうでしょう。内部クラスを通常のクラスと同じようにまったく独立したクラスとして扱うことができるのでしょうか。内部クラスが所属するクラスは確かに他から完全に独立していて再利用も可能です。でも内部クラスはそうではないと思います。それどころか、内部クラスの再利用はまったく不可能という話もよく聞きます。

内部: 内部クラスが通常のクラスに比べて再利用しにくい、場合によって再利用が不可能であるというのは本当です。それは外部クラスのインスタンス変数やメソッドと密接に結び付いているためです。ただ……。

著者: やはり私の言ったとおりじゃないですか。再利用もできないのに、独立したクラスにする意味がどこにあるんですか。インターフェースのインプリメンテーションを複数通り作るというのはわからないでもありませんが、それ以外は私には単なる弥縫策(びほう)のようなものにすぎないような気がします。

内部: IS-A関係やポリモーフィズムといったことについて考えると内部クラスの価値がよりわかると思いますよ。

著者: そのことについては私も言いたいことが……。

内部: 1つのクラスが同時に複数のクラスと「IS-A関係」になっていなければならない、ということがあるんですよ。先ほどの例のように1つのクラスの中に複数のイベントリスナーを作った場合、ボタンのリスナー登録用メソッドの引数には何を指定すればよいでしょうか。つまり、addActionListener()メソッドの引数の型がどのようなものになるか、ということです。これはAPIのドキュメントを見るとわかります。

著者: 引数は「リスナー」ですよね。特定のリスナーインターフェースをインプリメントするクラス、ということですが。addActionListener()メソッドの場合は、ActionListenerインターフェースをインプリメントするクラスということになりますね。それくらいはわかります。何が言いたいんですか。

内部: 引数の型は常に1つだということです。addActionListener()メソッドなら、引数は、必ずActionListenerと「IS-A関係」にあるものでなければなりません。問題は、引数がインターフェースではなくて、特定のクラスとIS-A関係になければならない場合にどうするか、ということです。

著者: 型として指定されたクラスを継承してさえいればいいんですよね。引数の型がAであれば、そのサブクラスBのオブジェクトは引数になれる。そういうことじゃないんですか。Animal型の引数にDogオブジェクトを指定することは可能ですよね。

内部: そのとおりです。では、A、Bという互いにまったく継承関係にはない2つのクラスがあったとして、この両者とIS-A関係にあるクラスを作らねばならないとしたらどうすれ

This PDF is prepared for tools@maruhouse.org, personal use only.

ばいいでしょうか。

著者：うーん……、そうですね……。おっしゃりたいことが段々わかってきました。複数のインターフェースをインプリメントすることはできるが、継承できるクラスは常に1つである、ということですね。複数のクラスと同時にIS-A関係にあるクラスはできません。

内部：そうです。DogクラスとButtonクラスを同時に継承するなどということはできません。しかし、クラスによっては、「基本的にはDogでよいが、時にはButtonとしての機能も必要（引数がButton型になっているメソッドにオブジェクトを渡さねばならない、など）」ということもあります。その場合には、Dogを継承するクラスの中に、Buttonを継承する内部クラスを作ればよいのです。そうすれば、Buttonとしての機能が必要になった時には、内部クラスを使うことができます。x.takeButton(this)と書く代わりに、x.takeButton(new MyInnerButton())と書けば、たとえ引数がButton型になっているメソッドでも呼び出せます。

著者：何かもっと具体的な例をあげてもらえませんか。

内部：たとえば描画パネルはJPanelを継承して作りますが、本書で紹介した描画パネルのクラスは内部クラスではなく、独立したクラスになっていました。これは、他のクラス（GUIプログラムのメインとなるクラスなど）のインスタンス変数などにアクセスする必要がなかったからです。ただ、常にそうであるとは限りません。描画パネル上の図形をユーザーの操作に応じて動かすプログラムを作る場合などには、描画パネルクラスから、他のクラスの変数にアクセスして操作の行われた座標を知る、といったことが必要でしょう。この場合は、描画パネルのクラスを内部クラスにします。描画パネルのクラスを内部クラスにするのであれば、外部クラスは他のどのクラスを継承するものであってもかまいません。

著者：わかりました。描画パネルのクラスはアプリケーションごとに機能的に異なることが多いので、独立したクラスであっても元々、再利用しにくいですよね。

内部：そうです。おわかりいただけたようですね。

著者：では次に、あなたと外部オブジェクトとの関係についてお話を伺いたいのですが。

内部：今日は真面目な話ができると思っていたのですが、結局そういうゴシップになってしまうんですか。

著者：これも皆の知りたがっている大切な話なんですよ。内部オブジェクトは産まれると同時に特定の外部オブジェクトと結合されてしまうと聞いたんですが。それは本当ですか。

内部：ええ本当です。「まるで政略結婚だ」と陰口を言う人もいるようです。確かにどのオブジェクトと「結婚」するかについて自分で意見を言うこともできないわけですからね。

著者：「結婚」という言葉が出ましたので、そのたとえを使いますが、「離婚」や「再婚」はできるのですか。

内部：いいえ。それは一生できません。

著者：誰の「一生」ですか。あなたですか、外部オブジェクトですか、それとも両方ですか。

内部：私ですよ。一度外部オブジェクトと「結婚」したら、ガーベージコレクションまで二度と離れることはできません。

著者：外部オブジェクトはどうなんですか。離婚や再婚ができるのですか。

内部：来ましたね。実はそれが聞きたかったんでしょう。まあ、いいです。そうです。私の「配偶者」である外部オブジェクトは、いくつの内部オブジェクトとでも結婚できます。

著者：相手が一度に複数ということもあるんですか。それとも一人ずつで、ただ入れ替わっていくだけですか。

内部：一度に複数でも可能です。これで満足ですか。

著者：ええ、お話、よくわかりました。あなた自身、「同じインターフェースのインプリメンテーションを複数通り作れる」というように、内部オブジェクトの良さを強調してらっしゃいましたが、やはりつらいこともあったんですね。それは、外部オブジェクトが3つのボタンのイベントを処理しなければならないのであれば、同時に3つの内部オブジェクトと「結婚」することもあるでしょう。仕方のない話です。お話いただいてありがとうございました。ハンカチ、ありますよ。

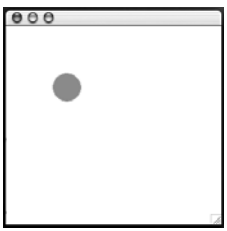
This PDF is prepared for tools@maruhouse.org, personal use only.

内部クラスをアニメーションに利用する

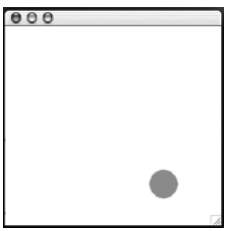
すでに見てきたとおり、内部クラスはイベントリスナーに使うのに便利です。同じイベント処理メソッドのインプリメンテーションを複数通り作ることができるからです。しかし、内部クラスの使い道はそれだけではありません。特に、外部クラスと内部クラスとで異なるクラスを継承する、というテクニックは、様々なプログラムに応用できます。ここではその応用例について見てみましょう。

例として使用するのは、画面の左上から右下に向かって円が移動していく、という簡単なアニメーションのプログラムです。

移動開始



移動終了



アニメーションプログラムの処理手順

① 画面上の指定の位置（座標）に円を描く。

```
g.fillOval(20, 50, 100, 100);
```

左端から 20 ピクセル、上端から 50 ピクセルの位置という意味

② はじめとは異なる座標に円を描き直す。

```
g.fillOval(25, 55, 100, 100);
```

左端から 25 ピクセル、上端から 55 ピクセルの位置という意味（はじめより少し右下に移動することになる）

③ 円が右下角に到達するまで、2 の処理を繰り返す。

素朴な疑問に答えます

Q：なぜここでアニメーションについて取りあげるのですか。ゲームプログラムを作りたい読者がそれほど多いとは思えないのですが。

A：ゲームを作る人は確かに多くないかもしれませんが、シミュレーションプログラムなどを作る人は意外に多いのではないでしょうか。また、メモリの残量を知らせるプログラムや、負荷分散のためにトラフィックの増減を監視するプログラムなどもアニメーションの応用です。ともかく、何かの連続的な変化についての情報を得るにはアニメーションが有効な手段であることは間違いありません。

「そんなのは後付の理屈だ」と思う人がいるかもしれません。特に、ビジネス用のアプリケーションを作るためにJavaを学んでいる人（そういう人はかなり多いはずです）はそうでしょう。確かにそのとおりです。著者がここでアニメーションについてとりあげるのは、第一に内部クラスの用途の例として非常にわかりやすいからです（もちろん、著者がアニメーション好きである、ということも大きな理由です。また次に予定している著書がJ2EEに関するもので、そこではアニメーションについて書けそうもないということも理由の 1 つと言えます）。

This PDF is prepared for tools@maruhouse.org , personal use only.

アニメーションが表示される描画パネルクラスのコードは以下のようなものになる……

```
class MyDrawPanel extends JPanel {
  public void paintComponent(Graphics g) {
    g.setColor(Color.orange);
    g.fillOval(x, y, 100, 100);
  }
}
```

paintComponent() が呼び出されるたびに、円が違う位置に描かれる

自分で考えてみよう

円の x、y 座標が次々に変わるようにするにはどうすればよいか。

また、repaint() はどこで呼び出せばよいか？

上記のような描画パネルクラスを使用した場合、円が描画パネルの左上から右下に向かって移動していくようなプログラムのコードがだいたいどのようなものになるかを考えてみてください。答えは次のページを見ればわかりますが、自分なりの答えが出るまでは見ないようにしてください。

ヒント 1：描画パネルクラスは内部クラスにする
ヒント 2：paintComponent() メソッドの中には、どのようなものであれループは入れない

答えは下に書いてください。完全なコードがわかればそれを書いてください。

This PDF is prepared for tools@maruhouse.org, personal use only.

アニメーションプログラムのコード

```
import javax.swing.*;
import java.awt.*;

public class SimpleAnimation {
  int x = 70;
  int y = 70;

  public static void main (String [] args) {
    SimpleAnimation gui = new SimpleAnimation();
    gui.go();
  }

  public void go() {
    JFrame frame = new JFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    MyDrawPanel drawPanel = new MyDrawPanel();

    frame.getContentPane().add(drawPanel);
    frame.setSize(300, 300);
    frame.setVisible(true);

    for (int i = 0; i < 130; i++) {

      x++;
      y++;

      drawPanel.repaint();

      try {
        Thread.sleep(50);
      } catch(Exception ex) { }
    }

  } // go() メソッドの終了

  class MyDrawPanel extends JPanel {

    public void paintComponent(Graphics g) {
      g.setColor(Color.green);
      g.fillOval(x, y, 40, 40);

    }
  } // 内部クラスの終了
} // 外部クラスの終了
```

円のx、y座標を納める変数をインスタンス変数にする

ウィジェットを作成し、フレームに組み込む。この種のコードについてはすでに説明済み

絵を動かすためのコード

130 回同じ処理を繰り返す

x、y座標をインクリメントする

パネルを描画し直す（その都度、円の位置が変わる）

動きを少し遅くする（そうしなければあまりに速すぎて円の動きが目に見えなくなる）。このコードは「スレッド」に関するもの。スレッドについては 15 章で詳述

描画パネルのクラスを内部クラスにする

外部クラスで次々に更新されるx、y座標を利用する

This PDF is prepared for tools@maruhouse.org, personal use only.

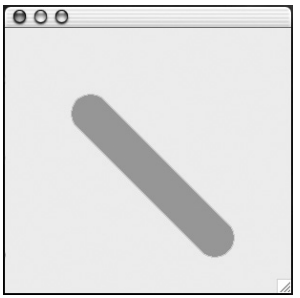
前ページのコードには問題が……

原因は？

paintComponent()メソッドのコードの欠陥です。

描画済みの円を消去せずに新しい円を描いているため、円は動かず、右の図のような奇妙な図形が描かれる

円を消去するためには、以下に示したような、描画パネル全体を背景色で塗りつぶすコード（2 行）をpaintComponent()メソッドに追加する必要があります。setColor()は背景色を指定するためのメソッドです。ここでは背景色として白を指定しています。fillRect()は長方形を塗りつぶすためのメソッドです。最初の 2 つの引数は塗りつぶしを開始する座標を指定するためのもので（ここではパネルの左上角が指定されている）、残りの 2 つは塗りつぶす範囲の縦、横の長さを指定するためのものです（ここでは、縦、横、ともに描画パネルと一致するようにしている）。



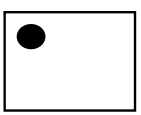
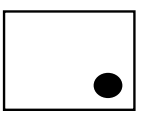
前ページのコードを実行するとこのような画面になる

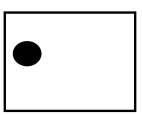
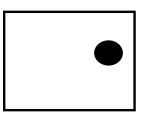
```
public void paintComponent(Graphics g) {
  g.setColor(Color.white);
  g.fillRect(0, 0, this.getWidth(), this.getHeight());

  g.setColor(Color.green);
  g.fillOval(x, y, 40, 40);
}
```

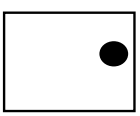
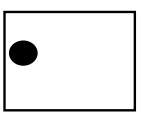
getWidth()、getHeight()はJPanelクラスから継承したメソッド

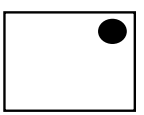
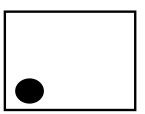
自分で考えてみよう（余裕のある人は挑戦してください）

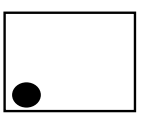
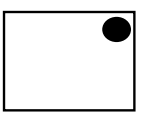
下に示したようなアニメーションを作るためには、それぞれx、y座標をどれだけ変化させればいいか、考えてみてください。（1 ～ 6 までありますが、1 はヒントです）

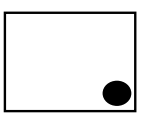
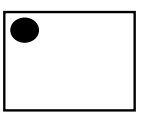
1

開始

終了
x +3
y +3

2

開始

終了
x____
y____

3

開始

終了
x____
y____

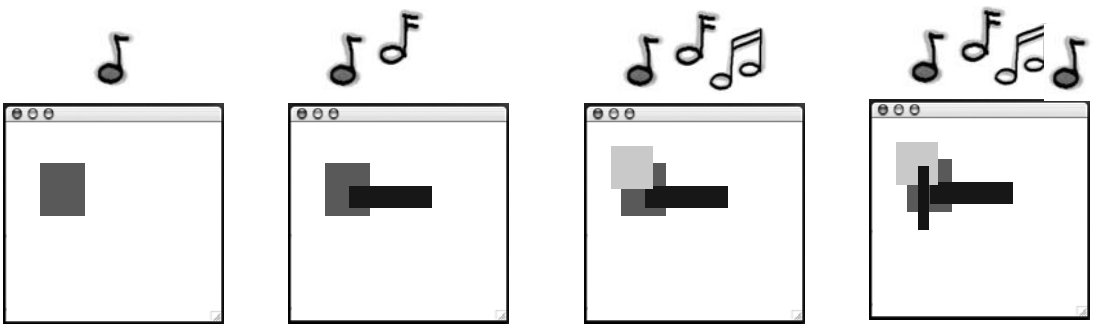
1

開始

終了
x____
y____

2

開始

終了
x____
y____

3

開始

終了
x____
y____

This PDF is prepared for tools@maruhouse.org, personal use only.

プログラム・キッチン



画面 1　画面 2　画面 3　画面 4……

それでは、ミュージックビデオプログラムを作ってみましょう。
これは、音楽に合わせてさまざまな図形を表示させるというプログラムです。
このプログラムを作るには、GUI部品ではなく音楽によって発生する
イベントを処理するためのコード（リスナー）を書く必要があります。

前の章と同じように、この「プログラム・キッチン」のページは、興味がない人は
飛ばして読んでもかまいません。しかし、読めば必ず役に立ち、楽しいはずです。
また、できたプログラムを見て、周囲の人は驚くに違いありません。
（このぐらいのプログラムでは、あまり驚いてくれない人もいるでしょうが……
でもできれば挑戦してみてください）

This PDF is prepared for tools@maruhouse.org, personal use only.

GUI部品以外で発生するイベントに対応するリスナーの作成

先述のとおり、ここで作成するのは、音楽に合わせて画面上に様々な図形を表示する、というプログラムです。音楽を作る時に発生するイベントに対応して図形（長方形）を描くプログラムと言ってもいいでしょう。単純なものなので、「ミュージックビデオプログラム」というのは少し大げさかもしれませんが……。

これまで本書で取りあげてきたイベントは、すべてGUI部品で発生するものでした。しかし、ここで作成するプログラムの場合、対応すべきイベントは音楽を演奏する上で発生する、「MIDIイベント」とでも呼ぶべきものです。ただ、この種のイベントもやはりイベントなので、対応そのものはGUIイベントとさほど変わるものではありません。リスナーインターフェースをインプリメントしてリスナーを作り、そのリスナーをイベントソースに登録するという点は同じです。イベントが発生すれば、イベントソースでは、リスナーに用意されたイベント処理メソッド（リスナーインターフェースに宣言されたもの）を呼び出すことになります。

最も簡単なのは、「音が出た」というイベント（NOTE ONイベント）に対応するリスナーを作り、それを登録することです。イベントが発生したら、リスナーが図形を描くようにするというわけです。しかし、困ったことに、実際にはそういうことはできません（バグが原因）。

次善の策としては、Controllerイベントという別のMIDIイベントを使うものが考えられます。このイベントに対応するリスナーを作り、登録するのです。そして、NOTE ONイベント発生時に必ずControllerイベントが発生するような仕組みを作る事も必要になります。そのために、具体的には、NOTE ONイベント発生を指示するMidiEventオブジェクトをTrackオブジェクトに追加する際、同時にControllerイベント発生を指示するMidiEventオブジェクトを追加するということになるでしょう。つまり、演奏の指示を以下のようなものにするということです。

1 -NOTE ON、CONTROLLER
2 -NOTE OFF
3 -NOTE ON、CONTROLLER
4 -NOTE OFF

プログラムの作り方についての解説の前に、次のページでは「MidiEventオブジェクトを作成し、Trackオブジェクトに追加する」という処理を簡単に行う方法を説明します。この処理は何度も行わねばならないため、少しでも効率化することが重要です。

「ミュージックビデオプログラム」で行うべき処理

1. 演奏に必要なMidiEventオブジェクトを作成し、Trackオブジェクトに追加する
2. イベントに対応するリスナーの登録
3. 演奏のスタート
4. リスナーのイベント処理メソッドが呼び出されるたびに、描画パネルに長方形を描く（描く長方形がどのようなものになるかは無作為に決定される）

プログラムは3つの段階に分けて完成させる

1. 第1段階：「MidiEventオブジェクトを作成し、Trackオブジェクトに追加する」という処理を簡単に行うためのコードを含むプログラムを作る
2. 第2段階：音が出るたびにコマンドラインにメッセージを表示するイベントリスナーを作り、登録する。この段階では、まだ図形の描画は行わない
4. 第3段階：プログラムを完成させる。第2段階のプログラムに図形を描画する機能を追加する

This PDF is prepared for tools@maruhouse.org, personal use only.

MidiEventオブジェクトの作成

「MidiEventオブジェクトを作成し、Trackオブジェクトに追加する」という処理は意外に煩雑なものです。逐一、ShortMessageオブジェクトを作成し、setMessage()メソッドで音の高さ、強さ、使用する楽器の種類などの設定をしてから、MidiEventオブジェクトを作成する、という手順を踏まねばならないからです。前の章ではそのためのコードを紹介しました。この方法だと、「音を出し、止める」というだけで8行ものコードを要することになってしまいます。NOTE ONイベントのために4行、NOTE OFFイベントに4行必要だからです。

```
ShortMessage a = new ShortMessage();
a.setMessage(144, 1, note, 100);
MidiEvent noteOn = new MidiEvent(a, 1);
track.add(noteOn);

ShortMessage b = new ShortMessage();
b.setMessage(128, 1, note, 100);
MidiEvent noteOff = new MidiEvent(b, 16);
track.add(noteOff);
```

MidiEventオブジェクトの作成とTrackオブジェクトへの追加に必要な処理

1. ShortMessageオブジェクトを作成する
   `ShortMessage first = new ShortMessage();`
2. setMessage()を呼び出す
   `first.setMessage(192, 1, instrument, 0)`
3. MidiEventオブジェクトを作成する
   `MidiEvent noteOn = new MidiEvent(first, 1);`
4. MidiEventオブジェクトをTrackオブジェクトに追加する
   `track.add(noteOn);`

コードを簡略化するためのユーティリティメソッドを作成する

setMessage()メソッドを呼び出すための4つの引数

このメソッドには5つの引数がある

```
public static MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
  MidiEvent event = null;
  try {
    ShortMessage a = new ShortMessage();
    a.setMessage(comd, chan, one, two);
    event = new MidiEvent(a, tick);

  } catch(Exception e) { }
  return event;
}
```

変数tickの値は、ShortMessageオブジェクトの指示をどのタイミングで実行するか、という情報。MidiEventオブジェクト作成に利用される

メソッドのパラメータを利用して、ShortMessageオブジェクト、MidiEventオブジェクトが作成される

必要な情報をすべて保持したMidiEventオブジェクトを戻す

This PDF is prepared for tools@maruhouse.org, personal use only.

ユーティリティメソッド makeEvent() の使用例

以下に示したのは、前ページで紹介した makeEvent() の使用例です。makeEvent() の使い方を知ってもらうことが目的なので、イベント処理や図形の描画は行わず、徐々に出す音の高さを上げていくだけの簡単なものになっています。この後に紹介するプログラムも、makeEvent() を使っているため、コードが短く簡単になっているという点に注目してください。

```
import javax.sound.midi.*;

public class MiniMusicPlayer1 {

  public static void main(String [] args) {

    try {

      Sequencer sequencer = MidiSystem.getSequencer();
      sequencer.open();

      Sequence seq = new Sequence(Sequence.PPQ, 4);
      Track track = seq.createTrack();

      for (int i = 5; i < 61; i+= 4)
        track.add(makeEvent(144, 1, i, 100, i));
        track.add(makeEvent(128, 1, i, 100, i + 2));
      } // ループの終了

      sequencer.setSequence(seq);
      sequencer.setTempoInBPM(220);
      sequencer.start();
    } catch(Exception ex) {ex.printStackTrace();}
  } // main の終了

  public static MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
      MidiEvent event = null;
      try {
        ShortMessage a = new ShortMessage();
        a.setMessage(comd, chan, one, two);
        event = new MidiEvent(a, tick);

      } catch(Exception e) { }
      return event;
  }
} // クラスの終了
```

必ずインポートをする

Sequencerオブジェクトの取得、オープン

Sequenceオブジェクト、Trackオブジェクトの作成

音を連続して出すため（音程を 5 ～ 61 まで徐々に上げていく）、MidiEventオブジェクトを複数作成

演奏の開始

MidiEventオブジェクトを作成し、Trackオブジェクトに追加する。MidiEventオブジェクトを作成するためには、makeEvent() を呼び出す（MidiEventオブジェクトはmakeEvent() の戻り値として得られる）。引数「144」は「NOTE ON」、「128」は「NOTE OFF」に対応

This PDF is prepared for tools@maruhouse.org, personal use only.

第 2 段階：Controllerイベントのリスナーの作成、登録

```
import javax.sound.midi.*;
public class MiniMusicPlayer2 implements ControllerEventListener {

  public static void main(String [] args) {
    MiniMusicPlayer2 mini = new MiniMusicPlayer2();
    mini.go();
  }

  public void go() {

     try {
       Sequencer sequencer = MidiSystem.getSequencer();
       sequencer.open();

       int [] eventsIWant = {127};
       sequencer.addControllerEventListener(this,eventsIWant);

       Sequence seq = new Sequence(Sequence.PPQ, 4);
       Track track = seq.createTrack();

       for (int i = 5; i < 60; i+= 4) {
           track.add(makeEvent(144, 1, i, 100, i));

           track.add(makeEvent(176, 1, 127, 0, i));

           track.add(makeEvent(128, 1, i, 100, i + 2));
       } // ループの終了

       sequencer.setSequence(seq);
       sequencer.setTempoInBPM(220);
       sequencer.start();
     } catch(Exception ex) {ex.printStackTrace();}
   } // メソッドの終了

   public void controlChange(ShortMessage event) {
     System.out.println("la");
   }

   public MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
     MidiEvent event = null;
     try {
       ShortMessage a = new ShortMessage();
       a.setMessage(comd, chan, one, two);
       event = new MidiEvent(a, tick);

     } catch(Exception e) { }
     return event;
   }
} // クラスの終了
```

Controllerイベント（ControllerEvent）に対応するリスナーを作るため、ControllerEventListenerインターフェースをインプリメントする

Controllerイベントに対応するリスナーをSequencerオブジェクトに登録する。メソッドの引数には、リスナーと、intの配列eventsIWantを指定する。eventsIWantを指定するのは、Controllerイベントのうち、番号が「127」となっているイベントのみを処理対象とするため

Controllerイベントを発生させるためのコード（176は、イベントのタイプがControllerEventであることを表す。127はイベントの番号）。このイベントが発生しても音楽の演奏にはまったく影響しない。ただ、「音が出された」ということを通知するために発生させる。音が出された時に、それに対応してリスナーで何かをするためだけに利用するイベント（NOTE ON/OFFイベントに対応するリスナーは作れない）。ここでは、ControllerイベントをNOTE ONイベントと同じタイミングで発生させなければならない点に注意

イベント処理のためのメソッド（ControllerEventlistenerインターフェースに宣言されているメソッド）。これにより、Controllerイベントが発生した時には、コマンドラインに「la」と表示される

前のページのプログラムと異なっている部分はグレーの四角をかぶせてある（main()が短くなり、go()というメソッドが増えている点にも注意）

This PDF is prepared for tools@maruhouse.org, personal use only.

第 3 段階：音楽に合わせて図形を描画する

第 3 段階では、前ページのプログラムに、図形を描く機能を追加することになります。まず、フレームを作成し、それに描画パネルを組み込んで、イベントが発生するたびにその上に新しい長方形が描かれるようにするのです。また、音楽の方も単純に音程が上がっていくだけのものから、無作為に演奏する音が選択されるよう改良します。

また、描画パネルクラス（内部クラス）自身がControllerEventListenerをインプリメントしたイベントリスナーになるのも、第 3 段階のプログラムの特徴です。従ってイベントに対応して長方形を描画する機能は、描画パネルオブジェクト自身が備えることになります。

以下に示すのは描画パネルクラスのコードです。プログラム全体のコードは次のページに載せてあります。

描画パネルクラス（内部クラス）

描画パネルクラスがイベントリスナーになる

```
class MyDrawPanel extends JPanel implements ControllerEventListener {

  boolean msg = false;

  public void controlChange(ShortMessage event) {
    msg =true;
    repaint();
  }

  public void paintComponent(Graphics g) {
    if (msg) {

      Graphics2D g2 = (Graphics2D) g;

      int r = (int) (Math.random() * 250);
      int gr = (int) (Math.random() * 250);
      int b = (int) (Math.random() * 250);

      g.setColor(new Color(r,gr,b));

      int ht = (int) ((Math.random() * 120) + 10);
      int width = (int) ((Math.random() * 120) + 10);
      int x = (int) ((Math.random() * 40) + 10);
      int y = (int) ((Math.random() * 40) + 10);
      g.fillRect(x, y, ht, width);
      msg = false;

    } // if 文の終了
  } // メソッドの終了
} // 内部クラスの終了
```

フラグをfalseに設定。このフラグはイベント発生時にのみtrueになる

イベントが発生したのでフラグをtrueにし、repaint()を呼び出す

フラグを使用するのは、そうしなければrepaint()がどのような理由で呼び出されたのかがわからないため。フラグを使用することで、Controllerイベント発生時にのみ描画をする、ということができる

色を無作為に選び、長方形を無作為に選んで描画をする

This PDF is prepared for tools@maruhouse.org, personal use only.

```
import javax.sound.midi.*;
import javax.swing.*;
import java.awt.*;

public class MiniMusicPlayer3 {

  static JFrame f = new JFrame("My First Music Video");
  static MyDrawPanel ml;

  public static void main(String [] args) {
    MiniMusicPlayer3 mini = new MiniMusicPlayer3();
    mini.go();
  } // メソッドの終了

  public void setUpGui() {
    ml = new MyDrawPanel();
    f.setContentPane(ml);
    f.setBounds(30, 30, 300, 300);
    f.setVisible(true);
  } // メソッドの終了

  public void go() {
    setUpGui();

    try {

      Sequencer sequencer = MidiSystem.getSequencer();
      sequencer.open();
      sequencer.addControllerEventListener(ml, new int [] {127});
      Sequence seq = new Sequence(Sequence.PPQ, 4);
      Track track = seq.createTrack();

      int r = 0;
      for (int i = 0; i < 60; i+= 4) {

        r = (int) ((Math.random() * 50) + 1);
        track.add(makeEvent(144, 1, r, 100, i));
        track.add(makeEvent(176, 1, 127, 0, i));
        track.add(makeEvent(128, 1, r, 100, i + 2));
      } // ループの終了

      sequencer.setSequence(seq);
      sequencer.start();
      sequencer.setTempoInBPM(120);
    } catch (Exception ex) {ex.printStackTrace();}
  } // メソッドの終了
```

自分で考えてみよう

ここに示したのは、「ミュージックビデオプログラム」の完成版のコードです。第 2 段階のコードを改良して作られているのですが、どの部分が変わっているでしょうか。第 2 段階のコードを見ずに考えてください。

This PDF is prepared for tools@maruhouse.org, personal use only.

```
    public MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
      MidiEvent event = null;
      try {
        ShortMessage a = new ShortMessage();
        a.setMessage(comd, chan, one, two);
        event = new MidiEvent(a, tick);

      } catch(Exception e) { }
      return event;
    } // メソッドの終了

  class MyDrawPanel extends JPanel implements ControllerEventListener {
    boolean msg = false;

    public void controlChange(ShortMessage event) {
      msg =true;
      repaint();
    }

    public void paintComponent(Graphics g) {
      if (msg) {

        Graphics2D g2 = (Graphics2D) g;

        int r = (int) (Math.random() * 250);
        int gr = (int) (Math.random() * 250);
        int b = (int) (Math.random() * 250);

        g.setColor(new Color(r, gr, b));

        int ht = (int) ((Math.random() * 120) + 10);
        int width = (int) ((Math.random() * 120) + 10);

        int x = (int) ((Math.random() * 40) + 10);
        int y = (int) ((Math.random() * 40) + 10);

        g.fillRect(x, y, ht, width);
        msg = false;

      } // if 文の終了
    } // メソッドの終了
  } // 内部クラスの終了
} // クラスの終了
```


This PDF is prepared for tools@maruhouse.org, personal use only.

Javaに関連する用語にはさまざまなものがあります。下の説明文に当てはまる用語が何かを解答欄に書いてください。同じ答えを何度書いてもかまいません。また、1つの説明文に複数の用語が当てはまる場合もあります。

ヒント：この章で学んだことを思い出せばどれもわかるはずです。

GUIの中ではじめに作成する _____

イベントリスナーを作る際に必要 _____

ボタンクリックに対応するイベントリスナーに必要なメソッド _____

JFrameのサイズを指定する _____

プログラマがこのメソッドを呼び出すコードを書くことはない _____

「ユーザーが何らかの操作をした」といった「できごと」を表す言葉 _____

ほぼすべてがイベントソースになる _____

イベントリスナーに情報を渡す役割を果たす _____

addXxxListener()という名前のメソッドがあればそのクラスは_____であることがわかる _____

イベントリスナーを登録するためのメソッド _____

グラフィックスを表示する際には必ず使用されるメソッド _____

必ず特定のオブジェクトと結合される _____

(Graphics g)の「g」は、このクラスのオブジェクト _____

間接的にpaintComponent()を呼び出すのに使用できるメソッド _____

Swingコンポーネントのほとんどが属するパッケージ _____

This PDF is prepared for tools@maruhouse.org, personal use only.

エクササイズ

コンパイラになってみよう

左のソースコードを、コンパイラになったつもりで見てみましょう。そして、コンパイルが正しく行えるかどうかを考えてください。コンパイルできないと判断した場合は、コンパイルできるようコードを修正してください。また、コンパイルできると判断した場合は実行結果がどのようになるか予測してください。

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

class InnerButton {

  JFrame frame;
  JButton b;

  public static void main(String [] args) {
    InnerButton gui = new InnerButton();
    gui.go();
  }

  public void go() {
    frame = new JFrame();
    frame.setDefaultCloseOperation (
                  JFrame.EXIT_ON_CLOSE);

    b = new JButton("A");
    b.addActionListener();

    frame.getContentPane().add(
                  BorderLayout.SOUTH, b);
    frame.setSize(200, 100);
    frame.setVisible(true);
  }

  class BListener extends ActionListener {
    public void actionPerformed(ActionEvent e) {
      if (b.getText().equals("A")) {
        b.setText("B");
      } else {
        b.setText("A");
      }
    }
  }
}
```

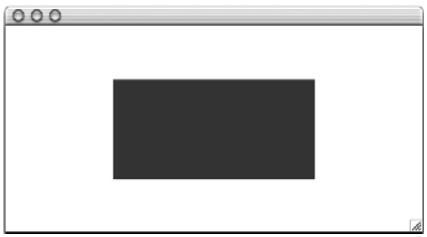

This PDF is prepared for tools@maruhouse.org, personal use only.

プールパズル

右のプログラムの欠落部分に、下のプールの中のコードを当てはめ、実行結果が以下に示すようなものになるようなプログラムを完成させてください。すべてのコードを使う必要はありません。同じコードを 2 回以上繰り返し使うこともできます。

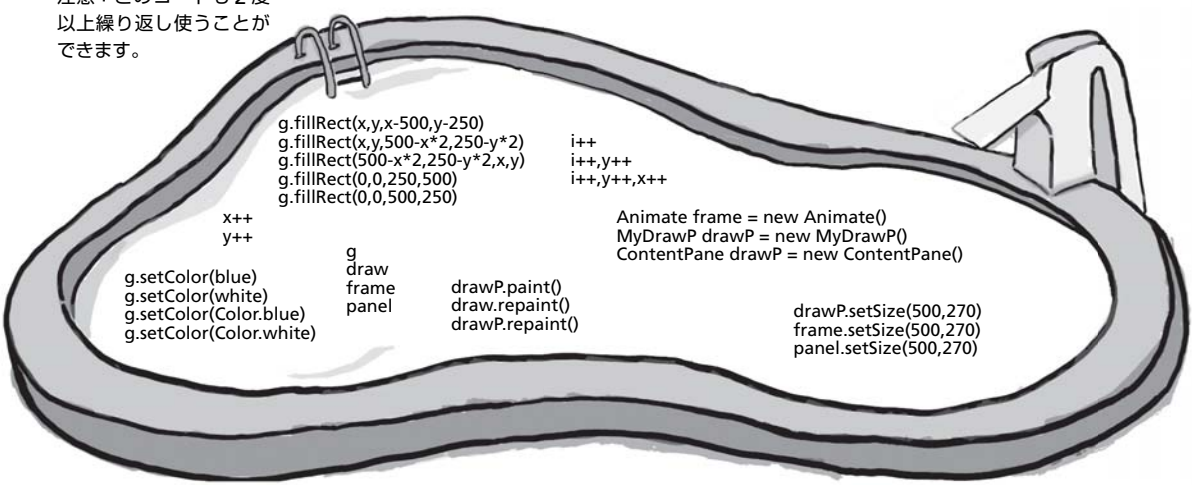
実行結果

プログラムを実行すると青い長方形が表示され、徐々に小さくなり最後には消えてしまいます。その後、画面は白くなります。



```
import javax.swing.*;
import java.awt.*;
public class Animate {
  int x = 1;
  int y = 1;
  public static void main(String [] args) {
    Animate gui = new Animate();
    gui.go();
  }
  public void go() {
    JFrame _________= new JFrame();
    frame.setDefaultCloseOperation(
                JFrame.EXIT_ON_CLOSE);
    ______________________________________;
    _________.getContentPane().add(drawP);
    ___________________________;
    _________.setVisible(true);
    for (int i=0; i<124; _________________) {
    ______________________;
    ______________________;
    try {
      Thread.sleep(50);
    } catch(Exception ex) { }
    }
  }
  class MyDrawP extends JPanel {
    public void paintComponent(
            (Graphics _________) {
      ______________________________________;
      ______________________________________;
      ______________________________________;
      ______________________________________;
    }
  }
}
```

注意：どのコードも 2 度以上繰り返し使うことができます。



This PDF is prepared for tools@maruhouse.org, personal use only.

エクササイズの答え

私は誰でしょう？

GUIの中ではじめに作成する	JFrame
イベントリスナーを作る際に必要	リスナーインターフェース
ボタンクリックに対応するイベントリスナーに必要なメソッド	actionPerformed()
JFrameのサイズを指定する	setSize()
プログラマがこのメソッドを呼び出すコードを書くことはない	paintComponent()
「ユーザーが何らかの操作をした」といった「できごと」を表す言葉	イベント
ほぼすべてがイベントソースになる	Swingコンポーネント
イベントリスナーに情報を渡す役割を果たす	イベントオブジェクト
addXxxListener()という名前のメソッドがあればそのクラスは_____であることがわかる	イベントソース
イベントリスナーを登録するためのメソッド	addActionListener()
グラフィックスを表示する際には必ず使用されるメソッド	paintComponent()
必ず特定のオブジェクトと結合される	内部オブジェクト
(Graphics g)の「g」は、このクラスのオブジェクト	Graphics2D
間接的にpaintComponent()を呼び出すのに使用できるメソッド	repaint()
Swingコンポーネントのほとんどが属するパッケージ	javax.swing

コンパイラになってみよう

このプログラムはこのままではコンパイルできない。太字部分を修正すれば、クリックする度にテキストが「A」から「B」、「B」から「A」へと変化するボタンのプログラムになる

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

class InnerButton {

  JFrame frame;
  JButton b;

  public static void main(String [] args) {
    InnerButton gui = new InnerButton();
    gui.go();
  }

  public void go() {
    frame = new JFrame();
    frame.setDefaultCloseOperation(
                    JFrame.EXIT_ON_CLOSE);
```

addActionListener()の引数には、ActionListenerインターフェースをインプリメントするクラスをTを指定する

```
    b = new JButton("A");
    b.addActionListener(new BListener());

    frame.getContentPane().add(
                    BorderLayout.SOUTH, b);
    frame.setSize(200, 100);
    frame.setVisible(true);
  }

  class BListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if (b.getText().equals("A")) {
        b.setText("B");
      } else {
        b.setText("A");
      }
    }
  }
}
```

ActionListenerはインターフェースなので、継承ではなくインプリメントする

This PDF is prepared for tools@maruhouse.org, personal use only.

プールパズル

```
import javax.swing.*;
import java.awt.*;
public class Animate {
  int x = 1;
  int y = 1;
  public static void main(String [] args) {
    Animate gui = new Animate();
    gui.go();
  }
  public void go() {
    JFrame frame = new JFrame();
    frame.setDefaultCloseOperation(
                JFrame.EXIT_ON_CLOSE);
    MyDrawP drawP = new MyDrawP();
    frame.getContentPane().add(drawP);
    frame.setSize(500,270);
    frame.setVisible(true);
    for (int i = 0; i < 124; i++,x++,y++) {
      x++;
      drawP.repaint();
      try {
        Thread.sleep(50);
      } catch(Exception ex) { }
    }
  }
  class MyDrawP extends JPanel {
    public void paintComponent(Graphics g) {
      g.setColor(Color.white);
      g.fillRect(0,0,500,250);
      g.setColor(Color.blue);
      g.fillRect(x,y,500-x*2,250-y*2);
    }
  }
}
```

青い長方形が徐々に小さくなり、消えていく

This PDF is prepared for tools@maruhouse.org, personal use only.

13章　Swingの基礎知識

Swingを利用する

Swingはさほど難しいものではありません。ウィジェットが画面上のどこに表示されるかを気にしなければ、の話ですが……。Swingを利用するコード自体は比較的簡単に書けるのですが、プログラムをコンパイルして実行してみると「あれ、なぜこれがこんなところに……」ということが起きやすいのです。実は、コードが簡単に書けるのも、ウィジェットが思いがけない位置に表示されるのも、その理由は「レイアウトマネージャ」というオブジェクトにあります。レイアウトマネージャは、JavaのGUIにおけるウィジェットのサイズや位置を決める役割を果たすオブジェクトです。レイアウトマネージャはプログラマの代わりにさまざまな仕事をしてくれるのですが、その結果がプログラマの望むものと異なる、ということが珍しくないのです。「ボタンを2つ作ったら、勝手にそれぞれ違ったサイズのものになってしまった」、「テキストフィールドの長さが驚くほど長くなった」、「ラベルの下に作りたかったテキストフィールドがラベルの隣にできてしまった」というようなことが頻繁に起こる、ということです。ただ、こうしたことは、少し手間をかければ防げます。この章では、レイアウトマネージャをはじめ、Swingに関する基本的なことがらについて解説します。ウィジェットを作り、思いどおりの場所に表示する方法、ウィジェットをプログラムの中で利用する方法などがわかるはずです。

This PDF is prepared for tools@maruhouse.org, personal use only.

Swingコンポーネント

「コンポーネント」と「ウィジェット」はほぼ同じ意味の言葉です。同じものを指すのですが、Javaでは、「コンポーネント」の方が正式な呼び方と言えるでしょう。コンポーネントはGUIの構成要素です。GUIを利用する際、ユーザーが目で見て、操作するものはすべてコンポーネントです。コンポーネントの例としては、テキストフィールド、ボタン、スクローラブルリスト、ラジオボタンなどがあげられます。こうしたコンポーネントはいずれもjavax.swing.JComponentを継承するオブジェクトです。

「ウィジェット」と「コンポーネント」はほぼ同じ意味。GUIのコンポーネントは、その多くがjavax.swing.JComponentを継承するオブジェクト

コンポーネントは他のコンポーネントの中に組み込むことができる

Swingの場合、コンポーネントのほとんどすべてに、他のコンポーネントを組み込むことができます。フレーム（ウィンドウ）やパネルに、ボタンやリストなど、ユーザーの操作を受け付ける「インタラクティブ」なコンポーネントを組み込む、といったことが可能なのです。そして、もちろん、逆に、ボタンの中にパネルを組み込むということも可能です（あまり有用性はありませんが）。

ただ、JFrame（フレーム）だけは例外ですが、コンポーネントがインタラクティブなものか、他のコンポーネントを組み込む単なる「容れ物」の役割を果たすだけのものか、という区別は恣意的なものです。たとえば、JPanel（パネル）は、一般に「容れ物」のコンポーネントとされますが、インタラクティブなコンポーネントにすることも不可能ではありません。イベントリスナーを作り、登録すれば、マウスやキーボードの操作に応じて動かすということもできるからです。

GUIを作成する手順（復習）

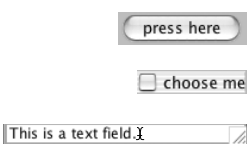
① ウィンドウ（フレーム）を作成する。
```
JFrame frame = new JFrame();
```

② ウィジェット（ボタン、テキストフィールドなど）を作成する。
```
JButton button = new JButton("click me");
```

③ ウィジェットをウィンドウに組み込む。
```
frame.getContentPane().add(BorderLayout.EAST, button);
```

④ 作成したGUIのサイズを指定し、画面に表示する。
```
frame.setSize(300, 300);
frame.setVisible(true);
```

インタラクティブなコンポーネント



JButton
JCheckBox
JTextField

他のコンポーネントの「容れ物」となるコンポーネント（バックグラウンドコンポーネント）

JFrame

JPanel

This PDF is prepared for tools@maruhouse.org, personal use only.

レイアウトマネージャ

レイアウトマネージャは、あるコンポーネントに他のコンポーネントが組み込まれる際、そのサイズや配置などを決定する役割を果たすオブジェクトです。たとえば、フレームにパネルが組み込まれ、パネルにボタンが組み込まれる場合には、フレームのレイアウトマネージャがパネルのサイズと配置を、パネルのレイアウトマネージャがボタンのサイズと配置を決定するというわけです。ボタンのように、一般に他のコンポーネントが組み込まれることのないコンポーネントは、レイアウトマネージャを必要としません。

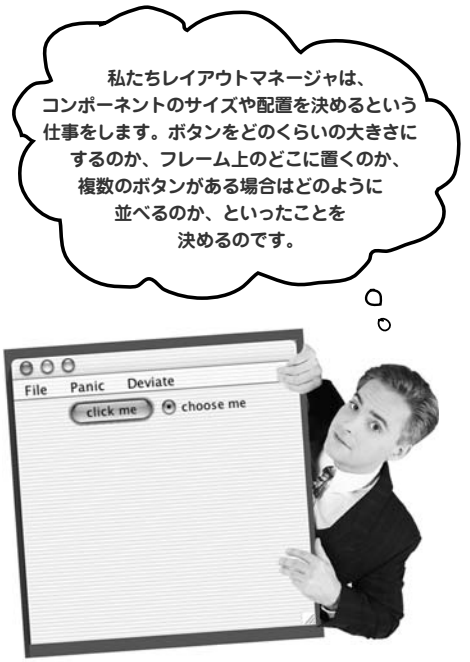
パネルに組み込まれたコンポーネントのサイズや配置を決めるのは、パネルのレイアウトマネージャの役目です。この点は、組み込まれたコンポーネントが自らのレイアウトマネージャを持っていたとしても同じです。組み込まれたコンポーネントのレイアウトマネージャは、そのコンポーネントにさらに他のコンポーネントが組み込まれた場合にのみ機能します。

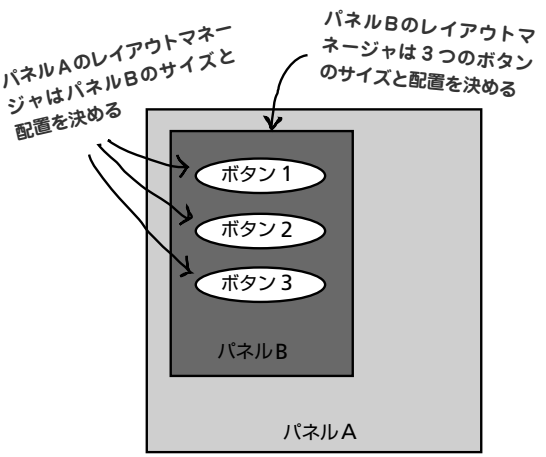
コンポーネントに他のコンポーネントを組み込むには、add() というメソッドを使用します。たとえば、ボタンをパネルに組み込む際にはだいたい以下のようなコードを書くことになります。

```
myPanel.add(button);
```

レイアウトマネージャにはいくつかの種類があり、持つべきレイアウトマネージャはコンポーネントによって違ってきます。レイアウトマネージャの種類が異なれば、コンポーネントのレイアウトの仕方も当然異なります。たとえば、組み込まれたコンポーネントのサイズを自動的にすべて同じにするもの、コンポーネントを組み込まれた順に並べるもの、コンポーネントごとにサイズを選ぶことができるもの……といった具合です。また、コンポーネントを縦に並べるレイアウトマネージャもあります。以下のコードは、パネルBにボタンを3つ組み込み、パネルBをパネルAに組み込むというものです。

```
JPanel panelA = new JPanel();
JPanel panelB = new JPanel();
panelB.add(new JButton("button 1"));
panelB.add(new JButton("button 2"));
panelB.add(new JButton("button 3"));
panelA.add(panelB);
```





パネルAのレイアウトマネージャは3つのボタンには一切関知しない。関知するのは直接組み込まれているコンポーネントのみ。間接的に組み込まれているコンポーネントのサイズや配置を決めることはない

This PDF is prepared for tools@maruhouse.org, personal use only.

レイアウトマネージャの機能

レイアウトマネージャは、種類ごとに異なったルール（コンポーネントを組み込まれた順に並べる、サイズをすべて同じにする、縦に並べる、など。このルールを「ポリシー」と呼ぶこともある）に基づいて機能します。ただ、コンポーネント側の「要望」を完全に無視してレイアウトをするわけではありません。一般に、レイアウトマネージャがバックグラウンドコンポーネントのレイアウトを行う際には次のような手順を踏むことになります。

レイアウトの手順

① パネルが作成され、ボタンが組み込まれる。

② パネルのレイアウトマネージャが、サイズに関するボタン側の要望を確認する。

③ パネルのレイアウトマネージャが、自らのレイアウトルールに基づき、ボタン側の要望をどの程度受け入れるか（全部受け入れるのか、一部だけを受け入れるのか、それともまったく受け入れないのか）を決定する。

④ パネルがフレームに組み込まれる。

⑤ フレームのレイアウトマネージャが、サイズに関するパネル側の要望を確認する。

⑥ フレームのレイアウトマネージャが、自らのレイアウトルールに基づき、パネル側の要望をどの程度受け入れるか（全部受け入れるのか、一部だけを受け入れるのか、それともまったく受け入れないのか）を決定する。

レイアウトのルールはレイアウトマネージャごとに異なる

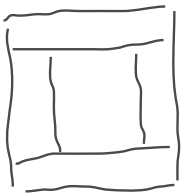
コンポーネント側の「要望」のうち何をどの程度受け入れるかはレイアウトマネージャごとに違います。サイズに関する要望を全面的に受け入れるものもあれば、一部のみを受け入れるものもあるのです。たとえば、「30 × 50 ピクセル」というのがボタンの希望のサイズであった場合、それをそのまま反映させるレイアウトマネージャもあれば、縦のサイズ 30 だけを受け入れ、横のサイズはパネルの幅に合わせるというレイアウトマネージャもあります。また、中には、ボタンが複数組み込まれる場合、すべてのボタンのサイズを最も大きい要望サイズに揃えてしまう、というレイアウトマネージャもあります。レイアウトマネージャが適用するルールの中には複雑なものもありますが、いったんルールを把握してしまえば、多くの場合、慎重に考えればどのような結果が得られるかは事前に予測できます。

This PDF is prepared for tools@maruhouse.org, personal use only.

レイアウトマネージャの分類

BorderLayout

BorderLayoutは、バックグラウンドコンポーネントを 5 つの領域に分割するレイアウトマネージャです。各領域に組み込めるコンポーネントは 1 つだけです。このレイアウトマネージャを使用した場合、コンポーネントのサイズが要望どおりになることは少ないと言えます。BorderLayoutはフレームのデフォルトレイアウトマネージャです。



各領域にコンポーネントを 1 つずつ配置

FlowLayout

FlowLayoutは、ワードプロセッサに似た機能を持つレイアウトマネージャです。ただ、扱うのが文字ではなくコンポーネントであるという点はワープロとは異なります。このレイアウトマネージャを使用した場合、コンポーネントのサイズはすべて要望どおりになります。また、コンポーネントは組み込まれた順に左から右に並べられていきます。右端に到達すれば、自動的に「改行」が行われます。FlowLayoutはパネルのデフォルトレイアウトマネージャです。

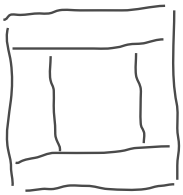
コンポーネントを左から右に順に配置。右端に到達したら自動的に改行

BoxLayout

BoxLayoutはコンポーネントのサイズが要望どおりになるという点、コンポーネントが組み込まれた順に並べられていくという点ではFlowLayoutに似ています。ただ、FlowLayoutと異なるのは、コンポーネントを縦あるいは横に並べることができるという点です。ある「行」がコンポーネントで埋まっていなくても、強制的に次の行の頭にコンポーネントを配置することが可能なのです。つまり、意図的に「改行」ができるということです。

コンポーネントを上から下に順に配置

This PDF is prepared for tools@maruhouse.org, personal use only.



BorderLayout はコンポーネントを east、west、north、south、center という 5 つの領域に分ける

ボタンを east 領域に組み込む

```
import javax.swing.*;
import java.awt.*;

public class Button1 {
  public static void main(String [] args) {
    Button1 gui = new Button1();
    gui.go();
  }

  public void go() {
    JFrame frame = new JFrame();
    JButton button = new JButton("click me");
    frame.getContentPane().add(BorderLayout.EAST,button);
    frame.setSize(200, 200);
    frame.setVisible(true);
  }
}
```

BorderLayoutはjava.awtパッケージのクラス

組み込む領域の指定

頭の体操

BorderLayoutを使ってボタンを組み込んだら、サイズが右の図のようになりました。なぜこのサイズになったと考えられますか。

BorderLayoutを使う場合、注意しなければならないことは何ですか。

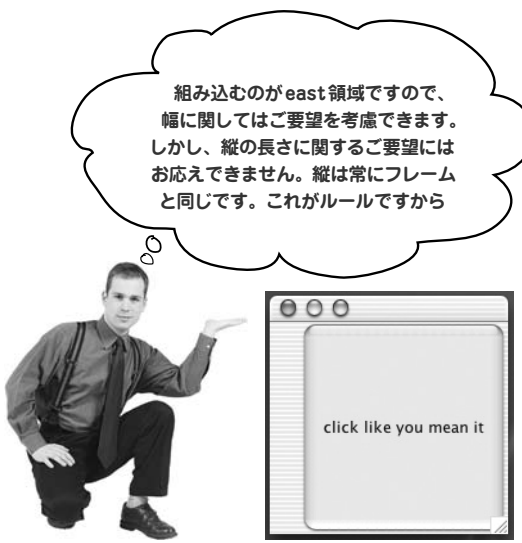
ボタンを、縦、あるいは横にさらに大きくすることはできますか。

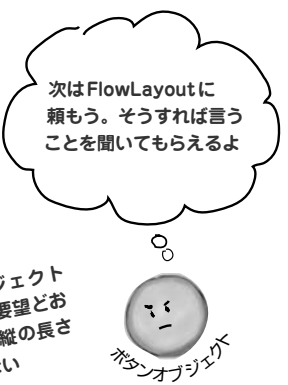
This PDF is prepared for tools@maruhouse.org, personal use only.

ボタンに表示される文字の数を増やすとどうなるか

ボタン上に表示されるテキストにのみ変更を加える

```
public void go() {
  JFrame frame = new JFrame();
  JButton button = new JButton("click like you mean it");
  frame.getContentPane().add(BorderLayout.EAST, button);
  frame.setSize(200,200);
  frame.setVisible(true);
}
```





Buttonオブジェクトの横の長さは要望どおりになるが、縦の長さはそうではない

This PDF is prepared for tools@maruhouse.org, personal use only.

north 領域にボタンを組み込む

```
public void go() {
  JFrame frame = new JFrame();
  JButton button = new JButton("there is no spoon...");
  frame.getContentPane().add(BorderLayout.NORTH ,button);
  frame.setSize(200,200);
  frame.setVisible(true);
}
```

ボタンの縦の長さは要望どおりになるが、横の長さは自動的にフレームと同じになる

ボタンの縦の長さをのばす

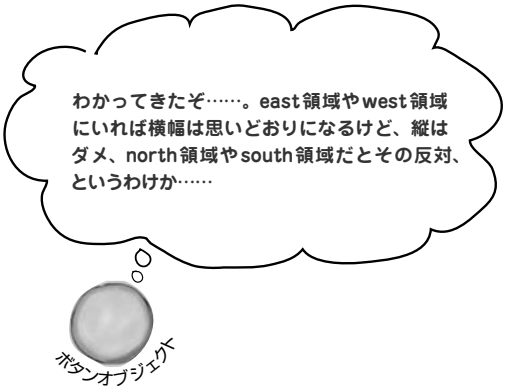
north 領域に組み込んだボタンの横の長さは変えられませんが、縦の長さは、テキストのフォントを大きくするとのばすことができます。大きくなったテキストが表示できるよう、ボタンは縦にのびるのです。

```
public void go() {
  JFrame frame = new JFrame();
  JButton button = new JButton("Click This!");
  Font bigFont = new Font("serif", Font.BOLD, 28);
  button.setFont(bigFont);
  frame.getContentPane().add(BorderLayout.NORTH ,button);
  frame.setSize(200,200);
  frame.setVisible(true);
}
```

フォントを大きくすることでボタンの縦の長さを強制的にのばす

ボタンの横の長さは変わらないが、縦の長さはのびる。north 領域では、縦の長さのみ調整可能

This PDF is prepared for tools@maruhouse.org, personal use only.



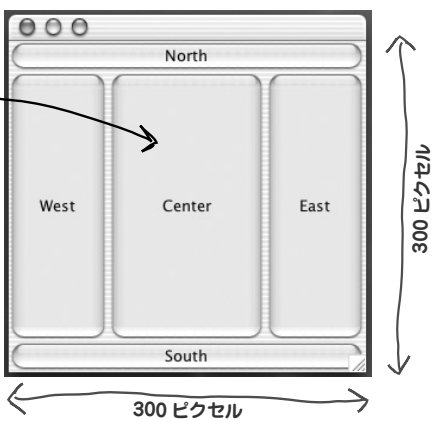
果たしてcenterの場合は？

center領域のコンポーネントは他の領域のコンポーネントが使っていないスペースをすべて使える（ただし例外もある。詳細は後述）

```
public void go() {
  JFrame frame = new JFrame();

  JButton east = new JButton("East");
  JButton west = new JButton("West");
  JButton north = new JButton("North");
  JButton south = new JButton("South");
  JButton center = new JButton("Center");

  frame.getContentPane().add(BorderLayout.EAST, east);
  frame.getContentPane().add(BorderLayout.WEST, west);
  frame.getContentPane().add(BorderLayout.NORTH, north);
  frame.getContentPane().add(BorderLayout.SOUTH, south);
  frame.getContentPane().add(BorderLayout.CENTER, center);
  frame.setSize(300,300);
  frame.setVisible(true);
}
```

center領域のコンポーネントは残っているスペースをすべて使える（縦、横がフレームより長くなることはない。上記のコードの場合なら 300 × 300 より大きくなることはないということ）

east、west領域のコンポーネントは、横の長さを必要に応じてのばせる。north、south領域のコンポーネントは、縦の長さを必要に応じてのばせる



north、south領域、east、west領域に同時にコンポーネントを組み込むと、north、south領域のコンポーネントの横の長さがフレームと同じになり、east、west領域のコンポーネントの縦の長さをその分削ることになる

This PDF is prepared for tools@maruhouse.org, personal use only.

FlowLayoutではコンポーネントが
組み込まれた順に並べられる
最初に組み込まれたものが
左上角に置かれ、その後、配置される場所は
左から右に、上から下に移動していく

パネルをフレームに組み込む

パネル（JPanel）のデフォルトレイアウトマネージャはFlowLayoutなのですが、パネルをフレームに組み込む際、パネル自体の配置はBorderLayoutによって決定されます。FlowLayoutによって配置が決められるのは、パネルに（panel.add(aComponent)というメソッドによって）組み込まれるコンポーネントです。以下に示すのは、フレームのeast領域に空のパネルを組み込むためのコードです。パネルにコンポーネントを組み込むコードは次のページ以降に示します。

パネルにはまだ何も組み込まれていないので、east領域の幅は小さくなっている

```
import javax.swing.*;
import java.awt.*;

public class Panel1 {

  public static void main(String [] args) {
    Panel1 gui = new Panel1();
    gui.go();
  }

  public void go() {
    JFrame frame = new JFrame();
    JPanel panel = new JPanel();
    panel.setBackground(Color.darkGray);
    frame.getContentPane().add(BorderLayout.EAST, panel);
    frame.setSize(200,200);
    frame.setVisible(true);
  }
}
```

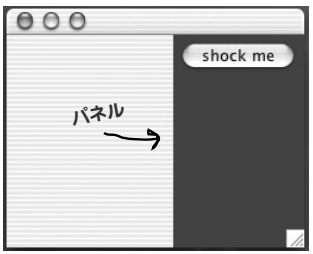
パネルの色をダークグレーにする。どこにパネルが組み込まれているかがすぐわかるようにするため

This PDF is prepared for tools@maruhouse.org, personal use only.

パネルにボタンを組み込む

```
public void go() {
  JFrame frame = new JFrame();
  JPanel panel = new JPanel();
  panel.setBackground(Color.darkGray);

  JButton button = new JButton("shock me");

  panel.add(button);
  frame.getContentPane().add(BorderLayout.EAST, panel);

  frame.setSize(250,200);
  frame.setVisible(true);
}
```

パネルにボタンを組み込み、さらにパネルをフレームに組み込む。パネルのデフォルトレイアウトマネージャはFlowLayoutだが、フレームのデフォルトレイアウトマネージャはBorderLayout



ボタンを組み込むとパネルの幅が広がる。レイアウトマネージャがFlowLayoutなので、ボタンのパネル上でのサイズには、縦、横ともに制約がない

This PDF is prepared for tools@maruhouse.org, personal use only.

パネルにボタンを 2 つ組み込むとどうなるか

```
public void go() {
  JFrame frame = new JFrame();
  JPanel panel = new JPanel();
  panel.setBackground(Color.darkGray);

  JButton button = new JButton("shock me");
  JButton buttonTwo = new JButton("bliss");

  panel.add(button);
  panel.add(buttonTwo);

  frame.getContentPane().add(BorderLayout.EAST, panel);
  frame.setSize(250,200);
  frame.setVisible(true);
}
```

ボタンを 2 つ作る

2 つのボタンをパネルに組み込む

作ろうとした画面

2 つのボタンが縦に重なって表示されるはず……

実際に得られた画面

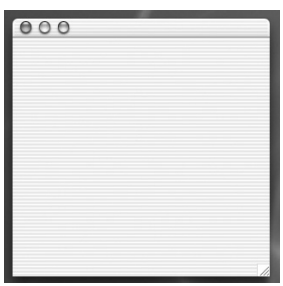
2 つのボタンが横に並び、それに合わせてパネルの幅が広がった

「bliss」ボタンの方が「shock me」ボタンより小さいことに注目……。FlowLayout の場合、サイズは必要に応じて変わるが、配置は強制的に決められてしまう

自分で考えてみよう

上記のコードを以下のように書き直した場合、GUI はどのように変わるでしょうか

```
JButton button = new JButton("shock me");
JButton buttonTwo = new JButton("bliss");
JButton buttonThree = new JButton("huh?");
panel.add(button);
panel.add(buttonTwo);
panel.add(buttonThree);
```



左に自分の予想を書き込んでください（その後、実際にプログラムを動かして確かめてください）

This PDF is prepared for tools@maruhouse.org, personal use only.

BoxLayoutで解決

BoxLayoutの場合は、
コンポーネントを縦に
重ねることができる

BoxLayoutでも、基本的にコンポーネントは組み込まれた順に並んでいくのですが、FlowLayoutとは異なり、「行」がコンポーネントで埋まっていなくても、強制的に「改行」することができます。つまり、コンポーネントを縦に重ねることもできるということです。

パネルのレイアウトマネージャをデフォルトの
FlowLayoutからBoxLayoutに変える

```
public void go() {
  JFrame frame = new JFrame();
  JPanel panel = new JPanel();
  panel.setBackground(Color.darkGray);

  panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));

  JButton button = new JButton("shock me");
  JButton buttonTwo = new JButton("bliss");
  panel.add(button);
  panel.add(buttonTwo);
  frame.getContentPane().add(BorderLayout.EAST, panel);
  frame.setSize(250,200);
  frame.setVisible(true);
}
```

レイアウトマネージャをBoxLayoutに変更

BoxLayoutのコンストラクタの引数には、レイアウトマネージャを適用するコンポーネントの名前と、コンポーネントを並べる方向を指定する（Y_AXISは縦に並べるという意味）

上記のコードを使うと、ボタンが縦に並ぶため、それに合わせてパネルの幅は狭くなる。幅は、大きい方の「shock me」ボタンに合わされる

This PDF is prepared for tools@maruhouse.org, personal use only.

素朴な疑問に答えます

Q：フレームに直接コンポーネントを組み込めないのはなぜですか。パネルには直接組み込めるようですが。

A：JFrameは、JavaとOSの間の「緩衝材」のような役割を果たす特殊なコンポーネントです。JFrame以外のSwingコンポーネントはすべてJava環境の中だけで機能するのですが、JFrameだけは、画面表示を行うためにその外のOSとコミュニケーションをとらねばなりません。JFrameにコンポーネントを組み込む際は、まず「コンテンツペイン」と呼ばれるコンポーネントを組み込みます。直接的には、このコンテンツペインに他のコンポーネントを組み込むことになります。コンテンツペインはJava環境の中だけで機能するコンポーネントです。JFrameを「窓枠」のようなもの、コンテンツペインを「窓ガラス」のようなものと考えてもいいでしょう。以下のようなコードを使えば、独自に作ったパネルオブジェクトをコンテンツペインとして指定することも可能です。

```
myFrame.setContentPane(myPanel);
```

Q：フレームのレイアウトマネージャを変更することはできますか。たとえば、BorderLayoutの代わりにFlowLayoutを使うということはできるのですか。

A：できます。最も簡単なのは、FlowLayoutをレイアウトマネージャとするパネルを作り、そのパネルを先述の方法でコンテンツペインとして指定する、という方法です。

Q：コンポーネントのサイズを自分の思いどおりに設定したい時はどうするのですか。各コンポーネントにsetSize()メソッドがあるのですか。

A：確かにsetSize()メソッドはあります。ただ、レイアウトマネージャはこのメソッドによる設定を無視します。ここで注意すべきなのは、「プログラマの望むサイズ」と「コンポーネント自身が望むサイズ」は同じではないということです。コンポーネント自身が望むサイズは、(ボタンに表示すべきテキストの文字数などの)いくつかの条件によって自動的に決まります。そのサイズを知るため、レイアウトマネージャは各コンポーネントのgetPreferredSize()メソッドを呼び出します。しかし、setSize()メソッドが呼び出されたことには関知しないのです。

Q：レイアウトマネージャの機能を停止して自分の思いどおりにコンポーネントを配置するということはできますか。

A：できます。その場合は、コンポーネントごとにsetLayout(null)というメソッドを呼び出すことになります。そして、コンポーネントの画面上での位置、大きさを正確に指定するコードを書くのです。ただ、後々のことを考えればレイアウトマネージャを使っておいた方が何かと便利であることは確かです。

重要ポイント

- レイアウトマネージャは、あるコンポーネントに他のコンポーネントが組み込まれる際、そのサイズや配置などを決定する役割を果たす。
- コンポーネントの配置、サイズは、そのコンポーネント自身のレイアウトマネージャではなく、組み込み先コンポーネントのレイアウトマネージャによって決定される。
- レイアウトマネージャは、コンポーネントのレイアウトを決定する際、サイズに関するコンポーネント側の要望を確認する。レイアウトマネージャは自らのルールに基づき、その要望をどの程度反映させるかを判断する。要望は場合によってすべて反映される場合もあれば、一部のみ反映される場合、まったく反映されない場合がある。
- レイアウトマネージャとして「BorderLayout」を使用した場合、バックグラウンドコンポーネント(組み込み先コンポーネント)は5つの領域に分割される。1つの領域につき、組み込めるコンポーネントは1つ。また、コンポーネント組み込み時には以下のように、「どの領域に組み込みたいか」を明確に指定する必要がある。
  `add(BorderLayout.EAST,panel);`
- BorderLayoutの場合、north、south領域では、縦の長さはコンポーネント側の要望に応じて変わるが、横の長さは自動的に決められる。east領域、west領域ではその逆に、横の長さにのみコンポーネント側の要望が反映され、縦の長さには反映されない。center領域のコンポーネントは、他の領域が使った残りのスペースを使うことになる(pack()を使用した場合は例外)。
- pack()メソッドを使用すると、まずcenter領域のコンポーネントのサイズを「要望どおり」にし、それに基づいて他の領域(そしてフレーム全体)のサイズを決めるということができる。
- FlowLayoutでは、コンポーネントは組み込まれた順に左から右に並べられる。右端に到達した場合は、自動的に「改行」が行われる。
- FlowLayoutでは、コンポーネントのサイズが縦、横ともコンポーネントの要望どおりになる。
- BoxLayoutでは、FlowLayoutと同様、コンポーネントが組み込まれた順に並べられていく。また、コンポーネントのサイズは要望どおりになる、ただ、FlowLayoutと異なるのは、コンポーネントを縦あるいは横に並べることができるという点。「行」がコンポーネントで埋まっていなくても、強制的に次の行の頭にコンポーネントを配置することが可能。
- BorderLayoutはフレームの、FlowLayoutはパネルのデフォルトレイアウトマネージャ。
- パネルのレイアウトマネージャとして、FlowLayout以外のものを使用する場合は、パネルのsetLayout()メソッドを使用する。

This PDF is prepared for tools@maruhouse.org, personal use only.

さまざまなSwingコンポーネント

次に、JTextField、JTextArea、JCheckBox、JListといった一般的なSwingコンポーネントについて見てみましょう。ここでは、概要をごく簡単に解説するにとどめます。

JTextField

「20」は、「20 列」という意味。「20 ピクセル」ではない。この引数は、テキストフィールドの横の長さを指定したもの（この指定は単なるプログラマの希望なので、必ず反映されるわけではない）

コンストラクタ

```
JTextField field = new JTextField(20);

JTextField field = new JTextField("Your name");
```

コンポーネントの使い方

① テキストフィールドのテキストを抽出する。

```
System.out.println(field.getText());
```

② テキストフィールドにテキストを入力する。

```
field.setText("whatever");
field.setText("");
```

このように書くとテキストを消去できる

③ ユーザーがreturn（Enter）キーを押した時に発生するActionEventを処理するイベントリスナーを登録する。

必要であれば、KeyEventに対応するイベントリスナーも登録できる

```
field.addActionListener(myActionListener);
```

④ フィールド中のテキストを選択（ハイライト表示）する。

```
field.selectAll();
```

⑤ テキストフィールドにカーソルを表示する（これによって新たなテキストが入力できるようになる）。

```
field.requestFocus();
```


This PDF is prepared for tools@maruhouse.org, personal use only.

JTextArea

JTextAreaは、JTextFieldとは異なり、複数行のテキストを入力、表示できるコンポーネントです。JTextAreaのオブジェクトを作成する際には、スクロールバーやラインラッピング（自動改行）などに関しての設定が必要になります。また、スクロール機能を持たせるためには、「スクロールペイン」というコンポーネントが必要になります。

コンストラクタ

```
JTextArea text = new JTextArea(10,20);
```

「10」は「10行」を意味する。テキスト領域の縦の長さを指定したもの（単なるプログラマの希望なので、必ず反映されるわけではない）

「20」は「20列」を意味する。テキスト領域の横の長さを指定したもの（これも単なる希望なので、必ず反映されるわけではない）

コンポーネントの使い方

① 縦のスクロールバーのみを持つテキスト領域を作成する場合のコードは以下のとおり。

```
JScrollPane scroller = new JScrollPane(text);
text.setLineWrap(true);

scroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
scroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

panel.add(scroller);
```

スクロールペイン（JScrollPaneオブジェクト）を作成する。引数には、スクロール機能を持たせるJTextAreaを指定する。これによってテキスト領域がスクロールペインに組み込まれる

ラインラッピング機能をオンにする

この引数の指定によって、縦のスクロールバーのみが作られるようになる

重要！ このコードによって、テキスト領域の組み込まれたスクロールペインをパネルに組み込む。テキスト領域をパネルに直接組み込むことはできない

② テキスト領域中のテキストを別のものに変える。

```
text.setText("Not all who are lost are wandering");
```

③ テキストを追加する

```
text.append("button clicked");
```

④ テキストを選択（ハイライト表示）する。

```
text.selectAll();
```

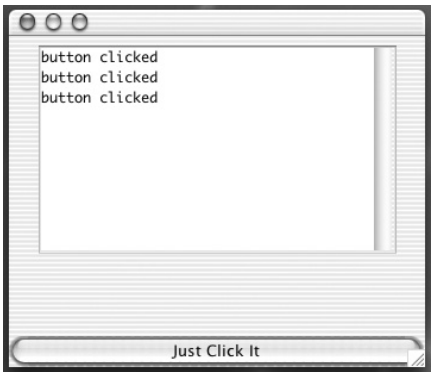
⑤ カーソルを表示する（これによって新たなテキストが入力できるようになる）。

```
text.requestFocus();
```

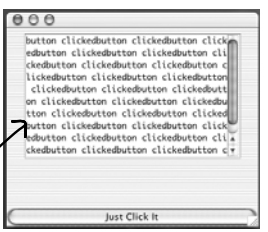

This PDF is prepared for tools@maruhouse.org, personal use only.

JTextArea の使用例

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class TextArea1 implements ActionListener {

  JTextArea text;

  public static void main (String [] args) {
    TextArea1 gui = new TextArea1();
    gui.go();
  }

  public void go() {
    JFrame frame = new JFrame();
    JPanel panel = new JPanel();
    JButton button = new JButton("Just Click It");
    button.addActionListener(this);
    text = new JTextArea(10,20);
    text.setLineWrap(true);

    JScrollPane scroller = new JScrollPane(text);
    scroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
    scroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

    panel.add(scroller);

    frame.getContentPane().add(BorderLayout.CENTER,panel);
    frame.getContentPane().add(BorderLayout.SOUTH,button);

    frame.setSize(350,300);
    frame.setVisible(true);
  }

  public void actionPerformed(ActionEvent ev) {
    text.append("button clicked \n ");
  }
}
```

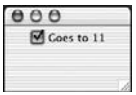


このように書けば、ボタンをクリックするたびに改行が行われる。さもなければ、右の図のようにテキストがつながって表示されることになる



This PDF is prepared for tools@maruhouse.org, personal use only.

JCheckBox



コンストラクタ

```
JCheckBox check = new JCheckBox("Goes to 11");
```

コンポーネントの使い方

① チェックが行われた時、解除された時に発生するItemEventに対応するリスナーを登録する。

```
check.addItemListener(this);
```

② イベントを処理する（また、チェックが行われているかどうかを確認する）。

```
public void itemStateChanged(ItemEvent ev) {
  String onOrOff = "off";
  if (check.isSelected()) onOrOff = "on";
  System.out.println("Check box is " + onOrOff);
}
```

③ チェックを行う、あるいはチェックを解除する。

```
check.setSelected(true);
check.setSelected(false);
```

素朴な疑問に答えます

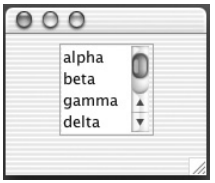
Q：レイアウトマネージャはトラブルの元になるばかりで役には立たないのではないでしょうか。これほど面倒なものなら、コンポーネントの配置やサイズを逐一指定する方がいいように思うのですが。

A：レイアウトマネージャを使って、コンポーネントの配置やサイズを完全に思いどおりにするのは確かに困難です。しかし、本当にレイアウトマネージャが役立たないかを判断するには、その役割についてもう一度よく考えてみる必要があるでしょう。コンポーネントの配置、サイズを決めるという仕事は一見、簡単そうに思えますが、状況によっては非常に難しくなることもあります。よほど注意しないと、コンポーネントどうしが重なり合ってしまう、組み込み先のコンポーネントからはみ出してしまう、などの問題が発生します。レイアウトマネージャを使用すれば、そのような問題が起きないための調整が常に自動的に行われるのです。プログラマが自分でそれをすることも不可能ではありませんが、小さいスペースにいくつものコンポーネントを組み込む場合には大変な手間がかかるでしょう。また、仮に正しく配置ができたとしても、プログラムを実行するプラットフォームが変わると、問題が起きる可能性があります。なぜでしょうか。それは、コンポーネントの形や大きさが、プラットフォームごとに少しずつ異なるからです。特に、各プラットフォームの「ネイティブ」のルック&フィールを使用するようなプログラムではその違いが大きくなります。「ボタンの角が、あるプラットフォームでは尖っているが、あるプラットフォームでは丸みを帯びている」というような些細な違いでも、コンポーネントの配列が乱れたり、コンポーネントどうしが重なり合ったり、といったことが起きてしまう恐れがあるのです。

レイアウトマネージャを使わないと困ることは他にもあります。たとえば、ユーザーがウィンドウのサイズを変えてしまったとしたらどうでしょうか。また、GUIが、表示されるコンポーネントが次々に変わっていくようなダイナミックなものだったとしたらどうでしょうか。ウィンドウのサイズが変化するたび、GUIを構成するコンポーネントが入れ替わるたびに、それに対応して適切なレイアウトをするようなプログラムを書く、というのが至難の業であるというのはすぐにわかるはずです。

This PDF is prepared for tools@maruhouse.org, personal use only.

JList



JListのコンストラクタの引数としては、あらゆるオブジェクトの配列を指定できる。オブジェクトの型はStringでなくてもかまわないが、配列の要素はリスト上では文字列として表示される

コンストラクタ

```
String [] listEntries = {"alpha", "beta", "gamma", "delta",
                         "epsilon", "zeta", "eta", "theta"};

list = new JList(listEntries);
```

コンポーネントの使い方

JTextAreaの場合と同じように、JScrollPaneにJListを組み込み、JScrollPaneをパネルに組み込む（JListをパネルに直接組み込むわけではない）

① 縦のスクロールバーを持たせる。

```
JScrollPane scroller = new JScrollPane(list);
scroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
scroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

panel.add(scroller);
```

② 画面に同時に何行表示するか（スクロールせずに何行見ることができるか）を指定する。

```
list.setVisibleRowCount(4);
```

③ ユーザーが同時に選択できる行を 1 行に制限する。

```
list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
```

④ 選択が行われた時に発生する SelectionEvent に対応するイベントリスナーを登録する。

```
list.addListSelectionListener(this);
```

⑤ イベントを処理する（またリスト中のどの項目が選択されているかを調べる）。

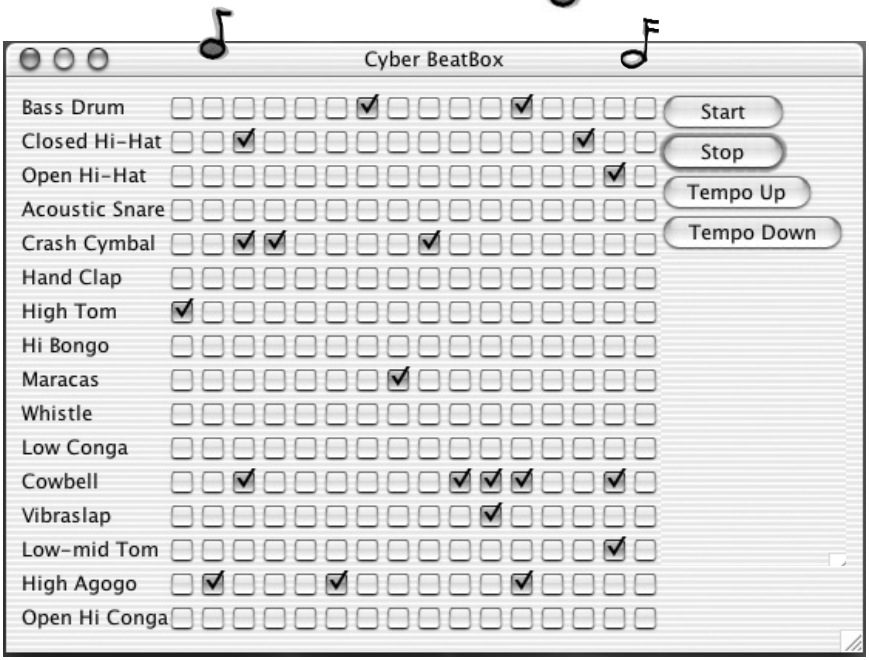
```
public void valueChanged(ListSelectionEvent lse) {
  if(!lse.getValueIsAdjusting()) {
    String selection = (String)list.getSelectedValue();
    System.out.println(selection);
  }
}
```

このif文がないとまだ選択が完了していない段階でイベント処理が行われることになる

getSelectedValue()が戻すのはオブジェクト。リストに格納されるのはStringオブジェクトとは限らない

This PDF is prepared for tools@maruhouse.org, personal use only.

プログラム・キッチン



この「プログラム・キッチン」のページは、これまでの章と同様、興味がない人は飛ばして読んでもかまいません。ここでは、GUIを備えたBeatBoxプログラムの作り方を解説します。作成したドラムパターンを保存したり、保存したパターンを再生したり、といったことがらについては14章で学ぶことになります。また、BeatBoxにネットワーク経由でドラムパターンをやりとりできるチャットクライアントの機能を持たせる方法については、15章で学びます。

This PDF is prepared for tools@maruhouse.org, personal use only.

BeatBoxプログラムのコード

では、BeatBoxプログラムのコードを見てみましょう。プログラムを構成するすべてのコードを漏らさず載せますので、よく内容を確認してください。コードには詳しい注釈がついていますが、一応、概要を以下にまとめておきます。

① 256のチェックボックス（JCheckBox）、16のラベル（JLabel）、4つのボタン（JButton）を備えたGUIを作成する。チェックボックスはいずれも初期状態ではチェックが解除されるようにする。ラベルは楽器名を表示するためのもの。

② 4つのボタンに対応するイベントリスナー（ActionListenerのインプリメンテーション）を登録する。ドラムパターンをダイナミックに変更する機能（演奏中にチェックボックスをチェックしてパターンを変更する機能）は持たせないので、チェックボックスに対応するイベントリスナーは必要ない。個々のチェックボックスがチェックされているかどうかの確認は、ユーザーが「Start」ボタンをクリックした時に行う。この確認の結果を基にTrackオブジェクトを作成することになる。

③ Sequencerオブジェクトの取得、Sequenceオブジェクトの作成、Trackオブジェクトの作成など、MIDIに関連する処理を行う。この処理には、Java 5.0で新たに追加された、SequencerインターフェースのsetLoopCount()メソッドを使用する。このメソッドを使用すると、ループの反復回数が指定できる。テンポの調整、ループ中のテンポ維持には「テンポ係数」を使用する。

④ ユーザーが「Start」をクリックした時に演奏が始まるようにする。「Start」ボタンのクリックに対応するイベントハンドラでは、buildTrackAndStart()メソッドが呼び出される。このメソッドでは、256個のチェックボックスの状態を逐一確認する（確認は楽器ごとに順に行う。1つの楽器の16個のチェックボックスが終わったら、次の楽器、という具合に進む）。確認の結果得られた情報はTrackオブジェクトの作成（これには、すでに紹介したmakeEvent()というメソッドを使用する）に使う。Trackオブジェクトができたら、Sequencerクラスのstart()メソッドで演奏をスタートさせる。上記の仕組みにより、ユーザーが「Stop」をクリックするまでは同じパターンが繰り返し演奏される。

This PDF is prepared for tools@maruhouse.org, personal use only.

```
import java.awt.*;
import javax.swing.*;
import javax.sound.midi.*;
import java.util.*;
import java.awt.event.*;

public class BeatBox {

  JPanel mainPanel;
  ArrayList<JCheckBox> checkboxList;
  Sequencer sequencer;
  Sequence sequence;
  Track track;
  JFrame theFrame;

  String [] instrumentNames = {"Bass Drum", "Closed Hi-Hat", "Open Hi-Hat", "Acoustic
  Snare", "Crash Cymbal", "Hand Clap", "High Tom", "Hi Bongo", "Maracas", "Whistle",
  "Low Conga", "Cowbell", "Vibraslap", "Low-mid Tom", "High Agogo", "Open Hi Conga"};

  int [] instruments = {35,42,46,38,49,39,50,60,70,72,64,56,58,47,67,63};

  public static void main (String [] args) {
    new BeatBox().buildGUI();
  }

  public void buildGUI() {
    theFrame = new JFrame("Cyber BeatBox");
    theFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    BorderLayout layout = new BorderLayout();
    JPanel background = new JPanel(layout);
    background.setBorder(BorderFactory.createEmptyBorder(10,10,10,10));

    checkboxList = new ArrayList<JCheckBox>();
    Box buttonBox = new Box(BoxLayout.Y_AXIS);

    JButton start = new JButton("Start");
    start.addActionListener(new MyStartListener());
    buttonBox.add(start);

    JButton stop = new JButton("Stop");
    stop.addActionListener(new MyStopListener());
    buttonBox.add(stop);

    JButton upTempo = new JButton("Tempo Up");
    upTempo.addActionListener(new MyUpTempoListener());
    buttonBox.add(upTempo);

    JButton downTempo = new JButton("Tempo Down");
    downTempo.addActionListener(new MyDownTempoListener());
```

チェックボックスオブジェクトをArrayListに格納する

ラベルに表示するテキスト（楽器名）を納めたStringの配列を作る

個々の数字は、ドラムを構成する楽器に対応する。ピアノなどの楽器であれば、この数字はそれぞれ「音程」に対応するが、ドラムの場合は、それとは違い、「35」が「ベースドラム」、「42」が「クローズドハイハット」というような対応関係になる

このように書くと、パネルの端の部分と、コンポーネントを組み込む部分の間に「縁取り」を作ることができる。完全にデザイン上の理由から行う処理なので、機能にはまったく影響がない

ここで登録しているリスナーは、これまでにいくつも紹介したものとほぼ同じで、特に際立った特徴はない

This PDF is prepared for tools@maruhouse.org, personal use only.

```
  buttonBox.add(downTempo);

  Box nameBox = new Box(BoxLayout.Y_AXIS);
  for (int i = 0; i < 16; i++) {
      nameBox.add(new Label(instrumentNames [i]));
  }

  background.add(BorderLayout.EAST, buttonBox);
  background.add(BorderLayout.WEST, nameBox);

  theFrame.getContentPane().add(background);

  GridLayout grid = new GridLayout(16,16);
  grid.setVgap(1);
  grid.setHgap(2);
  mainPanel = new JPanel(grid);
  background.add(BorderLayout.CENTER, mainPanel);

  for (int i = 0; i < 256; i++) {
    JCheckBox c = new JCheckBox();
    c.setSelected(false);
    checkboxList.add(c);
    mainPanel.add(c);
  } // ループの終了

  setUpMidi();

  theFrame.setBounds(50,50,300,300);
  theFrame.pack();
  theFrame.setVisible(true);
} // メソッドの終了

public void setUpMidi() {
  try {
    sequencer = MidiSystem.getSequencer();
    sequencer.open();
    sequence = new Sequence(Sequence.PPQ,4);
    track = sequence.createTrack();
    sequencer.setTempoInBPM(421);

  } catch(Exception e) {e.printStackTrace();}
} // メソッドの終了
```

このあたりのコードもGUIを作成するためのもの。特に難しくないはず

チェックボックスを作成し、初期状態を「false(つまりチェックされていない状態)」にする。チェックボックスはまずArrayListに、続いてパネルに組み込む

Sequencerオブジェクトを取得し、Sequenceオブジェクト、Trackオブジェクトを作成する、といったごく一般的なMIDI関連のコード

This PDF is prepared for tools@maruhouse.org, personal use only.

「Start」ボタンのクリックに対応するイベントハンドラによって呼び出されるメソッド。チェックボックスの状態を基にMidiEventオブジェクト、Trackオブジェクトを作るのはこのメソッド

```
  public void buildTrackAndStart() {
    int [] trackList = null;

    sequence.deleteTrack(track);
    track = sequence.createTrack();

    for (int i = 0; i < 16; i++) {
      trackList = new int [16];

      int key = instruments [i];

      for (int j = 0; j < 16; j++) {
        JCheckBox jc = (JCheckBox) checkboxList.get(j + (16*i));
        if (jc.isSelected()) {
          trackList [j] = key;
        } else {
          trackList [j] = 0;
        }
    } // 内側のループの終了

    makeTracks(trackList);
    track.add(makeEvent(176,1,127,0,16));
  } // 外側のループの終了

  track.add(makeEvent(192,9,1,0,16));
  try {

    sequencer.setSequence(sequence);
    sequencer.setLoopCount(Sequencer.LOOP_CONTINUOUSLY);
    sequencer.start();
    sequencer.setTempoInBPM(120);

  } catch(Exception e) {e.printStackTrace();}
} // buildTrackAndStart() メソッドの終了

public class MyStartListener implements ActionListener {
  public void actionPerformed(ActionEvent a) {
    buildTrackAndStart();
  }
} // 内部クラスの終了

public class MyStopListener implements ActionListener {
  public void actionPerformed(ActionEvent a) {
    sequencer.stop();
  }
} // 内部クラスの終了
```

楽器ごとに要素数16個の配列を作り、各楽器の16個のチェックボックスの状態を表す値を格納する。対応するチェックボックスがチェックされている要素には、変数keyの値を代入し、チェックされていない要素には0を代入する

古いTrackオブジェクトを削除し、新たに作成する

同じ処理を各楽器（ベースドラム、クローズドハイハットなど）について行う

変数keyに楽器（ベースドラム、クローズドハイハットなど）を表す値を設定する。instruments配列には、個々の楽器に対応する数字が格納される

同じ処理を同じ楽器に対応する16個のチェックボックスそれぞれについて行う

チェックボックスがチェックされているかどうかを確認し、チェックされていれば変数keyの値を代入し、チェックされていなければ0を代入する

チェックボックスの状態の確認結果を基にMidiEventオブジェクト、Trackオブジェクトを作る

256個のチェックボックスに対応するパターンが正しく繰り返されるようにするには、このコードが必要

繰り返し回数を指定する。ここでは無制限に繰り返す

このコードで演奏する

ボタン操作に対応する内部クラス。この種のクラスについては説明済み

This PDF is prepared for tools@maruhouse.org, personal use only.

```
  public class MyUpTempoListener implements ActionListener {
    public void actionPerformed(ActionEvent a) {
      float tempoFactor = sequencer.getTempoFactor();
      sequencer.setTempoFactor((float)(tempoFactor * 1.03));
    }
  } // 内部クラスの終了

  public class MyDownTempoListener implements ActionListener {
    public void actionPerformed(ActionEvent a) {
      float tempoFactor = sequencer.getTempoFactor();
      sequencer.setTempoFactor((float)(tempoFactor * .97));

    }
  } // 内部クラスの終了

  public void makeTracks(int [] list) {

    for (int i = 0; i < 16; i++) {
      int key = list [i];

      if (key != 0) {
        track.add(makeEvent(144,9,key,100,i));
        track.add(makeEvent(128,9,key,100,i + 1));
      }
    }
  }

  public MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
    MidiEvent event = null;
    try {
      ShortMessage a = new ShortMessage();
      a.setMessage(comd, chan, one, two);
      event = new MidiEvent(a, tick);

    } catch(Exception e) {e.printStackTrace(); }
    return event;
  }
} // クラスの終了
```

ボタンに対応するイベントリスナーの内部クラス

テンポはテンポ係数の指定によって調整する。テンポ係数のデフォルトは 1.0 なので、クリックあたり± 3%の調整ができる

このメソッドでは、楽器ごとに 16 個のチェックボックスに対応するMidiEventオブジェクト、Trackオブジェクトを作成する。引数となっているintの配列の各要素には、楽器に対応する数字、あるいは、0 が格納されている。「0」は「音を出さない」ということを意味する。要素の値が 0 でなかった場合にMidiEventオブジェクトとTrackオブジェクトを作成する

音を出すためのMidiEventオブジェクトと音を止めるためのMidiEventオブジェクトを作成し、それをTrackオブジェクトに組み込む

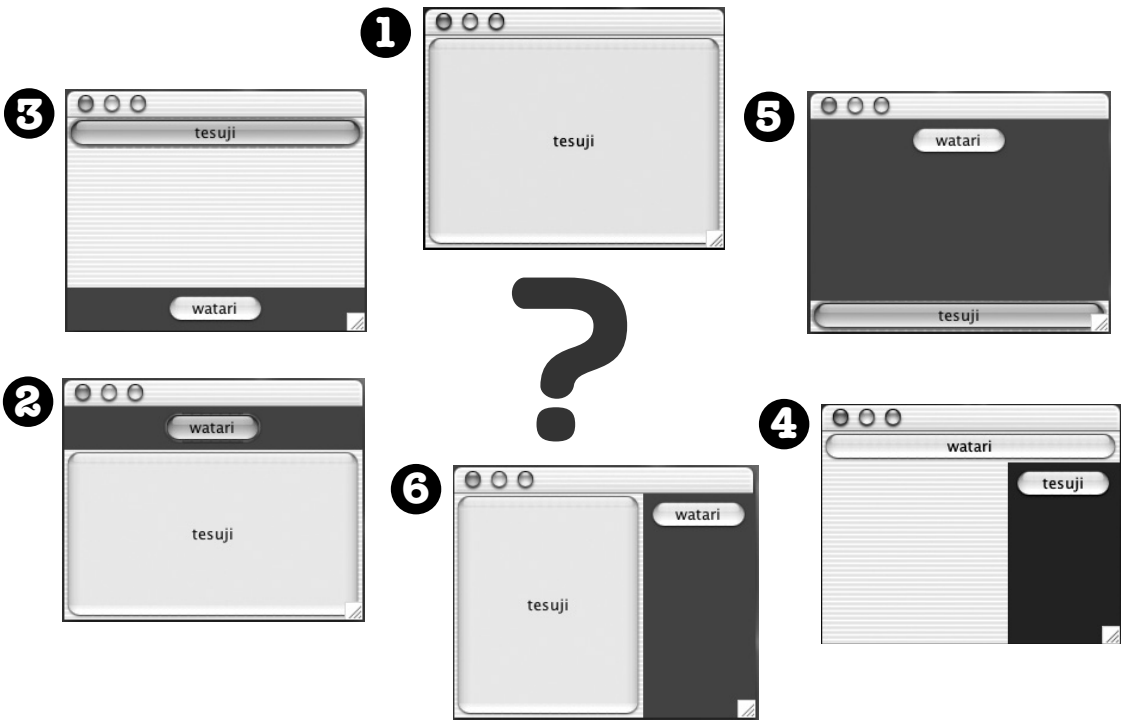
前章のプログラム・キッチンで紹介したユーティリティメソッド

This PDF is prepared for tools@maruhouse.org, personal use only.

エクササイズ

コードを当てよう

以下に示した 1 ～ 6 の画面を作るには、それぞれ次のページのどのコードを使用すればよいでしょうか。それを答えてください。ただ、1 ～ 6 の中には、1 つ対応するコードがないものがあるので注意してください。



This PDF is prepared for tools@maruhouse.org , personal use only.

A

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
panel.add(button);
frame.getContentPane().add(BorderLayout.NORTH,buttonTwo);
frame.getContentPane().add(BorderLayout.EAST,panel);
```

B

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.CENTER,button);
frame.getContentPane().add(BorderLayout.EAST,panel);
```

C

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.CENTER,button);
```

D

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
frame.getContentPane().add(BorderLayout.NORTH,panel);
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.CENTER,button);
```

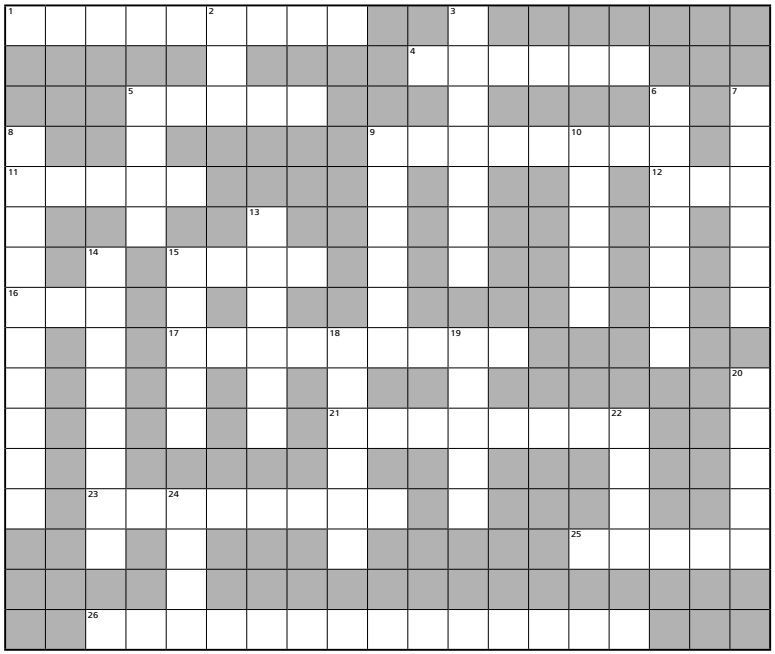
E

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
frame.getContentPane().add(BorderLayout.SOUTH,panel);
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.NORTH,button);
```


This PDF is prepared for tools@maruhouse.org, personal use only.

Java クロスワード

Javaに関する用語のクロスワードです（注：このクロスワードには英語で、すべて大文字で答えてください）。



ヨコのカギ

1. プログラマが自ら図形を描くことができるウィジェット
4. 他の領域が使った残りのスペースを使う
5. Javaの標準のルック＆フィール
9. イベントの発生を待ち受ける
11. できごと
12. コンポーネントの組み込みに使う
15. JPanelのデフォルトレイアウトマネージャは____Layout
16. スーパークラスとサブクラスの関係
17. 絵が動く
21. 複数行のテキストを入力、表示できる
23. 「✓」をつける
25. テキスト
26. actionPerformed() を定義するリスナーインターフェース

タテのカギ

2. Swingより古いグラフィックス関連のクラスライブラリ
3. フレームのサイズ指定
5. 機能を選択
6. テキストより楽しい
7. コンポーネントとほぼ同義の言葉
8. 画面表示
9. コンポーネントのサイズ、配置
10. BorderLayoutの上部の領域
13. レイアウトマネージャはこれに基づき機能する
14. イベントソースがイベントリスナーのメソッドを呼び出すこと（ヒント：「電話を折り返す」というのと同じ言葉です）
15. デフォルトがBorderLayout
18. ユーザーがボタンをクリックするなどの操作
19. 内部クラスを中に含むのは_____クラス
20. 他のコンポーネントを組み込むのによく使用する
22. Mac OS Xのルック＆フィール
24. BorderLayoutの右側の領域

ヒント

描画パネル = draw panel
外部 = outer

This PDF is prepared for tools@maruhouse.org, personal use only.

エクササイズの答え

1

C

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.CENTER,button);
```

2

D

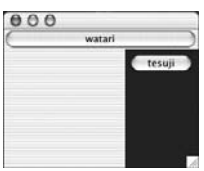
```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
frame.getContentPane().add(BorderLayout.NORTH,panel);
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.CENTER,button);
```

3

E

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
frame.getContentPane().add(BorderLayout.SOUTH,panel);
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.NORTH,button);
```

4



A

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
panel.add(button);
frame.getContentPane().add(BorderLayout.NORTH,buttonTwo);
frame.getContentPane().add(BorderLayout.EAST,panel);
```

6

B

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.CENTER,button);
frame.getContentPane().add(BorderLayout.EAST,panel);
```


This PDF is prepared for tools@maruhouse.org, personal use only.

Java クロスワード

1 D	R	A	W	P	2 A	N	E	L			3 S						
					W					4 C	E	N	T	E	R		
			5 M	E	T	A	L				T					6 G		7 W
8 S			E						9 L	I	S	T	E	10 N	E	R		I
11 E	V	E	N	T					A		I			O		12 A	D	D
T			U			13 P			Y		Z			R		P		G
V		14 C		15 F	L	O	W		O		E			T		H		E
16 I	S	A		R		L			U					H		I		T
S		L		17 A	N	I	M	18 A	T	I	19 O	N				C		
I		L		M		C		C			U							20 P
B		B		E		Y		21 T	E	X	T	A	R	E	22 A			A
L		A						I			E				Q			N
E		23 C	H	24 E	C	K	B	O	X		R				U			E
		K		A				N						25 L	A	B	E	L
				S														
		26 A	C	T	I	O	N	L	I	S	T	E	N	E	R			

This PDF is prepared for tools@maruhouse.org, personal use only.

14章　シリアライゼーションとファイルI/O

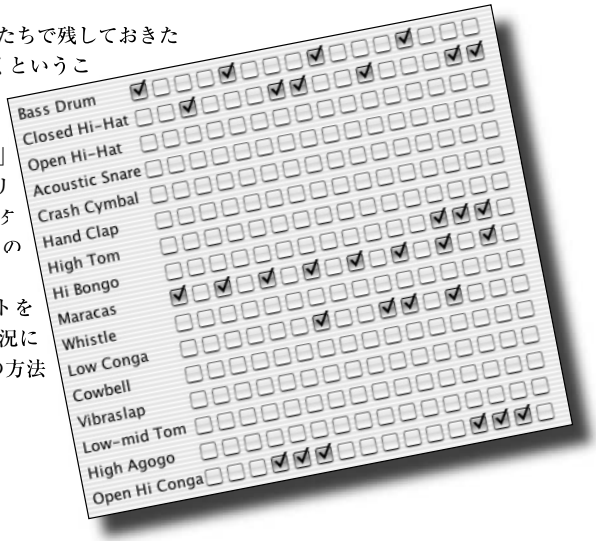
オブジェクトの保存

これ以上仕事が大変になったら、ボスをシリアライズしないと……。私はどんなオブジェクトでも簡単にシリアライズできるんだから。もちろん、ボスはそんなこと許さないでしょうけど、かまわないわ。でも、シリアライズされたボスってどんな感じかしら……。

プログラムの中には、オブジェクトをファイルに保存する機能を必要とするものがあります。では、オブジェクトのファイルへの保存はどのように行えばよいのでしょうか。オブジェクトは大きく「機能」と「状態」という2つの要素に分けることができるのですが、オブジェクトがどのような機能を持つのかは、そのオブジェクトが属するクラスのコードによってわかります。同じクラスのオブジェクトであれば、機能はすべて共通ということです。一方、「状態（ステートともいう）」はオブジェクトごとに異なります。ただ、オブジェクトの状態は、そのオブジェクトのインスタンス変数の値によってわかります。つまり、このインスタンス変数の値さえ保存しておけば、いつでも同じオブジェクトを再現できるということです。とはいえ、インスタンス変数の値を逐一確認し、それをファイルに保存するコードを書くのは大変です。まず、値をどのような形式で保存するかが問題です。そこで、Javaには、そうしたことを自動的に行う「シリアライゼーション」という機能が用意されています。シリアライゼーションによって保存されたオブジェクトは、「デシリアライゼーション」という処理によって、元に戻すことができます。14章では、このシリアライゼーション、デシリアライゼーションについて解説します。ただ、シリアライゼーションをしたファイルはJavaプログラム以外では利用できないので、他の言語のプログラムで利用する場合には、オブジェクトを別の方法で保存する必要があります。この章ではそのことについても説明します。

This PDF is prepared for tools@maruhouse.org, personal use only.

ドラムパターンを保存する



気に入ったドラムパターンができたら、それを何らかのかたちで残しておきたいと思うのが普通でしょう。もちろん、紙に書き留めておくということもできるでしょうが、やはりプログラムに「Save」ボタン（あるいは「Save」メニュー）などが用意されていて、それをクリックするだけで保存できるというのが理想でしょう。「Save」ボタンをクリックして、保存先のファイル名、ディレクトリを指定するのです。ファイルに保存しておけば、コンピュータが突然フリーズするなどしてもパターンが消えてしまわないので安心です。

ドラムパターンなどを保存するためには、「オブジェクトを保存する」ということが必要になりますが、そのためには状況によっていくつかの方法が選択できます。この章では、以下の方法について見ていきましょう。

保存したオブジェクトを、そのオブジェクトを生成したJavaプログラムのみで利用する場合

① **シリアライゼーション機能を利用する**
保存したオブジェクトをJavaプログラムでのみ利用することがわかっていれば、「シリアライゼーション」という機能によってファイルへの保存ができます。また、ファイルに保存したオブジェクトをヒープ上に再現するには、「デシリアライゼーション」という機能を使用します。

保存したオブジェクトをJava以外の言語のプログラムでも利用する場合

② **インスタンス変数の値をプレーンテキストファイルに書き込む**
Java以外の言語で書かれたプログラムでもオブジェクトを利用する可能性がある場合は、インスタンス変数の値をプレーンテキストファイルに保存します。値と値の間には、何らかの区切り文字を入れます。区切り文字は何種類か考えられますが、オブジェクトを利用するプログラムが表計算プログラムやデータベースアプリケーションである場合は、タブを使用すれば、区切り文字として認識されます

もちろん、選択肢は上記の2つだけではありません。他にも方法はあります。たとえば、上記のように値をテキストにするのではなく、2進数（バイト形式）にするということも可能です。この方法であれば、int、long、booleanなどプリミティブ型変数の値はそのままのかたちでファイルに書き込めます。ただ、どの方法を使うにしろ、覚えておかねばならないのは、Javaでデータの入出力（I/O）がどのように行われるのか、ということです。I/Oというのは、ファイルやディスク、ネットワークで接続された他のコンピュータとの間のデータのやりとりのことです。I/Oの処理は、オブジェクトの保存の仕方が変わっても基本的には変わることがありません。また、もちろん、ここではデータベースは使用しない前提で話を進めます。

This PDF is prepared for tools@maruhouse.org, personal use only.

オブジェクトの保存方法

たとえば、とても一度ではプレーが終わらないようなロールプレイングゲームのプログラムがあったとします。このゲームのキャラクタの特性はゲームの進行とともに変化していきます。強くなることもあれば、弱くなることもあり、知力がアップすることもあります。また、新たな武器を手に入れることもあれば、一度手に入れた武器を失ってしまうこともあります。プレーを中断するたびにゼロからやり直していたら、永遠にゲームは終了せず、キャラクタもある程度以上は成長しません。つまり、何らかの方法で、キャラクターの特性など、その時点でのゲームの進行状況を記録する必要があるわけです。記録したデータを利用すれば、中断した時点の状態からゲームを続行できます。また、プログラムを作る側からすれば、進行状況の記録（あるいは記録データの利用）ができるだけ簡単に、ミスのないように行えなければ困ります。

① 方法 1
キャラクターオブジェクトを「シリアライゼーション」によってファイルに保存する

キャラクターオブジェクトをシリアライズして、その結果をファイルに保存します。このファイルは、テキストエディタで開いても以下のような意味のわからない文字しか表示されません。

```
¨ I`srGameCharacter
¨ %ge^8MU^IpowerLjava/lang/
String;[weaponst[Ljava/lang/
String;xp2tlfur[Ljava.lang.String; ≠ "VA ´
E`{Gxptbowtswordtdustsq~》tTrolluq~tb
are handstbig axsq~xtMagicianuq~tspe
llstinvisibility
```

② 方法 2
プレーンテキストファイルにインスタンス変数の値を書き込む

テキストファイルにキャラクターオブジェクトのインスタンス変数の値を書き込みます。オブジェクトごとに改行をし、値と値の間には区切り文字としてカンマを入れます。

50,Elf,bow,sword,dust
200,Troll,bare hands,big ax
120,Magician,spells,invisibility

シリアライズされたデータは、人間には意味がわからないものだが、プログラムにとっては、テキストファイルに保存された変数の値よりも読み取りが容易である。また、テキストファイルに変数の値を保存して、それを基にオブジェクトを再生するという方法だと、誤りが起きやすい。たとえば、値の保存順序を間違えてしまうと、キャラクターの種類が"dust"で、武器が"Elf"である、などと認識されてしまう恐れがある。

注：
elf = 妖精
bow = 弓
troll = トロール
bare hand = 素手
ax = 斧
magician = 魔法使い
spell = 魔法
invisibility = 透明になる術

This PDF is prepared for tools@maruhouse.org, personal use only.

オブジェクトをシリアライズしてファイルに保存する

オブジェクトをシリアライズする手順を以下に簡単にまとめておきます。詳しくはこの後説明しますので、今の時点で完全に理解する必要はありません。

"MyGame.ser" はオブジェクトの保存先となるファイルの名前。この名前のファイルが存在しない場合は自動的に作成される

1 FileOutputStreamオブジェクトを作成する。

```
FileOutputStream fileStream = new FileOutputStream("MyGame.ser");
```

FileOutputStreamオブジェクトを作成する。FileOutputStreamオブジェクトは、プログラムと特定のファイルを接続する機能、ファイルを作成する機能などを備えたオブジェクト

2 ObjectOutputStreamオブジェクトを作成する。

```
ObjectOutputStream os = new ObjectOutputStream(fileStream);
```

ObjectOutputStreamオブジェクトは、オブジェクトをシリアライズし、その結果を外に書き出す機能を備えたオブジェクト。ただ、プログラムとファイルを接続する機能は持っていないので、ファイルにオブジェクトの内容を書き出すには、ファイルへの接続機能を備えたオブジェクト（このように処理の助けとなるオブジェクトを「ヘルパー」と呼ぶ）を必要とする。「ObjectOutputStream」や「FileOutputStream」などのオブジェクトを「ストリーム」と呼ぶが、2つ以上のストリームを組み合わせて使用することを「ストリームのチェーニング」という

3 オブジェクトをシリアライズし、その結果を書き出す。

```
os.writeObject(characterOne);
os.writeObject(characterTwo);
os.writeObject(characterThree);
```

このメソッドにより、characterOne、characterTwo、characterThreeといったパラメータに納められた参照に対応するオブジェクトをシリアライズし、"MyGame.ser" というファイルにシリアライズの結果を書き込む

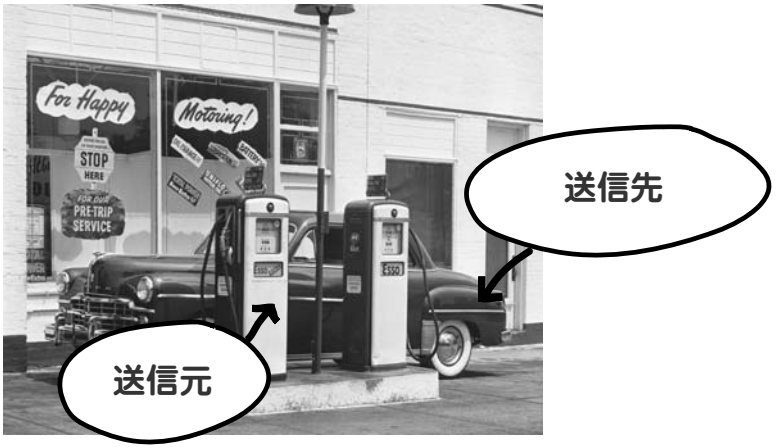
4 ObjectOutputStreamオブジェクトをクローズする。

```
os.close();
```

上位のストリームをクローズすると、下位のストリームは自動的にクローズされる。したがって、このコードによりFileOutputStream（とファイル）もクローズされることになる

This PDF is prepared for tools@maruhouse.org, personal use only.

データの通り道となる「ストリーム」



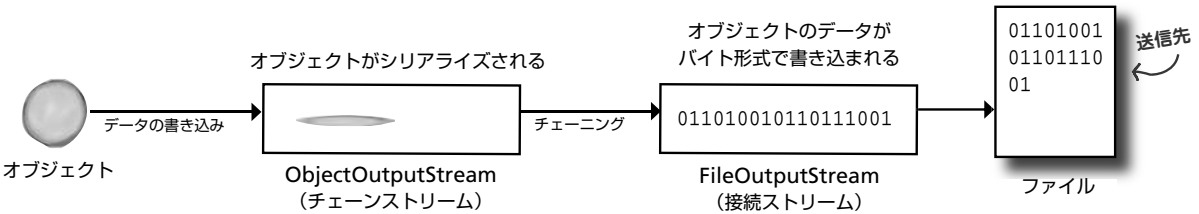
接続ストリームは、データの送信元、送信先の間の「接続」を表すオブジェクト。チェーンストリームは、この接続ストリームと組み合わせて使う。

JavaのAPIには、データを送受信する際にその「通り道」となる、「ストリーム」と呼ばれる種類のオブジェクトが用意されています。データの送信元や送信先となるものとしては、ファイルやネットワークソケットなどが考えられます。ストリームの中でも、送信元、送信先の間の「接続」に対応するもののことを「接続ストリーム」と呼びます。他のストリームと組み合わせることによってのみ機能するものを「チェーンストリーム」と呼びます。

ストリームは多くの場合、少なくとも2つを組み合わせて使用します。2つのうち一方は接続ストリームで、もう一方はチェーンストリームです。そして、何らかの処理を行うために、チェーンストリームのメソッドを呼び出すのです。なぜ2つを組み合わせるのかというと、接続ストリームは通常、あまりに「低水準」で、使い方が難しいからです。たとえば、FileOutputStreamは接続ストリームの一種ですが、このストリームには、データをバイト形式で出力するメソッドのみが用意されています。つまり、オブジェクトをシリアライズして出力する、といった目的には適さないのです。そうした目的のためには、より「高水準」のチェーンストリームを使用するのです。

なぜこんなことが必要なのでしょうか。なぜ、1種類のストリームであらゆる入出力ができるようにしないのでしょうか。オブジェクトをバイト形式に変換する機能を持ったストリームを作れば、ストリームを2つも使わずにファイルにオブジェクトを保存できるのではないでしょうか。それをしないのは、1つのオブジェクトに多様な機能を持たせることが、オブジェクト指向プログラミングにとって望ましいことではないからです。1つに多様な機能を持たせるより、FileOutputStreamはバイト形式でデータを出力する、ObjectOutputStreamはオブジェクトをストリームに書き込める形式のデータに変換する、というように、個々のオブジェクトに1つずつの「得意技」を持たせる方が良いのです。オブジェクトをファイルに保存する際に呼び出すのは、ObjectOutputStreamのwriteObject()メソッドですが、最終的に、オブジェクトのデータはFileOutputStreamによってバイト形式に変換されてファイルに書き込まれることになります。

ストリームが接続ストリームとチェーンストリームに分かれているということは、プログラマの自分の意志でさまざまなことを自由に行えるということを意味します。もし、ストリームが一種類だけだとしたら、できること、できないことをAPIのデザイナーにすべて決められてしまうことになります。APIのデザイナーがプログラマーのニーズを十分にくみとれなかった場合には、プログラマーは大変な不便を強いられるでしょう。



This PDF is prepared for tools@maruhouse.org, personal use only.

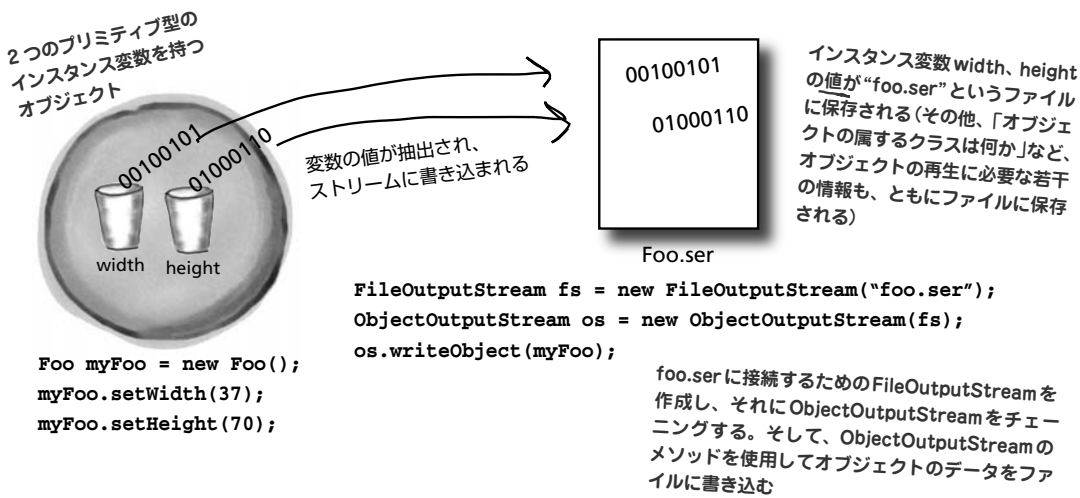
シリアライゼーションとは

❶ ヒープ上のオブジェクト

オブジェクトにはステート（インスタンス変数の値）がある。インスタンス変数の値は同じクラスのオブジェクトであってもそれぞれに異なっている

❷ シリアライズされたオブジェクト

オブジェクトをシリアライズする際には、インスタンス変数の値を保存する。この値を使用することにより、ヒープ上に元と同じオブジェクトを再度作ることができる



This PDF is prepared for tools@maruhouse.org, personal use only.

インスタンス変数の値はどのようなかたちで保存されるか

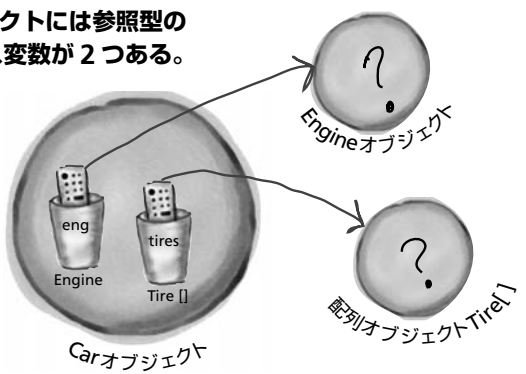
シリアライズにおいては、インスタンス変数の値を保存するのですが、いったいどのようなかたちで保存されるのでしょうか。プリミティブ型の変数であれば、値は37、70といった数値なのですから保存は非常に簡単です。しかし、参照型の変数の値はどうすればいいのでしょうか。たとえば、参照型のインスタンス変数を5つ持つオブジェクトがあったとしたら、値はどのように保存するのでしょうか。また、変数の値（つまりオブジェクトリファレンス）に対応するオブジェクトのインスタンス変数はどうするのでしょうか。

ここで重要なのは、「どこまでがそのオブジェクトの固有の特性か」ということです。ともかく、シリアライズしたオブジェクトは、元とまったく同じオブジェクトとしてヒープ上に再現できればよいのです。それに不要な情報はシリアライズには要りません。再現時には、メモリ上での位置などはどうしても元と変わりますので、「オブジェクトがメモリ上のどこにあったか」といった情報は不要ということになります。シリアライズした時点とステートを同じにするのに必要な情報以外はいらないのです。

頭の体操

右の図のCarオブジェクトを、後で同じステートで再生できるよう保存（シリアライズ）するとします。
保存（シリアライズ）に必要なのはどのような情報でしょうか。
また、もしEngineオブジェクトにさらにCarburatorオブジェクトへの参照を納めるインスタンス変数があったとしたらどうでしょうか。その他、配列オブジェクトTire []の要素をどのように扱うべきかも考えてください。

carオブジェクトには参照型のインスタンス変数が2つある。



Carオブジェクトをシリアライズする場合に保存すべき情報は？

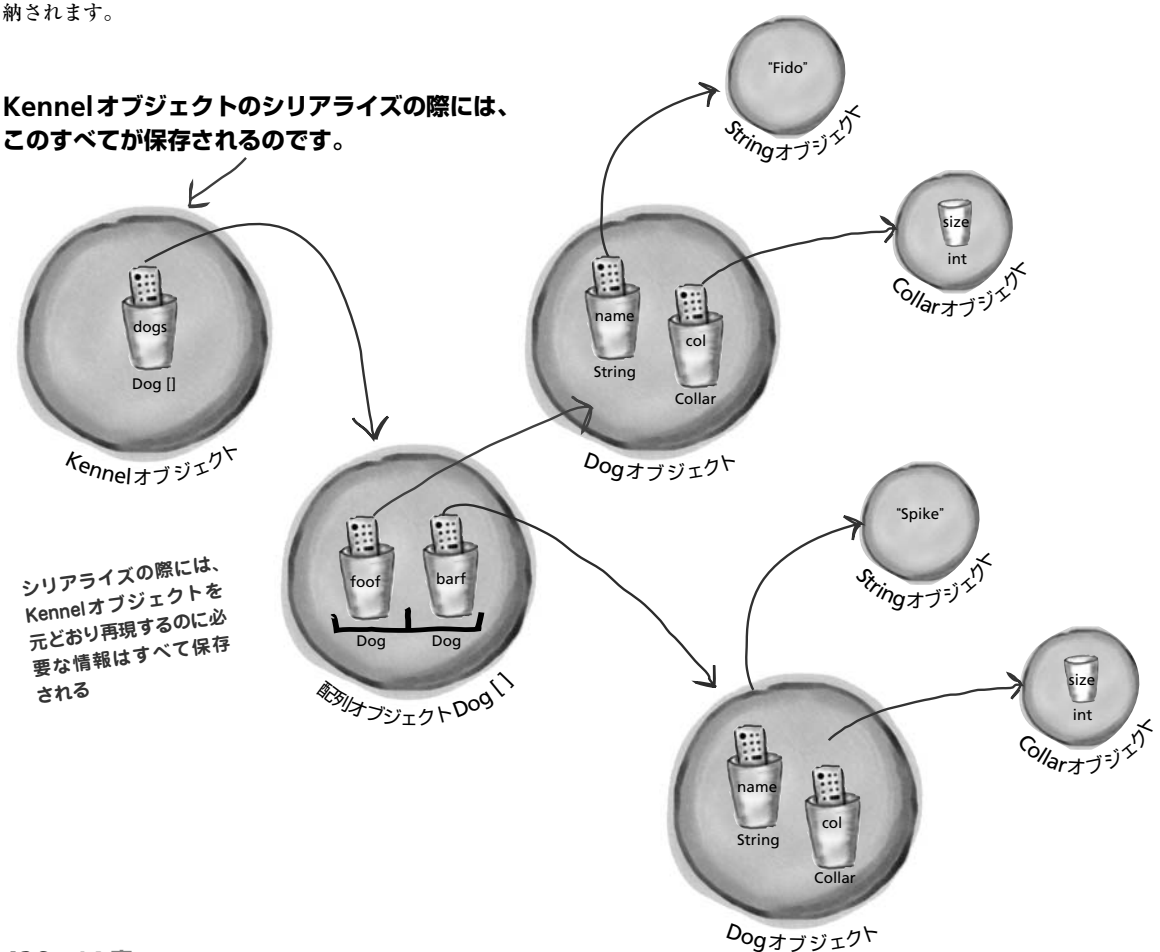
This PDF is prepared for tools@maruhouse.org, personal use only.

オブジェクトのシリアライズの際には、参照型のインスタンス変数の値（オブジェクトリファレンス）に対応するオブジェクトもすべて同時にシリアライズされます。また、オブジェクトリファレンスに対応するオブジェクトに参照型の変数があれば、その値に対応するオブジェクトもやはりすべてシリアライズされます。そして、素晴らしいのは、これがすべて自動的に行われるということです。

以下の図に示したKennelオブジェクトには、dogsというインスタンス変数があります。これは、Dogオブジェクトへの参照の配列です。この配列にはDogオブジェクトへの参照が2つ納められています。DogオブジェクトにはStringオブジェクトへの参照、Collarオブジェクトへの参照を納めるインスタンス変数があります。Stringオブジェクトには文字列が、Collarオブジェクトには整数値が格納されます。

シリアライズに際しては、参照型のインスタンス変数の値（オブジェクトリファレンス）に対応するオブジェクトもすべて同時にシリアライズされる

Kennelオブジェクトのシリアライズの際には、このすべてが保存されるのです。



シリアライズの際には、Kennelオブジェクトを元どおり再現するのに必要な情報はすべて保存される

This PDF is prepared for tools@maruhouse.org, personal use only.

クラスにシリアライズの機能を持たせるには Serializable インターフェースをインプリメントする

Serializableインターフェースは、「マーカーインターフェース」、「タグインターフェース」などと呼ばれる種類のインターフェースです。この種のインターフェースには、「インプリメントすべきメソッドが定義されていない」という特徴があります。「このクラスはシリアライズ対応である」ということを通知する以外、何の役割も持っていないのです。Serializableインターフェースをインプリメントしたクラスのオブジェクトはシリアライズが可能なのですが、スーパークラスがシリアライズ可能になっている場合、サブクラスは(implements Serializableと書かなくても)自動的にシリアライズ可能になるので注意が必要です(ただし、同様のことはすべてのインターフェースに当てはまります)。

```
objectOutputStream.writeObject(myBox);
```

ここに指定するオブジェクトのクラスはSerializableをインプリメントするものでなければならない。さもなければ実行時にエラーになる

Serializableはjava.ioパッケージのインターフェースなので、このコードが必要

```
import java.io.*;
public class Box implements Serializable {

  private int width;
  private int height;

  public void setWidth(int w) {
    width = w;
  }

  public void setHeight(int h) {
    height = h;
  }

public static void main (String [] args) {
    Box myBox = new Box();
    myBox.setWidth(50);
    myBox.setHeight(20);

    try {
      FileOutputStream fs = new FileOutputStream("foo.ser");
      ObjectOutputStream os = new ObjectOutputStream(fs);
      os.writeObject(myBox);
      os.close();
    } catch(Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

Serializableインターフェースにはインプリメントすべきメソッドはない。ただ、"implements Serializable"と書くことでJVMに「このクラスはシリアライズ可能」と知らせることができる

この変数の値は保存される

I/O関連のメソッドは例外を発生させる恐れがある

"foo.ser"というファイルに接続する。ファイルが存在しなければ作る

ObjectOutputStreamを作成し、接続ストリームにチェーニングするObjectOutputStreamのメソッドを使用して、オブジェクトのデータをファイルに書き込む

This PDF is prepared for tools@maruhouse.org, personal use only.

「0 か 100 か」のシリアライゼーション
必要なステートがすべて得られない場合にはシリアライゼーションは行われない

考えただけで怖いよ！　イヌに体重がなかったり、耳がなかったりしたら……、首輪のサイズが 30 センチじゃなくて 3 センチだったりしたら……、そんなのイヤだよ

関連するオブジェクトがすべて正しくシリアライズできない限り、シリアライズはエラーになる。たとえば、左の例では、Duckオブジェクトがシリアライズ可能でない限り、Pondオブジェクトはシリアライズできない

```
import java.io.*;
public class Pond implements Serializable {

  private Duck duck = new Duck();

  public static void main(String [] args) {
    Pond myPond = new Pond();
    try {
      FileOutputStream fs = new FileOutputStream("Pond.ser");
      ObjectOutputStream os = new ObjectOutputStream(fs);

      os.writeObject(myPond);
      os.close();

    } catch(Exception ex) {
      ex.printStackTrace();
    }
  }
}

public class Duck {
 // Duck クラスのコード
}
```

これによりPondオブジェクト自体はシリアライズ可能になる

Pondクラスには参照型のインスタンス変数Duckがある

Pondオブジェクトのシリアライズの際は、Duckオブジェクトもシリアライズする必要がある

Duckオブジェクトは、Serializableインターフェースをインプリメントしていないため、シリアライズ可能になっていない。これは、Pondオブジェクトのシリアライズがエラーになることを意味する

実行結果

```
File  Edit  Window  Help  Regret
% java Pond
java.io.NotSerializableException: Duck
                 at Pond.main (Pond.java:13)
```


This PDF is prepared for tools@maruhouse.org, personal use only.

インスタンス変数をシリアライズの対象から外すには「transient」宣言する

「transient」というキーワードを宣言に加えれば、インスタンス変数をシリアライゼーションの対象から外すことができます。

```
import java.net.*;
class Chat implements Serializable {
  transient String currentID;

  String userName;

  // etc...
}
```

「transient」は「この変数の値はシリアライゼーションの対象から外す」という意味

変数userNameの値はシリアライゼーションの対象となる

インスタンス変数の「型」となっているクラスがSerializableインターフェースをインプリメントしていない場合は、「transient」というキーワードを使うことで、その変数をシリアライゼーションの対象から外すことができます。

クラスがSerializableインターフェースをインプリメントしていない場合、その理由は単純に「プログラマが忘れていた」ということも多いのですが、そうではないこともあります。中には、実行時にのみ得られる情報によってオブジェクトの特性が変わってしまうために、あえてシリアライズができないようにしてある、というクラスもあり得るのです。Javaのクラスライブラリ中のクラスは、その多くがシリアライズ可能なのですが、ネットワーク接続、スレッド、ファイルに関するものなどにはシリアライズができないものも多いのです。それは、クラスのインスタンス化の仕方がプラットフォーム（JVM）の種類によって変わったり、プログラムを実行するたびに変わったり、といったことが起きるからです。プログラムを一度終了してしまえば、もう二度と同じインスタンスは作れない、というクラスに関してはシリアライズしても意味がありません。

This PDF is prepared for tools@maruhouse.org, personal use only.

素朴な疑問に答えます スペシャル

Q：シリアライゼーションが重要なものなら、なぜ全クラスをデフォルトでシリアライズ可能なものにしないのですか。Objectクラスで Serializableインターフェースをインプリメントすれば、サブクラスは自動的にシリアライズ可能になるのではないですか。

A：確かにSerializableインターフェースをインプリメントすべきクラスは多く、実際にほとんどのクラスがこのインターフェースをインプリメントすることになるのですが、それでもやはりクラスごとにインプリメントするかどうかを選択できるのは大切なことなのです。また、この選択は慎重に行う必要があります。

中には決してシリアライズをすべきではないというクラスも存在します。もし、デフォルトでシリアライズが可能になっていたら、どのようにして、そのクラスのシリアライズ機能を「オフ」にすればいいのでしょうか。Serializableのように、ある機能を持たせるために使用するインターフェースはありますが、その逆に、特定の機能を持たせないようにするためのインターフェースというのはありません。NonSerializableというインターフェースを設け、それでシリアライズができないことをJVMに通知する、ということは「ポリモーフィズム」の考え方とは合わないのです。

Q：クラスをシリアライズ不可能にするのはどのような場合ですか。

A：シリアライズ不可能にすべきクラスはめったにないのですが、たとえば、「パスワード」を表すクラスなどは、セキュリティ上の理由からファイルなどに保存できない方がいいでしょう。また、主要なインスタンス変数が、シリアライズ不可能なオブジェクトへの参照を格納する場合には、クラスをシリアライズする意味がほぼないと言えます。

Q：使用したいクラスが正当な理由もなく（単にプログラマがうっかり忘れたといった理由で）シリアライズ不可能になっている場合は、そのクラスを継承してシリアライズ可能なクラスを作ってもいいのでしょうか。

A：かまいません（ただし、クラスがfinal宣言されていない場合に限る）。シリアライズ可能なサブクラスを作った上で、元のクラスを利用するコードをサブクラスを利用するように改造すればいいのです。

Q：スーパークラスがシリアライズ不可能であることは、何かに影響しないのでしょうか。

A：大きな影響はありません。ただ、デシリアライズ（これについては次に詳しく説明します）の際には、シリアライズ不可能になっているスーパークラスがあると、そのスーパークラスのコンストラクタが、新規のオブジェクトが作られる場合とまったく同じように実行されるので、その点には注意しなくてはなりません。とはいえ、シリアライズ不可能にする正当な理由がないクラスに関しては、シリアライズ可能なサブクラスを作るのは、一応、有効な方法であると言えるでしょう。

Q：「transient」キーワードを宣言に加えれば、特定のインスタンス変数をシリアライゼーションの対象から外せるということですが、これで問題は解決するのでしょうか。そのインスタンス変数に対応するオブジェクトがシリアライズできなければ、後でオブジェクトを元に戻すことができないのではないでしょうか。そもそも、変数の値を保存するのは、オブジェクトを元に戻すことが目的のはずです。

A：おっしゃるとおり、その点は確かに問題なのですが、幸い、解決策はあります。transient宣言された参照型インスタンス変数の値は、デシリアライズの際、シリアライズ時点での値がどうであれ、nullということになります。つまり、そのインスタンス変数に対応するオブジェクトは保存されないということです。もし、変数がどうしてもnull以外の値を必要とするものであれば、困ったことになるでしょう。

この問題の解決策は二とおり考えられます。

1) デシリアライズ時に、transient宣言されたインスタンス変数を再初期化し、null以外の値を設定する。この対策は、デシリアライズによってできるオブジェクトの特性が、transient宣言されたインスタンス変数の値によって大きく左右されない場合に有効。たとえば、DogオブジェクトにCollarオブジェクト（首輪を表す）の参照を格納するインスタンス変数があった場合、一般に、イヌの首輪が前と変わってもさほどの不都合は生じないことから、このインスタンス変数の値は再初期化してもかまわないと考えられる。

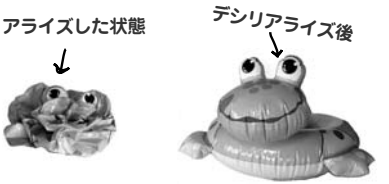
2) シリアライズできないオブジェクトの主要なインスタンス変数の値を保存しておき、それをオブジェクトの再現の際に利用する。この対策は、再現されたオブジェクトの特性がインスタンス変数の値によって大きく左右される場合（首輪の色やデザインでイヌを識別する、といった場合）に有効。この方法では、厳密にはオブジェクトを再現するというより、まったく新たに作り直すことになるが、元と実用上は同じ特性を持つオブジェクトが作れる。

Q：複数のオブジェクトのインスタンス変数に、同じオブジェクトに対応する参照が納められている時はどうなるのでしょうか。たとえば、Kennelオブジェクトがシリアライズ対象のオブジェクトで、このオブジェクトに、いずれもCat型の2つのインスタンス変数があったとします。2つのCatオブジェクトのOwner（「飼い主」を意味する）型のインスタンス変数の値が、どちらも同じOwnerオブジェクトへの参照である、という場合（つまり2匹のネコの飼い主が同じである場合）、シリアライズに際しては、Ownerオブジェクトが2つ保存されてしまうのでしょうか。そんなことはないと思うのですが……。

A：良い質問です。この場合、2つのCatオブジェクトに対応するOwnerオブジェクトがまったく同じものであるということは、シリアライゼーションの際に認識されるので心配は要りません。つまり、保存されるOwnerオブジェクトは1つだけということです。そして、デシリアライズの際には、1つのOwnerオブジェクトに対応する2つの参照も正しく再現されます。

This PDF is prepared for tools@maruhouse.org, personal use only.

デシリアライゼーション：保存したオブジェクトの再現



オブジェクトをシリアライズするのは、後で元の状態に戻す（デシリアライズする）ためです。元に戻すのは、JVMをいったん停止し、再度起動した後、ということもありますし、プログラムを実行するJVMそのものが、シリアライズした時点とは異なっているということもあります。デシリアライゼーションは、シリアライゼーションとはほぼ逆の処理と考えてよいでしょう。

1 FileInputStreamオブジェクトを作成する

```
FileInputStream fileStream = new FileInputStream("MyGame.ser")
```

"MyGame.ser"というファイルが存在しなければ例外が発生する

FileInputStreamオブジェクトを作成する。このオブジェクトは、すでに存在しているファイルへの接続を行う機能を備えている

2 ObjectInputStreamオブジェクトを作成する

```
ObjectInputStream os = new ObjectInputStream(fileStream);
```

ObjectInputStreamオブジェクトは、オブジェクトのデータを読み取る機能を備えているが、ファイルへの接続は単独ではできない。ファイルへの接続のためには、FileInputStreamという別のストリームへの「チェーニング」が必要

3 オブジェクトのデータを読み取る

```
Object one = os.readObject();
Object two = os.readObject();
Object three = os.readObject();
```

複数のオブジェクトのデータが保存されている場合は、readObject()メソッドを複数回呼び出せば、データを保存されたのと同じ順序で読み取ることができる。すでにすべてのデータを読み取ったにもかかわらずreadObject()メソッドを呼び出すと例外が発生する

4 オブジェクトをキャストする

```
GameCharacter elf = (GameCharacter) one;
GameCharacter troll = (GameCharacter) two;
GameCharacter magician = (GameCharacter) three;
```

readObject()の戻り値はObject型になるので（この点はArrayListに似ている）、完全に元に戻すにはシリアライズ前と同じ型にキャストする必要がある

5 ObjectInputStreamオブジェクトをクローズする

```
os.close();
```

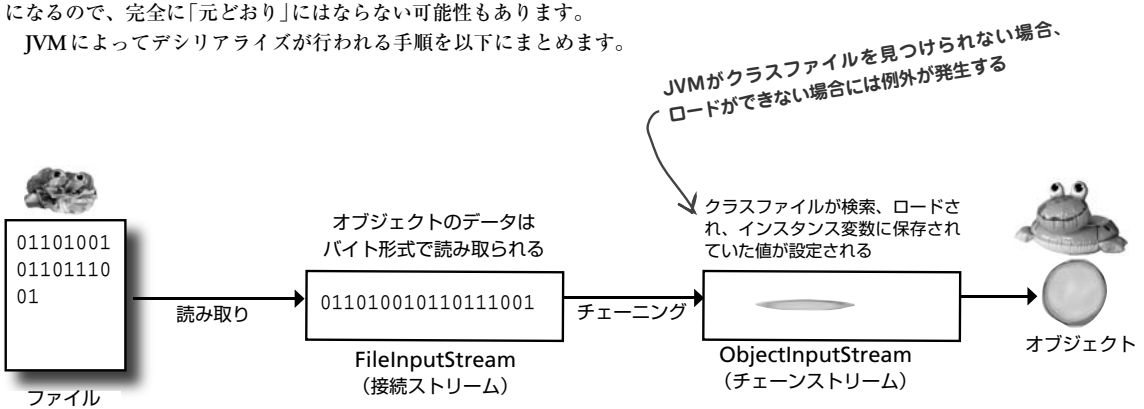
上位のストリームをクローズすると、下位のストリームは自動的にクローズされる。したがって、このコードによりFileInputStream（とファイル）もクローズされることになる

This PDF is prepared for tools@maruhouse.org, personal use only.

デシリアライズはどのように行われるのか

オブジェクトのデシリアライズというのは、厳密に言うと、シリアライズ時と同じステートを持つオブジェクトをヒープ上に新たに作る、という処理です。ただ、transient宣言されたインスタンス変数の値はnullになるか（参照型の変数の場合）、あるいはデフォルト値が設定（プリミティブ型の変数の場合）されることになるので、完全に「元どおり」にはならない可能性もあります。

JVMによってデシリアライズが行われる手順を以下にまとめます。



1. オブジェクトのデータがストリームから読み取られる。

2. （インスタンス変数の値とともに保存されている情報を基に）再現すべきオブジェクトのクラスが確認される。

3. 再現するオブジェクトのクラスファイルが検索、ロードされる。クラスファイルが見つからない場合、ロードできない場合には、例外が発生し、デシリアライゼーションは失敗に終わる。

4. 新しいオブジェクトがヒープ上に作られる。ただ、この時、コンストラクタは動作しないという点に注意。コンストラクタが動作すると、インスタンス変数の値がシリアライズした時点のものにはならず、まったく新たに設定されることになるため、デシリアライズの目的とは矛盾する。デシリアライズの際には、オブジェクトにシリアライズした時点とまったく同じステートを持たせなくてはならない。

This PDF is prepared for tools@maruhouse.org, personal use only.

5 デシリアライズ対象のオブジェクトの上位クラスの中にSerializableインターフェースをインプリメントしていないものがある場合は、デシリアライズ時にそのクラスのコンストラクタが実行される。また、それに伴いそのクラスより上位のクラス（シリアライズ可能なものも含む）のコンストラクタがすべて実行される。この現象は「コンストラクタチェーニング」と呼ばれるが、いったん始まると止めることができない。コンストラクタが実行されたクラスのオブジェクトに関してはステートがすべて初期化されることになる（シリアライズ時点の値は使われない）。

6 再現されたオブジェクトのインスタンス変数にシリアライズ時点の値が設定される。ただし、「transient」宣言された変数に関しては、null（参照型変数の場合）、あるいはデフォルト値（プリミティブ型変数の場合。デフォルト値は0、falseなど）が設定される。

素朴な疑問に答えます

Q：シリアライズの際、クラスファイルのデータが保存されないのはなぜですか。クラスファイルのデータも保存しておけば、クラスファイルが見付からないなどの問題は起きないと思うのですが。

A：確かにそういうシリアライゼーションの方法も考えられなくはありません。ただ、あまりに無駄、オーバーヘッドが大きくなり過ぎます。もし、シリアライゼーションの結果得られるデータをローカルのディスク上のファイルにのみ保存するのであれば、それでも特に問題はないでしょう。しかし、時にはネットワーク経由でデータを転送することもあり得るのです。シリアライズの際にクラスデータも保存するとなると、必要なネットワーク帯域が大幅に増えてしまい、それによってさまざまな問題が引き起こされることになります。

シリアライズしたオブジェクトのデータをネットワーク経由で転送する際には、データに、クラスファイルの所在を示すURLを添付することが可能です。この機能は、18章で詳述するJava RMI（Remote Method Invocation）に利用されます。それによって、シリアライズされたオブジェクトをメソッドの引数などに指定するといったことができるのです。JVMは、引数に指定されたオブジェクトのクラスデータがローカルに存在しなければ、URLを手掛かりにネットワーク経由で入手しロードします。すべてJVMによって自動的に行われるためプログラマはそれに関知する必要はありません。

Q：スタティック変数はシリアライズできるのですか。

A：できません。先述のとおり、スタティック変数は「オブジェクトに1つ」ではなく「クラスに1つ」という変数です。スタティック変数の値はオブジェクトのシリアライズ時に保存されることはなく、デシリアライズ時には、その時点でのカレント値が適用されます。このことから、スタティック変数の値によって大きく影響を受けるようなオブジェクトはシリアライズの対象にしない方が無難と言えます。シリアライズ時とデシリアライズ時のオブジェクトが同等にならない恐れが大きいからです。

This PDF is prepared for tools@maruhouse.org, personal use only.

ゲームキャラクターオブジェクトのシリアライズとデジリアライズ

```
import java.io.*;

public class GameSaverTest {
  public static void main(String[] args) {
    GameCharacter one = new GameCharacter(50,"Elf", new String[] {"bow", "sword", "dust"});
    GameCharacter two = new GameCharacter(200, "Troll", new String[] {"bare hands", "big ax"});
    GameCharacter three = new GameCharacter(120, "Magician", new String[] {"spells",
"invisibility"});
    // この部分に、キャラクターオブジェクトのステートを変えるような処理を行うコードが入る
    try {
      ObjectOutputStream os = new ObjectOutputStream(new FileOutputStream("Game.ser"));
      os.writeObject(one);
      os.writeObject(two);
      os.writeObject(three);
      os.close();
    } catch(IOException ex) {
      ex.printStackTrace();
    }
    one = null;
    two = null;
    three = null;

    try {
      ObjectInputStream is = new ObjectInputStream(new FileInputStream("Game.ser"));
      GameCharacter oneRestore = (GameCharacter) is.readObject();
      GameCharacter twoRestore = (GameCharacter) is.readObject();
      GameCharacter threeRestore = (GameCharacter) is.readObject();

      System.out.println("One's type: " + oneRestore.getType());
      System.out.println("Two's type: " + twoRestore.getType());
      System.out.println("Three's type: " + threeRestore.getType());
  } catch(Exception ex) {
    ex.printStackTrace();
  }
}
```

キャラクターの作成

nullを設定して、ヒープ上のオブジェクトにアクセスできないようにする

ファイルから読み出す

正しく処理が行われたことを確認

```
File  Edit  Window  Help  Resuscitate
% java GameSaver
Elf
Troll
Magician
```

This PDF is prepared for tools@maruhouse.org, personal use only.

GameCharacterクラス

```
import java.io.*;

public class GameCharacter implements Serializable {
  int power;
  String type;
  String[] weapons;

  public GameCharacter(int p, String t, String[] w) {
        power = p;
        type = t;
        weapons = w;
  }

  public int getPower() {
        return power;
  }

  public String getType() {
        return type;
  }

  public String getWeapons() {
        String weaponList = "";

        for (int i = 0; i < weapons.length; i++) {
        weaponList += weapons[i] + " ";
        }
        return weaponList;
  }
}
```

ここで紹介したコードはシリアライゼーションが実際に行われることを確かめるためのもの。ゲームのキャラクターのクラスだけで、ゲームそのものに対応するクラスは作っていないが、作ってみれば練習になるはずである

This PDF is prepared for tools@maruhouse.org, personal use only.

オブジェクトのシリアライゼーションの重要ポイント

- オブジェクトのシリアライズの際はステートが保存される。
- オブジェクトのシリアライズには、ObjectOutputStreamオブジェクトが必要(ObjectOutputStreamはjava.io パッケージのクラス)。
- ストリームは大きく接続ストリームとチェーンストリームに分けられる。
- 接続ストリームは、データの送信元、送信先(ファイル、ネットワークソケット、コンソールなど)の間の「接続」を表すオブジェクト。
- チェーンストリームだけでは、データの送信元、送信先との間の接続ができないので、接続ストリームへのチェーニングが必要になる。
- オブジェクトをシリアライズし、データをファイルに保存するには、ObjectOutputStreamとFileOutputStreamとのチェーニングが必要。
- オブジェクトのシリアライズの際には、ObjectOutputStreamクラスの writeObject (Object theObject)メソッドを呼び出す。FileOutputStreamクラスのメソッドは使用できない。
- シリアライズ可能なオブジェクトは、Serializableインターフェースをインプリメントするもののみ。スーパークラスがSerializableインターフェースをインプリメントしていれば、そのサブクラスも(implements Serializableという記述がなくても)自動的にシリアライズ可能になる。
- シリアライズの際には、直接その対象となるオブジェクトだけでなく、関連するオブジェクトもすべてシリアライズされる。参照型のインスタンス変数に対応するオブジェクトも、さらにそのインスタンス変数に対応するオブジェクトもすべてシリアライズされるということ。
- 関連するオブジェクトの中にシリアライズ不可能なものがあると、例外が発生する。ただ、あえてそのオブジェクトに対応するインスタンス変数をシリアライズ対象から外すという方法によって、例外発生を回避することはできる。
- インスタンス変数をあえてシリアライズの対象から外すには、transientというキーワードを使用する。transient宣言されたインスタンス変数には、シリアライズ時にnull(参照型の変数の場合)か、デフォルト値(プリミティブ型の変数の場合)が設定される。
- デシリアライズ時には、JVMは再現すべきすべてのオブジェクトのクラスデータを探す。見付からない場合は例外が発生する。
- オブジェクトのデータは、保存されたのと同じ順序で読み取る。読み取りにはreadObject()メソッドを使用する。
- readObject()の戻り値はObject型なので、オブジェクトを完全に再生するにはキャストが必要になる。
- スタティック変数はシリアライズされない。スタティック変数を特定のオブジェクトの「ステート」に含めても何の意味もない。スタティック変数に関しては、同じクラスのオブジェクトがすべて同じ値を共有するため。

This PDF is prepared for tools@maruhouse.org, personal use only.

オブジェクトのデータをプレーンテキストファイルに書き込む

オブジェクトのデータを保存する方法としては、ここまでに紹介してきたシリアライゼーションが最も簡単です。ただ、データを保存した時と同じプログラムで利用するのならそれで問題はないのですが、別の（Java以外の言語で書かれた）プログラムで利用する必要がある場合にはシリアライゼーションは使えません。その場合には、プレーンテキストファイルにデータを書き込むことになります。たとえば、サーブレット（Webサーバーで動作する特殊なJavaプログラム）では、Webブラウザ上でユーザーが入力したデータを受け付けることがありますが、このデータをシリアライゼーションによって保存してしまうと、Javaのプログラムでしか利用できません。表計算ソフトウェアなど、他の言語で書かれたプログラムで利用する場合には、データをプレーンテキストファイルに保存しておかねばならないのです。

テキストファイルにデータを書き込むという処理で行うことは、シリアライゼーションとさほど変わりません。ただ、異なるのは、書き込むのが文字列である、という点です。そのため、FileInputStreamは使わず、FileWriterを使うことになります（ストリームのチェーニングは必要ない）。

ゲームキャラクターに関するデータをテキストファイルに文字列で書き込むと以下のようになる

```
50,Elf,bow,sword,dust
200,Troll,bare hands,big ax
120,Magician,spells,invisibility
```

シリアライゼーションでのデータの書き込み

```
objectOutputStream.writeObject(someObject);
```

文字列の書き込み

```
fileWriter.write("My first String to save");
```

```
import java.io.*;
class WriteAFile {
  public static void main (String [] args) {

    try {
      FileWriter writer = new FileWriter("Foo.txt");

      writer.write("hello foo!");

      writer.close();

    } catch(IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

FileWriterクラスを使用するためにjava.ioパッケージをインポート

"Foo.txt"というファイルが存在しなければ自動的に作られる

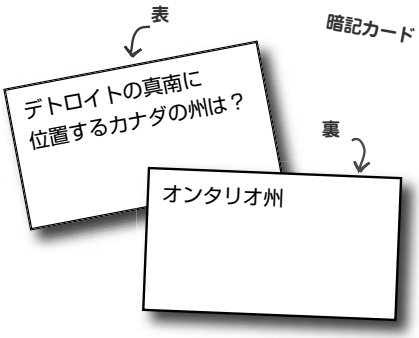
I/Oのためのコードはすべてtry/catchブロックに入れる。IOExceptionという例外が発生する恐れがあるため

write()メソッドの引数は文字列

処理が終わったらクローズする

This PDF is prepared for tools@maruhouse.org, personal use only.

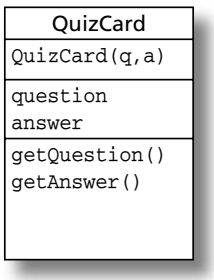
暗記カードプログラム



学生時代、勉強に暗記カードを使っていたという人も多いでしょう。表に問題、裏に答えを書いたカードのことです。何かを考えて理解するというような勉強にはあまり役立ちませんが、ともかく丸暗記すればいい、という場合には大変な武器になるものです。また、この暗記カードは、雑学知識を試すゲームなどにも使えます。

ここでは、暗記カードのコンピュータバージョンを作ってみましょう。この暗記カードプログラムは、以下のような3つのクラスから構成されます。

1) **QuizCardBuilder**
   暗記カードを作り、保存する。
2) **QuizCardPlayer**
   カードのデータをロードし、画面上に表示する
3) **QuizCard**
   「カードのデータ」を表すクラス。



QuizCardBuilder、QuizCardPlayerに関しては、そのコードがどのようなものになるかを解説します。また、QuizCardに関しては、右の図を基に、読者自らにコードを作成してもらうことになります。

QuizCardBuilder

Fileメニューには、画面への入力内容をテキストファイルに保存するためのSaveオプションが用意される

QuizCardPlayer

Fileメニューには、テキストファイルのデータを読み込み、画面に表示するためのLoadオプションが用意される

This PDF is prepared for tools@maruhouse.org, personal use only.

QuizCardBuilder クラスの概要

```
public class QuizCardBuilder {

  public void go() {
    // GUI の作成、表示
  }
```

GUIの作成、表示。イベントリスナーの登録も行う

内部クラス

```
  public class NextCardListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
      // 画面上のデータのリストへの追加。テキスト領域のリセット
    }
  }
```

ユーザーが「Next Card」ボタンをクリックした時（つまり、ユーザーが画面上のデータをカードリストに追加して、新たなカードの作成を始めることを希望した時）に実行されるイベントリスナー

内部クラス

```
  public class SaveMenuListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
      // File ダイアログボックスの表示
      // このダイアログボックスでカードセットの名前の設定、
      // カードの保存ができる
    }
  }
```

ユーザーがFileメニューから「Save」を選択した時（つまり、ユーザーがカードリスト中のすべてのカードをまとめて「カードセット」として保存することを希望した時）に実行されるイベントリスナー

内部クラス

```
  public class NewMenuListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
      // カードリストとテキスト領域のリセット
    }
  }
```

ユーザーがFileメニューから「New」を選択した時（つまり、ユーザーがカードリストとテキスト領域をリセットして新たなカードセットの作成を始めることを希望した時）に実行されるイベントリスナー

```
  private void saveFile(File file) {
      // カードリスト中の個々のカードの内容をテキストファイルに書き込む
      // 質問の部分と答えの部分は明確に分ける
  }
}
```

SaveMenuListenerによって呼び出され、データのファイルへの書き込みを行う

This PDF is prepared for tools@maruhouse.org, personal use only.

```
import java.util.*;
import java.awt.event.*;
import javax.swing.*;
import java.awt.*;
import java.io.*;

public class QuizCardBuilder {

  private JTextArea question;
  private JTextArea answer;
  private ArrayList<QuizCard> cardList;
  private JFrame frame;

  public static void main (String[] args) {
       QuizCardBuilder builder = new QuizCardBuilder();
       builder.go();
  }

  public void go() {
       // GUI の作成

       frame = new JFrame("Quiz Card Builder");
       JPanel mainPanel = new JPanel();
       Font bigFont = new Font("sanserif", Font.BOLD, 24);
       question = new JTextArea(6,20);
       question.setLineWrap(true);
       question.setWrapStyleWord(true);
       question.setFont(bigFont);

       JScrollPane qScroller = new JScrollPane(question);
       qScroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
       qScroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

       answer = new JTextArea(6,20);
       answer.setLineWrap(true);
       answer.setWrapStyleWord(true);
       answer.setFont(bigFont);

       JScrollPane aScroller = new JScrollPane(answer);
       aScroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
       aScroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

       JButton nextButton = new JButton("Next Card");

       cardList = new ArrayList<QuizCard>();

       JLabel qLabel = new JLabel("Question:");
       JLabel aLabel = new JLabel("Answer:");

       mainPanel.add(qLabel);
       mainPanel.add(qScroller);
       mainPanel.add(aLabel);
       mainPanel.add(aScroller);
       mainPanel.add(nextButton);
       nextButton.addActionListener(new NextCardListener());
       JMenuBar menuBar = new JMenuBar();
       JMenu .leMenu = new JMenu("File");
       JMenuItem newMenuItem = new JMenuItem("New");
```

GUI関連のコード。この種のコードについてはすでに解説済み。ただ、メニューバー、メニュー、メニュー項目等の扱いについては注意が必要

This PDF is prepared for tools@maruhouse.org, personal use only.

```
        JMenuItem saveMenuItem = new JMenuItem("Save");
        newMenuItem.addActionListener(new NewMenuListener());

        saveMenuItem.addActionListener(new SaveMenuListener());
        fileMenu.add(newMenuItem);
        fileMenu.add(saveMenuItem);
        menuBar.add(fileMenu);
        frame.setJMenuBar(menuBar);
        frame.getContentPane().add(BorderLayout.CENTER, mainPanel);
        frame.setSize(500,600);
        frame.setVisible(true);
}

public class NextCardListener implements ActionListener {
  public void actionPerformed(ActionEvent ev) {

        QuizCard card = new QuizCard(question.getText(), answer.getText());
        cardList.add(card);
        clearCard();
  }
}

public class SaveMenuListener implements ActionListener {
  public void actionPerformed(ActionEvent ev) {
        QuizCard card = new QuizCard(question.getText(), answer.getText());
        cardList.add(card);

        JFileChooser fileSave = new JFileChooser();
        fileSave.showSaveDialog(frame);
        saveFile(fileSave.getSelectedFile());
  }
}

public class NewMenuListener implements ActionListener {
  public void actionPerformed(ActionEvent ev) {
        cardList.clear();
        clearCard();
  }
}

private void clearCard() {
  question.setText("");
  answer.setText("");
  question.requestFocus();
}

private void saveFile(File file) {
  try {
        BufferedWriter writer = new BufferedWriter(new FileWriter(file));

        for(QuizCard card:cardList) {
        writer.write(card.getQuestion() + "/");
        writer.write(card.getAnswer() + "\n");
        }
        writer.close();

        } catch(IOException ex) {
        System.out.println("couldn't write the cardList out");
        ex.printStackTrace();
        }
  }
}
```

メニューバー、Fileメニューを作り、「New」、「Save」といった項目を作成する。メニューはメニューバーに組み込み、メニューバーはフレームに組み込む。メニュー項目はActionEventを発生させる

ダイアログボックスを表示し、ユーザーが「Save」をクリックするまで待機する。ダイアログボックス上での処理、ファイルの選択などはJFileChooserによって自動的に行われるので、コードは非常に簡単なものになる

ファイルへのデータ書き込みそのものはこのメソッドで行われる（このメソッドは、SaveMenuListenerのイベントハンドラによって呼び出される）。引数には、データを書き込むファイルに対応するオブジェクトを指定する（Fileクラスの詳細については次のページを参照）

データ書き込み処理の効率化のため、BufferedWriterとFileWriterとのチェーニングを行う（このことについては後述）

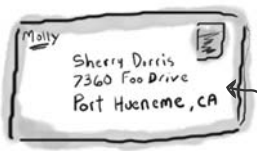
カードリスト（ArrayList）の要素を1つずつ書き込む。各カードのデータにそれぞれ1行を使う。質問と答えは"/"で区切る。答えの後には改行文字（\n）を入れる

This PDF is prepared for tools@maruhouse.org, personal use only.

java.io.File クラス

java.io.Fileクラスは、ディスク上のファイルに対応するクラスです。ただ、注意すべきなのは、あくまでファイル（あるいはディレクトリ）のパス名に対応するクラスであって、「内容」に対応するクラスではないということです。したがって、Fileクラスには、ファイルからデータを読み込む、ファイルにデータを書き込むといったことを行うためのメソッドはありません。Fileオブジェクトが便利なのは、文字列（String）のファイル名の代わりに使用できるという点です。たとえば、FileWriter、FileInputStreamなどのコンストラクタの引数として文字列のファイル名を指定する代わりに、Fileオブジェクトを指定するということもできます。まずFileオブジェクトを作り、それに対応するパス名が正当なものであることを確認してからコンストラクタの引数として指定する方が、直接文字列のファイル名を指定するより安全と言えるでしょう。

Fileオブジェクトはディスク上のファイル、ディレクトリの以下のようなパス名に対応する
/Users/Kathy/Data/GameFile.txt
しかし、ファイルのデータには対応せず、データへのアクセスのための機能もない



手紙の宛名は家そのものではない。Fileオブジェクトは、手紙で言えば宛名に対応するオブジェクト。ファイルの名前、保存場所に対応するがファイルそのものには対応しない

Fileクラスの使い方

① **すでに存在するファイルに対応するFileオブジェクトを作成する。**

```
File f = new File("MyCode.txt");
```

② **新規のディレクトリを作成する。**

```
File dir = new File("Chapter7");
dir.mkdir();
```

③ **ディレクトリの内容を表示する。**

```
if (dir.isDirectory()) {
  String [] dirContents = dir.list();
  for (int i = 0; i < dirContents.length; i++) {
    System.out.println(dirContents [i]);
  }
}
```

④ **ファイル、ディレクトリの絶対パスを表示する。**

```
System.out.println(dir.getAbsolutePath());
```

⑤ **ファイル、あるいはディレクトリを削除する（成功すればtrueが戻される）。**

```
boolean isDeleted = f.delete();
```

Fileオブジェクトは、ファイル名"GameFile.txt"に対応する

GameFile.txt

```
50,Elf,bow,sword,dust
200,Troll,bare hands,big ax
120,Magician,spells,invisibility
```

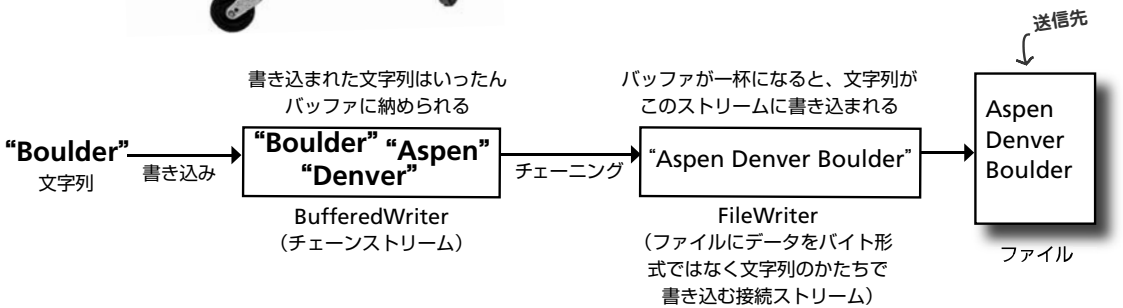
Fileオブジェクトはファイル中のデータには対応しない（また、ファイル中のデータへのアクセスのための機能も備えていない）

This PDF is prepared for tools@maruhouse.org, personal use only.

バッファを使うメリット

バッファはショッピングに使うカートのような役割を果たします。カートがなければ、品物を一つ一つ買って車まで運ぶようなことになり、非常に不便でしょう。

バッファがあれば、一定の限度内の量のデータを一時的にそこに蓄えておくことができ、書き込みがある度に逐一データをファイルに送るような手間は必要なくなる




```
BufferedWriter writer = new BufferedWriter(new FileWriter(aFile));
```

FileWriterオブジェクトへの参照が必要でない点に注意。使用するのはBufferedWriterオブジェクトのメソッド。また、BufferedWriterをクローズすれば、チェーニングされているFileWriterも自動的にクローズされる

バッファを使えば、使わない場合よりはるかに処理を効率化できます。FileWriterオブジェクトのwrite()メソッドを使用してファイルへのデータ転送を行うということも不可能ではありませんが、その場合はデータの書き込みがある度に逐一ファイルへのデータ転送も行うということになります。これは、何度も繰り返しディスクにアクセスするということで、メモリ上だけでデータを扱うのに比べてオーバーヘッドがはるかに大きくなってしまいます。その点、BufferedWriterとFileWriterとをチェーニングすれば、BufferedWriterに一定量のデータを一時的に（バッファが一杯になるまでの間）蓄えておくことができるので、ディスクへのアクセス回数を減らすことができます。

また、flush()メソッドを使用すれば、バッファが一杯になる前にデータをファイルに転送することも可能です。

This PDF is prepared for tools@maruhouse.org, personal use only.

テキストファイルから データを読み込む

テキストファイルからデータを読み込むという処理そのものはさほど難しくありませんが、ここでは、Fileオブジェクトと効率化のためにバッファを使った場合にそれがどのようになるかを簡単にまとめておきます。

読み取りは、whileループを使用して1行ずつ行っていきます。readLine()の戻り値がnullになった時、つまり読み取るデータがなくなった時にループは終了します。これは、ごく一般的なデータ読み取りの方法です(シリアライズしたオブジェクトのデータの場合は方法が異なる)。

テキストが1行書き込まれたファイル

```
What's 2+2?/4
What's 20+22/42
```

MyText.txt

インポートが必要

```
import java.io.*;

class ReadAFile {
  public static void main (String [] args) {

    try {
      File myFile = new File("MyText.txt");
      FileReader fileReader = new FileReader(myFile);

      BufferedReader reader = new BufferedReader(fileReader);

      String line = null;

      while ((line =reader.readLine()) != null) {
        System.out.println(line);
      }
      reader.close();

    } catch(Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

FileReaderはテキストファイルへの接続を行う接続ストリーム

FileReader と BufferedReader をチェーニングすることで読み込み処理を効率化できる。バッファが(データがすべて読み取られて)空になった時のみファイルにアクセスすればよいため

読み取った文字列を保持するための変数

readLine()の戻り値がnullになるまでの間、「テキストを1行読み取り、それをline変数に代入した後、画面に表示する」という処理を繰り返す
つまり、読み取る行が残っている間は、それを読み取り、画面に表示するということ

This PDF is prepared for tools@maruhouse.org, personal use only.

QuizCardPlayer クラスの概要

```
public class QuizCardPlayer {

  public void go() {
    // GUI の作成、表示
  }

  public class NextCardListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
      // その時点で表示されているのが問題であれば答えを表示し、答えであれば次の問題
      // を表示する。表示されているのが問題なのか答えなのかを示すフラグを設定
    }
  }

  public class OpenMenuListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
      // File ダイアログボックスを表示する。このダイアログボックスでユーザーは
      // 開くカードセットを選択する
    }
  }

  private void loadFile(File file) {
      // データをテキストファイルから読み取り、カードの ArrayList を作成する
      // このメソッドは、OpenMenuListener のイベントハンドラによって呼び出され、
      // ファイルのデータを 1 行ずつ読み取る
      // また、makeCard() を呼び出して、読み取ったデータを基に新たなカードを作成する
  }

  private void makeCard(String lineToParse) {
      // loadFile() メソッドによって呼び出される。引数には、テキストファイルから
      // 読み取られたデータが指定される。
      // このデータを問題と答えに分け、QuizCard オブジェクトを作成し、
      // CardList という ArrayList に組み込む
  }
}
```


This PDF is prepared for tools@maruhouse.org, personal use only.

```
import java.util.*;
import java.awt.event.*;
import javax.swing.*;
import java.awt.*;
import java.io.*;

public class QuizCardPlayer {

  private JTextArea display;
  private JTextArea answer;
  private ArrayList<QuizCard> cardList;
  private QuizCard currentCard;
  private int currentCardIndex;
  private JFrame frame;
  private JButton nextButton;
  private boolean isShowAnswer;

  public static void main (String[] args) {
       QuizCardPlayer reader = new QuizCardPlayer();
       reader.go();
  }

  public void go() {
       // GUI の作成

       frame = new JFrame("Quiz Card Player");
       JPanel mainPanel = new JPanel();
       Font bigFont = new Font("sanserif", Font.BOLD, 24);

       display = new JTextArea(10,20);
       display.setFont(bigFont);

       display.setLineWrap(true);
       display.setEditable(false);

       JScrollPane qScroller = new JScrollPane(display);
       qScroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
       qScroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);
       nextButton = new JButton("Show Question");
       mainPanel.add(qScroller);
       mainPanel.add(nextButton);
       nextButton.addActionListener(new NextCardListener());

       JMenuBar menuBar = new JMenuBar();
       JMenu fileMenu = new JMenu("File");
       JMenuItem loadMenuItem = new JMenuItem("Load card set");
       loadMenuItem.addActionListener(new OpenMenuListener());
       fileMenu.add(loadMenuItem);
       menuBar.add(fileMenu);
       frame.setJMenuBar(menuBar);
       frame.getContentPane().add(BorderLayout.CENTER, mainPanel);
       frame.setSize(640,500);
       frame.setVisible(true);

  } // go() メソッドの終了
```

ごく一般的なGUIコード。
特に解説の必要なし。

This PDF is prepared for tools@maruhouse.org, personal use only.

```
public class NextCardListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
      if (isShowAnswer) {
         // 問題はすでに表示されたので答えを表示する
         display.setText(currentCard.getAnswer());
         nextButton.setText("Next Card");
         isShowAnswer = false;
      } else {
         // 次の問題を表示
         if (currentCardIndex < cardList.size()) {

            showNextCard();

         } else {
            // カードがもうない場合
            display.setText("That was last card");
            nextButton.setEnabled(false);
         }
      }
    }
  }
```

boolean 型の変数（フラグ）であるisShowAnswerで、表示しているのが問題なのか答えなのかを確認し、その結果に応じた処理を行う

```
public class OpenMenuListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
      JFileChooser fileOpen = new JFileChooser();
      fileOpen.showOpenDialog(frame);
      loadFile(fileOpen.getSelectedFile());
    }
  }

private void loadFile(File file) {

  cardList = new ArrayList<QuizCard>();
  try {
      BufferedReader reader = new BufferedReader(new FileReader(file));
      String line = null;
      while ((line = reader.readLine()) != null) {
         makeCard(line);
      }
      reader.close();

      } catch(Exception ex) {
         System.out.println("couldn' t read the card file");
         ex.printStackTrace();
      }

    // 最初のカードを表示する
    showNextCard();
}

private void makeCard(String lineToParse) {
    String[] result = lineToParse.split("/");
    QuizCard card = new QuizCard(result[0], result[1]);
    cardList.add(card);
    System.out.println("made a card");
  }

  private void showNextCard() {
      currentCard = cardList.get(currentCardIndex);
      currentCardIndex++;
      display.setText(currentCard.getQuestion());
      nextButton.setText("Show Answer");
      isShowAnswer = true;
  }
} // クラスの終了
```

開くべきファイルを選ぶためのダイアログボックスを表示する

BufferedReaderとFileReaderをチェーニングする。FileReaderのコンストラクタの引数には、ユーザーがダイアログボックスで選択したファイルに対応するFileオブジェクトを指定する

ファイルから 1 行ずつ読み取り、それを引数に指定してmakeCard() メソッドを呼び出す。makeCard() メソッドでは、カードに対応するQuizCardオブジェクトが作られ、ArrayListに組み込まれる

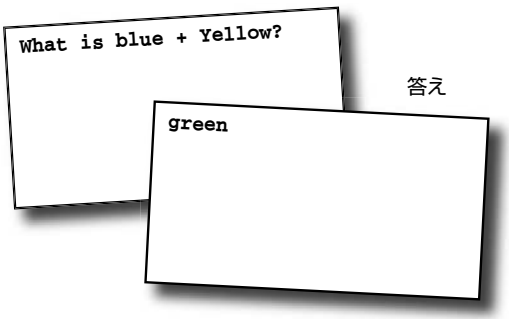
テキストファイルの各行は 1 枚のカードに対応するが、ここでは、問題と答えを別のものとして扱う。そのため、StringのSplit() メソッドを使って、行を 2 つの部分（一方は問題に、一方は答えに使う）に分けることになる。Split() メソッドについては次ページで詳述。

This PDF is prepared for tools@maruhouse.org, personal use only.

Sprit()メソッドの使用方法

カードは問題と答えに分かれる

問題



この場合、テキストファイルには次のようなデータが保存される

```
What is blue + Yellow?/green
What is red + blue?/purple
```

問題に使う文字列と答えに使う文字列をどのように分けるか

テキストファイルには、各カードのデータが1行にまとめて保存されますが、問題と答えは「/」で区切られます（QuizCardBuilderクラスのコードがそのようになっている）。

StringのSplit()メソッドを使えば文字列を分割できる

Sprit()メソッドには、「区切り文字」を指定されれば、それを境に文字列を分割する、という機能があります。

QuizCardPlayerのコードでは、ファイルから各行のデータをこの形式で読み取る

```
String toTest = "What is blue + yellow?/green";
String[] result = toTest.split( "/" );
  for (String token:result) {
System.out.println(token);
}
```

split()メソッドの引数として"/"を指定し、文字列を（ここでは）2つの部分に分割する（注意：split()の機能はここで紹介したものがすべてではない。フィルタ、ワイルドカードなどを伴う複雑な処理も可能）

ループを使用して配列の各要素を順に出力する。この場合、配列の要素は"What is blue + yellow?"と"green"の2つだけ

This PDF is prepared for tools@maruhouse.org, personal use only.

素朴な疑問に答えます

Q：java.ioパッケージにはあまりにも多くのクラスがありますが、自分がどれを使えばよいかをどのように見分ければいいのですか。

A：java.ioパッケージのクラスは、複数のストリームを組み合わせる「チェーニング」と呼ばれる手法に使うことが多くなっています。チェーニングによって組み合わせるのは、接続ストリームとチェーンストリーム（フィルタストリームとも呼ばれる）ですが、組み合わせ方のパターンにはさまざまなものが考えられ、また、3つ以上のストリームを組み合わせることもあり得ます。ともかく、必要となっている処理に最適な組み合わせをその都度考えるのです。

ただ、一般に使用されるクラスの種類はさほど多くありません。たとえば、テキストファイルへのデータの読み書きを行う場合には、BufferedReader、BufferedWriterとFileReader、FileWriterをチェーニングするのが普通でしょう。また、オブジェクトのシリアライズ、デシリアライズを行う場合なら、ObjectOutputStream、ObjectInputStreamとFileInputStream、FileOutputStreamをチェーニングすることになります（保存先にファイルを使用する場合）。

この章で紹介したクラスの組み合わせで、ほぼ90%のケースには対応できると考えて間違いはないでしょう。

Q：Javaのバージョン1.4で追加されたNew I/O（NIO）パッケージとはどのようなものですか。

A：NIO（java.nio）パッケージには、プログラムの動作するマシンのネイティブ環境を利用し、パフォーマンスを大幅に向上するのに役立つクラスが納められています。中でも重要なのが、バッファをダイレクトにコントロールする機能、ノンブロッキングI/Oの機能をサポートしていることです。ノンブロッキングI/Oというのは、読み取り/書き込みの対象となるデータが存在しない時に、読み取り/書き込み対象データの発生を待つことなく別の処理を進めることができる、という機能のことです。こうした機能は、FileInputStream、FileOutputStreamといった旧来のクラスと組み合わせて使うことができます。ただ、NIOパッケージのクラスは使い方が難しいので、どうしても必要な時を除き使わない方が無難でしょう。使い方を誤ると、かえってパフォーマンスの低下にもつながるからです。特に初心のプログラマの場合、NIOパッケージの使用がメリットになることはほとんどないと言っていいでしょう。

ただ、FileInputStreamのgetChannel()メソッド（バージョン1.4でFileInputStreamに追加されたメソッド）など、NIOパッケージのクラスを比較的簡単に利用する手段があることも確かです。

重要ポイント

- ■テキストファイルへのデータの書き込みを行う場合は、接続ストリームとしてFileWriterを使用する。
- ■FileWriterとBufferedWriterをチェーニングするのは、処理の効率化のため。
- ■Fileオブジェクトはファイルのパス名に対応する。ファイルそのもの（ファイルの中のデータ）に対応するわけではない。
- ■Fileオブジェクトは、ディレクトリの作成、移動、削除などにも使える。
- ■ストリームには、多くの場合、文字列のファイル名の代わりに、Fileオブジェクトを使用することができる。Fileオブジェクトを使う方が無難。
- ■テキストファイルからのデータの読み取りを行う場合は、接続ストリームとしてFileReaderを使用する。
- ■FileReaderとBufferedReaderをチェーニングするのは、処理の効率化のため。
- ■テキストファイル中のデータを分割するには、何らかの方法で「どこで分割するのか」を判別できるようにする必要がある。通常は、そのために「区切り文字」が使用される。
- ■区切り文字を境に文字列を分割するには、StringのSplit()メソッドが使用できる。区切り文字が1つであれば、文字列は2つに分割できる。区切り文字自身は2つの文字列のどちらにも属さない。

This PDF is prepared for tools@maruhouse.org, personal use only.

クラスのバージョン

ここまでの説明で、Javaの場合、I/O関連のコードを書くのがさほど難しくないということはわかったと思います。接続ストリームとチェーンストリームの組み合わせをいくつか覚えるだけでほとんどのケースに対応できるからです。ただ、その際、若干注意すべきこともあるので、ここではそのことについて触れておきましょう。

バージョンコントロールの問題

シリアライズしたオブジェクトを元に戻す（デシリアライズする）には、クラスファイルが必要になります。しかし、シリアライズしてから、デシリアライズするまでの間に、クラスに改造が加えられていたとしたらどうでしょうか。たとえば、Dogオブジェクトのインスタンス変数（transient宣言されていないもの）の型が知らない間にdoubleからStringに変わっていたとしたら……。Javaは型を厳密に扱う言語なので、これは大きな問題です。デシリアライズの際に問題となる改造は、型の変更だけではありません。それを次にまとめておきます。

デシリアライズ時に問題となる改造

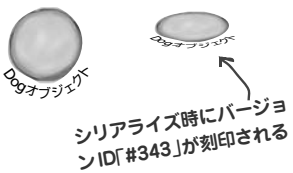
- 元々あったインスタンス変数をなくす
- インスタンス変数の型を変更する
- transient宣言されていなかったインスタンス変数をtransient宣言する
- 元々存在しなかったスーパークラスを作る、あるいは存在していたスーパークラスをなくす
- 関連するクラスのいずれかをシリアライズ可能なものからシリアライズ不可能なものに変える（つまり「implements Serializable」という記述を削除する）
- インスタンス変数をスタティック変数に変える

デシリアライズ時に通常は問題とならない改造

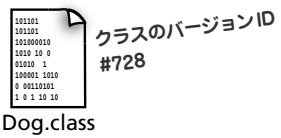
- 新たなインスタンス変数を追加する（デシリアライズ時には、新しいインスタンス変数にはデフォルト値が設定される）
- 下位のクラス、あるいは同じ階層のクラスを追加する
- 下位のクラス、あるいは同じ階層のクラスを削除する
- インスタンス変数のアクセスレベルを変更する（これによって、デシリアライズ時の変数への値の設定に影響することはない）
- インスタンス変数のtransient宣言をやめる（元々transient宣言されていた変数には、デシリアライズ時にデフォルト値が設定される）

① Dogクラスを作る。

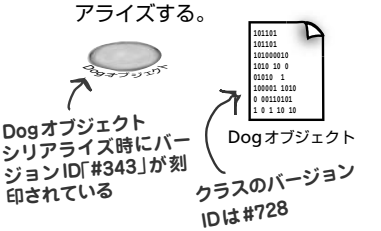
② Dogオブジェクトをシリアライズする。



③ Dogクラスを改造する。



④ 改造したクラスファイルを使ってDogオブジェクトをデシリアライズする。



⑤ デシリアライゼーションがエラーになる。
「古いバージョンのDogクラスのコードがないとデシリアライゼーションができない」という意味のメッセージが出る。

This PDF is prepared for tools@maruhouse.org, personal use only.

バージョンID

オブジェクトがシリアライズされる時には、オブジェクトそのものに関するデータとともに、デシリアライズ時に必要になるすべてのクラスのバージョンIDが記録されます。このバージョンIDは、「シリアルバージョンUID」と呼ばれ、システムによって自動的に生成されます。オブジェクトのデシリアライズの際、その時点でのクラスのバージョンIDが元と変わっていることがわかると、デシリアライズはエラーになります。ただ、このエラーの発生は回避することができます。

シリアライズ後にクラスが改造される可能性がある場合は、クラスのコードにバージョンIDを入れておく

オブジェクトのデシリアライズ時には、記録されているIDと、使用するクラスのIDとが比較されます。たとえば、Dogオブジェクトのシリアライズの際、DogクラスのIDが「23(実際のIDはもっと長くなります)」だったとします。デシリアライズ時には、この「23」と、その時点でシステムに存在するDogクラスのIDが比較されるわけです。2つのIDが一致しなければ、JVMがクラスのバージョンが異なると判断し、例外が発生します。

この例外が発生しないようにするには、クラスのコードの中であらかじめIDを指定しておくという方法が使えます。こうしておけば、たとえクラスが改造されても、IDは常に同じになるので、バージョン違いによってデシリアライゼーションができないということはなくなります。

ただ、この方法を使う際には、クラスの改造に常に気を配っていなくてはなりません。改造の仕方によっては問題が起きることもあるのですが、その問題にも自分で対処しなければならないのです。

クラスに指定するシリアルバージョンUIDは、JDKに納められているserialverというツールを使うことで取得できます。

```
File  Edit  Window  Help  serialKiller
% serialver Dog

Dog: static final long
serialVersionUID = -
5849794470654667210L;
```

クラスのコードの中でIDを指定する方法

1. serialverコマンドを実行し、クラスのバージョンIDを取得する。

```
File  Edit  Window  Help  serialKiller
% serialver Dog

Dog: static final long
serialVersionUID =-
5849794470654667210L;
```

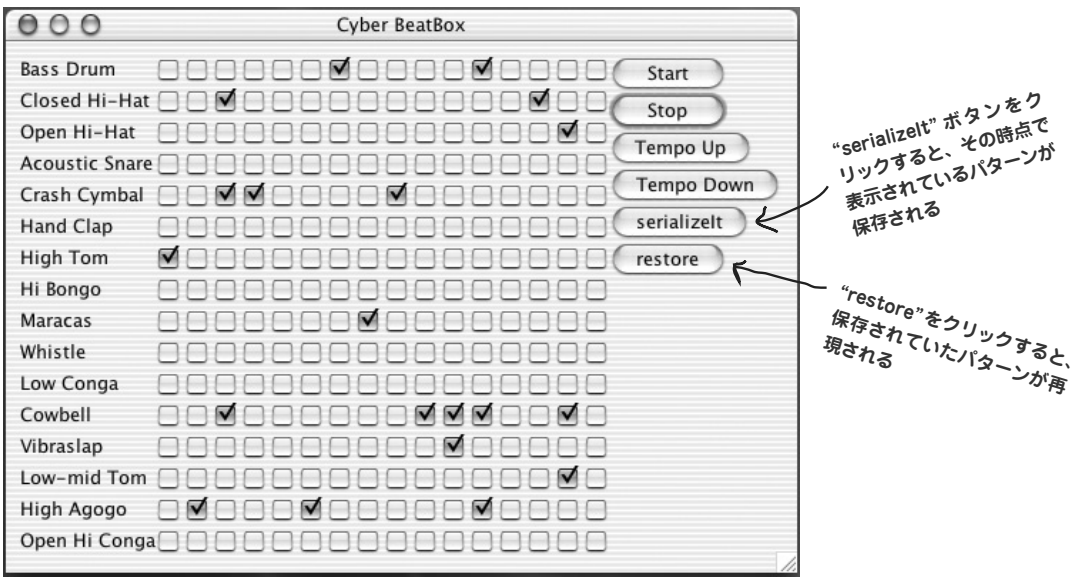
2. 取得できたIDを以下のようなコードで指定する(1の出力結果からコピー＆ペーストするとよい)。

```
public class Dog {

  static final long serialVersionUID =
  -5849794470654667710L;

  private String name;
  private int size;
  // メソッドのコード
}
```

3. このようにIDを指定した場合は、クラスの改造に常に注意を払う必要がある。改造の仕方によっては問題が発生するが、その問題にも対処しなくてはならない。たとえば、Dogオブジェクトのシリアライズ後、Dogクラスに新しいインスタンス変数を追加した場合、その変数にはデシリアライズ時にデフォルト値が自動的に設定されることになるので、デフォルト値が設定された場合の動作がどうなるかをよく確認する。

This PDF is prepared for tools@maruhouse.org, personal use only.

プログラム・キッチン



この章では、BeatBoxプログラムに、ドラムパターンを保存し、後で再現する機能を持たせる方法を解説します。

This PDF is prepared for tools@maruhouse.org, personal use only.

ドラムパターンの保存

BeatBoxプログラムのドラムパターンは、突き詰めれば「チェックボックスの集まり」に過ぎません。256個のチェックボックスの状態のみによって、どのようなパターンを演奏するかが決められるからです。したがって、「ドラムパターンを保存する」というのは、この場合、「チェックボックスの状態を保存する」というのと同義である、ということになります。

256個のチェックボックスの状態を保存するためには、まず、boolean型の配列を用意します。配列はオブジェクトであり、他のオブジェクトと同様にシリアライズが可能です。ただ、そのためには配列の要素がシリアライズ可能なものでなければなりませんが、要素がboolean型の値であればまったく問題はありません。

逆に、保存したパターンを再現するには、シリアライズした配列をデシリアライズし、配列を元に戻します。BeatBoxプログラムのコードの大部分はすでに紹介したので、この章では、配列の保存と再現に関係する部分だけについて見ていくことにします。

ここで、ファイルにデータを書き込む、ファイルからデータを読み取る、といったテクニックについて理解しておけば、次章で、データをネットワーク経由でサーバに送る、サーバからデータを受け取るといったテクニックを学ぶ際に役立つでしょう。

ドラムパターンをシリアライズするためのコード

このコードはBeatBoxの内部クラスとなる

```
public class MySendListener implements ActionListener {

  public void actionPerformed(ActionEvent a) {

    boolean [] checkboxState = new boolean [256];

    for (int i = 0; i < 256; i++) {

      JCheckBox check = (JCheckBox) checkboxList.get(i);
      if (check.isSelected()) {
        checkboxState [i] = true;
      }
    }

    try {
      FileOutputStream fileStream = new FileOutputStream(new File("Checkbox.ser"));
      ObjectOutputStream os = new ObjectOutputStream(fileStream);
      os.writeObject(checkboxState);
    } catch(Exception ex) {
      ex.printStackTrace();
    }

  } // メソッドの終了
} // 内部クラスの終了
```

ユーザーが"serializeIt"ボタンをクリックした時に呼び出される

チェックボックスの状態を保存するためのboolean型の配列を作成

チェックボックスのリスト(ArrayList)中の個々のチェックボックスの状態を確認し、その結果を配列に格納

配列をシリアライズし、データをファイルに書き込むためのコード。この章を読んでいれば簡単に理解できるはず！

This PDF is prepared for tools@maruhouse.org, personal use only.

ドラムパターンの再現

ドラムパターンの再現のための処理では、保存のための処理のほぼ逆のことをすることになります。まず、ファイルに納められたデータを読み取り、それをboolean型の配列に格納します。そして、配列の内容を基にチェックボックスの状態を決めていくのです。再現の処理は、ユーザーが“restore”ボタンをクリックした時に行われるようにします。

ドラムパターンをデシリアライズするためのコード

このコードはBeatBoxの内部クラスとなる

```
public class MyReadInListener implements ActionListener {

  public void actionPerformed(ActionEvent a) {
    boolean [] checkboxState = null;
    try {
      FileInputStream fileIn = new FileInputStream(new File("Checkbox.ser"));
      ObjectInputStream is = new ObjectInputStream(fileIn);
      checkboxState = (boolean [])is.readObject();

    } catch(Exception ex){ex.printStackTrace();}

    for (int i = 0; i < 256; i++) {
      JCheckBox check = (JCheckBox) checkboxList.get(i);
      if (checkboxState [i]) {
        check.setSelected(true);
      } else {
        check.setSelected(false);
      }
    }

    sequencer.stop();
    buildTrackAndStart();

  } // メソッドの終了
} // 内部クラスの終了
```

ファイルに保存されたデータを読み取り、それをキャストしてboolean型の配列に再現する（readObject()はObject型のデータを戻すためキャストが必要）

リスト（ArrayList）中の個々のチェックボックスをシリアライズ時の状態に戻す

演奏が行われていればストップし、ArrayList中のチェックボックスの状態を基に新たなSequenceオブジェクトを作成する

自分で考えてみよう

ここで紹介したコードは機能的にあまり優れているとは言えません。“serializeIt”ボタンをクリックすると、自動的にCheckbox.serファイルにデータが保存されるのですが（ファイルが存在しなければ作られる）、このファイルは毎回上書きされてしまうのです。

この点を改良するには、JFileChooserを使用します。それによって保存先のファイルを毎回ユーザーが指定できるようにし、ドラムパターンを何種類でも保存できるようにするのです。ファイル名の指定も行えるようにすることが必要でしょう。コードに具体的にどのような修正を加えればよいか考えてみてください。

This PDF is prepared for tools@maruhouse.org, personal use only.

自分で考えてみよう

シリアライズすべきオブジェクト、すべきでないオブジェクト

以下に示すオブジェクトがそれぞれシリアライズすべきものかどうかを答えてください。もし、シリアライズすべきでないとしたら、なぜそうなのかその理由も答えてください。理由としては、シリアライズしても意味がない。セキュリティ上危険である、次回以降のプログラム実行時に同じオブジェクトを再現しても正しく機能しない恐れがある、といったものが考えられるでしょう（APIのドキュメントは見ずに考えること）

オブジェクト	シリアライズすべきか	シリアライズすべきでない理由
Object	すべき/すべきでない	
String	すべき/すべきでない	
File	すべき/すべきでない	
Date	すべき/すべきでない	
OutputStream	すべき/すべきでない	
JFrame	すべき/すべきでない	
Integer	すべき/すべきでない	
System	すべき/すべきでない	

正しいコードはどれ？

右に示したコードのうち、正しいもの（コンパイルできるもの）に丸を付けてください

```
FileReader fileReader = new FileReader();
BufferedReader reader = new BufferedReader(fileReader);
```

```
FileOutputStream f = new FileOutputStream(new File("Foo.ser"));
ObjectOutputStream os = new ObjectOutputStream(f);
```

```
BufferedReader reader = new BufferedReader(new FileReader(file));
String line = null;
while ((line = reader.readLine()) != null) {
  makeCard(line);
}
```

```
ObjectInputStream is = new ObjectInputStream(new FileOutputStream("Game.ser"));
GameCharacter oneAgain = (GameCharacter) is.readObject();
```


This PDF is prepared for tools@maruhouse.org, personal use only.

エクササイズ

○か×か

次の文は、いずれもこの章で学んだことがらについて述べたものです。正しいものに○を、誤っているものに×をつけてください。

1. シリアライゼーションによって保存したデータは、Java以外の言語のプログラムで使用するのに適している。
2. オブジェクトのステートは、シリアライゼーションによってのみ保存できる。
3. ObjectOutputStreamはオブジェクトのシリアライゼーションに使用するクラスである。
4. チェーンストリームは、単独で使用することも、接続ストリームと組み合わせて使用することもできる。
5. writeObject()メソッドは一度呼び出すだけで複数のオブジェクトを保存できる。
6. オブジェクトはすべてデフォルトでシリアライズ可能になっている。
7. transient宣言するとインスタンス変数はシリアライズ可能になる。
8. スーパークラスがシリアライズ不可能だとサブクラスもシリアライズ不可能になる。
9. 複数のオブジェクトのデータが保存されている場合、その読み取りは、データを保存した時と逆の順序で行わねばならない。
10. オブジェクトのデシリアライズ時、コンストラクタは実行されない。
11. オブジェクトのシリアライゼーションの処理、テキストファイルへのデータの書き込みの処理は、いずれも例外を発生させる恐れがある。
12. BufferedWriterとFileWriterはチェーニングできる。
13. Fileオブジェクトは、ファイル名には対応するが、ディレクトリ名には対応しない。
14. バッファ中のデータはバッファが一杯になるまで絶対に転送できない。
15. テキストファイルへのデータの書き込み、テキストファイルからのデータの読み取りにはいずれもバッファが使用できる。
16. StringのSplit()メソッドを使用した場合、区切り文字自身も分割によってできた文字列のいずれかに属することになる。
17. オブジェクトのシリアライズ後にクラスを改造すると、その改造がどのようなものであってもデシリアライゼーションが正しくできなくなる。

This PDF is prepared for tools@maruhouse.org, personal use only.

プログラミングマグネット

バラバラになったJavaプログラムを並べ替え、下のような実行結果が得られるようにしてください。もし、{ 、} などが抜けているところがあれば付け加えてもかまいません（すべてのコードを使わない可能性もあります。同じコードを2回使うこともできます）。

```
class DungeonGame implements Serializable {
try {
FileOutputStream fos = new
FileOutputStream("dg.ser");
short getZ() {
return z;
e.printStackTrace();
oos.close();
ObjectInputStream ois = new
  ObjectInputStream(fis);
int getX() {
  return x;
System.out.println(d.getX()+d.getY()+d.getZ());
FileInputStream fis = new
  FileInputStream("dg.ser");
public int x = 3;
transient long y = 4;
private short z = 5;
long getY() {
  return y;
class DungeonTest {
ois.close();
import java.io.*;
fos.writeObject(d);
} catch (Exception e) {
d = (DungeonGame) ois.readObject();
ObjectOutputStream oos = new
  ObjectOutputStream(fos);
oos.writeObject(d);
public static void main(String [] args) {
  DungeonGame d = new DungeonGame();
```

```
File  Edit  Window  Help  Torture
% java DungeonTest
12
8
```

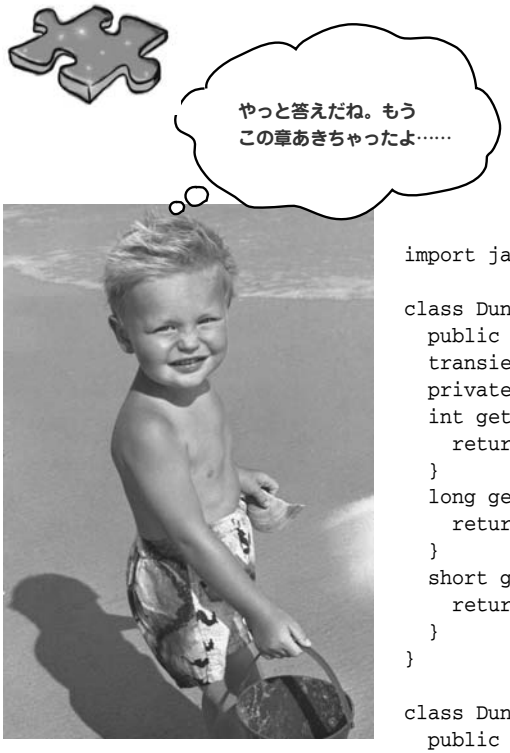

This PDF is prepared for tools@maruhouse.org, personal use only.

エクササイズの答え

○か×か

1. シリアライゼーションによって保存したデータは、Java以外の言語のプログラムで使用するのに適している。 ×
2. オブジェクトのステートは、シリアライゼーションによってのみ保存できる。 ×
3. ObjectOutputStreamはオブジェクトのシリアライゼーションに使用するクラスである。 ○
4. チェーンストリームは、単独で使用することも、接続ストリームと組み合わせて使用することもできる。 ×
5. writeObject()メソッドは一度呼び出すだけで複数のオブジェクトを保存できる。 ○
6. オブジェクトはすべてデフォルトでシリアライズ可能になっている。 ×
7. transient宣言するとインスタンス変数はシリアライズ可能になる。 ×
8. スーパークラスがシリアライズ不可能だとサブクラスをシリアライズ可能にすることができない。 ×
9. 複数のオブジェクトのデータが保存されている場合、その読み取りは、データを保存した時と逆の順序で行わねばならない。 ×
10. オブジェクトのデシリアライズ時、コンストラクタは実行されない。 ○
11. オブジェクトのシリアライゼーションの処理、テキストファイルへのデータの書き込みの処理は、いずれも例外を発生させる恐れがある。 ○
12. BufferedWriterとFileWriterはチェーニングできる。 ○
13. Fileオブジェクトは、ファイル名には対応するが、ディレクトリ名には対応しない。 ×
14. バッファ中のデータはバッファが一杯になるまで絶対に転送できない。 ×
15. テキストファイルへのデータの書き込み、テキストファイルからのデータの読み取りにはいずれもバッファが使用できる。 ○
16. StringのSplitメソッドを使用した場合、区切り文字自身も分割によってできた文字列のいずれかに属することになる。 ×
17. オブジェクトのシリアライズ後にクラスを改造すると、その改造がどのようなものであってもデシリアライゼーションが正しくできなくなる。 ×

This PDF is prepared for tools@maruhouse.org, personal use only.




```
import java.io.*;

class DungeonGame implements Serializable {
  public int x = 3;
  transient long y = 4;
  private short z = 5;
  int getX() {
    return x;
  }
  long getY() {
    return y;
  }
  short getZ() {
    return z;
  }
}

class DungeonTest {
  public static void main(String [] args) {
    DungeonGame d = new DungeonGame();
    System.out.println(d.getX() + d.getY() + d.getZ());
    try {
      FileOutputStream fos = new FileOutputStream("dg.ser");
      ObjectOutputStream oos = new ObjectOutputStream(fos);
      oos.writeObject(d);
      oos.close();
      FileInputStream fis = new FileInputStream("dg.ser");
      ObjectInputStream ois = new ObjectInputStream(fis);
      d = (DungeonGame) ois.readObject();
      ois.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
    System.out.println(d.getX() + d.getY() + d.getZ());
  }
}
```

File Edit Window Help Escape

```
% java DungeonTest
12
8
```


This PDF is prepared for tools@maruhouse.org, personal use only.

This PDF is prepared for tools@maruhouse.org, personal use only.

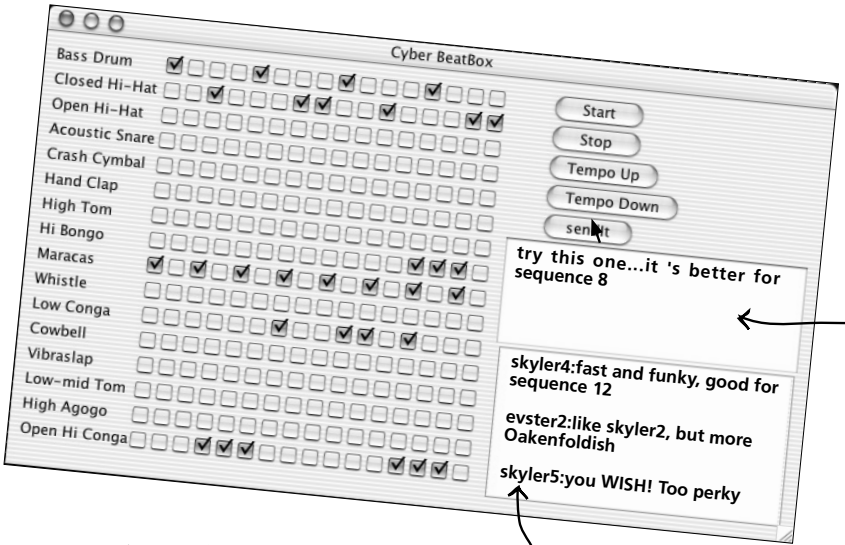
15章　ネットワークとスレッド

ネットワーク接続のための プログラムを書く

Javaではネットワーク接続のためのプログラムも簡単に書くことができます。複雑な、いわゆる「低水準（low-level）の」処理は、java.netライブラリのクラスによってすべて自動的に行われるからです。その他、ネットワーク経由でのデータ送受信を単なるI/Oのように扱うことができる、というのもJavaの大きな特徴でしょう。ただ、同一マシン上のI/Oとは使用する接続ストリームが若干異なるだけです。データの読み取りにはやはりBufferedReaderが使用できます。BufferedReaderにとっては、データがファイルから来たものであろうと、イーサネットケーブルを流れてきたものであろうとまったく関係がないのです。この章では、ネットワーク経由でのデータ送受信、そして、複数の処理を同時に行うために必要な「スレッド」という概念について解説します。この章を最後まで読めば、BeatBoxプログラムに、ドラムパターンをサーバとの間でやりとりできる「マルチスレッドチャットクライアント」としての機能が追加できるようになるでしょう。

This PDF is prepared for tools@maruhouse.org, personal use only.

チャットクライアントとチャットサーバ

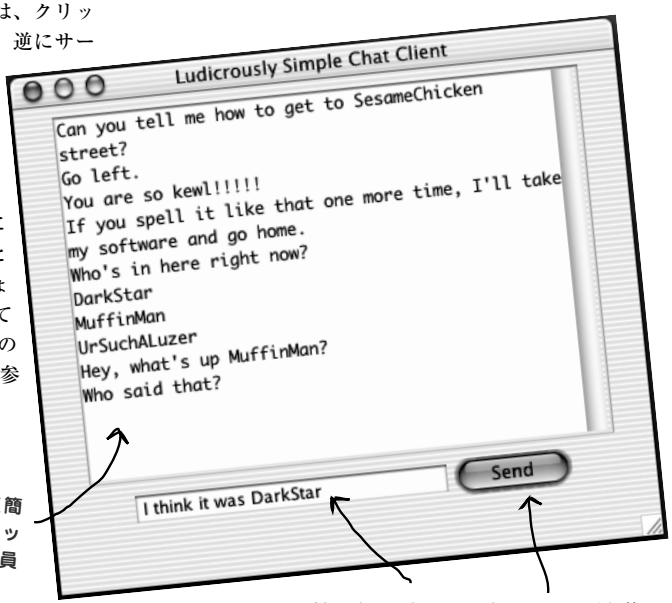


ここにメッセージを入力し、"sendIt"ボタンをクリックすると、ドラムパターンとメッセージが送信できる

受信したメッセージをクリックすると、パターンがロードされ、再生される

この章では、右に示すようなごく簡単なチャットクライアント、チャットサーバのプログラムを例にとり、ネットワーク経由でのデータ送受信、スレッドといったことがらについて解説していきます。チャットクライアントプログラムでは、クリック1つでテキストのメッセージをサーバに送ったり、逆にサーバから(他のユーザーから送られた)テキストのメッセージを受け取ったりということができます。主に取りあげるのはこのクライアントプログラムの方で、サーバについては簡単に説明するにとどめます。

この章の内容を理解すれば、BeatBoxプログラムに、ドラムパターンを(メッセージとともに)サーバに送信する、サーバからドラムパターンを受信する、といった機能を追加することができるようになるでしょう(上記の図を参照)。BeatBoxプログラム、そしてBeatBoxプログラムとデータのやりとりをするサーバの完成版のコードがどのようなものになるかは付録Aを参照してください。



テキストをやりとりする、ごく簡単なチャットプログラム。メッセージは、チャットの参加者全員に送られることになる

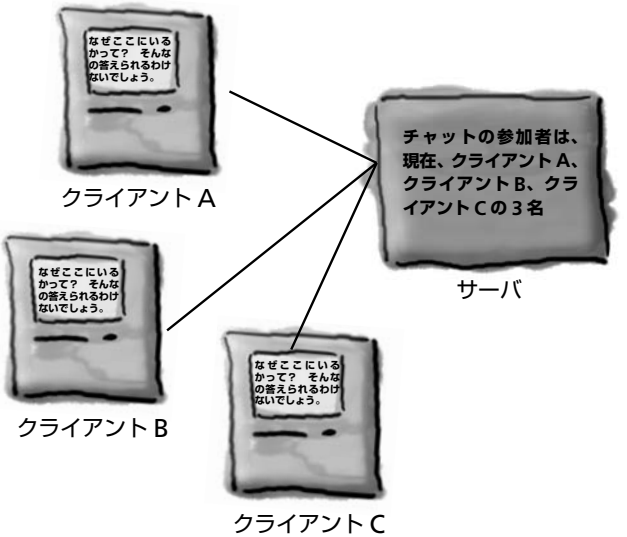
ボタンクリックでサーバにメッセージを送る

This PDF is prepared for tools@maruhouse.org, personal use only.

チャットプログラムの概要

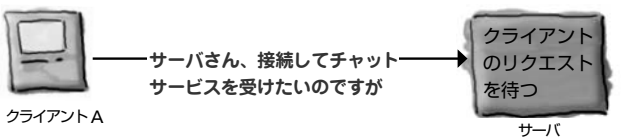
クライアントはサーバとのみやりとりする

サーバはすべてのクライアントとやりとりする

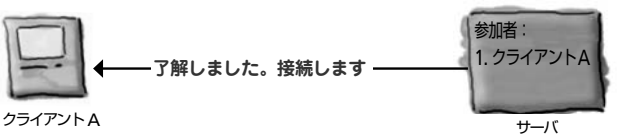


チャットプログラムの動作

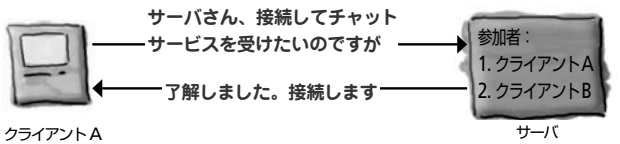
❶ クライアントがサーバに接続をリクエストする。



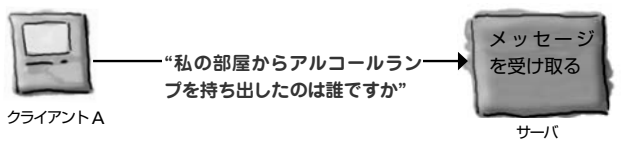
❷ サーバが接続を確立し、クライアントを参加者のリストに加える。



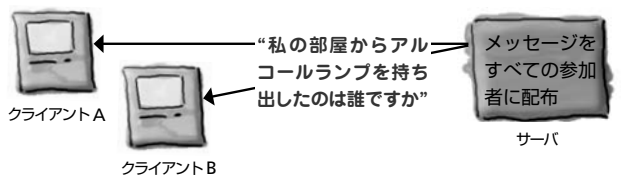
❸ 別のクライアントがサーバに接続をリクエストする。



❹ クライアント A がサーバにメッセージを送る。



❺ サーバが、受け取ったメッセージをすべての参加者（メッセージの元々の送信者も含む）に配布する。



This PDF is prepared for tools@maruhouse.org , personal use only.

接続、送信、受信

チャットクライアントが行うべきことは、主に次の3つです。

1) サーバとの間の接続を確立する
2) サーバにメッセージを送信する
3) サーバからメッセージを受信する

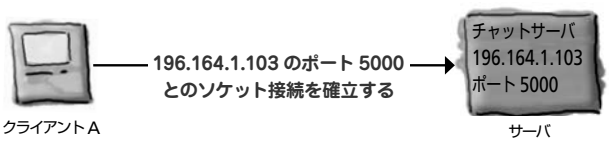
上記のような処理を行うプログラムをゼロから書こうとすれば、本来、大変な困難が伴うはずです。しかし、幸いなことに、Java APIのjava.netパッケージには、こまごまとした「低水準」の処理を行うためのクラスがすでに用意されています。こうしたクラスを使用すれば、ネットワーク関連のプログラミングは非常に容易になります。チャットクライアントのプログラムでも、ネットワークやI/Oに関するコードよりGUIに関するコードの方がむしろ多くなるくらいです。

では、ここで紹介するチャットクライアントには何も難しいところがないか、というとそうでもありません。

このチャットクライアントには、「2つの処理を同時に行う」という、本書でこれまでに取りあげたプログラムにはない特徴があるからです。サーバとの間の接続を確立した後は、サーバへのメッセージの送信と、サーバからのメッセージの受信を同時に行わねばならない可能性があるのです。このことについては、ネットワーク関連のコードの書き方について説明した後に触れることにします。

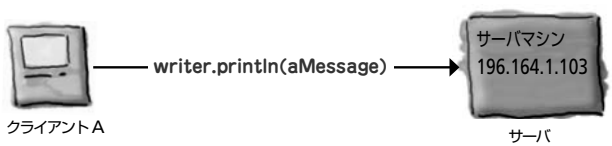
❶ 接続

サーバとの間に接続（「ソケット接続」とも呼ばれる）を確立する。



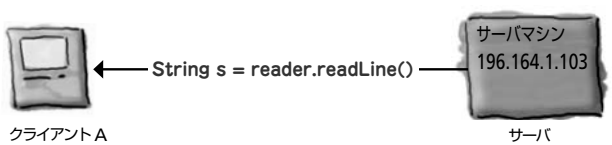
❷ 送信

サーバにメッセージを送信する。



❸ 受信

サーバからメッセージを受信する。



This PDF is prepared for tools@maruhouse.org , personal use only.

ネットワーク（ソケット）接続を確立する

ネットワーク接続（ソケット接続）の際、Javaでは、Socketクラス（java.net.Socket）というクラスを使用します。Socketクラスは、2つのマシンの間の接続を表すクラスです。2つのマシンが接続される、というのは、言い換えれば、2つのマシン上のソフトウェアがお互いに相手の存在を認識し、コミュニケーションがとれるようになる、ということです。また、「コミュニケーション」とは、この場合、データを送り合うことを意味します。

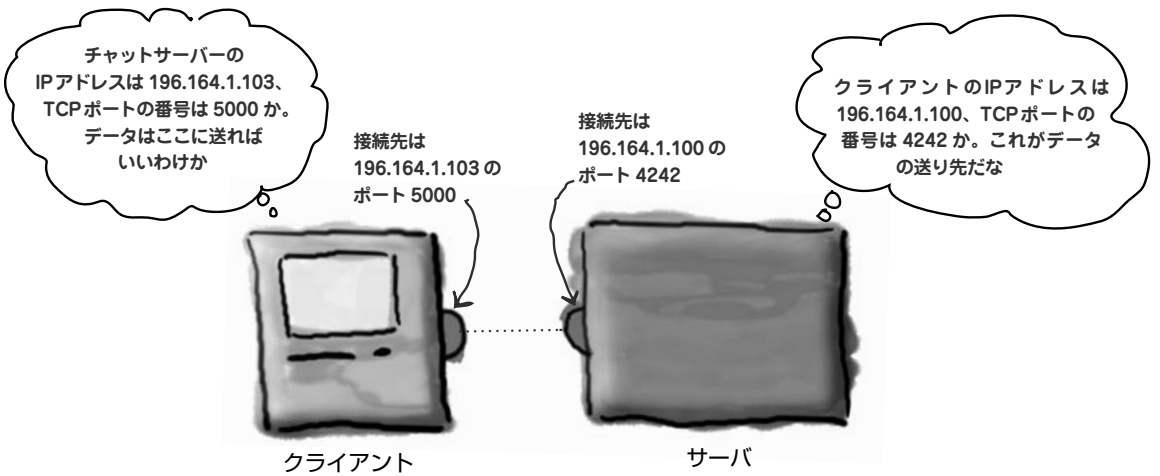
すでに述べたとおり、ネットワーク接続に伴うこまごまとした処理は、java.netパッケージのクラスを使えばすべて目に見えないところで自動的に行われるため、プログラマがそれについて考える必要はありません。また、JVMやOS、イーサネットケーブルをはじめとするハードウェアなどがネットワーク接続に際してどのように機能するかといったことも考えなくてかまいません。プログラマは、ネットワーク接続のために、以下に示すような驚くほど簡単なコードを書けばよいのです。

ソケット接続の確立には、接続先のサーバに関して、IPアドレスとTCPポートの番号という2つの情報が必要

TCPポートの番号

```
Socket chatSocket = new Socket("196.164.1.103", 5000);
```

サーバのIPアドレス



ソケット接続が行われると、2つのマシンは、互いについての情報（ネットワーク上での位置を示すIPアドレス、TCPポート番号など）を得ることになる

This PDF is prepared for tools@maruhouse.org, personal use only.

TCPポート番号もアドレスの一部

マシン上で動作するプログラムに対応する16ビットの数値

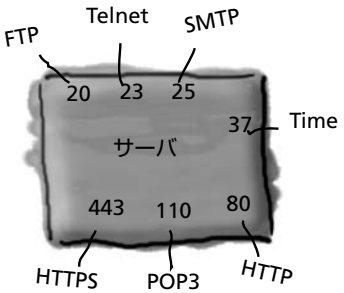
ネットワーク接続というのは厳密にはマシン（ハードウェア）ではなく、ソフトウェア（プログラム）に対して行うものです。したがって接続に際しては、対象となるマシンとソフトウェアを指定しなくてはなりません。マシンの指定にはIPアドレスが、ソフトウェアの指定にはポート番号が使用されます。ポート番号は16ビットの数値で、どのソフトウェアとどの番号が対応するかは基本的にプログラマが決めるのですが、中には、Web（HTTP）サーバの80、Telnetサーバの23、FTPサーバの20、POP3メールサーバの110、SMTPサーバの25、Timeサーバの37など、あらかじめ対応するポート番号が規格で決まっているソフトウェアもあります（この番号は変えてもよい）。このようにあらかじめ決まっているポート番号を「ウェルノウンポート」といいます。勘違いしやすいのですが、マシンに「ポート」という接続用の場所（部品）があって、それに番号がついている、というわけではないので注意してください。ポート番号はあくまで単にソフトウェアを区別するための番号にすぎません。ポート番号には0から65535までの数字が使われます。

ポート番号がなければ、接続をどのソフトウェアに対して行えばよいのかわかりません。ソフトウェアはそれぞれ「プロトコル（動作のルール）」が異なるので、もし、接続が誤ったソフトウェアに対して行われてしまえば、接続をしても何の意味もないのです。たとえば、WebブラウザがHTTPサーバではなく、POP3サーバと接続されてしまったとしたらどうでしょうか。POP3メールサーバはブラウザから送られるHTTPリクエストを理解できません。たとえ理解できたとしてもWebサービスを提供することはまったくできないでしょう。

サーバプログラムを自ら書く場合は、そのプログラムをどのポート番号と対応させるかを指定します（Javaでそれをどのように行うのかは後述）。ここで例に使用するチャットサーバのポート番号は「5000」にしてあります。この番号に特に意味はありません。適当に選んだだけです。ただ、プログラマが自分でポート番号を選ぶ際は、0から1023までの数字は使えないので注意してください。この範囲の番号はウェルノウンポートに使うと決められているからです。

自分で作るサーバプログラムを会社のネットワークで使用するという場合は、使おうと考えているポート番号がすでに社内の誰か他の人に使われていないかをシステム管理者に確認しなくてはなりません。会社によっては、たとえば「3000より小さい数字は使ってはいけない」などの独自のルールが存在することもあり得ます。どんなネットワークであれ、「自分以外にユーザーがいない」という場合を除けば、自由にポート番号を選んでいいというようなことはあり得ないと考えてください。

主なウェルノウンポート



ポート番号には0か65535までの数字が使える

0から1023までのTCPポート番号は、ウェルノウンポートに使うと定められている（これを「予約されている」という）。プログラマが独自に作るプログラムに使うことはできない＊

ここで紹介するチャットサーバーはポート番号「5000」を使用する。これは、1024から65535までの数字の中から適当に選んだもの

＊ 無理に使えば使えないことはありませんが、システム管理者の大変な怒りを買うことは間違いないでしょう

This PDF is prepared for tools@maruhouse.org, personal use only.

素朴な疑問に答えます

Q：接続するサーバプログラムのポート番号はどうすればわかるのですか。

A：それはサーバプログラムの種類によって異なってきます。前ページで紹介したような、ウェルノウンポートの割り当てられたサーバ（HTTP、SMTP、FTPなど）であれば、インターネットで調べれば（Googleなどで「ウェルノウンTCPポート」といったキーワードを検索する）、すぐにポート番号がわかるでしょう。システム管理者に知り合いがいれば、その人に聞いてもいいかもしれません。
しかし、サービスプログラムがHTTPサーバなどのように広く使われているものでない場合は、開発者を突き止めてポート番号を確認する必要があるでしょう。通常、ネットワークでサービスを提供するプログラムを作った人であれば、当然、クライアントを作ってもらいたいので、何らかのかたちでIPアドレスやポート番号、サービスのプロトコルなどを公開しているはずです。公開の手段としてはそのためのWebサイトを設けるというのが一般的でしょう。また、開発者以外の有志が同種のサイトを作っている可能性もあります。

Q：複数のプログラムに同じポート番号を重複して割り当てるということはあるのですか。つまり、同じマシン上の2つ以上のプログラムのポート番号が同じということがあり得るか、ということですが……。

A：あり得ません。もし、すでにいずれかのプログラムに割り当てられているポート番号を他のプログラムに割り当てようとすると、BindException という例外が発生します。ポート番号の割り当ては、プログラムの起動時に行うことになります。その具体的な方法については、この章の中で後に触れます。

IPアドレスはショッピングモールの名前

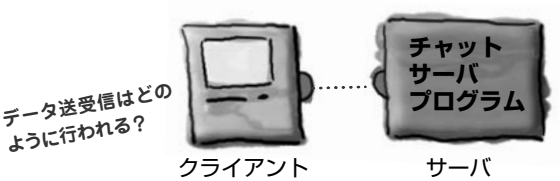
IPアドレスは数多くの店舗が並んだショッピングモールの名前のようなもの

ポート番号は特定の店舗名

ポート番号は、モールの中の特定の店舗の名前のようなもの

頭の体操

クライアントとサーバが接続され、どちらもお互いのIPアドレスとポート番号を認識している、という状態になったとします。その後、両者の間でデータを送り合うとすると、Javaではどのようなコードを書くことになると思いますか。バイト形式でデータを送受信する場合と、テキストデータを送受信する場合とでどのような違いがあるでしょうか。少し考えてみてください。

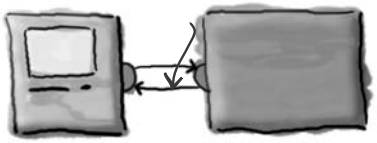


This PDF is prepared for tools@maruhouse.org, personal use only.

ソケット接続経由でのデータの読み取りにもBufferedReaderが使用できる

前章で「ストリーム」について解説しましたが、このストリームはソケット接続経由でのデータの送受信にも使用できます。中でも、データをやりとりする相手がファイルだろうが、ネットワークを隔てた他のマシンだろうが、「高水準」のチェーンストリームは常に同じように使える、というのはJavaの大きな特徴と言えるでしょう。たとえば、BufferedReaderのようなストリームの使い方は、ファイルからデータを読み取る場合にも、ネットワーク経由でデータを読み取る場合にも違いはないのです。

ソケット接続経由でデータを送受信する入力、出力ストリーム



❶ サーバへのソケット接続を確立する

```
Socket chatSocket = new Socket("127.0.0.1", 5000);
```

チャットサーバーのポート番号。先述のとおり、ここでは「5000」にする

127.0.0.1はローカルホスト、つまりこのプログラム自身が動作しているのと同一のマシンのIPアドレス。テストなどで便宜上1台のマシンをクライアントとサーバの両方に使う際にこのようなアドレスを指定することがある

❷ InputStreamReaderを、ソケット接続に対応する接続ストリームとチェーニングする

```
InputStreamReader stream = new InputStreamReader(chatSocket.getInputStream());
```

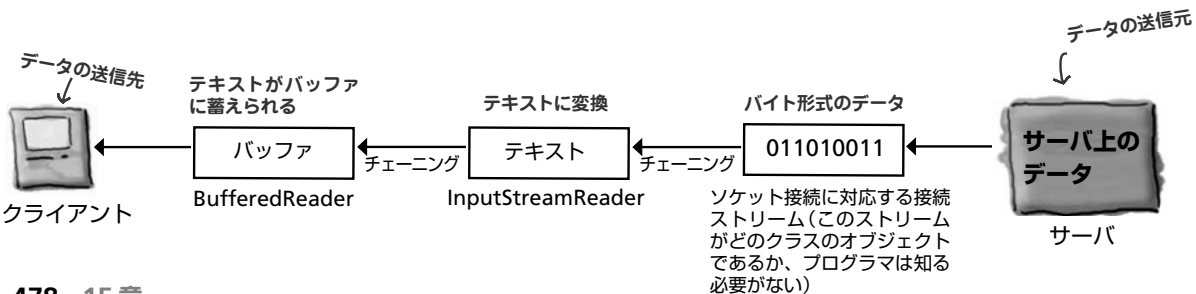
InputStreamReaderは、低水準の接続ストリームと高水準のチェーンストリーム（BufferedReaderなど）の橋渡しの役割を果たす

このコードによってソケット接続に対応する接続ストリームを取得する。この接続ストリームはより使いやすい高水準のチェーンストリームとチェーニングする

❸ BufferedReaderオブジェクトを作成し、読み取りを行う

BufferedReaderとInputStreamReader（すでにソケットに対応する接続ストリームとチェーニング済み）とをチェーニングする

```
BufferedReader reader = new BufferedReader(stream);
String message = reader.readLine();
```



This PDF is prepared for tools@maruhouse.org, personal use only.

ソケット接続経由でのデータ書き込みには PrintWriterを使用する

ソケット接続経由でのデータ書き込みの場合、前章のようにBufferedWriterを使用する方法と、PrintWriterを使用する方法が選択できます。ただ、まとまった文字列を一度に書き込む場合は、後者の方法を選択するのが一般的と言えるでしょう。PrintWriterクラスには、print()、println()といったメソッド（System.out.print()、System.out.println()と同じように使える）が用意されています。

1 サーバへのソケット接続を確立する

```
Socket chatSocket = new Socket("127.0.0.1", 5000);
```

この部分は、前ページのコードと同じ。データを書き込む際にも接続は必要

2 PrintWriterを、ソケット接続に対応する接続ストリームとチェーニングする

```
PrintWriter writer = new PrintWriter(chatSocket.getOutputStream);
```

PrintWriterには、文字列をバイト形式のデータに変換して接続ストリームに渡す機能がある。PrintWriterをソケット接続に対応する接続ストリームとチェーニングすれば、文字列をソケット接続経由で書き込めるようになる

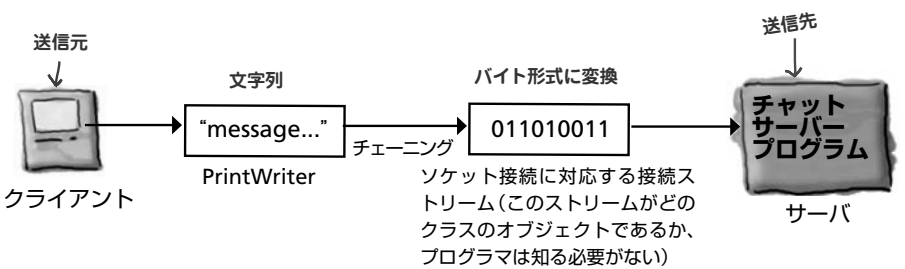
このコードによってソケット接続に対応する接続ストリームを取得する。この接続ストリームはより使いやすい高水準のチェーンストリームとチェーニングする

3 データ（テキスト）を書き込む

```
writer.println("message to send");
writer.print("another message");
```

println()では文字列の末尾に改行文字が追加される

print()では改行文字が追加されない



This PDF is prepared for tools@maruhouse.org, personal use only.

簡単なクライアントプログラム

この後、チャットクライアントの作り方について本格的に学んでいくわけですが、その前にもう少し簡単なプログラムを作ってみましょう。サーバに接続し、短いテキストのメッセージを受け取る、というだけのクライアントプログラムです。

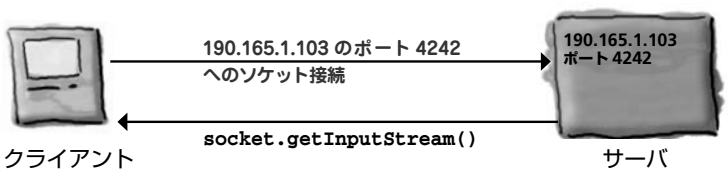
クライアントの後には、メッセージを提供するサーバの作り方についても説明します。このサーバは、クライアントのリクエストに応じてメッセージを作成し、送信します。

あまり実用的とは言えないプログラムですが、ネットワーク経由でのデータのやりとりの基本について学ぶには適していると言えるでしょう。

Mr. サーバからのメッセージ

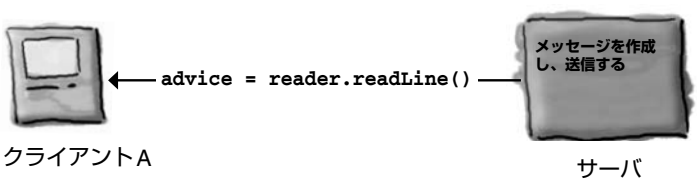
1 接続

サーバへの接続を行い、
接続ストリームを取得する



2 データの読み取り

クライアントがサーバからのデータを読み取る（受信する）



This PDF is prepared for tools@maruhouse.org, personal use only.

クライアントプログラムのコード

このプログラムでは、Socketオブジェクト、BufferedReaderオブジェクト（他のストリームにチェーニングする）を作成し、その後、サーバプログラム（ポート番号 4242 が割り当てられたプログラム）からテキストを 1 行読み取る、という処理を行います。

```
import java.io.*;
import java.net.*;

public class DailyAdviceClient {

  public void go() {
    try {
      Socket s = new Socket("127.0.0.1", 4242);

      InputStreamReader streamReader = new InputStreamReader(s.getInputStream());
      BufferedReader reader = new BufferedReader(streamReader);

      String advice = reader.readLine();
      System.out.println("Today you should: " + advice);

      reader.close();

    } catch(IOException ex) {
      ex.printStackTrace();
    }
  }

  public static void main(String [] args) {
    DailyAdviceClient client = new DailyAdviceClient();
    client.go();
  }
}
```

Socketはjava.netのクラス

例外が発生する可能性が非常に高いのでtry/catchブロックを使う

ローカルホスト上の、ポート番号「4242」のプログラムを接続先とする

BufferedReaderをInputStreamReaderにチェーニングする（InputStreamReaderはソケット接続に対応するストリームにチェーニングする）

このコードは、データの読み取り先がファイルであっても同様のものになる。同じことは、BufferedWriterのメソッドに関しても言える

すべてのストリームをクローズする

This PDF is prepared for tools@maruhouse.org, personal use only.

自分で考えてみよう

ここまで、ネットワーク経由でのデータ送受信について解説してきましたが、どの程度理解できているでしょうか。以下の練習問題でそれを確かめてください（前のページはできるだけ見ないようにしてください）。

ネットワーク経由でのデータの読み取り

クライアント

ストリームのチェーニングを行い、データを読み取るコードを書いてください

送信元

サーバ

ネットワーク経由でのデータの書き込み

クライアント

ストリームのチェーニングを行い、データを書き込むコードを書いてください

送信先

サーバ

自分で考えてみよう

下の質問に答えてください。

ソケット接続の確立には 2 つの情報が必要です。何と何ですか？

_____ _____

ウェルノウンポートのために予約されているTCPポート番号は、何番から何番までですか？

_____

TCPポート番号は、short型の変数に納めることができますか？

_____

This PDF is prepared for tools@maruhouse.org, personal use only.

サーバプログラムの作成

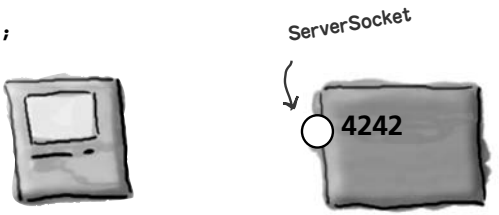
サーバプログラムはクライアントプログラムと異なり、Socketオブジェクトだけでなく、ServerSocketというオブジェクトを必要とします。ServerSocketは、クライアントからのリクエスト（このリクエストはクライアントでSocketオブジェクトが作られる時に送られる）に対応するオブジェクトです。

サーバとクライアントの接続

❶ サーバプログラムでServerSocketオブジェクトが作成される。引数には、「どのポートへのリクエストに対応するのか」が指定される。

```
ServerSocket serverSock = new ServerSocket(4242);
```

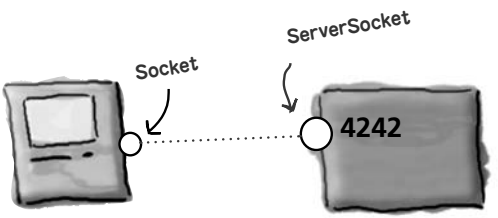
これによって、クライアントからのポート4242へのリクエストに対応できるようになる。



❷ クライアントプログラムでサーバプログラムへのソケット接続がリクエストされる。

```
Socket sock = new Socket("190.165.1.103", 4242);
```

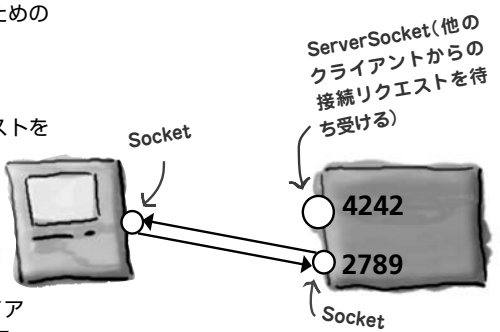
クライアント作成者はIPアドレスとポート番号を知っていなくてはならない（公開されている資料を調べるか、知っている人に尋ねることになる）。



❸ サーバプログラムでクライアントプログラムとの通信のためのソケット接続が確立される

```
Socket sock = serverSock.accept();
```

accept()は、クライアントによるソケット接続のリクエストを待ち受け、リクエストが出されたらSocketオブジェクトを戻す（接続を確立する）、というメソッド。この時作成されるSocketオブジェクトは、クライアントのIPアドレス、ポート番号といった情報を保持する。接続確立後に使用されるサーバのポートが、リクエスト時とは異なることに注意。これにより、ServerSocketは、他のクライアントからの新たな接続リクエストを待ち受けることができる。



This PDF is prepared for tools@maruhouse.org, personal use only.

サーバプログラムのコード

サーバプログラムでは、まずクライアントからのリクエスト（Socketオブジェクトの作成によって発行される）に対応するためのServerSocketオブジェクトを作成します。クライアントからのリクエストがあれば、サーバは新たにクライアントとの通信のためのSocketオブジェクトを作成します。クライアントへのメッセージの送信には、PrintWriter（ソケット接続に対応する接続ストリームとチェーニングする）を使用します。

```
import java.io.*;
import java.net.*;

public class DailyAdviceServer {

  String [] adviceList = {"Take smaller bites", "Go for the tight jeans. No they do NOT make
  you look fat.", "One word: inappropriate", "Just for today, be honest. Tell your boss what you
  *really* think", "You might want to rethink that haircut."};

  public void go() {

    try {
      ServerSocket serverSock = new ServerSocket(4242);

      while(true) {

        Socket sock = serverSock.accept();

        PrintWriter writer = new PrintWriter(sock.getOutputStream());
        String advice = getAdvice();
        writer.println(advice);
        writer.close();
        System.out.println(advice);
      }

    } catch(IOException ex) {
      ex.printStackTrace();
    }
  } // go() メソッドの終了

  private String getAdvice() {
    int random = (int) (Math.random() * adviceList.length);
    return adviceList [random];
  }

  public static void main(String [] args) {
    DailyAdviceServer server = new DailyAdviceServer();
    server.go();
  }
}
```

パッケージのインポートが必要

（紙面の都合で途中で次の行に送られているが、改行文字は入っていないことに注意。自分でコードを書くときは、ここでリターンキーを押さないよう注意する）

メッセージはこの配列を基に作る

このサーバプログラムでは、「ポート 4242」へのクライアントリクエストを待ち受ける

クライアントからのリクエストを待ち受ける（また、リクエストに対応する）ための無限ループ

accept() メソッドはリクエストがあるまで何もしない。リクエストがあれば、クライアントとの通信のためのSocketオブジェクトを戻す

ソケット接続に対応するストリームを利用してPrintWriterオブジェクトを作成する。クライアントへのメッセージは、このPrintWriterオブジェクトのprintln() メソッドを使用して書き込む。それが終わったらストリーム（ソケット接続）をクローズする

This PDF is prepared for tools@maruhouse.org, personal use only.

頭の体操

サーバはどのようにしてクライアントのIPアドレス、ポート番号を知るのか

先述のとおり、クライアントプログラムでは、プログラマがサーバのIPアドレス、ポート番号を指定しています。しかし、サーバプログラムは、ソケット接続の確立に必要なクライアントのIPアドレス、ポート番号といった情報を、いつ、どのようにして知るのでしょうか。自分なりに考えてみてください。

素朴な疑問に答えます

Q：前ページのサーバプログラムのコードには重大な欠陥があるように思えます。このままのコードでは結局、1度に1つのクライアントとしか通信できないのではないですか。

A：おっしゃるとおりです。このコードでは、1つのクライアントとの通信が完了するまで、別のクライアントからのリクエストに対応することはできません。リクエストに対応するaccept()メソッドは無限ループの中で呼び出されるのですが、ループ内の処理が一度完了しなければ、次回の呼び出しが行われないからです。accept()メソッドが呼び出されなければ、新たなクライアントとの間のソケット接続が行われ、通信が開始されるということはありません。

Q：では、複数のクライアントに同時に対応できるサーバを作るにはどうすればいいのですか。それができなければ、チャットサーバなどは絶対に作れないのではないでしょうか。

A：それには、「スレッド」という概念を覚える必要があります。クライアントごとに1つずつスレッドを割り当てていく、ということをするのですが、それについては後で詳しく説明します。

重要ポイント

- クライアントプログラムとサーバプログラムは、ソケット接続経由で通信ができる。
- Socketは、（通常は別々のマシン上で動作する）2つのプログラムの間の接続を表すオブジェクト。
- クライアントプログラムを作る際は、サーバプログラムのIPアドレス（あるいはドメイン名）とTCPポート番号を知る必要がある。
- TCPポート番号は符号なし（正の数のみ）の16ビットの数値。マシン上の特定のプログラムに対応する。TCPポート番号を使用することで、複数のクライアントが同じサーバマシンに同時に接続する、ということができる（ただし、利用するサーバプログラムはそれぞれに異なる）。
- 0から1023までのTCPポート番号は、ウェルノウンポート（HTTP、FTP、SMTPといった一般的なサーバプログラムに対応するポート番号）に使うと定められている。
- クライアントがサーバへのソケットを接続を行うためのコードは以下のようになる。

```
Socket s = new Socket("127.0.0.1", 4200);
```

- サーバへの接続ができれば、クライアントは、以下のようなコードで、ソケット接続に対応する接続ストリーム（入力ストリーム、あるいは出力ストリーム）を取得できる。

```
sock.getInputStream();
```

- サーバからのテキストデータの読み取りのためには、BufferedReaderを作り、InputStreamReaderとチェーニングする。InputStreamReaderは、ソケット接続に対応する接続ストリームとチェーニングする。
- InputStreamReaderは、バイト形式のデータをテキストデータに変換する機能を持ち、低水準の接続ストリームと高水準のチェーンストリーム（BufferedReaderなど）の橋渡しの役割を果たす。
- サーバへのテキストデータの書き込みのためには、PrintWriterを作り、ソケット接続に対応する接続ストリームと直接チェーニングする。テキストの書き込みそのものは、print()メソッド、あるいはprintln()メソッドを使って行う。
- ServerSocketは、サーバがクライアントからのリクエストを待ち受けるためのオブジェクト。コンストラクタの引数として、待ち受けに使用するポート番号を指定する。
- ServerSocketオブジェクトはクライアントからのリクエストに応え、ソケット接続を確立する役割を果たす。

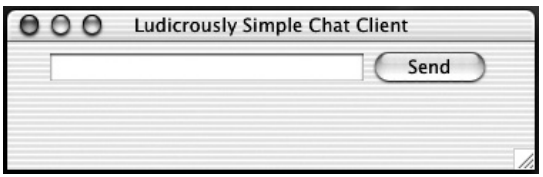
This PDF is prepared for tools@maruhouse.org, personal use only.

簡単なチャットクライアントの作成

それでは、チャットクライアントプログラムの作成に移りましょう。このプログラムは2段階に分けて作っていきます。まず、サーバにメッセージを送信する機能だけを持ち、他の参加者からのメッセージを(サーバ経由で)受信する機能は持っていない、というバージョンを作ります(これで「チャットクライアント」と呼べるかどうかは疑問ですが……)。

そして、その後、送受信両方の機能を備えた「フルバージョン」のチャットクライアントプログラムを作ります。

バージョン1:送信機能のみのチャットクライアント



メッセージを入力して"Send"ボタンをクリックすると、サーバーに送信される。このバージョンにはメッセージの受信機能がないため、受信メッセージを表示するためのテキスト領域もない

コードの概要

```
public class SimpleChatClientA {

  JTextField outgoing;
  PrintWriter writer;
  Socket sock;

  public void go() {
    // GUI を作成し、Send ボタンのリスナーを登録する
    // setUpNetworking() メソッドを呼び出す
  }

  private void setUpNetworking() {
    // Socket オブジェクト、PrintWriter オブジェクトを作成する
    // PrintWriter オブジェクトへの参照をインスタンス変数 writer に代入する
  }

  public class SendButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
      // テキストフィールドに入力されたテキストを
      // PrintWriter オブジェクトを利用してサーバに送信する
    }
  } // 内部クラス SendButtonListener の終了

} // 外部クラスの終了
```


This PDF is prepared for tools@maruhouse.org, personal use only.

```
import java.io.*;
import java.net.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class SimpleChatClientA {

  JTextField outgoing;
  PrintWriter writer;
  Socket sock;

  public void go() {
    JFrame frame = new JFrame("Ludicrously Simple Chat Client");
    JPanel mainPanel = new JPanel();
    outgoing = new JTextField(20);
    JButton sendButton = new JButton("Send");
    sendButton.addActionListener(new SendButtonListener());
    mainPanel.add(outgoing);
    mainPanel.add(sendButton);
    frame.getContentPane().add(BorderLayout.CENTER,mainPanel);
    setUpNetworking();
    frame.setSize(400,500);
    frame.setVisible(true);
  } // go メソッドの終了

  private void setUpNetworking() {
    try {
      sock = new Socket("127.0.0.1", 5000);
      writer = new PrintWriter(sock.getOutputStream());
      System.out.println("networking established");
    } catch(IOException ex) {
      ex.printStackTrace();
    }
  } // setUpNetworking メソッドの終了

  public class SendButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
      try {
        writer.println(outgoing.getText());
        writer.flush();

      } catch(Exception ex) {
        ex.printStackTrace();
      }
      outgoing.setText("");
      outgoing.requestFocus();
    }
  } // 内部クラス SendButtonListener の終了

  public static void main(String [] args) {
    new SimpleChatClientA().go();
  }
} // 外部クラスの終了
```

ストリーム、ソケット接続、GUIのためのパッケージのインポート

ごく一般的なGUI作成のコード。ネットワーク、I/Oには無関係

ローカルホストのIPアドレスを指定。ここではクライアントとサーバが物理的には同一のマシンとなる

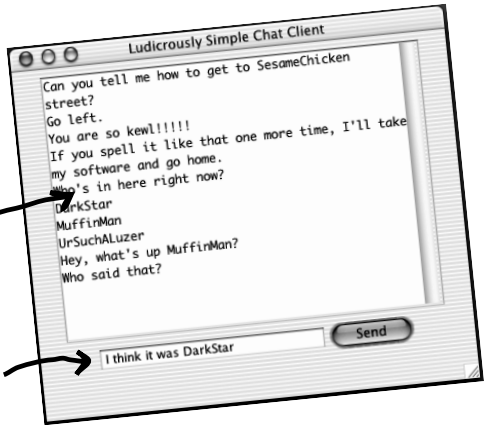
Socketオブジェクト、PrintWriterオブジェクトを作成する（このメソッドは、go()メソッドによって、GUIの画面表示の前に呼び出される）

ここでデータの書き込みを行う。PrintWriterは、サーバとのソケット接続に対応する接続ストリーム（出力ストリーム）にチェーニングされるため、println()の出力先はサーバになる

このクライアントプログラムを実際に動かす場合には、後に紹介するサーバプログラムも作る必要があります。両方のプログラムができたら、サーバ、クライアントの順に起動します。

This PDF is prepared for tools@maruhouse.org, personal use only.

バージョン 2：送受信両方の機能を備えたチャットクライアント



サーバは、クライアントからメッセージを受け取ると、チャットの全参加者のクライアントに即座に転送する。「全参加者のクライアント」には、メッセージを送信したクライアントも含まれる

バージョン 2 作成上の問題点 1：サーバからどのようにしてメッセージを受信すればよいのか。
これはさほど難しくありません。ソケット接続を確立し、接続ストリームと適切なチェーンストリーム（BufferedReaderが最適）を組み合わせ、readLine()メソッドを使用してデータを読み取ればいいのです。

バージョン 2 作成上の問題点 2：サーバからのメッセージ受信をどのタイミングで行えばよいのか。
これには以下のような選択肢があります。

① **定期的（20 秒に 1 回程度）に、新しいメッセージがないかをサーバに問い合わせ、あれば受け取る。**

この方法は不可能ではありませんが、いくつか問題があります。まず、各クライアントにとってどれが新しいメッセージでどれがすでに受け取ったメッセージなのかを区別するための仕組みが必要です。また、いずれにしてもサーバがクライアントから受け取ったメッセージを蓄えておかねばなりません。「どこかからメッセージが届いたら全クライアントに即座に配り、後のことは関知しない」というわけにはいかないのです。また、クライアントへのメッセージの配布が遅れるというのも問題です。常に 20 秒遅れでメッセージが送られてくるようでは、実用的なチャットシステムとは言えないでしょう。ただ、遅延を減らそうとすれば、クライアントが問い合わせを頻繁に行うことになり、処理速度の低下につながります。

② **メッセージ送信の度に、他のクライアントからのメッセージがサーバに届いていないかを確認し、届いていれば受信する。**

実現は非常に簡単な方法ですが、チャットシステムとしてはまったく使えません。第一、自分からはメッセージをまったく送らない、いわゆる「ROM」の人は何も受け取れないことになります。

③ **サーバにメッセージが送られたら、それをすべてのクライアントが即座に受け取る。**

この方法なら効率的です。非常に使いやすいチャットシステムが作れます。ただし、メッセージの送信と受信という 2 つの処理を同時に行えるような仕組みが必要になります。「Send」ボタンがクリックされたら、入力されたテキストをサーバに送り、同じテキストを同時にサーバから受け取る、というようなことができなければならないのです。

This PDF is prepared for tools@maruhouse.org, personal use only.

彼女は歩きながら ガムを噛んでいます……

厳密には同時に複数の処理を行っているわけではないのですが、各スレッドの処理を少しずつ行うことで見かけ上、同時に処理が進行しているように見せるのです（たとえば、スレッド1と2がある場合は、1と2の処理を交互に行う。切り換えはごく短時間で行われるのでユーザの目にはそれとわからない）。スレッドを複数にすることを「マルチスレッド」と言いますが、Javaでは、マルチスレッドのプログラムが簡単に作れます。そのために使用するThreadというクラスが用意されているからです。

```
Thread t = new Thread();
t.start();
```

上記のように、Threadオブジェクトを作成し、start()メソッドを使用すれば、新しいスレッドを起動することができます。スレッドを起動すると、それぞれに専用のスタックが割り当てられます。

ただ、これでマルチスレッドのプログラムが完成、というわけではありません。

スレッド自身は起動しても何もしないからです。起動しただけだと、スレッドはすぐに終了し、割り当てられたスタックも消滅してしまいます。

スレッドを持続させるためには、そのスレッドでどのような処理（「ジョブ」と呼ばれる）を行うのか、それを明確にしておかねばなりません。これは、言い換えれば、各スレッドには、そのスレッドで実行するためのコードが必要、ということです。

このコードは、run()というメソッドに書くことになります（run()メソッドの詳細については後述）。Threadクラスは、java.langパッケージに属するクラスなので、インポートのためのコードを書かなくても使用できます（java.langパッケージは、String、Systemなど、Java言語にとって特に重要なクラスの属するパッケージです）。

実用的なチャットシステムを作るなら選択肢3を選ぶべき

選択肢3を選ぶということは、サーバへのメッセージ到着を絶えず待ち受ける機能が必要、ということになります。また、同時に、ユーザのGUI操作にも常に対応できなくてはなりません。ユーザのメッセージ入力や、受信メッセージ表示画面のスクロールに対応しながら、サーバに新規メッセージが届けば即座にそれを読み込めるよう待機するのです。

それには、すでに述べたとおり、「スレッド」という概念が必要になります。

元来、データを送信すると同時に受信する、GUIを機能させながらサーバのデータの読み取りも行う、などということは通常のコンピュータにはできないはずです。

もし、本当にそういうことをしようとすれば、コンピュータに複数のプロセッサが必要になります。ただ、「スレッド」を利用すれば、たとえプロセッサは1つでも、同時に複数のことが行われているように見せることができるのです。

マルチスレッドプログラムの作成

「スレッド」というのは、プログラムの実行単位のことです。どのプログラムも最低1つのスレッドを持つのですが、このスレッドを複数にすれば、同時に複数の処理を行っているように見せることができます。

This PDF is prepared for tools@maruhouse.org, personal use only.

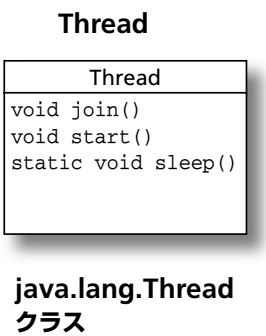
スレッドを表すThreadクラス

Javaには、スレッドを表すThreadというクラスが用意されています（クラスなので名前の先頭が大文字になる点に注意）。プログラムの中で新たにスレッドを起動したい場合はThreadオブジェクトを作成することになります。スレッドを起動すると、以下の図に示すように、それぞれに専用のスタックが割り当てられます。呼び出されたメソッドは、スレッドごとに別のスタックに格納されます。Threadはjava.langパッケージのクラスなので、インポートのためのコードは必要ありません。スレッドを起動する必要のある時は、いつでも使用することができます。

スレッドにはそれぞれ専用のスタックが割り当てられる

Threadはスレッドを表すクラス

スレッドを新たに起動するには、Threadオブジェクトを作成する



Threadオブジェクトを作成してスレッドを起動する、という操作を行わなければ、プログラムの処理はすべて「メインスレッド」と呼ばれるスレッドだけで行われることになります。メインスレッドのスタックの最下部にはmain()メソッドが置かれ、その上に呼び出されたメソッドが積み重ねられていきます。メインスレッドはJVMによって自動的に起動されるので、メインスレッドを起動するためのコードをわざわざ書くことはありません（メインスレッドの他にも、ガーベージコレクションのためのスレッドなど、JVMによって自動的に起動されるスレッドはある）。

Threadはスレッドを表すクラス。このクラスには、スレッドを起動するためのメソッド、複数のスレッドを結合するためのメソッド、スレッドを「スリープ」状態にするためのメソッドなどがある（他にもメソッドはあるが、ここではさしあたって知っておく必要のあるものだけをあげた）。

This PDF is prepared for tools@maruhouse.org, personal use only.

「マルチスレッド」の意味

マルチスレッドのプログラムでは複数の処理が同時に進行しているように見えます。しかし、本当に複数の処理を同時に行えるのは複数のプロセッサを持ったコンピュータだけです。マルチスレッドというのはあくまで、複数の処理が同時に行われているように「見せる」だけのものです。そのためには、複数のスタックを交互に有効にするという手段が使用されます。処理の進行するスタックがごく短時間で次々に入れ替わるので、プログラムのユーザから見れば、同時に処理が進んでいるように見えるというわけです。また、Javaプログラムは、OSから見れば、いくつか存在するプロセスの1つに過ぎない、という点にも注意が必要です。OSが他のプロセスの処理を進めている間、Javaプログラムのプロセスは待機状態におかれます。マルチスレッドのプログラムにおいて問題となるのは、いったん待機状態になってからの処理の再開の仕方です。スタック（スレッド）が複数あっても、必ず、待機状態になった時点で有効になっていたスタックから処理を再開しなければなりません。

つまり、今、有効になっているスタックはどれか、また、処理が進行中のメソッドはどれか、といった情報が常に保持されるということです。

マルチスレッドプログラムの処理は、おおまかには以下のように進行します。

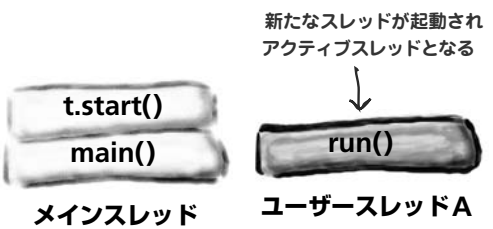
❶ JVMがmain()メソッドを呼び出す。

```
public static void main(String [] args) {
...
}
```

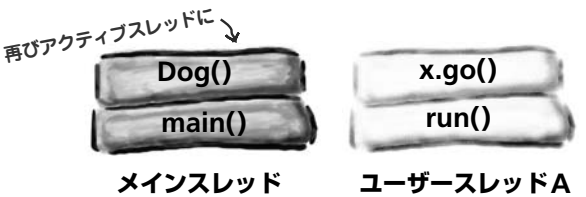
❷ main()メソッドが新たなスレッドを起動する。新たなスレッドが起動される際、メインスレッドの処理は一時凍結される。

```
Runnable r = new MyThreadJob();
Thread t = new Thread(r);
t.start();
Dog d = new Dog();
```

このコードの意味は後述



❸ JVMはメインスレッドとユーザースレッドAの処理を、両者の処理が完了するまで交互に進める。



This PDF is prepared for tools@maruhouse.org, personal use only.

スレッドを起動するためのコード

❶ Runnableインターフェースをインプリメントするオブジェクトの作成

```
Runnable threadJob = new MyRunnable();
```

スレッドを起動するには、まず、Runnableというインターフェースをインプリメントしたクラスのオブジェクト（Runnableオブジェクト）を作成する必要があります。Runnableインターフェースをインプリメントしたクラスには、スレッドで行うべき処理（ジョブ）がどのようなものなのかが記述されます（Runnableインターフェースの詳細については後述）。新たなスレッドのスタックにはまず、このクラスに属するメソッドが格納されることになります。

❷ Threadオブジェクトの作成

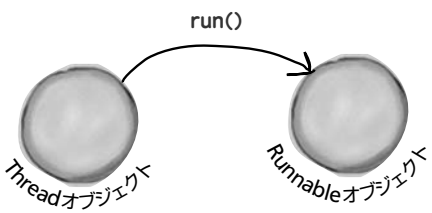
```
Thread myThread = new Thread(threadJob);
```

Threadオブジェクトを作成する際、コンストラクタの引数としては、Runnableオブジェクトを指定します。

❸ スレッドの起動

```
myThread.start();
```

スレッドを起動するには、start()メソッドを呼び出します。このメソッドを呼び出すまでは、ただThreadインスタンスが存在するというだけで、新たなスレッドでの処理は何も行われません。スレッドが起動されると、Runnableオブジェクトのrun()メソッドが呼び出され、スタックの最下部に置かれます。



This PDF is prepared for tools@maruhouse.org, personal use only.

スレッドには「ジョブ」が必要

Threadオブジェクトのコンストラクタの引数にRunnableオブジェクトを指定することで、スレッドで行うべき処理が決まる

スレッドのスタックの最下部にはRunnableオブジェクトのrun()メソッドが置かれる

Threadオブジェクトはジョブを必要とします。「ジョブ」とは事実上、新しいスレッドに割り当てられるスタックの最下部の置かれるメソッドのことである、と言っていいでしょう。このメソッド（run()メソッド）は必ず、以下のような形式のものになります。

Runnableインターフェースのメソッドは「public void run()」のみ

```
public void run() {
  // スレッドで行うべき処理について書かれたコード
}
```

run()メソッドは、Runnableインターフェースのメソッドです。Threadオブジェクトを作成する際には、コンストラクタの引数にRunnableインターフェースをインプリメントするクラスのオブジェクト（Runnableオブジェクト）を指定します。これにより、どのクラスのrun()メソッドがスタックの最下部に置かれるのかが決まるのです。run()メソッドにはそれぞれスレッドでどのような処理を行うのかが記述されます。

コンストラクタの引数にRunnableオブジェクトを指定することで、呼び出すべきrun()メソッドが指定され、スレッドで行うべき処理が指定される、ということになるのです。

This PDF is prepared for tools@maruhouse.org, personal use only.

Runnableインターフェースをインプリメントする

Runnableはjava.langパッケージのインターフェースなので、インポートは必要ない

```
public class MyRunnable implements Runnable {

  public void run() {
    go();
  }

  public void go() {
    doMore();
  }

  public void doMore() {
    System.out.println("top o' the stack");
  }
}
```

❷

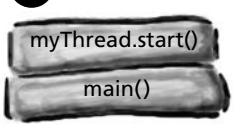
Runnableインターフェースのインプリメントすべきメソッドはrun()のみ(引数はない)。スレッドが起動された時に行うべき処理はこのメソッドに記述される。run()メソッドは、スレッドに割り当てられるスタックの最下部に置かれる

```
class ThreadTester {

  public static void main (String [] args) {

    Runnable threadJob = new MyRunnable();
    Thread myThread = new Thread(threadJob);

❶   myThread.start();

    System.out.println("back in main");
  }
}
```

Threadのコンストラクタの引数としては、Runnableオブジェクトを指定する。これによって、同時に、スレッド起動時にどのrun()を呼び出すかも指定される

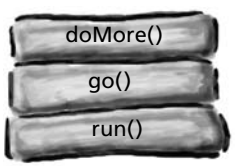
スレッドはstart()メソッドによって起動される。スレッドが起動されるまでは、たとえThreadオブジェクトは存在しても、事実上、スレッドは存在しないのと同じ。start()メソッドを呼び出さない限り、Threadオブジェクトはまったく「スレッドらしさ」を持つことができない

❶



メインスレッド

❷



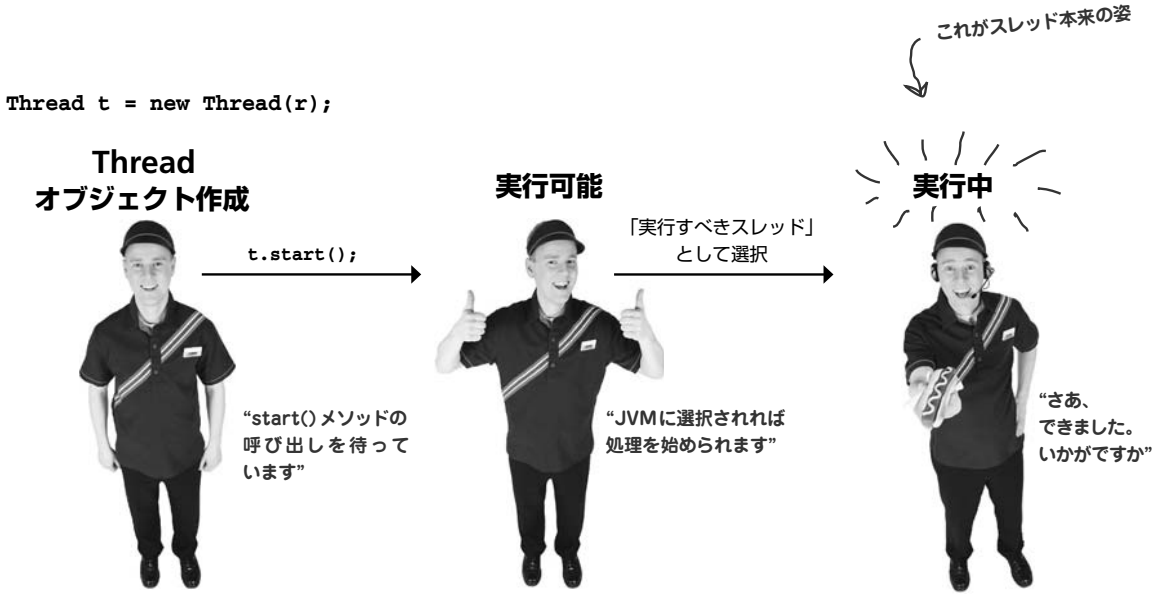
ユーザースレッド

頭の体操

ThreadTesterクラスを実行すると、どのような結果が得られるでしょうか(答えはしばらく後に書いてあります)

This PDF is prepared for tools@maruhouse.org, personal use only.

スレッドの状態の変化



Thread t = new Thread(r);

Threadオブジェクトは作成されたが、スレッドはまだ起動されていないという状態です。Threadオブジェクトは存在しても、事実上、スレッドは存在しない状態と言えます。

t.start();

少しわかりにくいのですが、スレッドは起動されてもすぐに処理を始めるわけではありません。起動されたスレッドは、JVMによって、「実行すべきスレッド」として選択されるのを待つのです。選択を待っている状態を「実行可能状態」と呼びます。この状態ですでにスレッドに対応するスタックは存在します。

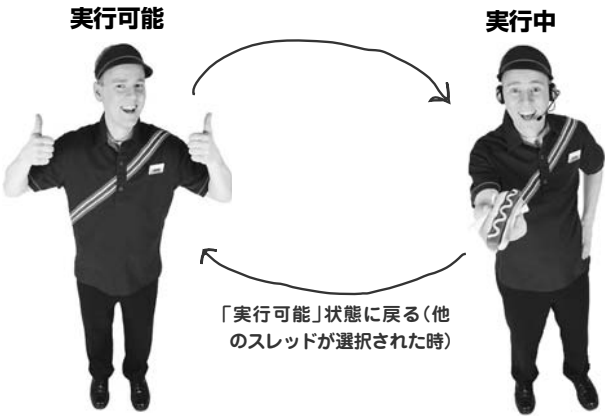
「実行すべきスレッド」として選択されれば、スレッドは処理を始められる状態になります。この状態がいわば最も「スレッドらしい」状態と言えるでしょう。特定のスレッドを「実行すべきスレッド」として選択する、という決定は、JVMのスレッドスケジューラによって自動的に下されます。プログラマは、プログラムの書き方によってこの決定に影響を与えられることもありますが、強制的に特定のスレッドを選択させることはできません。スレッドがこの状態になると、スタックが有効になり（他のスレッドのスタックはすべて無効になる）、最上部のメソッドから実行されていきます。

「実行中」の状態になったスレッドはずっとそのままの状態というわけではなく、いったん「実行可能」の状態や、後に説明する「ブロック」という状態に切り替わり、再び「実行中」に戻る、ということを繰り返します。

This PDF is prepared for tools@maruhouse.org, personal use only.

通常は実行可能/実行中という2つの状態の間で切り換えが行われる

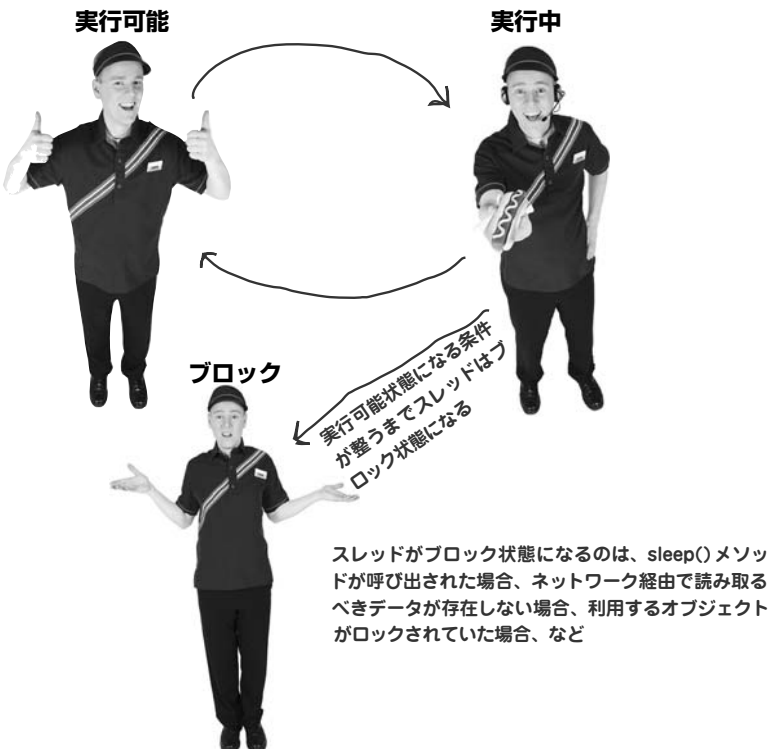
通常のケースでは、スレッドの状態は、「実行可能」と「実行中」の間で切り替わっていきます。JVMのスレッドスケジューラによって「実行すべきスレッド」として選択されれば「実行中」状態になり、他のスレッドが選択されれば、「実行可能」状態になるというわけです。



「ブロック」状態

スレッドの状態は基本的には上記の2つですが、場合によっては「ブロック」という状態になることもあり得ます。これは、スレッドが一時的に実行を止めた状態です。スレッドが「ブロック」状態になるのは、たとえば「ソケット接続経由でデータを読み取ろうとしたが、その時点では読み取るべきデータが存在しなかった」というようなケースです。このケースでは、(スレッドスケジューラが)スレッドをブロック状態にして、読み取るデータが生じるのを待つのです。また、スレッドの実行コードの中でsleep()メソッドを呼び出した場合や、ロックされているオブジェクトのメソッドを呼び出そうとした場合などにも、スレッドはブロック状態になります。オブジェクトがロックされていた場合は、ロックをかけたスレッドでロックの解除が行われるまで、ブロック状態が続きます。

スレッドがブロック状態になるケースは他にもありますが、取りあえず、ここでは上記の3つのケースを覚えておいてください。



This PDF is prepared for tools@maruhouse.org, personal use only.

スレッドスケジューラ

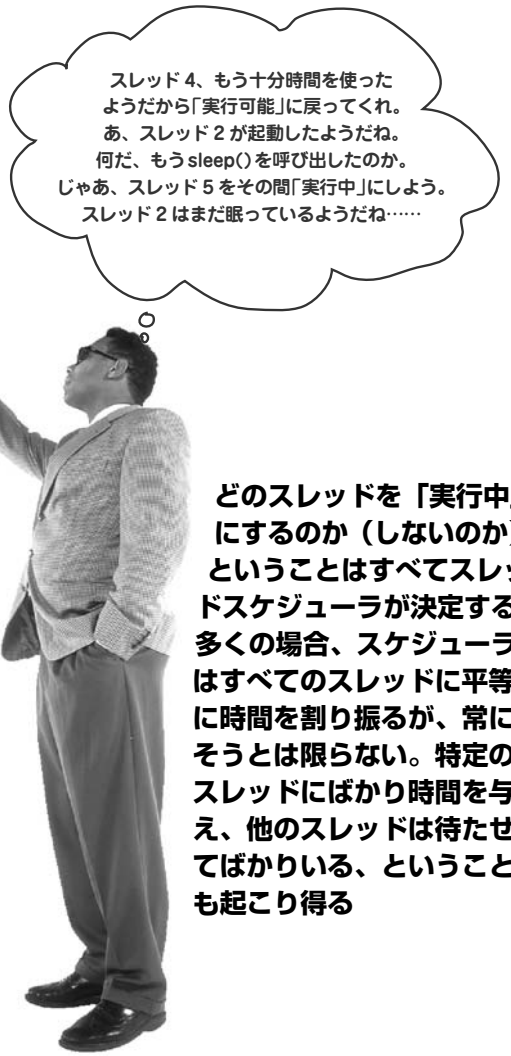
スレッドスケジューラは、「実行可能」状態のスレッドを「実行中」状態にする、また、逆に「実行中」状態のスレッドを「実行可能」状態に戻す、といった決定のすべてを下します。あるスレッドがいつ処理を開始し、どれだけの間、処理を続行するのか、「実行中」状態でなくなったスレッドを次にどのような状態に置くのか、といったことはすべてスレッドスケジューラが決めるのです。

プログラマがスケジューラをコントロールするということはできません。そのためのメソッドなどもAPIには用意されていません。また、重要なことは、スケジューラが「条件が同じであれば常に同じように機能する」というようなものではない、ということです（多くの場合、条件が同じであれば同じように機能するのですが、例外もあるということです）。

つまり、「スケジューラはこう動くはず」という予測のもとにプログラムを書いてはいけないのです。スケジューラの動きは、JVMの種類が異なれば当然異なってきますし、たとえ同じマシン上で同じプログラムを動かしたとしても、スケジューラの動きが同じとは限りません。プログラミングを始めて日の浅い人は、スケジューラが常に同じように機能すると思いこみ、マルチスレッドプログラムを1台のマシンだけでテストして安心するということが多いので注意しましょう。

Javaのプログラムはいくつものプラットフォームで動かすということが多いので、マルチスレッドプログラムが特定のプラットフォームのスレッドスケジューラでなければ正しく機能しないということでは困ります。たとえば、スレッドスケジューラはすべてのスレッドを平等に「実行中」状態にするとは限りません。それを期待してプログラムを書くのは危険でしょう。最近では少なくなっていますが、中には、個々のスレッドをいったん「実行中」状態にすると、run()メソッドの処理が完了するまでそれを維持してしまう、というスケジューラもあるからです。

スケジューラを直接コントロールすることはできませんが、sleep()メソッドを利用すれば、多少それに似たことはできます。sleep()メソッドを呼び出してスレッドの処理を一時停止すると、その間だけスレッドが「実行中」状態でなくなるからです。つまり、その間（数ミリ秒という場合もありますが）は他のスレッドが「実行中」になれる可能性が生まれるわけです。ともかく、sleep()メソッドを呼び出せば、引数に指定した時間そのスレッドが「実行中」状態に戻ることはない、ということだけは確実です。たとえば、引数に「2000」と指定してsleep()メソッドを呼び出した場合は、確実に2秒間そのスレッドは「実行中」状態には戻りません。



どのスレッドを「実行中」にするのか（しないのか）、ということはすべてスレッドスケジューラが決定する。多くの場合、スケジューラはすべてのスレッドに平等に時間を割り振るが、常にそうとは限らない。特定のスレッドにばかり時間を与え、他のスレッドは待たせてばかりいる、ということも起こり得る

This PDF is prepared for tools@maruhouse.org, personal use only.

スケジューラは予測とは異なる動きをすることもある……

マルチスレッドプログラムの例

```
public class MyRunnable implements Runnable {

  public void run() {
    go();
  }

  public void go() {
    doMore();
  }

  public void doMore() {
    System.out.println("top o' the stack");
  }
}

class ThreadTestDrive {

  public static void main (String [] args) {

    Runnable threadJob = new MyRunnable();
    Thread myThread = new Thread(threadJob);

    myThread.start();

    System.out.println("back in main");
  }
}
```

実行結果の例

```
File  Edit  Window  Help  PickMe
% java ThreadTestDrive
back in main
top o' the stack
% java ThreadTestDrive
top o' the stack
back in main
% java ThreadTestDrive
top o' the stack
back in main
% java ThreadTestDrive
top o' the stack
back in main
% java ThreadTestDrive
top o' the stack
back in main
% java ThreadTestDrive
top o' the stack
back in main
% java ThreadTestDrive
back in main
top o' the stack
```

スレッドの実行の順序が頻繁に入れ替わっていることに注意。メインスレッドの処理より、新たに起動されたスレッドの処理の方が先に完了することもあれば、逆の場合もある

This PDF is prepared for tools@maruhouse.org, personal use only.

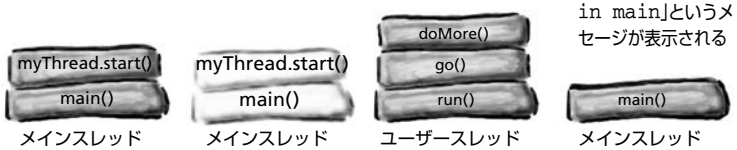
メインスレッドの完了が後になる場合

main() メソッドで新たなスレッドが起動される

スケジューラがメインスレッドを「実行中」状態から「実行可能」状態にし、新しいスレッドの処理が進行できるようにする

スケジューラがユーザースレッドの処理を完了させる。すると、画面には「top o' the stack」というメッセージが表示される

run() メソッドの処理が完了すると、ユーザースレッドのスタックはなくなる。その後、メインスレッドが再び「実行中」状態になり、処理を完了させる。その際、画面には「back in main」というメッセージが表示される



時間の経過

メインスレッドの完了が先になる場合

main() メソッドで新たなスレッドが起動される

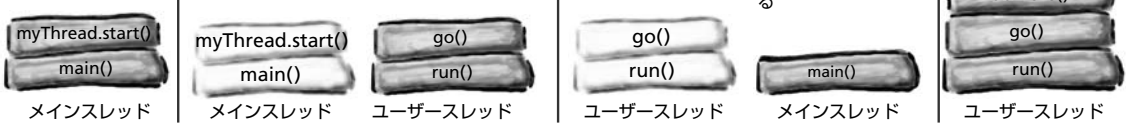
スケジューラがメインスレッドを「実行中」状態から「実行可能」状態にし、新しいスレッドの処理が進行できるようにする

スケジューラはユーザースレッドの処理をしばらく進行させるが、run() メソッドが終了するだけの時間は与えない

スケジューラがユーザースレッドを「実行中」状態から「実行可能」状態にする

スケジューラはメインスレッドを「実行中」状態にし、処理を再開させる。そのままメインスレッドの処理は完了し、画面には「back in main」というメッセージが表示される

ユーザースレッドが再び「実行中」状態になり、そのまま処理が完了する。画面には「top o'the stack」というメッセージが表示される



時間の経過

This PDF is prepared for tools@maruhouse.org, personal use only.

素朴な疑問に答えます

Q：マルチスレッドのプログラムを作る際、Runnableインターフェースをインプリメントするクラスを独自に作るのではなく、Threadのサブクラスを作り、Threadのrun()メソッドをオーバーライドしている、という例を見たことがあるのですが……。Threadオブジェクトを作成するコードでは、以下のように、引数なしのコンストラクタを使用するのです。

```
Thread t = new Thread();
        // 引数にRunnableオブジェクトを指定しない
```

A：確かにそういうことは可能です。ただ、「オブジェクト指向」という観点から見てあまりいい方法とは言えないでしょう。まず、サブクラスというものは本来、どういう目的で作るのか、ということを考えてみてください。ご質問の場合のように、Threadのサブクラスとして、「スレッドのジョブ」に対応するクラスを作ると、スーパークラスとサブクラスがあまりにかけ離れたものになってしまいます。「スレッドのジョブ」に対応するクラスは、やはり「スレッドそのもの」に対応するThreadクラスとはまったく関係のないものとして作るべきでしょう。Threadのサブクラスは、「スレッドの中でも特別なものにのみ対応するクラスが必要」という場合に作るべきものです。ジョブに対応するクラスはあくまでRunnableインターフェースをインプリメントし、Threadクラスからは独立したものとして作るようにしてください。
Threadのサブクラスを作りrun()メソッドをオーバーライドしたとしても、Javaの文法に反するわけでも、パフォーマンスの低下につながるわけでもありません。プログラムのデザインとして優れているかどうか、という話です。

Q：Threadオブジェクトは再利用できないのですか。いったんstart()で起動し、run()の処理が完了した後、もう一度start()で再起動するということはできないのでしょうか。

A：できません。run()の処理が完了してしまえば、スレッドは二度と起動し直すことはできません。run()の処理が完了したスレッドは、いわば「死んだ」状態になってしまうのです。その時点でThreadオブジェクト自体はまだヒープ上に残っているということはあり、そのオブジェクトのメソッドを呼び出すことができる場合もあるのですが、Threadオブジェクトのスレッドとしての性質はまったくなくなってしまいます（スレッドに割り当てられていたスタックも消滅している）。たとえオブジェクトが残っていても、他のオブジェクトと何ら変わるところのない普通のオブジェクトに過ぎません。
一度に複数のThreadオブジェクトを作り、それを順に起動していく、というプログラムを書くことはできますが、これはご質問のThreadオブジェクトの再利用とは異なります。

重要ポイント

- 「スレッド」とは、プログラムの実行単位のこと。
- スレッドには必ず専用のスタックが割り当てられる。
- Threadはjava.langパッケージのクラスで、スレッドを表す。
- スレッドには行うべき仕事（ジョブ）を与える必要がある。その仕事の内容は、Runnableインターフェースをインプリメントするクラスに記述する。
- Runnableインターフェースのメソッドはrun()だけ。このメソッドは、スレッドのスタックの最下部に置かれる。つまり、スレッドで最初に実行されるメソッドであるということ。
- Threadオブジェクトのコンストラクタの引数には、Runnableオブジェクトを指定する。
- Threadオブジェクトを作成しても、start()メソッドを呼び出すまでの間は、事実上、スレッドは存在しない状態と言える。
- start()メソッドを呼び出してスレッドを起動すると、新しいスタックが作成され、最下部にrun()メソッドが置かれる。この時、スレッドは「実行可能」状態になり、「実行すべきスレッド」として選択されるのを待つことになる。
- スレッドは、JVMのスレッドスケジューラによって「実行すべきスレッド」として選択されると、「実行可能」状態から「実行中」状態になる。コンピュータのプロセッサが1つであれば、同時に2つのスレッドが「実行中」になることはない。
- スレッドの状態は「実行中」から「ブロック（一時的に実行を止めた状態）」に変わることもある。「ブロック」状態になるのは、たとえば、「ソケット接続経由でデータを読み取ろうとしたが、読み取るべきデータが存在しなかった」というようなケース。また、コードの中でsleep()メソッドを呼び出した、ロックされているオブジェクトのメソッドを呼び出そうとした、といった場合にも、スレッドはブロック状態になる。
- スレッドスケジューラの動きは完全には予測できない。すべてのスレッドに平等に時間を与える場合もあるが、そうでない場合もある。sleep()メソッドをうまく使えば、スレッドへの時間の割り当てをある程度思いどおりに操作できる。

This PDF is prepared for tools@maruhouse.org, personal use only.

スレッドをスリープさせる

スケジューラの選択するスレッドを強制的に変更させる方法として最適なのは、スレッドを一時的にスリープさせる、というものでしょう。スレッドをスリープさせるには、sleep() というスタティックメソッドを呼び出すだけなので簡単です。引数にはスリープの時間（単位はミリ秒）を指定します。

sleep() メソッドを呼び出す際にはたとえば以下のようなコードを書きます。

処理を進行するスレッドを意図的に切り換えたい場合は、sleep() メソッドを使用する

```
Thread.sleep(2000);
```

これにより、スレッドが 2 秒間、「実行中」でも「実行可能」でもない、という状態になります。2 秒間は決して「実行中」状態に戻ることができないのです。ただ、注意すべきなのは、sleep() メソッドが InterruptedException というチェック例外を発生させる可能性があるということです。したがって、sleep() メソッドの呼び出しは必ず try/catch ブロックで行うようにします（先述のとおり InterruptedException を宣言するだけ、という方法も使える）。つまり、たとえば、以下のようなコードを書くことになります。

スレッドがスリープを脱すると、いったん「実行可能」状態になり、スケジューラに選択されるのを持つことになる

```
try {
  Thread.sleep(2000);
} catch(InterruptedException ex) {
  ex.printStackTrace();
}
```

現実には、この例外が発生することはまずあり得ません。それでも sleep() メソッドの宣言ではこの例外が発生する可能性があることになっているので、ルール上、try/catch ブロックを使わざるを得ません。sleep() メソッドが InterruptedException を発生させるよう宣言されているのは、スレッド通信メカニズムをサポートするためです（このメカニズムを利用する人はほとんどいない）。

スリープさせられたスレッドは、指定の時間内は決して元に戻らないということはすでにわかったと思いますが、それでは、元に戻るのが指定の時間より少し後になる、ということはあり得るのでしょうか。ないとは言えません。指定の時間になれば自動的に元に戻る、つまり「実行中」の状態に戻る、というわけではなく、いったん「実行可能」の状態になるからです。その後スレッドがどうなるかはスレッドスケジューラの決定次第です。スレッドの数が少ないプログラムであれば、指定の時間が経過した後すぐにスレッドが元に戻ったように見えることが多いでしょうし、それほどタイミングが厳密でなくてもいいプログラムなら多少の遅れは問題にならないでしょう。しかし、どのプログラムでも常にそうというわけにはいきません。

This PDF is prepared for tools@maruhouse.org, personal use only.

sleep()メソッドを使用すれば スケジューラをある程度コントロールできる

先に示したとおり、マルチスレッドのプログラムは実行するたびに結果が異なる、という場合があります。スレッドがどのような順序で実行されるかは決まっていないためです。先に示した例では、ユーザースレッドが完了するのを待ってメインスレッドが実行されることもあれば、ユーザースレッドが少し実行された後、メインスレッドが再び「実行中」状態に戻って最後まで実行されることもある、といった具合でした。常に結果が同じになるようにするにはどうすればよいのでしょうか。それにはsleep()メソッドを使用します。たとえば、先のプログラム例で、"back in main"というメッセージが必ず"top o' the stack"よりも先に表示されるようにするには、どこでsleep()メソッドを呼び出せばよいか考えてみてください。答えは1つとは限りません。

以下に一例をあげておきます。

```
public class MyRunnable implements Runnable {

  public void run() {
    go();
  }

  public void go() {

    try {
      Thread.sleep(2000);
    } catch(InterruptedException ex) {
      ex.printStackTrace();
    }

    doMore();
  }

  public void doMore() {
    System.out.println("top o' the stack");
  }
}

class ThreadTestDrive {
  public static void main (String [] args) {
    Runnable theJob = new MyRunnable();
    Thread t = new Thread(theJob);
    t.start();
    System.out.println("back in main");
  }
}
```

ここでsleep()を呼び出せば、ユーザースレッドは「実行中」状態でなくなる

メインスレッドが再び「実行中」状態になり、まず"back in main"というメッセージが表示される。ユーザースレッドは少なくとも2秒間、「実行中」状態に戻らない。「実行中」状態に戻った後はdoMore()メソッドを呼び出して"top o'the stack"というメッセージを表示する

実行結果はこのように常に同じになる

```
File  Edit  Window  Help  SnoozeButton
% java ThreadTestDrive
back in main
top o' the stack
% java ThreadTestDrive
back in main
top o' the stack
% java ThreadTestDrive
back in main
top o' the stack
% java ThreadTestDrive
back in main
top o' the stack
% java ThreadTestDrive
back in main
top o' the stack
```


This PDF is prepared for tools@maruhouse.org, personal use only.

ユーザースレッドを 2 つ作る

スレッドには名前を付けることができます。また、特に指定しない場合でも、デフォルトの名前が付けられます。この名前を利用すれば、「今、どのスレッドが実行されているか」を確認することができます。以下に示すプログラムでは、ユーザースレッドを 2 つ作成し、どちらにも、「自分の名前を繰り返し表示する」という同じ「ジョブ」を与えています。このプログラムを実行すると、画面に実行中のスレッドの名前が（実行が続く限り何度も繰り返し）表示されることになります。

```
public class RunThreads implements Runnable {

  public static void main(String [] args) {
    RunThreads runner = new RunThreads();
    Thread alpha = new Thread(runner);
    Thread beta = new Thread(runner);
    alpha.setName("Alpha thread");
    beta.setName("Beta thread");
    alpha.start();
    beta.start();
  }

  public void run() {
    for (int i =0; i < 25; i++) {
      String threadName = Thread.currentThread().getName();
      System.out.println(threadName + " is running");
    }
  }
}
```

Runnableオブジェクトの作成

2 つのスレッドを作成する。どちらもコンストラクタの引数には同じRunnableオブジェクトを指定する（2 つのスレッドに同じジョブを与えるということ。このことについては後述）

スレッドに名前を付ける

スレッドの起動

どちらのスレッドもこのループを実行し、自分の名前を繰り返し画面に表示する

繰り返し回数を 25 回にした時の実行結果（の一部）

```
File  Edit  Window  Help  Centauri
Alpha thread is running
Alpha thread is running
Alpha thread is running
Beta thread is running
Alpha thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Alpha thread is running
```

実行結果

上記のプログラムの実行結果はどうなるでしょうか。2 つのスレッドの名前が交互に表示されるのでしょうか。それとも、一方の名前が繰り返し表示されてからもう一方の名前が表示されるという具合になるのでしょうか。スレッドの切り換えの頻度はどのくらいなのでしょうか。

ここまで読んできた方には、こうした問いに対する答えがもうわかっているはずです。答えは「実行してみなければわからない」です。スケジューラ次第でどうにでも実行結果が変わりうるからです。OS、JVM、CPU などが変われば、実行結果は違ったものになるでしょう。

Mac OS X 10.2（Jaguar）で実行した場合、ループの繰り返し回数が 5 回以下であれば、ほぼ、まず Alpha スレッドの処理が完了し、その後に Beta スレッドの処理が完了するという結果になりました。絶対とは言えませんが、かなりの確率でそういう結果が得られたのです。

しかし、繰り返しの回数が 25 回を超えると、実行結果に変化が見られるようになりました。Alpha スレッドが完了しないうちに Beta スレッドの処理が始まることが増えたのです。

This PDF is prepared for tools@maruhouse.org, personal use only.

スレッドの問題点、「競合」

スレッドにも問題はあります。同時に複数のスレッドを動かすと「競合」という現象が起きる恐れがあるのです。競合が起きるとデータの破壊につながることがあります。データが破壊されるというのは、周知のとおり、非常に困った事態です。

競合というのは、複数のスレッドが同時に同じオブジェクトのデータにアクセスする、という現象です。つまり、別々のスタック上に置かれたメソッドが、ヒープ上の同じオブジェクトのゲッターメソッド、セッターメソッドなどを呼び出してしまう、ということです。

スレッドはどれも他のスレッドの動きにまったく関知しません。どのスレッドも世の中にスレッドは他に存在しないかのように動くのです。他のスレッドが動いている間は、「意識がない」のと同じような状態に置かれるのですから、その間に何が起きているのかは知りようがないとも言えます。いったん「実行可能」、あるいは「ブロック」の状態になって、再び「実行中」に戻った時は、処理を止めた事実すらなかったかのように動きます。

This PDF is prepared for tools@maruhouse.org, personal use only.

離婚の危機？

Dr.steve ショー

このカップルは修復可能か

[番組の第 42 回より抜粋]

みなさん、こんにちは。「Dr.Steve ショー」の時間です。

今回は「お金と睡眠」という、特に離婚の原因になりやすい問題を抱えるご夫婦のお話です。

Ryan、Monica のお二人は、ベッドと銀行口座を共有にされています。しかし、1 つの口座を二人で使った場合には誰しも直面する問題に悩まされ、有効な解決策を見つけられずにいます。

Monica さんに詳しいお話を伺ってありますので、聞いてみましょう。

「Ryan と私は、預金を引き出す前にはよく確認して、残高が絶対にマイナスにならないようにしようって 2 人で話しあって決めたんです。いつも残高より少ない額を引き出すようにすればいいことだから簡単なはずですよね。なのに、残高がマイナスになるってことが何回も起きるんです」

「そんなこと、あり得ない。自分たちのやり方に間違いはないはず……、って思っていたところに、その謎が解けるような事件がありました」

「その日、Ryan は 50 ドル必要になり、残高を確認しました。残高は 100 ドルだったので、問題ないと思い、50 ドル引き出すことにしたのはいいんですが、彼はすぐに引き出しをせずに眠ってしまったんです」

Ryan は口座の残高を確認した後、引き出しをする前に眠ってしまった。目を覚ました彼は再度残高を確認することなく、引き出しをした

「Ryan が寝ている間に 100 ドル必要になった私は、まず残高が 100 ドルあることを確認しました。Ryan は寝ていたので、まだお金が引き出されていなかったんです。問題ないと思った私は 100 ドル引き出しました。その時点でもまだ残高はマイナスにはなっていません。しばらくして Ryan は目を覚ましました。そして、すぐに 50 ドル引き出してしまったんです。彼は自分が眠っていたことに気付いていなかったので、もう一度残高の確認が必要だとはまったく思っていませんでした。Dr.Steve、どうしたらいいんでしょうか！」

このお二人を救う何かいい方法はないものでしょうか。もう別れるしかない運命なのでしょうか。Ryan さんが突然眠ってしまうのを防ぐことはできないようです。だからといってご主人が目覚めるまで Monica さんが口座に手を付けないということも不可能でしょう。

すぐに CM が入りますから、その間にみなさんで考えてみてください。

二人で銀行口座を共有する Ryan と Monica は問題を抱えていた……

This PDF is prepared for tools@maruhouse.org , personal use only.

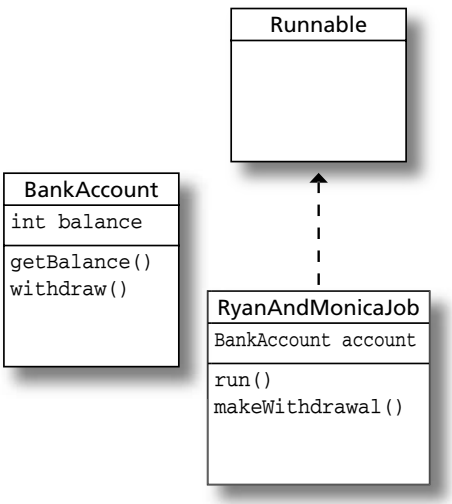
複数のスレッドによるオブジェクトの共有

Ryan と Monica の問題は、2 つのスレッドが 1 つのオブジェクトを共有していることによって起きていると言えます。

この状況を Java プログラムにしたとすると、そのプログラムは、2 人の行動を表すクラス（ここでは MonicaAndRyanJob クラスとする）と、銀行口座を表すクラス（ここでは BankAccount クラスとする）という 2 つのクラスから構成されることになります。MonicaAndRyanJob は、スレッドの「ジョブ」に対応するクラスなので、Runnable をインプリメントします。また、この場合の「ジョブ」は、「口座の残高のチェック」と「現金の引き出し」です。スレッドは、残高チェック終了後、引き出しの前にスリープします。

MonicaAndRyanJob クラスには、BankAccount 型の変数があり、この変数には、銀行口座に対応するオブジェクトへの参照が格納されます。

プログラムのコードは、おおまかには以下のような機能を備えたものになるでしょう。



① RyanAndMonicaJob クラスのインスタンスを作成する。

RyanAndMonicaJob クラスは Runnable インターフェースをインプリメントするクラス（ジョブに対応する）。Monica と Ryan はまったく同じ行動（残高の確認、現金の引き出し）をとるため、ここではインスタンスは 1 つでかまわない。

```
RyanAndMonicaJob theJob = new RyanAndMonicaJob();
```

② 同じ引数（RyanAndMonicaJob オブジェクト）を指定して 2 つの Thread オブジェクトを作成する。

```
Thread one = new Thread(theJob);
Thread two = new Thread(theJob);
```

③ スレッドに名前を付け、起動する。

```
one.setName("Ryan");
two.setName("Monica");
one.start();
two.start();
```

④ 2 つのスレッドに対応する run() メソッド（残高の確認と引き出しを行う）を適宜実行する。

一方のスレッドは Ryan に、もう一方のスレッドは Monica に対応する。どちらのスレッドでも、残高を確認し、残高が十分であれば現金を引き出す、という処理を行う。

```
if (account.getBalance() >= amount) {
  try {
    Thread.sleep(500);
  } catch(InterruptedException ex) {ex.printStackTrace(); }
}
```

run() メソッドでは、残高を確認し、十分であれば引き出す、という処理を行う

残高がマイナスになることを防止する仕組みが必要

残高確認の後、引き出しの前にスレッドがスリープしてしまうことが問題……

This PDF is prepared for tools@maruhouse.org, personal use only.

「Ryan と Monica」プログラム

```
class BankAccount {
  private int balance = 100;

  public int getBalance() {
    return balance;
  }
  public void withdraw(int amount) {
    balance = balance - amount;
  }
}

public class RyanAndMonicaJob implements Runnable {

  private BankAccount account = new BankAccount();

  public static void main (String [] args) {
    RyanAndMonicaJob theJob = new RyanAndMonicaJob();
    Thread one = new Thread(theJob);
    Thread two = new Thread(theJob);
    one.setName("Ryan");
    two.setName("Monica");
    one.start();
    two.start();
  }

public void run() {
  for (int x = 0; x < 10; x++) {
    makeWithdrawal(10);
    if (account.getBalance() < 0) {
      System.out.println("Overdrawn!");
    }
  }
}

private void makeWithdrawal(int amount) {
  if (account.getBalance() >= amount) {
  System.out.println(Thread.currentThread().getName() + " is about to withdraw");
    try {
      System.out.println(Thread.currentThread().getName() + " is going to sleep");
      Thread.sleep(500);
    } catch(InterruptedException ex) {ex.printStackTrace(); }
    System.out.println(Thread.currentThread().getName() + " woke up.");
    account.withdraw(amount);
    System.out.println(Thread.currentThread().getName() + " completes the withdrawl");
    }
  else {
    System.out.println("Sorry, not enough for " + Thread.currentThread().getName());
  }
}
```

残高を格納する変数。初期値は仮に「100 ドル」とする

RyanAndMonicaJob オブジェクトは 1 つだけ作成する。銀行口座は 1 つだけで、2 つのスレッドがいずれも同じ口座にアクセスするため

Runnable オブジェクト（ジョブに対応する）を作成

同じ Runnable オブジェクトを引数に指定して 2 つのスレッドを作成する。これにより、2 つのスレッドは、同じ Runnable オブジェクトの同じインスタンス変数（account）にアクセスすることになる

run() メソッドでは、ループを使って現金引き出しを繰り返し試行する。引き出しの度に残高がマイナスになっていないかを確認する

残高を確認し、十分でなければメッセージを出力するだけにとどめる。残高が十分であれば、まずスレッドをスリープさせ、スリープ解除後に引き出しを行う

プログラムの実行状況がわかりやすいよう、メッセージ出力のための println() メソッドを多用している

This PDF is prepared for tools@maruhouse.org, personal use only.

```
File Edit Window Help Visa
File Edit Window Help Visa
Ryan is about to withdraw
Ryan is going to sleep
Monica woke up.
Monica completes the withdrawl
Monica is about to withdraw
Monica is going to sleep
Ryan woke up.
Ryan completes the withdrawl
Ryan is about to withdraw
Ryan is going to sleep
Monica woke up.
Monica completes the withdrawl
Monica is about to withdraw
Monica is going to sleep
Ryan woke up.
Ryan completes the withdrawl
Ryan is about to withdraw
Ryan is going to sleep
Monica woke up.
Monica completes the withdrawl
Sorry,not enough for Monica
Sorry,not enough for Monica
Sorry,not enough for Monica
Sorry,not enough for Monica
Sorry,not enough for Monica
Ryan woke up.
Ryan completes the withdrawl
Overdrawn!
Sorry, not enough for Ryan
Overdrawn!
Sorry, not enough for Ryan
Overdrawn!
Sorry, not enough for Ryan
Overdrawn!
```

この現象が起きる理由は？ ⟶

makeWithdrawal()メソッドでは引き出しの前に必ず残高をチェックするが、実際には残高を超える額の引き出しが行われてしまう

この現象は以下のような経緯で起きると考えられる。

1. Ryanのスレッドが、残高が十分にあることを確認した後、スリープする。
2. Ryanのスレッドがスリープしている間に、Monicaのスレッドも、残高が十分であることを確認する。Ryanのスレッドがスリープから脱した後に引き出しを行うということには、Monicaのスレッドは関知しない。
3. 残高確認後、Monicaのスレッドはスリープする。
4. Ryanのスレッドがスリープから脱し、引き出しを行う。
5. Monicaのスレッドがスリープから脱し、引き出しを行う。引き出し前に再度残高を確認するわけではないので、引き出し額が残高を上回る可能性がある。Ryanのスレッドがすでに引き出しを行ってしまったため、残高は変わっている。

Monicaのスレッドでの残高確認は妥当なタイミングで行われているとは言えない。残高確認後、Ryanのスレッドによる引き出しが行われてしまうので、確認が意味をなさなくなっている。

Monicaのスレッドは、Ryanのスレッドがスリープから脱し、引き出しを完了するまで口座へのアクセスを待つ必要がある。MonicaとRyanの立場が逆の場合も同様のことが言える。

This PDF is prepared for tools@maruhouse.org, personal use only.

オブジェクトへのアクセスには「ロック」をかけることができる

ここでの「銀行口座」オブジェクトには以下のような特性のロックが必要

① ロックは、口座取引（残高の確認、引き出し）に対してかけるようにする。ロックを解除する「鍵」は 1 つだけ。誰も口座にアクセスしていない時には、「鍵」はいつでも使える状態になる。

誰も口座にアクセスしていない時には、口座取引へのロックは解除されている

② スレッドが口座にアクセスする際（残高のチェック、引き出しをする際）には、ロックをかける。ロックがかけられると、ロックをかけたスレッド以外、口座にはアクセスできなくなる。また、「鍵」も、ロックをかけたスレッド以外には利用できなくなる。

口座にアクセスするスレッドは、ロックをかける。「鍵」は他のスレッドには利用できなくなる

③ ロックは口座取引が終わるまで続けられる。1 つしかない「鍵」も独占し続け、他のスレッドには利用できないようにする。口座取引が完了し、ロックが解除され、「鍵」が元の状態に戻されるまで、他のスレッドが口座にアクセスすることはない。

このようにロックが使えれば、ライアンのスレッドが残高確認後にスリープしてしまっても、引き出す時点になって残高が変わってしまっているという事態になることはない。

口座取引が完了後、ロックは解除され、「鍵」は元に戻される。これで他のスレッドも口座にアクセスできるようになる

This PDF is prepared for tools@maruhouse.org, personal use only.

makeWithdrawal()メソッドに同時にアクセスできるスレッドを1つに限定する

「synchronized」キーワードを使用すれば、メソッドに同時にアクセスするスレッドを1つに限定できる

この方法で、そのメソッドで操作するデータ(銀行口座のデータ)の整合性を維持することができる

Ryan と Monica の問題を解決するには、makeWithdrawal() に同時にアクセスできるスレッドを1つに限定するということが必要です。そうすれば、1つのスレッドの口座取引の途中で別のスレッドが取引を行ってしまう、ということは防げます。

これにより、残高を確認した後でスリープしてしまっても、引き出しの時点で残高が変わってしまっているということは起きなくなるのです。

メソッドにアクセスできるスレッドを1つに限定するためには、「synchronized」というキーワードを使用します。

これが、実は、前ページで説明した「ロック」になります。つまり、厳密にはロックは「口座」オブジェクトそのものにかけるのではなく、口座取引を行うためのメソッドにかけるのです。makeWithdrawal() の宣言に「synchronized」というキーワードを加えれば、1つのスレッドがこのメソッドの処理を完了させるまでの間、別のスレッドが同じスレッドを使用することはできなくなります。

スレッドによるオブジェクトへのアクセスにロックをかけると言っても、アクセスするオブジェクトそのもの、あるいはRunnableオブジェクト、Threadオブジェクトなどにロックをかけるわけではない、という点には注意が必要です。

このロックに関しては次のページでさらに詳しく説明します。ここでは、メソッドの宣言にどのように「synchronized」キーワードを加えるのかを見ておいてください。

```
private synchronized void makeWithdrawal(int amount) {

  if (account.getBalance() >= amount) {
     System.out.println(Thread.currentThread().getName() + "is about to withdraw");
     try {
        System.out.println(Thread.currentThread().getName() + "is going to sleep");
        Thread.sleep(500);
     } catch(InterruptedException ex) {ex.printStackTrace(); }
     System.out.println(Thread.currentThread().getName() + "woke up.");
     account.withdraw(amount);
     System.out.println(Thread.currentThread().getName() + "completes the withdrawl");
  } else {
     System.out.println("Sorry, not enough for" + Thread.currentThread().getName());
  }
}
```


This PDF is prepared for tools@maruhouse.org, personal use only.

すべてのオブジェクトが備える「ロック」

オブジェクトはすべて「ロック」を備えています。通常、このロックは解除されているため、その存在（ロックそのものだけでなく、ロックの施錠、解錠に使用する「鍵」の存在も）を意識することはありません。オブジェクトのロックが機能するのは、メソッドの宣言に「synchronized」というキーワードが使われた時だけです。ロックは、「synchronized」宣言されたメソッドに対してはたらき、そのメソッドにアクセスできるスレッドが1つに限定されます（「鍵」を利用できるスレッドが1つに限定されるということ）。

わかりづらいのは、1つのオブジェクトに「synchronized」宣言されたメソッドが複数存在する場合です。この場合、個々のメソッドに1つずつのスレッドがアクセスできるわけではありません。オブジェクト中の「synchronized」宣言されたメソッドのうち、1つにでもアクセスがあれば、他のsynchronizedメソッドにはどのスレッドもアクセスできないのです。

なぜこのようになっているかというと、同じオブジェクトのメソッドであれば、操作するインスタンス変数が共通している可能性が高いからです。2つのスレッドが同じオブジェクトのメソッドにアクセスすれば、両者のアクセスしているメソッドがたとえ異なったとしても、インスタンス変数の値に整合性がなくなる恐れが大きいのです。

メソッドを「synchronized」宣言するのは、重要なデータを保護するためです。ただ、ロックをかけるのはデータそのものではなく、データへのアクセスを行うメソッドであるという点に注意してください。

スレッドは、アクセスしようとしたメソッドが「synchronized」宣言されたものであった場合、まずロックの「鍵」を探します（この処理は自動的に行われるのでプログラマは関知しなくてよい。APIにも「鍵」を探すためのメソッドなどは用意されていない）。鍵が使えるようになっていれば、スレッドはメソッドへのアクセスを開始します（この時、スレッドは鍵を「隠し」、他のスレッドには使えないようにする）。

いったん鍵を手にしたスレッドは、synchronizedメソッドの処理が完了するまでの間、その鍵を大切に持ち続け、決して手放しません。その間、他のスレッドは鍵を使えないため、同じオブジェクトのすべてのsynchronizedメソッドにアクセスできなくなります。鍵は各オブジェクトに1つしか用意されていないからです。

Javaオブジェクトはすべて「ロック」を備えている。個々のロックには鍵が1つだけ用意されている。通常、ロックは解除されているため、誰も関知しない

synchronized宣言されたメソッドは、鍵がなければ利用できない。つまり、synchronized宣言されたメソッドを利用できるスレッドは1つに限定される

This PDF is prepared for tools@maruhouse.org, personal use only.

「更新が無効になる」という問題

マルチスレッドのプログラムには、先の「RyanとMonica」のような問題以外に、更新したはずのデータがもとに戻ってしまう、という問題もよく発生します。これは、データベースに詳しい人にとっては馴染み深い問題でしょう。この問題はたとえば、以下のように、2ステップでデータを更新するコードを書いた時に発生します。

ステップ1：ある変数（ここではbalance）の値を別の変数（ここではi）に代入する

```
int i =balance;
```

ステップ2：代入した値に1を加え、元の変数に再び代入する

```
balance = i + 1;
```

このコードは下に示すように、1行にまとめて書くこともできるので、わざわざ2行に分けて書くことは多くありません。ただ、ここでは、マルチスレッドの問題点がわかりやすいよう、あえて2行に分けて書いています。

```
balance++;
```

プログラムがただ、変数をインクリメントするだけの単純なものであれば、特にどうということもないのですが、このコードを以下のような複雑なプログラムの中で使った場合に問題は起きます。下のプログラムでは、スレッドが2つ起動され、いずれのスレッドも変数「balance」のインクリメントを行おうとします。

```
class TestSync implements Runnable {

  private int balance;

  public void run() {
    for (int i = 0; i < 50; i++) {
      increment();
      System.out.println("balance is " + balance);
    }
  }

  public void increment() {
    int i = balance;
    balance = i + 1;
  }
}

public class TestSyncTest {
  public static void main (String [] args) {
    TestSync job = new TestSync();
    Thread a = new Thread(job);
    Thread b = new Thread(job);
    a.start();
    b.start();
  }
}
```

どちらのスレッドもbalanceのインクリメントを50回繰り返す

ここが重要。このコードだと、変数iに代入を行った時点でのbalanceの値に1を加えることになる。インクリメントの時点でのbalanceの値そのものに1を加えるわけではない

This PDF is prepared for tools@maruhouse.org, personal use only.

コードを実行してみる……

① **スレッドAの処理がしばらく行われる。**

変数balanceの値が変数iに代入される。
Balanceの値は 0 なので、iの値も 0 になる。
iに 1 が加えられ、その結果が変数balanceに代入される。
これにより、balanceの値は 1 になる。
変数balanceの値が変数iに代入される。
Balanceの値は 1 なので、iの値も 1 になる。
iに 1 が加えられ、その結果が変数balanceに代入される。
これにより、balanceの値は 2 になる。

② **スレッドBの処理がしばらく行われる。**

変数balanceの値が変数iに代入される。
Balanceの値は 2 なので、iの値も 2 になる。
iに 1 が加えられ、その結果が変数balanceに代入される。
これにより、balanceの値は 3 になる。
変数balanceの値が変数iに代入される。
Balanceの値は 3 なので、iの値も 3 になる。

[この後本来は、balanceの値が 4 になるはずだが、
ここでスレッドBが「実行可能」状態になってしまう]

③ **スレッドAが処理を続行する。**

変数balanceの値が変数iに代入される。
Balanceの値は 3 なので、iの値も 3 になる。
iに 1 が加えられ、その結果が変数balanceに代入される。
これにより、balanceの値は 4 になる。
変数balanceの値が変数iに代入される。
Balanceの値は 4 なので、iの値も 4 になる。
iに 1 が加えられ、その結果が変数balanceに代入される。
これにより、balanceの値は 5 になる。

④ **スレッドBが処理を続行する。**

iに 1 が加えられ、その結果が変数balanceに代入される。
これにより、balanceの値は 4 になる。

注意!!

スレッドAが値を 5 に更新したにもかかわらず、スレッドBは、その更新がまったく行われなかったかのように処理を続ける

スレッドAが行った更新は失われている。スレッドBは、balanceの値をiに代入した時点でいったん動きを止めたが、その後、止まったという事実さえなかったかのように実行を続ける

This PDF is prepared for tools@maruhouse.org, personal use only.

increment()メソッドをsynchronized宣言する

このように、「更新が無効になる」という問題も、メソッドをsynchronized宣言することで解決できます。ここでは、increment()メソッドをsynchronized宣言することになります。これによって、2ステップの処理が途中で分割されるようなことはなくなるのです。

```
public  synchronized  void increment() {
  int i = balance;
  balance = i + 1;
}
```

increment()メソッドをsynchronized宣言すれば、1つのスレッドがこのメソッドの処理を完了するまで、別のスレッドがこのメソッドを利用することはできなくなる

素朴な疑問に答えます

Q：synchronized宣言は、どのメソッドでも行っておくべきではないですか。そうすれば、マルチスレッドで問題が起きることはまったくなくなるはずですが。

A：それは得策とは言えません。synchronized宣言にはデメリットもあるからです。まず、synchronized宣言をすると、スレッドがメソッドにアクセス可能かを確かめる分だけオーバーヘッドが増えます。つまり、synchronized宣言されたメソッドが利用される時にはパフォーマンスが低下しやすいということです（パフォーマンス低下の度合いが、ユーザーにはほとんどわからないほど小さいこともありますが）。

また、複数の処理を同時に進めることができないためにプログラムの実効速度が落ちることもあります。1つのスレッドが処理を終えるまで他のスレッドは待機することになるからです。これがプログラムの種類によっては大きな問題になることがあります。

そして、デメリットの中でも最も深刻なのはsynchronized宣言をしたメソッドがデッドロックの原因になる恐れがあるということです（詳しくは次のページを参照）。

以上のようなことから、synchronized宣言をするメソッドはどうしても必要というものだけにとどめるのが無難と言えるでしょう。また、メソッドの中の一部のコードだけをsynchronized宣言する、というテクニックもあるので、場合によってはそれを利用すべきです。このテクニックについては、本書では詳しくとりあげませんが、一応、コード例を右に示しておきます。

doStuff()を呼び出している部分はsynchronized宣言する必要はない。そこでメソッド全体ではなく、一部のコードだけをsynchronized宣言する

```
public void go() {
   doStuff();

   synchronized(this) {
      criticalStuff();
      moreCriticalStuff();
   }
}
```

ここでは、2つのメソッドを呼び出す部分だけをsynchronized宣言している。これにより、2行のコードが分割できないものになる。synchronizedの後の「this」という引数は、「コードへのアクセスにどのオブジェクトの鍵を利用するか」を指定したもの

この場合の引数には必ず「this」を指定しなければならない、というわけではないが、通常は「this」を指定する。これにより、メソッドのsynchronized宣言の場合と同様、コードが属するオブジェクトの鍵が利用されることになる

This PDF is prepared for tools@maruhouse.org, personal use only.

① スレッドAの処理がしばらく行われる。

increment()メソッドの利用が試みられる。
このメソッドはsynchronized宣言されているので、利用には「鍵」が必要。
変数balanceの値が変数iに代入される。
balanceの値は0なので、iの値も0になる。
iに1が加えられ、その結果が変数balanceに代入される。
これにより、balanceの値は1になる。
increment()メソッドの処理が完了したため、スレッドAが独占していた「鍵」がいったん解放される（すべてのスレッドが利用できる状態に戻される）。
再度increment()メソッドへのアクセスが行われ、「鍵」が再びスレッドAの独占になる。
変数balanceの値が変数iに代入される。
balanceの値は1なので、iの値も1になる。
［ここでスレッドAは「実行可能」状態になるが、increment()メソッドの処理がまだ完了していないので、鍵はスレッドAが独占し続ける］

② スレッドBが「実行すべきスレッド」として選択される。

increment()メソッドへのアクセスが試みられる。このメソッドはsynchronized宣言されているので、利用には「鍵」が必要だが、鍵はスレッドAが独占しているので利用できない。
［スレッドBは、鍵が解放されるのを待つことになる］

③ スレッドAが再び「実行中」状態になり、処理を続行する（鍵はそのまま独占し続けている）

iに1が加えられ、その結果が変数balanceに代入される。
これにより、balanceの値は2になる。
鍵が解放される。
［ここでスレッドAが「実行可能」状態に戻る。increment()メソッドの処理は完了しているので、もう鍵は独占していない］

④ スレッドBが「実行すべきスレッド」として選択される

increment()メソッドへのアクセスが試みられる。このメソッドはsynchronized宣言されているので、利用には「鍵」が必要。
鍵は解放されているので、スレッドBは鍵を独占できる。
変数balanceの値が変数iに代入される。
［以下、同様に処理が続く……］

This PDF is prepared for tools@maruhouse.org, personal use only.

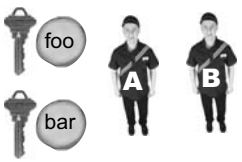
synchronized宣言の問題点

synchronized宣言を使う際に特に注意すべきこととしては、スレッドのデッドロックがあげられるでしょう。「デッドロック」とは、プログラムの2つのスレッドが、どちらもお互いが必要とする「鍵」を独占してしまう、という現象のことです。この状況に陥ってしまえば、もう抜け出す術はありません。2つのスレッドはただ、ひたすら鍵が解放されるのを待って待って待ち続けるだけで、プログラムの処理はまったく前に進みません。

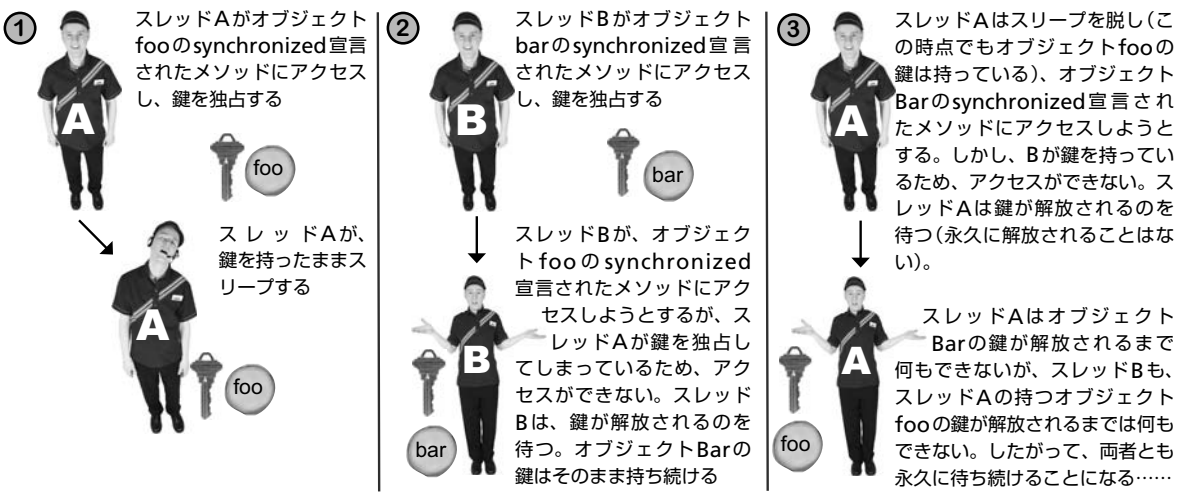
データベース、アプリケーションサーバに詳しい人であれば、このデッドロックについてもよく知っているでしょう。データベースも、通常、synchronized宣言と同じようなロック機能を備えているからです。ただ、データベースなどの場合には、デッドロックに対処する機能も同時に備えていることが多くなっています。トランザクション（常にひとかたまりで行われ、分割できない処理のこと）が複数になり、どちらも長時間を要する、というような場合には、あらかじめデッドロックが発生することを見越した対策がとられるのです。トランザクションの途中で問題が起きた場合には、システム、データをトランザクション開始前の状態に戻すことができる、「ロールバック」の機能なども備えているのが普通です。

Javaにはデッドロックに対処するための機能は用意されていません。それどころかデッドロックが起きたことを関知することさえできないのです。したがって、プログラムを書く段階で十分に注意してデッドロックが起きないようにする必要があります。マルチスレッドのプログラムを頻繁に書く人は、『Javaスレッドプログラミング』（Scott Oaks、Henry Wong 共著、戸松豊和監訳、西村利浩訳、オライリー・ジャパン発行）などの本で、デッドロックの回避方法について勉強した方がよいでしょう。デッドロック対策としてまず注意すべきなのは、スレッドの起動順序です。

2つのオブジェクトと2つのスレッドがあれば、デッドロックが起きる可能性がある



デッドロックの発生例



This PDF is prepared for tools@maruhouse.org, personal use only.

重要ポイント

- Thread.sleep()というスタティックメソッドを使うと、最低限、引数に指定した時間が経過するまでの間、スレッドの実行を止めることができる。たとえば、Thread.sleep(2000)というメソッドを呼び出すと、スレッドの実行は少なくとも 2 秒間止まることになる。
- sleep()メソッドは、InterruptedExceptionというチェック例外を発生させる可能性があるので、呼び出す際にはtry/catchブロックを使用するか、「throws」キーワードでInterruptedExceptionを宣言しておくかのいずれかの対応が必要。
- sleep()は、実行されるスレッドを意図的に変えるのに利用できる。sleep()を使用しても、処理を進行するために時間が平等に割り当てられる保証はない。1 つのスレッドの処理ばかりが進み、他のスレッドの処理があまり進まないこともある。だが、sleep()をうまく使えば、こうした偏りは解消できることが多い。
- スレッドには、setName()メソッドを使用して名前を指定できる。このメソッドを使用しなくても、すべてのスレッドにはデフォルトで名前が指定されるが、自分で名前を指定した方が、各スレッドの動きの確認が容易。動きの確認はデバッグに役立つ。
- 2 つ以上のスレッドが、ヒープ上の同じオブジェクトにアクセスすると深刻な問題が起きる恐れがある。
- 同じオブジェクトに 2 つ以上のスレッドがアクセスすると、データが破壊される（整合性がなくなる）可能性がある。たとえば、オブジェクトの重要なインスタンス変数を操作している最中にスレッドの状態が「実行中」から「実行可能」に変わったりすると、大きな問題につながりやすい。
- データの安全を確保するためには、スレッドによって行う一連の処理をひとまとめにし、途中で他のスレッドが割り込めないようにするという対策が有効。これは他のスレッドによる同じオブジェクトへのアクセスを、一連の処理が完了するまで禁止するということ。
- 同じオブジェクトへの重複アクセスを防止するには、アクセスのためのメソッドの宣言に「synchronized」というキーワードを加えるという方法が使われる。
- オブジェクトはすべて「ロック」を 1 つ備えており、それに対応する「鍵」も 1 つだけ備えている。このロックについては、多くの場合、プログラマが関知する必要はない。関知するのは、メソッドをsynchronized宣言する時のみ。
- スレッドは、synchronized宣言されたメソッドにアクセスする際、そのメソッドの属するオブジェクトの鍵を独占する必要がある。鍵がすでに他のスレッドによって独占されている場合には、鍵が解放されるのを待つ。
- 1 つのオブジェクトにsynchronized宣言されたメソッドが複数ある場合でも、オブジェクトに用意される鍵は 1 つだけ。いずれかのスレッドがsynchronized宣言されたメソッドのうちの 1 つにアクセスしてしまうと、他のスレッドは、同じオブジェクトのsynchronizedメソッドにはまったく（たとえ同じメソッドでなくても）アクセスできなくなる。同じオブジェクトのメソッドであれば、操作するインスタンス変数が共通している可能性が高いため。

This PDF is prepared for tools@maruhouse.org, personal use only.

改良型チャットクライアント

この章ではすでに簡単なチャットクライアントを紹介しています。ただ、そのクライアントはメッセージをサーバに送信する機能だけを持っていて、受信する機能を持たないというものでした。長々とスレッドの話をしてきたのは、そもそも、そのクライアントに、「メッセージの送信（ユーザーのGUI操作への対応）と受信（受信したメッセージのテキスト領域への表示）を同時に行う」という機能を持たせるためでした。

このチャットクライアントとチャットサーバーの紹介で、この章もだいたい終わりである……

```
import java.io.*;
import java.net.*;
import java.util.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class SimpleChatClient {

  JTextArea incoming;
  JTextField outgoing;
  BufferedReader reader;
  PrintWriter writer;
  Socket sock;

  public static void main(String [] args) {
    SimpleChatClient client = new SimpleChatClient();
    client.go();
  }

  public void go() {

    JFrame frame = new JFrame("Ludicrously Simple Chat Client");
    JPanel mainPanel = new JPanel();
    incoming = new JTextArea(15,50);
    incoming.setLineWrap(true);
    incoming.setWrapStyleWord(true);
    incoming.setEditable(false);
    JScrollPane qScroller = new JScrollPane(incoming);
    qScroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
    qScroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);
    outgoing = new JTextField(20);
    JButton sendButton = new JButton("Send");
    sendButton.addActionListener(new SendButtonListener());
    mainPanel.add(qScroller);
    mainPanel.add(outgoing);
    mainPanel.add(sendButton);
    setUpNetworking();

    Thread readerThread = new Thread(new IncomingReader());
    readerThread.start();

    frame.getContentPane().add(BorderLayout.CENTER,mainPanel);
    frame.setSize(400,500);
    frame.setVisible(true);
  } // go メソッドの終了
```

このあたりはほとんどが、ごく普通のGUIコードだが、コンポーネントを画面に表示するためのコードが少しこれまでと異なる……

新しいスレッドを起動する。Runnableオブジェクトとしては内部クラスを使用する。このスレッドの「ジョブ」は、サーバーとのソケット接続からデータを読み取り、受信したメッセージをテキスト領域に表示すること

This PDF is prepared for tools@maruhouse.org, personal use only.

```
private void setUpNetworking() {

  try {
    sock = new Socket("127.0.0.1", 5000);
    InputStreamReader streamReader = new InputStreamReader(sock.getInputStream());
    reader = new BufferedReader(streamReader);
    writer = new PrintWriter(sock.getOutputStream());
    System.out.println("networking established");
  } catch(IOException ex) {
    ex.printStackTrace();
  }
} // setUpNetworking メソッドの終了

public class SendButtonListener implements ActionListener {
  public void actionPerformed(ActionEvent ev) {
    try {
      writer.println(outgoing.getText());
      writer.flush();

    } catch(Exception ex) {
      ex.printStackTrace();
    }
    outgoing.setText("");
    outgoing.requestFocus();
  }
} // 内部クラスの終了
```

入力ストリーム、出力ストリームの取得にはSocketオブジェクトを使用する。先に示したチャットクライアントでは、サーバへのメッセージ送信のための出力ストリームだけを取得していたが、このチャットクライアントでは、サーバからメッセージを受け取るための入力ストリームも取得する

ユーザーによってSendボタンがクリックされた時に、テキストフィールドのメッセージをサーバに送信するためのコード。この種のコードについてはすでに説明済み

```
public class IncomingReader implements Runnable {
  public void run() {
    String message;
    try {

      while ((message = reader.readLine()) != null) {
        System.out.println("read" + message);
        incoming.append(message + "\n");

      } // while の終了
    } catch(Exception ex) {ex.printStackTrace();}
  } // run メソッドの終了
} // 内部クラスの終了
```

スレッドのジョブ。このrun()メソッドでは、ループを使用して、サーバーから読み取ったデータから一行ずつメッセージを読み取りテキスト領域に表示する（データがnullになればループを抜ける）。行の終わりには改行文字を挿入する

```
} // 外部クラスの終了
```


This PDF is prepared for tools@maruhouse.org, personal use only.

チャットサーバのコード

ここで紹介するサーバのコードは、この章で取りあげた2つのチャットクライアントの両方に対応できるものです。ただ、本当に必要最低限の機能以外は持っていないので、実用的なサーバとはとても言えません。一応、サーバとしての用はなすのですが、考えられ得るさまざまなトラブルに対応することはまったくできないので、安定して動かない可能性が高いのです。意欲のある方は、このサーバを、より安定して動くものに改良してみても勉強になるでしょう。

また、このコードには注釈が付けてありませんが、自分で注釈を付けるというのも面白いかもしれません。人が付けた注釈を読むより、自分でコードの動きをよく見て注釈を付ける方がより理解が深まるはずです。もちろん、「まだそこまでは……」という人は、特に個々のコードの意味は考えず、ともかく書いてあるとおりにサーバを作り、チャットクライアントを動かしてみる、というだけでかまいません。

チャットクライアントを動かすには、端末が2つ必要になります。一方の端末でまずサーバを起動し、その後、もう一方の端末でクライアントを起動するのです

```
import java.io.*;
import java.net.*;
import java.util.*;

public class VerySimpleChatServer {

  ArrayList clientOutputStreams;

  public class ClientHandler implements Runnable {
    BufferedReader reader;
    Socket sock;

    public ClientHandler(Socket clientSocket) {
      try {
        sock = clientSocket;
        InputStreamReader isReader = new InputStreamReader(sock.getInputStream());
        reader = new BufferedReader(isReader);

      } catch(Exception ex) {ex.printStackTrace();}
    } // コンストラクタの終了

    public void run() {
      String message;
      try {
        while ((message =reader.readLine()) != null) {
          System.out.println("read" + message);
          tellEveryone(message);

        } // while の終了
      } catch(Exception ex) {ex.printStackTrace();}
    } // run メソッドの終了
  } // 内部クラスの終了
```


This PDF is prepared for tools@maruhouse.org, personal use only.

```
  public static void main (String [] args) {
    new VerySimpleChatServer().go();
  }

  public void go() {
    clientOutputStreams = new ArrayList();
    try {
      ServerSocket serverSock = new ServerSocket(5000);

      while(true) {
        Socket clientSocket = serverSock.accept();
        PrintWriter writer = new PrintWriter(clientSocket.getOutputStream());
        clientOutputStreams.add(writer);

        Thread t = new Thread(new ClientHandler(clientSocket));
        t.start();
        System.out.println("got a connection");
      }
    } catch(Exception ex) {
      ex.printStackTrace();
    }
  } // go メソッドの終了

  public void tellEveryone(String message) {

    Iterator it = clientOutputStreams.iterator();
    while(it.hasNext()) {
      try {
        PrintWriter writer = (PrintWriter) it.next();
        writer.println(message);
        writer.flush();
      } catch(Exception ex) {
          ex.printStackTrace();
      }

    } // while の終了
  } // tellEveryone メソッドの終了
} // クラスの終了
```


This PDF is prepared for tools@maruhouse.org, personal use only.

素朴な疑問に答えます

Q：synchronized宣言によってスタティック変数の保護はできるのですか。スタティック変数の値を操作するスタティックメソッドをsynchronized宣言するということはできるのでしょうか。

A：できます。ご存知のように、スタティックメソッドは、オブジェクトへの参照を納めた変数の名前ではなく、クラス名を使って呼び出すメソッドです。先述のとおり、「ロック」はオブジェクトに用意されるものなので、「スタティックメソッドに使ってもいいのだろうか……」という疑問が湧くのは当然だと思います。しかし、幸いなことに、「ロック」を持つのはオブジェクトだけではないのです。クラスも、ロードされればロックを持つことになります。これは、たとえばヒープ上にDogオブジェクトを3つ作った場合には、Dogクラス、Dogオブジェクトに関連する「ロック」が合計で4つできる、ということを意味します。そのうちの3つはDogオブジェクトに属し、残りの1つはDogクラスに属するわけです。スタティックメソッドをsynchronized宣言した場合には、クラスに属するロックを利用することになります。1つのクラスにsynchronized宣言されたスタティックメソッドが複数ある場合でも、クラスに用意されるロックは1つだけです。いずれかのスレッドがsynchronized宣言されたスタティックメソッドのうちの1つにアクセスしてしまうと、他のスレッドは、同じクラスのsynchronizedスタティックメソッドにはまったく（たとえ同じメソッドでなくても）アクセスできなくなります。

Q：スレッドにプライオリティを付ければ、スケジューラをコントロールできると聞いたのですが……。実際はどうなのでしょうか。

A：確かにスレッドにプライオリティを付けることで、スケジューラの動きに多少影響を与えることはできます。しかし、「意のままに動かす」というわけにはいかないので注意してください。スレッドのプライオリティは数値で指定します。数値によって「このスレッドはどの程度重要か」ということをスケジューラに知らせるのです。スケジューラは一般に、プライオリティの高いスレッドができる限り長い時間「実行中」状態になり、プライオリティの低いスレッドができるだけ「実行中」にならないように動きます。とはいえ、これはあくまで一般論に過ぎません。プライオリティは「パフォーマンスが多少改善するかもしれない」というくらいの気持ちで使用し、過度な期待はしない方が無難でしょう。

Q：synchronized宣言は、保護したいデータにアクセスするゲッターメソッド、セッターメソッドだけに使えばいいのではないでしょうか。RyanとMonicaの例で、makeWithdrawal()メソッドをsynchronized宣言するのではなく、BankAccount クラスのgetBalance()メソッド、withdraw()メソッドをsynchronized宣言するというのではいけないのですか。

A：確かに、getBalance()メソッド、withdraw()メソッドもsynchronized宣言する方が望ましいとは言えます。ただ、ここでは2人の口座にアクセスするプログラムが他に存在するわけではないのでそこまではしていません。

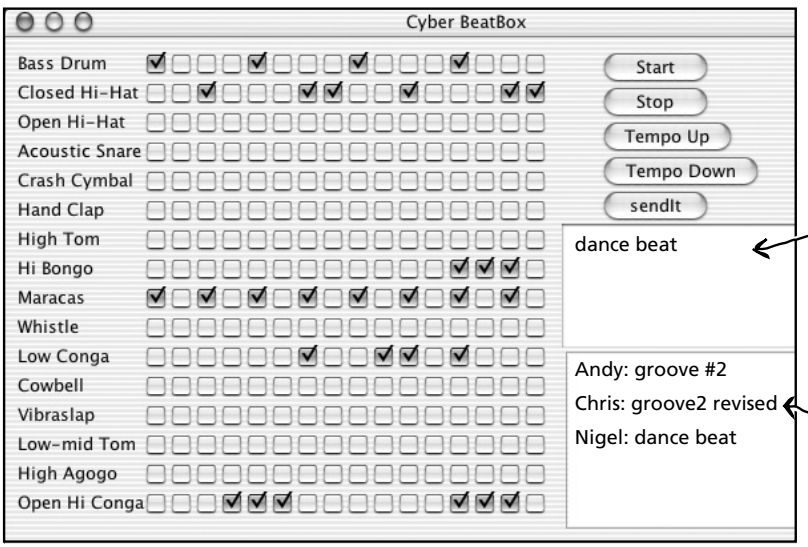
しかし、いずれにしても、makeWithdrawal()メソッドのsynchronized宣言は必要です。synchronized宣言には、あるメソッドで行う処理（あるいはメソッドの内の一部のコードで行う一連の処理）をひとかたまりの、分割できないものにする、という意味もあるのです。「分割できない」というのは、あるスレッドがその処理を開始したら、それが完了するまでの間は、他のスレッドが処理を行うことができない、という意味です。makeWithdrawal()メソッドをsynchronized宣言していいなければ、Ryanのスレッドが（getBalance()メソッドを呼び出して）口座の残高を確認しただけでスリープした場合、スリープを脱するまでの間にMonicaのスレッドが口座の確認、引き出しを行うのを防ぐことはできません。残高確認が終われば、BankAccountオブジェクトの鍵は解放されてしまうからです。

Ryanのスレッドはスリープから脱した後には再びBankAccountオブジェクトの鍵を独占し、withdraw()メソッドを呼び出すことになります。これでは、「残高がマイナスになってしまう」という問題の解決にはまったくなりません。Ryanがいくら残高を確認しても、その後スリープすれば、Monicaが引き出しを行うことは自由にできるからです。

ゲッターメソッド、セッターメソッドをsynchronized宣言することも大切ですが、「必ずひとかたまりで実行しなければならない」というコードをsynchronized宣言することも忘れてはなりません。

This PDF is prepared for tools@maruhouse.org, personal use only.

プログラム・キッチン



ここに書き込んだメッセージは、"sendIt"ボタンをクリックすると、ドラムパターンとともにサーバを経由して他のBeatBoxプログラムに送られる

他のBeatBoxプログラムからサーバを経由して送られてきたメッセージが表示される。このメッセージをクリックすると、メッセージに対応するドラムパターンがロードされる。そして、"Start"ボタンをクリックすると、パターンの演奏が開始される

さて、それでは、この章で学んだことを活かしてBeatBoxプログラムの完成バージョンを作成してみましょう。このバージョンは、サーバに接続し、他のBeatBoxプログラムとの間でドラムパターンをやりとりする機能を備えたものです（表示画面は上の図のようになる）。前章までのバージョンをどのように修正すればよいか、自分なりに考えてみてください。
完成版プログラムのコードは非常に長いものなので、付録Aに載せてあります。付録Aにはサーバのコードも載せておきました。

This PDF is prepared for tools@maruhouse.org, personal use only.

次のページに示したJavaプログラムの断片を並べ替え、下のような実行結果が得られるようにしてください。このプログラムは、4つのクラスから構成されており、それぞれのクラスの先頭の部分だけを以下に示してあります。もし、{ 、} などが抜けているところがあれば付け加えてもかまいません。

```
public class TestThreads {
```

```
class Accum {
```

```
class ThreadOne
```

```
class ThreadTwo
```

```
% java TestThreads
one 98098
two 98099
```

ボーナスクエスチョン：Accumクラスには、"private static Accum a = new Accum();" というコードが含まれるが、それはなぜか？

This PDF is prepared for tools@maruhouse.org, personal use only.

```
Accum a = Accum.getAccum();
```

```
Thread one = new Thread(t1);
```

```
System.out.println("two "+a.getCount());
```

```
} catch(InterruptedException ex) { }
```

```
ThreadTwo t2 = new ThreadTwo();
```

```
try {
```

```
return counter;
```

```
counter += add;
```

```
Thread two = new Thread(t2);
```

```
implements Runnable {
```

```
one.start();
```

```
Accum a = Accum.getAccum();
```

```
Thread.sleep(50);
```

```
public static Accum getAccum() {
```

```
} catch(InterruptedException ex) { }
```

```
private static Accum a = new Accum();
```

```
private int counter = 0;
```

```
public void run() {
```

```
a.updateCounter(1);
```

```
Thread.sleep(50);
```

```
for(int x = 0; x < 99; x++) {
```

```
implements Runnable {
```

```
a.updateCounter(1000);
```

```
public int getCount() {
```

```
return a;
```

```
System.out.println("one "+a.getCount());
```

```
public void updateCounter(int add) {
```

```
public static void main(String [] args) {
```

```
for(int x = 0; x < 98; x++) {
```

```
two.start();
```

```
try {
```

```
public void run() {
```

```
private Accum() { }
```

```
ThreadOne t1 = new ThreadOne();
```


This PDF is prepared for tools@maruhouse.org, personal use only.

エクササイズの答え

```
public class TestThreads {
  public static void main(String [] args) {
    ThreadOne t1 = new ThreadOne();
    ThreadTwo t2 = new ThreadTwo();
    Thread one = new Thread(t1);
    Thread two = new Thread(t2);
    one.start();
    two.start();
  }
}

class Accum {
  private static Accum a = new Accum();
  private int counter = 0;

  private Accum() { }

  public static Accum getAccum() {
    return a;
  }

  public void updateCounter(int add) {
    counter += add;
  }

  public int getCount() {
    return counter;
  }
}

class ThreadOne implements Runnable {
  Accum a = Accum.getAccum();
  public void run() {
    for(int x = 0; x < 98; x++) {
      a.updateCounter(1000);
      try {
        Thread.sleep(50);
      } catch(InterruptedException ex) { }
    }
    System.out.println("one "+a.getCount());
  }
}
```

Accumクラスのスタティックインスタンスを作る

private宣言したコンストラクタ

```
class ThreadTwo implements Runnable {
  Accum a = Accum.getAccum();
  public void run() {
    for(int x=0; x < 99; x++) {
      a.updateCounter(1);
      try {
        Thread.sleep(50);
      } catch(InterruptedException ex) { }
    }
    System.out.println("two "+a.getCount());
  }
}
```

このプログラムでは、それぞれに対応するRunnableオブジェクトが異なる2つのスレッドが、同じAccumオブジェクトにアクセスすることになる。

private static Accum a = new Accum();

というコードは、Accumクラスのスタティックインスタンス(この場合、「スタティック」という言葉が「クラスに1つ」という意味になることに注意)を作成するためのもの。また、Accumクラスのコンストラクタがprivate宣言されているのは、Accum以外のクラスではAccumオブジェクトを作成できないようにするため。private宣言したコンストラクタとスタティックゲッターメソッドを組み合わせて使用すると、「シングルトン」と呼ばれるオブジェクトが作成できる。シングルトンは、あるクラスのインスタンスの数を制限する手段として利用される。また、通常、シングルトンを作る場合、インスタンスの数は1つに限定される(「シングルトン」という名前はそこから来ている)。

This PDF is prepared for tools@maruhouse.org, personal use only.

エアロックの抱える深刻な問題

5分間ミステリー

会議室に入ったSarahがふと窓の外に目を向けると、インド洋の夜明けが見えた。船内の会議室は実に狭苦しかったが、夜の闇におおわれていた地球が次第に青と白に変わっていくのを見ているうちに、サラの心は畏れと感謝の念で満たされていった。

今朝の会議では、エアロックの制御システムについて話し合うことになっていた。宇宙ステーション建設の工程も終わりに近づき、船外活動が必要な場面も大幅に増えている。それにつれ、当然、エアロックを通って出入りする乗組員の数も増えることになる。「おはよう、Sarah。いいタイミングだね。これから詳しいデザインレビューを始めるところだよ」とTomが言った。

「君も知っているとおり、エアロックの内側と外側にはGUI端末が取り付けられている。船外の環境にも耐えられるよう特別な強化を施した端末だ。とにかく、エアロックを通って船外に出る者、船内に戻って来る者は、必ずこの端末を使ってエアロックを操作するためのプログラムを動かすことになる」Tomの言葉にSarahはうなずくと「船外へ出る場合、船内に戻る場合に、どのような順序でメソッドが実行されるのか、教えてくれないかしら」と尋ねた。するとTomは立ち上がり、「じゃあ、まずは船外へ出る場合からだ」と言ってホワイトボードにメソッドの名前を書き始めた。

```
orbiterAirlockExitSequence()
  verifyPortalStatus();
  pressurizeAirlock();
  openInnerHatch();
  confirmAirlockOccupied();
  closeInnerHatch();
  decompressAirlock();
  openOuterHatch();
  confirmAirlockVacated();
  closeOuterHatch();
```

「途中で割り込みが入らないよう、orbiterAirlockExitSequence()によって呼び出されるメソッドはすべてsynchronized宣言してある」とTomは説明した。「ハッチが開いたら、中にズボンをおろしているヤツがいた、なんてことになったら困るからな」

Tomの冗談に皆笑ったが、Sarahはホワイトボードを消すTomの姿を見ながら何か腑に落ちないものを感じていた。そして、再びTomがホワイトボードにメソッド名を書き連ね始めた時、彼女はある問題に気付き、叫んだ。「ちょっと待って！このプログラムには重大な欠陥があるわ。会議はいったん中止して、もう一度よくコードを確かめましょう。このままだと、とんでもないことになるかもしれない」

Sarahはなぜ、会議の中止を提案したのだろうか。彼女は何に気付いたというのだろうか。

This PDF is prepared for tools@maruhouse.org , personal use only.

5 分間ミステリーの答え

サラは、orbiterAirlockExitSequence() メソッド自身を synchronized 宣言しなければ、割り込みを防止できないことに気付いたのである。このままだと、船外へ出るためのプログラムを動かしている時に、同時に船内へ戻るためのプログラムが動いてしまう恐れがある。確かに、個々のメソッドが実行されている間には割り込みができないのだが、1 つのメソッドの実行が終わり、次のメソッドの実行が開始されるまでの間には割り込みが可能である。ここでは、orbiterAirlockExitSequence() メソッドの処理全体を、ひとかたまりの、分割のできない（途中で割り込みのできない）ものにしなくてはならない。そのためには orbiterAirlockExitSequence() メソッドの synchronized 宣言が必要なのである。

This PDF is prepared for tools@maruhouse.org, personal use only.

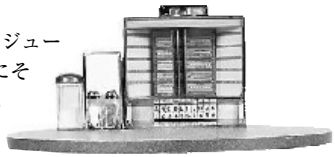
16章　コレクションとジェネリック型

データ構造

Javaはソートが簡単にできる言語です。そのためのツールがあらかじめ用意されているので、プログラマが自らソートのためのコードを書く必要はありません（プログラミングを学び始めたばかりの人は、よくソートのコードを書かされますが、Javaの場合はその種のコードを書かず、Java APIのメソッドを呼び出すだけで済むのです）。Javaのコレクションフレームワークに用意されたクラスを使用すれば、複数のデータをまとめ、操作するためのコードはほぼどのようなものでも書けると言っていいでしょう。データのリストを作ることも、リストに新たな要素を追加することも、リスト中から名前を手掛かりに必要なデータを見付け出すことも簡単にできます。データの重複を自動的に排除する機能を持たせることなども可能です。社員名簿を業績順にソートする、覚えている芸の数が多い順にペットの名前をソートする、といったことも容易です。

This PDF is prepared for tools@maruhouse.org, personal use only.

ジュークボックスで人気のある曲を調べる



あるお店のジュークボックスを管理するプログラムの作成を依頼されたとします。ジュークボックス自体がJavaプログラムを動かせるわけではないのですが、曲がかかる度にその曲に関するデータが簡単なテキストファイルに書き込まれる、という仕組みはできているものと仮定します。

これから作るプログラムには、曲の人気を調べる機能、レポート作成の機能、プレイリストの操作機能などを持たせなくてはいけません。プログラムのコードの多くは他のデベロッパが書いてくれるので、自分で書く部分は多くないのですが、データの管理、ソートなどのためのコードは書かなくてはなりません。また、店主がなぜかデータベースを嫌っているので、データベースを一切使わずにデータを扱う必要があります。その前提で、ジュークボックスから提供されるデータをうまく整理、操作しなくてはならないのです。

ファイルのデータの読み取りや操作について、またArrayListへのデータの保持については本書ですでに触れたのでわかるはずです。

SongList.txt

```
Pink Moon/Nick Drake
Somersault/Zero 7
Shiva Moon/Prem Joshua
Circles/BT
Deep Channel/Afro Celts
Passenger/Headmix
Listen/Tahiti 80
```

ジュークボックスがデータを書き込むことで作られるファイル。このファイルのデータを読み込み、操作するプログラムを書かなくてはならない

曲名をアルファベット順にソートする

曲名のリストを収めた、図のような内容のファイルがあるとします。曲のタイトルとアーチスト名が各行に1つずつ書き込まれています。タイトルとアーチスト名の間は順スラッシュで区切られています。このファイルの各行のデータをArrayListに保持することは特に難しくはないでしょう。

この場合、発注元が必要とする情報は曲名だけなので、曲名だけのリストにしてしまうこともできます。

ただし、図のリストで問題なのは、曲名がアルファベット順に並んでいないことです。アルファベット順にするにはどうすればいいでしょうか。

すでに見てきたとおり、ArrayListには、追加された順序で要素が保持されることになります。つまり、ArrayListに保持しただけでアルファベット順に並べ替えられるわけではないということです。ArrayListクラスにsort()メソッドでもあれば、ソートできそうですが、どうでしょうか。

This PDF is prepared for tools@maruhouse.org, personal use only.

ArrayListに曲名を保持するコード（ソート機能なし）

```
import java.util.*;
import java.io.*;

public class Jukebox1 {

  ArrayList<String> songList = new ArrayList<String>();

  public static void main(String[] args) {
    new Jukebox1().go();
  }

  public void go() {
    getSongs();
    System.out.println(songList);
  }

  void getSongs() {
    try {
      File file = new File("SongList.txt");
      BufferedReader reader = new BufferedReader(new FileReader(file));
      String line = null;
      while ((line= reader.readLine()) != null) {
      addSong(line);
      }

    } catch(Exception ex) {
      ex.printStackTrace();
    }
  }

  void addSong(String lineToParse) {
    String[] tokens = lineToParse.split("/");
    songList.add(tokens[0]);
  }
}
```

曲のタイトルをStringのArrayListに保持する

ファイルのロードをスタートし、ArrayList（songList）の内容を出力するメソッド

ファイルを読み取り、各行ごとにaddSong()メソッドを呼び出す。特別難しいことはしていない……

addSong()メソッドと同様のコードは、14章の「暗記カードプログラム」でも使った。split()メソッドを使って、1行を2つの部分（曲のタイトルとアーチスト名）に分割する

必要なのは曲のタイトルだけなので、1つ目の部分だけをsongList（ArrayList）に追加

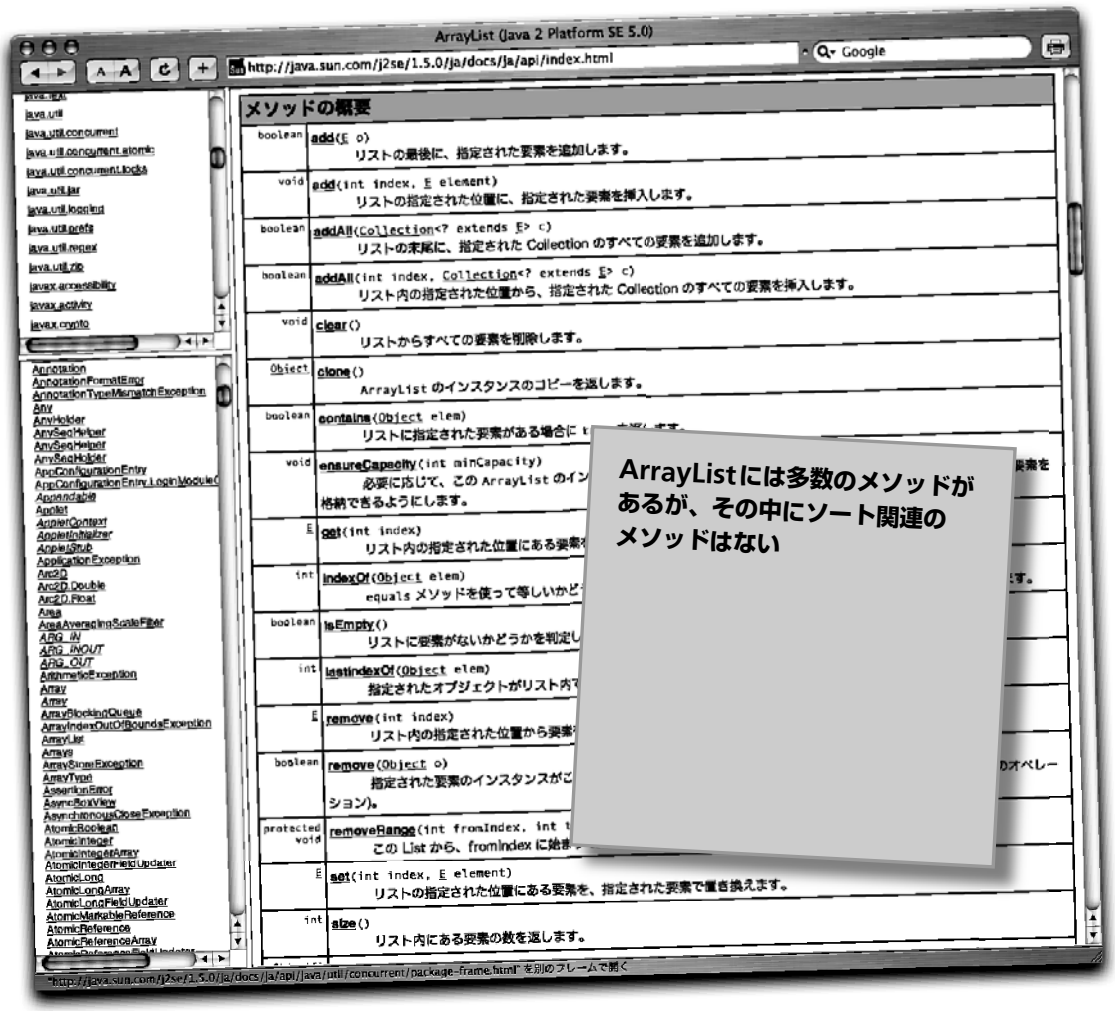
```
File  Edit  Window  Help  Dance
%java Jukebox1
[Pink Moon, Somersault,
Shiva Moon, Circle,
Deep Channel, Passenger,
Listen]
```

出力結果を見ると、ArrayList（songList）には、追加された順序（元のファイルに保存されていた順序）で曲のタイトルが保持されているとわかる。アルファベット順にはなっていない

This PDF is prepared for tools@maruhouse.org, personal use only.

ArrayListクラスにはsort()メソッドはない

ドキュメントを見てみると、ArrayListには、ソートに関連するメソッドがないことがわかります。ArrayListのスーパークラスなどを調べても無駄です。ともかく、ArrayListを使ってソートのメソッドを呼び出すことはできないということは明らかです。



This PDF is prepared for tools@maruhouse.org, personal use only.

ArrayListだけがコレクション（複数のデータをひとまとめにするクラス）ではない

ArrayListは最も使用頻度の高いコレクションですが、状況が違えば他のコレクションを使う場合もあります。中でも重要なものについて以下にまとめます。

こうしたコレクションについては後で詳述するので、今は説明がよく理解できなくても心配はいりません。

▶ TreeSet

要素を常にソートされた状態に保ち、重複を自動的に排除する。

▶ HashMap

キーと値のペアを要素として保持する。

▶ LinkedList

ArrayListに比べ、リスト中の任意の位置に要素を挿入する、あるいは任意の位置から要素を削除する、といった処理のパフォーマンスが若干高いとされる（同じ用途にArrayListを使ってもまったく問題はない）。

▶ HashSet

要素の重複を自動的に排除する。リスト中の特定の要素へのアクセスが高速に行える。

▶ LinkedHashMap

HashMapとほぼ同じだが、個々の要素（キーと値のペア）の操作を、リストに追加された順に行うことも、最後にアクセスされてから時間の立っていないものから順に行うこともできる。

This PDF is prepared for tools@maruhouse.org, personal use only.

TreeSetを使用してもよいが……
Collections.sort()メソッドを使用することもできる

曲のタイトルをArrayListではなくTreeSetに入れるようにすれば、自動的にアルファベット順にソートされます。リストを出力してみれば、要素が常にアルファベット順に並んでいることがわかるでしょう。

TreeSetはいわゆる「Set系」のコレクションクラスです（Setについては後述）。要素を常にアルファベット順にソートされた状態にしておきたいという場合には便利です。

しかし、アルファベット順にソートされた状態を常に保つということが必要ない場合には非効率なだけになってしまいます。新たな要素が追加される度に、同時にソートの処理も行われるため、その分だけ時間がかかるのです。その点、ArrayListなら、新たな要素は単純に末尾に追加するだけなので時間が短くて済むのは明らかです。

java.util.Collections

```
public static void copy(List destination, List source)
public static List emptyList()
public static void fill(List listToFill, Object objToFillItWith)
public static int frequency(Collection c, Object o)
public static void reverse(List list)
public static void rotate(List list, int distance)
public static void shuffle(List list)
public static void sort(List list)
public static boolean replaceAll(List list, Object oldVal, Object newVal)
// その他多数……
```

Collectionsクラスにはsort()メソッドがある。引数は"List"になっている。ArrayListはListインターフェースをインプリメントしているので、引数として指定できる（ポリモーフィズムのおかげ）。

Q：ArrayListのadd()メソッドにはオーバーロードバージョンもあり、新たな要素をリストの末尾でなく、指定の位置（int型の引数で指定する）に追加することもできます。このメソッドは、末尾に要素を追加するものより速度が遅いのですか。

A：その通りです。ArrayListの場合、リストの末尾以外に要素を追加しようとすれば、速度は低下します。オーバーロードバージョンであるadd(index, element)メソッドは、add(element)よりも遅いわけです。ただ、ArrayListを使用していて、特定の位置への要素の追加が必要になることは多くありません。

Q：LinkedListというクラスもありますが、要素を指定の位置に追加するのであれば、ArrayListよりこちらの方がよいのではないでしょうか。少なくとも、データがどのような順序で並んでいるかを把握していれば、こちらの方がいいと思うのですが。

A：良い質問です。LinkedListは、ArrayListに比べ、リスト中の任意の位置に要素を挿入する、任意の位置から要素を削除する、といった処理が高速に行えるとされています。とはいえ、両者の速度の違いは、ほとんどの場合、問題にならないほどのものです。要素の数が相当多くなれば違ってくるかもしれませんが。LinkedListについては後で触れます。

注意：上記のCollectionsクラスについての情報は、もちろん十分なものではない。特に「ジェネリック型」に関する説明がないのは問題（ジェネリック型については後述）。

This PDF is prepared for tools@maruhouse.org, personal use only.

Collections.sort()を使用してコードを改良する

```
import java.util.*;
import java.io.*;

public class Jukebox1 {

  ArrayList<String> songList = new ArrayList<String>();

  public static void main(String[] args) {
    new Jukebox1().go();
  }

  public void go() {
    getSongs();
    System.out.println(songList);
    Collections.sort(songList);
    System.out.println(songList);
  }

  void getSongs() {
    try {
      File file = new File("SongList.txt");
      BufferedReader reader = new BufferedReader(new FileReader(file));
      String line = null;
      while ((line= reader.readLine()) != null) {
      addSong(line);
    }

    } catch(Exception ex) {
      ex.printStackTrace();
    }
  }

  void addSong(String lineToParse) {
    String[] tokens = lineToParse.split("/");
    songList.add(tokens[0]);
  }
}
```

Collections.sort()メソッドにより、Stringのリストをアルファベット順にソートできる

Collectionsクラスのsort()メソッドを呼び出し、再度リストを出力すると、アルファベット順にソートされていることがわかる

```
File  Edit  Window  Help  Chill
%java Jukebox1
[Pink Moon, Somersault, Shiva Moon, Circles, Deep
Channel, Passenger, Listen]
[Circles, Deep Channel, Listen, Passenger, Pink
Moon, Shiva Moon, Somersault]
```

sort()呼び出し前
sort()呼び出し後

This PDF is prepared for tools@maruhouse.org, personal use only.

Stringではなく Songオブジェクトをソートする

次はStringではなく、独自に作ったSongクラスのインスタンスのリストを扱うプログラムを作ってみましょう。Songクラスでは、曲のタイトル、アーチスト名以外のデータも扱うことになります。それに対応して、ジュークボックスも、2種類ではなく4種類のデータをファイルに書き込むものとします。

Songクラスのコードは非常に簡単なものです。目立った特徴と言えば、toString()メソッドがオーバーライドされていることくらいです。先述のとおり、toString()は、Objectクラスのメソッドであり、Javaのあらゆるクラスはこのメソッドを継承します。toString()は、"System.out.println(anObject)"というようなコードを書いた時に呼び出されるメソッドです。デフォルトでは、このコードにより、anObjectオブジェクトの識別子を文字列にしたものが出力されることになります。ここでtoString()メソッドをオーバーライドするのは、それ以外の情報(人間が見て価値のある情報)が出力されるようにするためです。

SongListMore.txt

```
Pink Moon/Nick Drake/5/80
Somersault/Zero 7/4/84
Shiva Moon/Prem Joshua/6/120
Circles/BT/5/110
Deep Channel/Afro Celts/4/120
Passenger/Headmix/4/100
Listen/Tahiti 80/5/90
```

ファイルには、曲に関して2種類ではなく4種類の情報が書き込まれる。この4種類の情報をすべてリストに入れるため、4つのインスタンス変数を持ったSongというクラスを作る

```
class Song {
  String title;
  String artist;
  String rating;
  String bpm;

  Song(String t, String a, String r, String b) {
    title = t;
    artist = a;
    rating = r;
    bpm = b;
  }

  public String getTitle() {
    return title;
  }

  public String getArtist() {
    return artist;
  }

  public String getRating() {
    return rating;
  }

  public String getBpm() {
    return bpm;
  }

  public String toString() {
    return title;
  }
}
```

4種類のデータを保持するための4つのインスタンス変数

Songのインスタンスが作成される際に4つの変数の値はすべて設定される

4つのデータのゲッターメソッド

toString()をオーバーライドして、Songオブジェクトを引数にしてSystem.out.println()が実行された時に曲のタイトルが出力されるようにする。Songオブジェクトのリストを引数に指定してSystem.out.println()が実行された場合には、リストの各要素のtoString()メソッドが呼び出される

This PDF is prepared for tools@maruhouse.org, personal use only.

Songクラスを利用してコードを改造する

改造と言っても変わる部分は多くありません。ファイルI/Oのコードは同じで、ファイルのデータを操作するコード（String.split()）もほぼ同じです。ただ、違うのは、各行（個々の曲）のデータが2つではなく、4つの部分に分かれているという点です。4つのデータはSongオブジェクトの作成に使用されます。また、もちろん、ArrayListは<String>ではなく、<Song>のリストになります。

String型のArrayListを
Song型のArrayListに変える

```
import java.util.*;
import java.io.*;

public class Jukebox3 {

  ArrayList<Song> songList = new ArrayList<Song>();
  public static void main(String[] args) {
    new Jukebox3().go();
  }

  public void go() {
    getSongs();
    System.out.println(songList);
    Collections.sort(songList);
    System.out.println(songList);
  }
  void getSongs() {
    try {
      File file = new File("SongListMore.txt");
      BufferedReader reader = new BufferedReader(new FileReader(file));
      String line = null;
      while ((line= reader.readLine()) != null) {
      addSong(line);
      }
    } catch(Exception ex) {
      ex.printStackTrace();
    }
  }

  void addSong(String lineToParse) {
    String[] tokens = lineToParse.split("/");
    Song nextSong = new Song(tokens[0], tokens[1], tokens[2], tokens[3]);
    songList.add(nextSong);
  }
}
```

4種類のデータ（ファイルの各行の曲に関するデータ）を
使ってSongオブジェクトを作りリストに加える

This PDF is prepared for tools@maruhouse.org, personal use only.

コンパイルできない！

実は、前述のコードはコンパイルできません。どこかに問題があるということです。一体どこでしょうか。Collectionsクラスにsort()メソッドがあるのは確かです。引数としてListを指定できるのも間違いありません。

ArrayListはListインターフェースをインプリメントしているので、引数として指定できるはずです。そう考えると何も問題はないようですが、実はそうではないのです。

前述のコードをコンパイルすると「ArrayList<Song>を引数として受け取れるsort()メソッドが見付からない」というエラーが出ます。つまり、SongオブジェクトのArrayListに問題の原因があるということになります。ArrayList<String>なら問題はないわけですから、両者の違いがどこにあるかを考えなくてはなりません。両者の間のどのような違いがコンパイルエラーの原因となっているのかを知る必要があります。

```
File  Edit  Window  Help  Bummer
%javac Jukebox3.java
Jukebox3.java:15: cannot find symbol
symbol  : method sort(java.util.ArrayList<Song>)
location: class java.util.Collections
                Collections.sort(songlist)
                                        ^
1 error
```

気付いている人も多いでしょうが、ここでまず問題になるのは「何を基準にソートすればいいのか」ということです。ただSongオブジェクトを渡されても、sort()の方では、何を基準にソートすればいいのかわかりません。基準が曲のタイトルなのか、テンポなのかそれ以外なのか、いずれにしても、それを何らかのかたちでメソッドに伝えなくてはならないのです。

その点についてはしばらく後に説明します。その前に、なぜSongオブジェクトのArrayListをsort()メソッドの引数として指定するとコンパイルエラーになるか、その理由について説明することにしましょう。

This PDF is prepared for tools@maruhouse.org, personal use only.

sort()メソッドの宣言

Collections (Java 2 Platform SE 5.0)

file:///Users/kathy/Public/docs/api/index.html

Google

rm SE 5.0) Caffeinated ...d Brain Day Brand Noise Diva Marketing craftster.org...clever ideas Brain Waves

Method Detail

sort

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

Sorts the specified list into ascending order, according to the *natural ordering* of its elements. All elements in the list must implement the `Comparable` interface. Furthermore, all elements in the list must be *mutually comparable* (that is, `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the list).

APIのドキュメントを見ると（java.util.Collectionsクラスのsort()メソッドを見る）、sort()メソッドの宣言が奇妙なものになっていることがわかります。少なくとも本書のこれまでの説明だけでは理解できない宣言のはずです。

sort()メソッド（に限らず、Javaのコレクションフレームワークでは全般的に）には「ジェネリック型」が多用されます。Javaのソースコードやドキュメントに<>という記号が出てきたら、それは「ジェネリック型」を意味していると考えて間違いありません。ジェネリック型はJava 5.0から新たに導入された概念です。sort()の引数として、StringのArrayListは使えてもSongのArrayListは使えない理由を知るには、まずこのジェネリック型について学び、ドキュメントが理解できるようになる必要があります。

This PDF is prepared for tools@maruhouse.org, personal use only.

コレクションを「タイプセーフ」にするジェネリック型

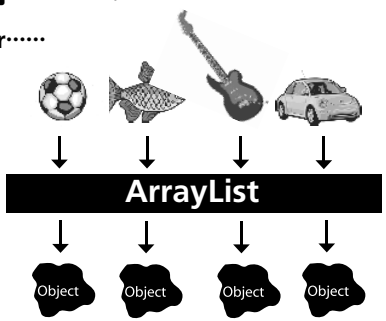
「ジェネリック型を使うコードは、ほぼすべてがコレクション関連のコードである」と言い切っても問題はないでしょう。他の用途に使うこともできるのですが、主に、コレクションを「タイプセーフ」なもの（型についての問題が起きにくいもの）にするのがジェネリック型の目的であると言えます。ジェネリック型導入により、Duck型のリストにDogオブジェクトが入ってしまうといったことが防げるのです。

ジェネリック型導入以前（Java 5.0が出る前）、コレクションに入るものの型について、コンパイラはまったく関知しませんでした。コレクションはすべて、Object型のオブジェクトに対応すると宣言されていたからです。ArrayListにもあらゆる型のオブジェクトを入れることができました。以前のArrayListは、Java 5.0で言えばArrayList<Object>と宣言されていたも同然だったのです。

ジェネリック型導入以前

ジェネリック型導入以前は、ArrayListの型を宣言することはできなかった。add()メソッドの引数の型はObjectだった

SoccerBall、Fish、Guitar、Car……
あらゆる型のオブジェクトへの参照をArrayListに入れることができた



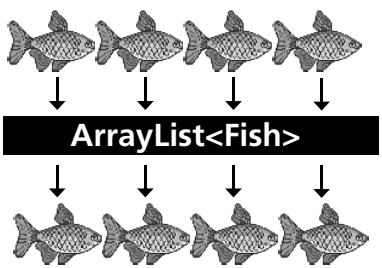
ArrayListから取り出される時はすべてがObject型のオブジェクトへの参照に

ジェネリック型導入により「タイプセーフ」なコレクションが作成できるようになった。型に関する問題があれば、実行時ではなく、コンパイル時に発見が可能

ジェネリック型導入以前は、Carオブジェクトに対応させるつもりで作ったArrayListにPumpkinオブジェクトを入れることも自由にできた

ジェネリック型導入後

宣言された型（ここではFish）のオブジェクトへの参照のみに対応



Fish型のオブジェクトへの参照として取り出せる

ジェネリック型導入後は、ArrayList<Fish>にはFishオブジェクトのみ入れることができる。取り出す時もFishオブジェクトのまま。知らない間にCarオブジェクトが入っていたということが起きる心配はなく、取り出した時にFishへのキャストの必要もない

This PDF is prepared for tools@maruhouse.org, personal use only.

ジェネリック型

ジェネリック型に関して学ぶべきことは数多くありますが、通常は次の3つを覚えておけば困ることは少ないでしょう。

① **ジェネリッククラス（ArrayListなど）のインスタンス作成**
ArrayListなどのクラスのインスタンスを作成する際は、従来の配列と同様、保持するオブジェクトの型を指定する必要があります。

```
new ArrayList<Song>()
```

② **ジェネリック型の変数の宣言、値の代入**
ジェネリック型の変数に関して「ポリモーフィズム」はどうなるのでしょうか。たとえば、ArrayList<Animal>と宣言された変数に、ArrayList<Dog>を代入することはできるのでしょうか。また、List<Animal>と宣言された変数にArrayList<Animal>を代入することはできるでしょうか。その点については後述します。

```
List<Song> songList =
      new ArrayList<Song>()
```

③ **ジェネリック型の引数を持つメソッドの宣言（呼び出し）**
たとえば、AnimalオブジェクトのArrayListを引数として指定するよう宣言されているメソッドがあるとします。これは正確にはどういう意味でしょうか。DogオブジェクトのArrayListを引数として渡すことはできるのでしょうか。この種のメソッドには、ポリモーフィズムに関して、従来の配列を引数とするメソッドとは大きく違った難しい問題があるので、後で詳しく触れます。
（2と3は厳密には同じことですが、重要なポイントであることを強調するためにあえて重複させています）

```
void foo(List<Song> list)
x.foo(songList)
```

Q：ジェネリッククラスのインスタンスを作成するには、逐一、それがどのクラス（型）に対応するかを指定するわけですが、そうしたジェネリッククラスをプログラマが自分で作るようなことはあるのですか。

A：そういう必要はまずないでしょう。APIには、必要になりそうなデータ構造のほとんどをカバーできるコレクションクラスが用意されています。コレクションクラス以外に、ジェネリッククラスである必要性のあるクラスはほぼないと言って間違いはありません。インスタンス化や変数の宣言の際に要素の型を指定しなくてはならないようなクラスは、コレクションクラスくらいでしょう。
プログラマがどうしても自らジェネリッククラスを作らなければならないというケースは皆無ではないでしょうが、それはあくまで例外ですので、ここでは触れません（ただし、説明を読んでいて、自然にどうすればよいかわかる人はいると思います）。

This PDF is prepared for tools@maruhouse.org, personal use only.

ジェネリッククラスを使用する

ArrayListはジェネリッククラスの中でも最も使用頻度の高いものなので、まずはArrayListのドキュメントの読み方について説明します。ジェネリッククラスのドキュメントで重要になるのは以下の2つです。

1) クラスの宣言

3) 要素を追加するためのメソッドの宣言

"E"はコレクションで扱う「要素の型」という意味だと考える("Element" =「要素」の略)

ArrayListのドキュメントを読む("E"は正確には何を意味するのか)

"E"の部分には、実際にはArrayListを宣言、作成する際に指定する型が入る

ArrayListはAbstractListのサブクラスなので、ArrayListに型を指定すると、それはすべてAbstractListにも適用されることになる

```
public class ArrayList<E> extends AbstractList<E> implements List<E> ... {

    public boolean add(E o)

    // 以下略……
}
```

これが重要なポイント。ArrayListに追加できるオブジェクトの型が"E"の指定によって決まる

<E>の指定はListインターフェースにも同時に適用される

"E"には、ArrayListのインスタンス作成に使用する型を指定します。ArrayListのドキュメント中に"E"という記述を見付けたら、頭の中で「インスタンス作成に使用する型」という記述に置き換えて読んでいくといいでしょう。

"new ArrayList<Song>"というコードでは、"E"が"Song"になっています。メソッドや変数の宣言に出てくる"E"もすべて同様に置き換えればいいのです。

This PDF is prepared for tools@maruhouse.org, personal use only.

ArrayListの型パラメータを使用する

たとえば、以下のようなコードを書いた場合は、

```
ArrayList<String> thisList = new ArrayList<String>
```

以下の"E"の部分に"String"が入ります。

```
public class ArrayList<E> extends AbstractList<E> ... {

      public boolean add(E o)
      // 以下略……
}
```

つまり、コンパイラは次のように解釈することになります。

```
public class ArrayList<String> extends AbstractList<String>... {

      public boolean add(String o)
      // 以下略……
}
```

"E"は、ArrayList作成の時にパラメータに指定された型に置き換えられると言ってもいいでしょう。ArrayListのadd()に、指定の型と互換性のない引数を渡すことができないのはそのためです。ArrayList<String>という指定をすると、add()メソッドはadd(String o)となり、ArrayList<Dog>という指定をすると add()メソッドはadd(Dog o)となります。

Q：使えるのは"E"だけですか。sort()メソッドのドキュメントには"T"が使われているようですが……

A：Javaの識別子として正当なものであれば、どれでも使えます。メソッド名、変数名として使えるものなら、型パラメータとして使えるということです。ただ、慣例として、一文字にすることにはなっているので、それは守るようにすべきです。その他、"E"はコレクションクラスの要素の型にしか使わないが、"T"はそれ以外にも使える、というのも慣例になっています。

This PDF is prepared for tools@maruhouse.org, personal use only.

ジェネリックメソッドを使用する

「ジェネリッククラス」は、クラスの宣言に型パラメータ（<E>など）を含むクラスのことです。「ジェネリックメソッド」も、同様に、宣言に型パラメータを含むメソッドのことです。

メソッドに使う型パラメータには以下のような種類があります。

① クラス宣言の中の型パラメータ

```
public class ArrayList<E> extends AbstractList<E> ... {
      public boolean add(E o)
```

ここでは<E>ではなく、Eとだけ書くことができる。クラスの一部としてすでに宣言されているため

クラスの一部として型パラメータを宣言した場合は、通常のクラス名、インターフェース名と同様のかたちで、型パラメータ名を使用できます。メソッドの引数に使用された型パラメータ（上記の"E"）は、クラスがインスタンス化された時に、指定の型に置き換えられます。

② クラス宣言にない型パラメータ

```
public <T extends Animal> void takeThing(ArrayList<T> list)
```

ここでは単に<T>と書いている。すでに宣言されているため

クラス宣言にない型パラメータを、メソッドのために宣言することもできます。その場合は、戻り値の型の前、という変則的な位置に型パラメータの宣言を置きます。上記の場合、Tは、Animalを継承するあらゆるクラスに置き換えることが可能、ということになります。

This PDF is prepared for tools@maruhouse.org, personal use only.

重要ポイント：

このコードは、

```
public <T extends Animal> void takeThing(ArrayList<T> list)
```

このコードと同じではありません。

```
public void takeThing(ArrayList<Animal> list)
```

どちらも正しいのですが、意味は異なります。

上のコードでは、<T extends Animal>がメソッド宣言の一部になっています。これは、AnimalのArrayList、あるいはAnimalのあらゆるサブクラスのArrayListはすべて引数にできる、という意味になります。従って、Dog、CatがAnimalのサブクラスだとすれば、ArrayList<Dog>、ArrayList<Cat>、ArrayList<Animal>はどれも引数にできることになります。

しかし、"ArrayList<Animal>"と書いている下のコードの場合、ArrayList<Animal>以外の引数は指定できません。上のコードなら、ArrayList<Dog>、ArrayList<Cat>にも対応できるのですが、下のコードではArrayList<Animal>にのみ対応するのです。

Javaの「ポリモーフィズム」という特徴から見て、それはおかしいのでは、と感じる人もいるでしょう。その点に関しては、この章の終わりの方で明らかにします。今のところは、まず、sort()メソッドを使ってsongListをソートする方法、sort()のドキュメントの記述（ジェネリック型の宣言）を理解することを優先してください。

現時点では、上のコードが正しく、下のコードは誤っているという理解でもかまわないでしょう。上のコードなら、Animal、あるいはAnimalのサブクラスを指定したArrayListを引数にできます。

では、次にsort()メソッドについて見てみましょう……

This PDF is prepared for tools@maruhouse.org, personal use only.

問題はこのエラー……

```
File  Edit  Window  Help  Bummer
%javac Jukebox3.java
jukebox3.java:15: cannot find symbol
symbol  :method sort(java.util.ArrayList<Song>)
location: class java.util.Collections
                    Collections.sort(songList)
                                             ^
1 error
```

```
import java.util.*;
import java.io.*;

public class Jukebox3 {
  ArrayList<Song> songList = new ArrayList<Song>();
  public static void main(String[] args) {
    new Jukebox3().go();
  }
  public void go() {
    getSongs();
    System.out.println(songList);
    Collections.sort(songList);
    System.out.println(songList);
  }
  void getSongs() {
    try {
      File file = new File("SongListMore.txt");
      BufferedReader reader = new BufferedReader(new FileReader(file));
      String line = null;
      while ((line= reader.readLine()) != null) {
        addSong(line);
      }
    } catch(Exception ex) {
      ex.printStackTrace();
    }
  }
  void addSong(String lineToParse) {
    String[] tokens = lineToParse.split("/");
    Song nextSong = new Song(tokens[0], tokens[1], tokens[2], tokens[3]);
    songList.add(nextSong);
  }
}
```

ここが問題。引数がArrayList<String>なら良いがArrayList<Song>だとエラーになる

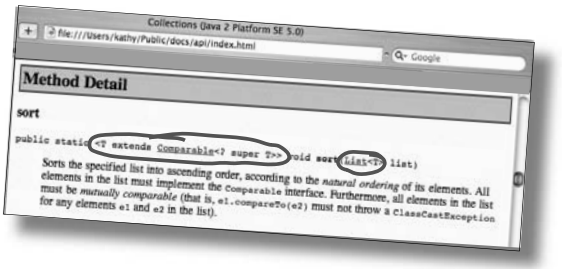
This PDF is prepared for tools@maruhouse.org, personal use only.

sort()メソッド

前頁までの説明を基にドキュメントを見ると、なぜ、sortメソッドの引数としてStringのArrayListは指定できるのに、SongのArrayListは指定できないのかがわかります。理由を簡単にまとめると以下のようになります。

sort()メソッドの引数として指定できるのは、ComparableオブジェクトのリストListのみ。

SongはComparableのサブクラスではないので、Songのリストは引数にできない(少なくとも現時点ではそう理解してよい)。




```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

これは「TはComparableを継承するものでなくてはならない」という意味

(この部分は今は無視してよい。どうしても気になるのであれば、「Comparableの型パラメータは、Tか、そのスーパークラスでなくてはならない」という意味、と理解しておけばよい)

引数としては、Comparableを継承する型を型パラメータに指定したList(あるいはそのサブクラス。ArrayListなど)を渡す

一応、ドキュメント見てみたら、String ってComparableを「継承(extend)」してるんじゃなくて、「インプリメント」してるって書いてあるけど。Comparable ってインターフェースなんでしょ。<T extends Comparable> ってコード、おかしいと思うんだけど。

```
public final class String extends Object implements Serializable,
                                                    Comparable<String>, CharSequence
```


This PDF is prepared for tools@maruhouse.org, personal use only.

この場合の"extends"は「"extends"あるいは"implements"」という意味

型パラメータは、対応する型の限定をするのですが、その場合、あるクラスの「サブクラス」に限定したい、ということもあれば、あるインターフェースの「インプリメンテーション」に限定したい、ということもあります。また、時には、あるクラスのサブクラスか、あるインターフェースのインプリメンテーションのどちらかに限定したい、ということもあります。つまり、それを表現するためのコードも必要、ということです。継承とインプリメントの両方の意味を持つキーワードがあれば便利です。

そこで選ばれたのが"extends"です。この"extends"は「継承」というより、"IS-A"という意味だと考えればいいでしょう。右側にはクラス名も、インターフェース名も指定できます。

ここでのキーワード"extends"は、"IS-A"を意味し、クラスにもインターフェースにも使える

Comparableはインターフェースなので、これは正確には「TはComparableインターフェースをインプリメントするクラスでなければならない」という意味

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

右側に指定するものはクラスでもインターフェースでもかまわない。どちらの場合も"extends"と書く

Q：キーワードを"is"にしなかったのはなぜですか。

A：言語に新たなキーワードを追加するのは大変なことです。それによって、過去のバージョンに対応して書かれたコードを無効にしてしまう危険があるからです。少し考えてみてください。過去に書かれたコードの中に"is"という変数が使われていたとしたらどうでしょうか（実際、本書でも、入力ストリームを表すものとして使っている）。キーワードになってしまった言葉を、各プログラマが勝手に独自の識別子として使うことはできません。ある言葉を急にキーワードにしてしまうと、それ以前にその言葉を使っていたコードは正しく機能しなくなります。そういう理由から、言語を開発する側は、できる限り、従来からあるキーワードを再利用しようとするわけです。"extends"が使えそうなら、新たなキーワードを導入するよりはそちらを選ぶことになります。時には使えそうな言葉がない場合もありますが……

数は多くない（非常に少ない）のですが、Javaに新たなキーワードが追加されたという例がないわけではありません。たとえば、Java 1.4で追加された"assert"、Java 5.0で追加された"enum"などがそうです（"enum"に関しては付録Bで詳述します）。当然、これによって過去のコードが無効になってしまう危険が生じました。そのため、新バージョンのJavaで古いJavaコードを扱う際には、コンパイル時や実行時にオプションを付けるなどの対応が必要となる場合があります。コンパイラやJVMを動かす時に、コマンドラインで特殊なフラグを指定し「これがJava 1.4であることはわかっているが、Java 1.3のように動いて欲しい」といった指示をするのです。assertがキーワードでなかった頃に変数に使ってしまっていた、というような場合にはそれで対処します。

（実際にどのようなフラグが使えるかは、コンパイラならjavac、JVMならjavaとコマンドラインに入力して確かめてください。後ろに何も付けずに入力すれば、使用できるオプションのリストが表示されます。フラグについては次章でも触れます）

This PDF is prepared for tools@maruhouse.org, personal use only.

これで問題が明らかに……

SongクラスはComparableをインプリメントするものでなくてはならない

ArrayList<Song>をsort()メソッドの引数にするためには、SongクラスがComparableをインプリメントするものでなくてはなりません。sort()メソッドの宣言がそのようになっているからです。ドキュメントを見るとComparableインターフェースは非常に簡単なものであることがわかります。インプリメントすべきメソッドは1つだけです。

問題は、何を基準に、あるSongを他のSongより「大きい」「小さい」「同じ」とみなすかということ

それを決めなければComparableインターフェースのインプリメントはできない

java.lang.Comparable

```
public interface Comparable<T> {
  int compareTo(T o);
}
```

compareTo()メソッドに関してドキュメントには、次のようなことが書かれています。

> **戻り値：**
> 指定のオブジェクトがこのオブジェクトより小さければ負の整数、同じであれば0、大きければ正の整数を戻す。

Comparableインターフェースをインプリメントして Songクラスを作るのであれば、compareTo()メソッドは、Songオブジェクトを使い、もう1つ別のSongオブジェクトを引数として呼び出されることになります。その場合、Songクラスには、メソッド呼び出しに使われたSongオブジェクトと、引数のSongオブジェクトのうち、どちらがリストの中で先になり、後になるか（あるいは同等になるか）、を判定できる機能が必要になります。

大きな問題となるのが、どちらのSongオブジェクトが「大きい」かを何を基準に決めるのか、ということです。基準を決め、それを反映したかたちでcompareTo()メソッドをインプリメントする必要があります。ドキュメントによれば、引数として渡されたSongオブジェクトの方が、呼び出しに使われたSongオブジェクトよりも「大き」ければ、負の数（具体的にどの数とは書かれていない）が戻されることになります。逆に、呼び出しに使われたSongオブジェクトの方が「大き」ければ、正の数が戻されます。両者が「同等」（少なくともソートという目的から見て同等、ということ。必ずしも両者が同じオブジェクトでなくてもよい）であれば、0が戻されます。たとえば、タイトルを基準に「大きさ」を決める場合は、タイトルが同じであれば、両者は同等ということになります。
（このあたりのことに関しては、複雑な問題があるので、後でまた触れます……）

自分で考えてみよう

compareTo()メソッドをインプリメントする仮コードを書いてみましょう（もちろん、リアルコードが書ければその方が良いですが）。Songオブジェクトを曲のタイトルを基準にソートできるようなものにしてください。

ヒント：これまでの説明を正しく理解できていれば、おそらくコードは3行以内で書けるはずです。

This PDF is prepared for tools@maruhouse.org, personal use only.

comparableインターフェースをインプリメントしたSongクラス

ソートの基準を「曲のタイトル」にするのであれば、引数として渡されたSongオブジェクトと、メソッドの呼び出しに使われたSongオブジェクトの曲のタイトルを比較できるよう、compareTo()メソッドをインプリメントする必要があります。これは、Songクラスに、曲のタイトルの「大小」を判定する機能を持たせなくてはならないということも意味します。

Stringクラスには「アルファベット順」を判定する機能があるのは間違いありません。sort()メソッドが、Stringのリストに対応できるのは、そのためです。Stringには、やはりcompareTo()メソッドがあります。つまり、単純にこのメソッドを呼び出してやれば、アルファベット順を判定することは可能だということです。そうすれば、曲のタイトル（これはString）の比較は簡単にできます。わざわざ自分で新たに比較、判定のためのコードを書く必要はないのです。

通常、この型は一致させる。Comparableをインプリメントする型と、比較対象となる型を指定する。ここでは、ソートのため、Sortオブジェクトどうしを比較することになる

sort()メソッドは、Songオブジェクトをcompareto()メソッドに渡す。それによって、そのSongオブジェクトと、メソッドの呼び出しに使われたSongオブジェクトとが比較される

曲のタイトル（Stringオブジェクト）を基準にソートするのであれば、コードは簡単に書ける。Stringには、はじめからcompareTo()メソッドがある。

```
class Song implements Comparable<Song> {
  String title;
  String artist;
  String rating;
  String bpm;

  public int compareTo(Song s) {
    return title.compareTo(s.getTitle());
  }

  Song(String t, String a, String r, String b) {
    title = t;
    artist = a;
    rating = r;
    bpm = b;
  }

  public String getTitle() {
    return title;
  }

  public String getArtist() {
    return artist;
  }

  public String getRating() {
    return rating;
  }

  public String getBpm() {
    return bpm;
  }

  public String toString() {
    return title;
  }
}
```

これでコードは正しく機能する。sort()メソッド呼び出し前、呼び出し後のリストの出力を見ると、曲のタイトルのアルファベット順にソートが行われていることがわかる

```
File  Edit  Window  Help  Ambient
%java Jukebox3
[Pink Moon, Somersault, Shiva Moon, Circles, Deep
Channel, Passenger, Listen]
[Circles, Deep Channel, Listen, Passenger, Pink Moon,
Shiva Moon, Somersault]
```


This PDF is prepared for tools@maruhouse.org, personal use only.

リストのソートは確かにできたが……

これでソートはできたのですが、まだ問題はあります。曲のタイトルだけでソートできれば十分、というのならいいのですが、アーチスト名でもソートできた方が良いに違いありません。

しかし、コレクションの要素となるクラス（Comparableインターフェースをインプリメントするクラス）には、comparaTo()を1つしか作ることができません。どうすればいいのでしょうか。

Songクラスの中でフラグを使うという方法もありますが、良いとは言えません。compareTo()メソッドの中でif文を使い、フラグの値を調べ、それが曲のタイトルに対応するものか、アーチスト名に対応するかによって、動作を変えるようにするわけですが、得策とは言えないでしょう。

そんなぎこちなくて危ないコードを書くよりも、良い方法があります。実はAPIにはまさにそのためのコードがあらかじめ組み込まれているのです。それにより、同じものを複数とおりの方法でソートすることができます。

Collectionsクラスのドキュメントをもう一度見てみましょう。Comparatorという型の引数を持つ、もう一つのsort()メソッドがあるのがわかるはずです。

Collections (Java 2 Platform SE 5.0)

file:///Users/kathy/Public/docs/api/index.html

Jellyvision, Inc.　Collections ...form SE 5.0)　Caffeinated ...d Brain Day　Brand Noise　Diva Marketing

static <K,V> Map<K,V>	singletonMap(K key, V value) Returns an immutable map, mapping only the specified key to the specified value.
static ...ble<? super T>> void	sort(List<T> list) Sorts the specified list into ascending order, according to the *natural ordering* of its elements.
static <T> void	sort(List<T> list, Comparator<? super T> c) Sorts the specified list according to the order induced by the specified comparator.

sort()メソッドはオーバーロードされており、Comparatorを引数にできるものもある

注：Comparatorを使えば、Songオブジェクトを、曲のタイトルだけでなく、アーチスト名によってもソートできる。Comparatorの使い方については次に述べる

This PDF is prepared for tools@maruhouse.org, personal use only.

Comparatorを使用する

先述のcompareTo()を使えば、リスト中の要素の比較はできますが、比較の基準を複数設定するということはできません。しかし、これから解説するComparatorは、リストの要素となるクラスに含まれるものではなく、独立したクラスであり、必要に応じていくつでも作ることができます。もし、アーチスト名を基準にSongオブジェクトの比較をしたければ、そのために"ArtistComparator"といったクラスを作ればいいのです。テンポを基準に比較をするなら、たとえば"BPMComparator"というようなクラスを作ることになります。

そうしたクラスができれば、後は、リストと、適切なComparatorを引数に指定して、オーバーロードバージョンのsort()メソッドを呼び出すだけです。sort()メソッドは、Comparatorを利用してソートを行います。

Comparatorを引数にするsort()メソッドでは、compareTo()メソッド(リストの要素となっているオブジェクトのメソッド)の代わりにComparatorを使用してソートを行います。Comparatorが引数として指定されると、sort()は、compareTo()メソッドを呼び出さないのです。その代わりにComparatorのcompare()メソッドを呼び出します。

ここでの要点をまとめると次のようになります。

- 引数が1つのsort(List o)メソッドを呼び出すと、リストの要素のcompareTo()メソッドを利用してソートが行われる。従って、リストの要素はComparableインターフェースをインプリメントしていなくてはならない。
- sort(List o, Comparator c)メソッドを呼び出すと、リストの要素のcompareTo()メソッドは呼び出されず、代わりにComparatorのcompare()メソッドが使用される。これは、リストの要素がComparableインターフェースをインプリメントしてなくてもよいということ。

java.util.Comparator

```
public interface Comparator<T> {
  int compare(T o1, T o2);
}
```

Comparatorを引数にするsort()メソッドでは、リストの要素のcompareTo()メソッドの代わりにComparatorを使用してソートを行う

Q：たとえComparableをインプリメントしていないクラスであっても、またそのソースコードが手に入らなくても、Comparatorを作成すれば、ソートが可能だということですか。

A：そのとおりです。また、Comparableをインプリメントして、要素のサブクラスを作るという方法も考えられます(これはできない場合もあり得る)。

Q：リストの要素となるクラスは、すべてComparableをインプリメントして作っておけばいいということではないのですか。

A：「ソート」という処理があらゆる要素に必要なものだと本当に思いますか。その性質上、ソートをしてもまったく意味がない、という要素があったとして、その要素がComparableをインプリメントしていたら他のプログラマに誤解される恐れがあります。それに、Comparableをインプリメントしてなかったとしても、新たにComparatorを作れば好きな方法でソートができるのですから、さほど大きな問題にはなりません。

This PDF is prepared for tools@maruhouse.org, personal use only.

Comparatorを使用してコードを改造する

改造したのは以下の3点です。

1) Comparator（従って、前述のcompareTo()と同様の機能を持つcompare()メソッドも）をインプリメントする内部クラスを作成。
2) Comparatorをインプリメントする内部クラスのインスタンスを作成。
3) オーバーロードバージョンのsort()メソッドを呼び出す。Songオブジェクトのリストと、Comparatorをインプリメントする内部クラスのインスタンスを引数とする。

注意：SongクラスのtoString()メソッドも、曲のタイトルとアーチスト名の両方を（リストがどのようにソートされたかとは無関係に）出力するよう改造します。

```
import java.util.*;
import java.io.*;

public class Jukebox5 {
  ArrayList<Song> songList = new ArrayList<Song>();
  public static void main(String[] args) {
    new Jukebox5().go();
  }

  class ArtistCompare implements Comparator<Song> {
    public int compare(Song one, Song two) {
      return one.getArtist().compareTo(two.getArtist());
    }
  }

 public void go() {
    getSongs();
    System.out.println(songList);
    Collections.sort(songList);
    System.out.println(songList);

    ArtistCompare artistCompare = new ArtistCompare();
    Collections.sort(songList, artistCompare);

    System.out.println(songList);
  }

  void getSongs() {
    // I/Oコード
  }

  void addSong(String lineToParse) {
    // ファイルの各行のデータを分割し、リストに追加するコード
  }
}
```

Comparatorをインプリメントする内部クラス作成（型パラメータは、比較対象となるオブジェクトに合わせる。ここではSongにする）

これがString（アーチスト名）になる

（アーチスト名の）比較の処理には、Stringのcompare To()メソッドを使用する。Stringクラスには、あらかじめ「アルファベット順」を判定する機能が用意されている

Comparatorをインプリメントする内部クラスのインスタンスを作成

sort()を呼び出す。引数には、リストと、Comparatorオブジェクトへの参照を指定する

注意：ここでは、曲のタイトルによるソートには、デフォルトのsort()メソッド、SongクラスのcompareTo()メソッドを使用しています。しかし、曲のタイトルによるソート、アーチスト名によるソートの両方に、Comparatorをインプリメントする内部クラスを使用するということも可能です。その場合には、SongがComparableをインプリメントする必要はありません。その代わり、Collections.sort()の引数は常に2つにします。

This PDF is prepared for tools@maruhouse.org, personal use only.

```
import ___________________;
public class SortMountains {
  LinkedList_________________ mtn = new LinkedList_____________();
  class NameCompare ____________________________________ {
    public int compare(Mountain one, Mountain two) {
      return ____________________________;
    }
  }
  class HeightCompare ________________________________ {
    public int compare(Mountain one, Mountain two) {
      return (___________________________);
    }
  }
  public static void main(String [] args) {
    new SortMountains().go();
  }
  public void go() {
    mtn.add(new Mountain("Longs", 14255));
    mtn.add(new Mountain("Elbert", 14433));
    mtn.add(new Mountain("Maroon", 14156));
    mtn.add(new Mountain("Castle", 14265));

    System.out.println("as entered:\n" + mtn);
    NameCompare nc = new NameCompare();
    ________________________________;
    System.out.println("by name:\n" + mtn);
    HeightCompare hc = new HeightCompare();
    ________________________________;
    System.out.println("by height:\n" + mtn);
  }
}

class Mountain {
  ____________________;
  ___________________;

  ________________________ {
    ___________________;
    ___________________;
  }
  ___________________________ {
    ________________________________;
  }
}
```

自分で考えてみよう

リバースエンジニアリング

左のコードはすべて 1 つのファイルに収められています。下に示した結果が得られるよう、空白部分を埋めてください。

注意：解答は章の最後にあります。

出力：

```
File  Edit  Window  Help  ThisOne'ForBob
%java SortMountains
as entered:
[Longs 14255, Elbert 14433, Maroon 14156, Castle 14265]
by name:
[Castle 14265, Elbert 14433, Longs 14255, Maroon 14156]
by height:
[Elbert 14433, Castle 14265, Longs 14255, Maroon 14156]
```


This PDF is prepared for tools@maruhouse.org, personal use only.

自分で考えてみよう

空欄を埋めよ

1 ～ 9 の質問それぞれについて、下に用意した答えの中から適当なものを選んで空欄を埋めてください（解答は章末にあります）。

答えは以下から選ぶ：

Comparator
Comparable
compareTo()
compare()
yes
no

1 ～ 3 の質問には、下のようなコード（これはコンパイル可能な正しいコードです）が存在するという前提で答えてください。

```
Collections.sort(myArrayList);
```

1. myArrayList に保持されるオブジェクトのクラスがインプリメントしていなくてはならないインターフェースは？ _____
2. myArrayList に保持されるオブジェクトのクラスがインプリメントしていなくてはならないメソッドは？ _____
3. myArrayList に保持されるオブジェクトのクラスが同時に Comparator と Comparable をインプリメントすることはできるか。 _____

4 ～ 9 の質問には、下のようなコード（これはコンパイル可能な正しいコードです）が存在するという前提で答えてください。

```
Collections.sort(myArrayList, myCompare);
```

4. myArrayList に保持されるオブジェクトのクラスが Comparable をインプリメントすることはできるか。 _____
5. myArrayList に保持されるオブジェクトのクラスが Comparator をインプリメントすることはできるか。 _____
6. myArrayList に保持されるオブジェクトのクラスは Comparable をインプリメントしなくてはならないか。 _____
7. myArrayList に保持されるオブジェクトのクラスが Comparator をインプリメントしなくてはならないか。 _____
8. myCompare オブジェクトのクラスがインプリメントしなくてはならないインターフェースは？ _____
9. myCompare オブジェクトのクラスがインプリメントしなくてはならないメソッドは？ _____

This PDF is prepared for tools@maruhouse.org, personal use only.

ソートは問題なくできるが、要素の重複が……

前述の改造でソートは問題なくできるようになりました。曲のタイトルによるソート（Songオブジェクトのcompare To()メソッドを使う）も、アーチスト名によるソート（Comparatorのcompare()メソッドを使う）もできます。しかし、実は問題は他にもあります。ジュークボックスがテキストファイルに書き込むデータには重複が起きる場合があるのです。

ジュークボックスは、それ以前にかかった（つまりデータがファイルに書き込まれた）曲であるかどうか、ということには関知せず、ただひたすらかかった曲のデータをファイルに書き込むだけです。従って当然重複は起こり得ます。SongListMore.txtファイルには、かかったすべての曲のデータが完全に記録されるのですが、何度もかかった曲のデータは重複するのです。

```
File  Edit  Window  Help  TooManyNotes
%java Jukebox4

[Pink Moon: Nick Drake, Somersault: Zero 7, Shiva Moon: Prem
Joshua, Circles: BT, Deep Channel: Afro Celts, Passenger:
Headmix, Listen: Tahiti 80, Listen: Tahiti 80, Listen: Tahiti
80, Circles: BT]

[Circles: BT, Circles: BT, Deep Channel: Afro Celts, Listen:
Tahiti 80, Listen: Tahiti 80, Listen: Tahiti 80, Passenger:
Headmix, Pink Moon: Nick Drake, Shiva Moon: Prem Joshua,
Somersault: Zero 7]

[Deep Channel: Afro Celts, Circles: BT, Circles: BT, Passenger:
Headmix, Pink Moon: Nick Drake, Shiva Moon: Prem Joshua, Listen:
Tahiti 80, Listen: Tahiti 80, Listen: Tahiti 80, Somersault:
Zero 7]
```

ソート前

曲のタイトルによるソート（Songオブジェクトのcompare To()メソッドを使う）後

アーチスト名によるソート（ArtistCompareのcompare()メソッドを使う）後

SongListMore.txt

```
Pink Moon/Nick Drake/5/80
Somersault/Zero 7/4/84
Shiva Moon/Prem Joshua/6/120
Circles/BT/5/110
Deep Channel/Afro Celts/4/120
Passenger/Headmix/4/100
Listen/Tahiti 80/5/90
Listen/Tahiti 80/5/90
Listen/Tahiti 80/5/90
Circles/BT/5/110
```

ジュークボックスは、かかったすべての曲のタイトルを順に書き込んでいくだけなので、SongListMore.txtファイルにはデータの重複が生じる可能性がある。左のファイルの場合、"Listen"という曲が3回連続でかかり、"Circles"という曲は間をあけて2回かかっている。場合によっては、重複したデータもすべて必要になることがあるため、書き込みの方式を変更することはできない。ここではあくまでJavaコードの改造によって重複に対処する必要がある。

This PDF is prepared for tools@maruhouse.org, personal use only.

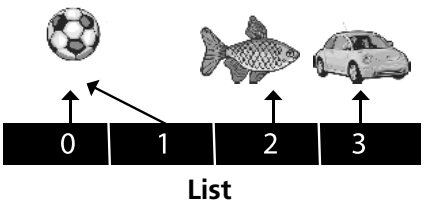
必要なのはListではなくSet

Collection APIには、List、Set、Mapという3つの主要インターフェースがあります。ArrayListはListのインプリメンテーションです。しかし、ここでの問題を解決するにはSetを使うべきです。

▶ List ― 要素の「並び方」が重要になる際に使用する

リストでは、要素にインデックスが付き、それぞれ決まった「位置」に保持される。
Listの場合、要素がどこにあるかがわかる。複数の要素が同じオブジェクトを参照する場合もある。

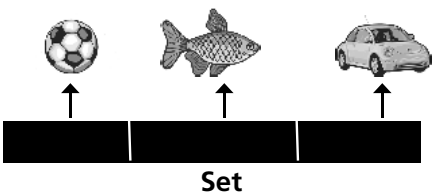
重複は起こり得る



▶ Set ―「重複がないこと」が重要な際に使用する

Setでは重複が許容されない。
Setでは、ある要素がすでにコレクション中に存在すれば、新たに同じものを追加することはできない。
同じオブジェクトを複数の要素が参照するということはない（あるいは、2つの要素が、同等とみなせる2つのオブジェクトを同時に参照する、ということもない。ここでの「同等」の意味については後述）。

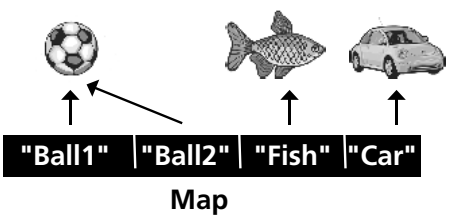
重複は許容されない



▶ MAP ―「キーによって要素を見付けられる」ことが重要な際に使用する

キーと値のペアを使用するコレクション。
Mapでは「キーと値のペア」が保持される。2つのキーが同時に同じ値に対応する場合もあるが、キーの重複は許容されない。キーは通常は名前（文字列）になるが、他のどのようなオブジェクトでもかまわない（キーを名前にした場合は、名前と値から成るプロパティリストを作ることができる）。

値の重複はいいが、キーの重複は許容されない

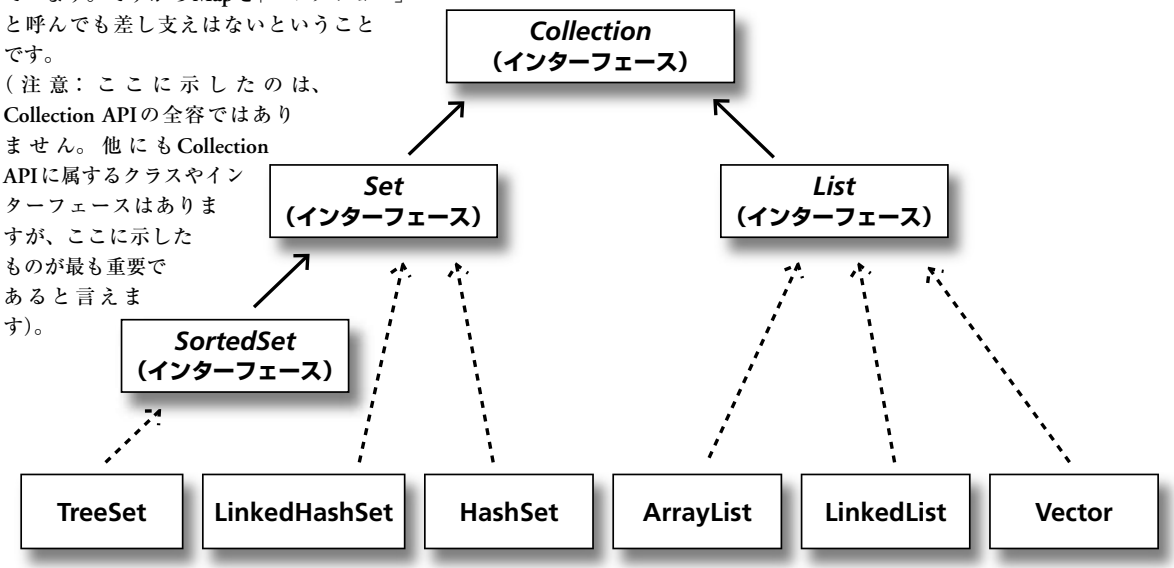


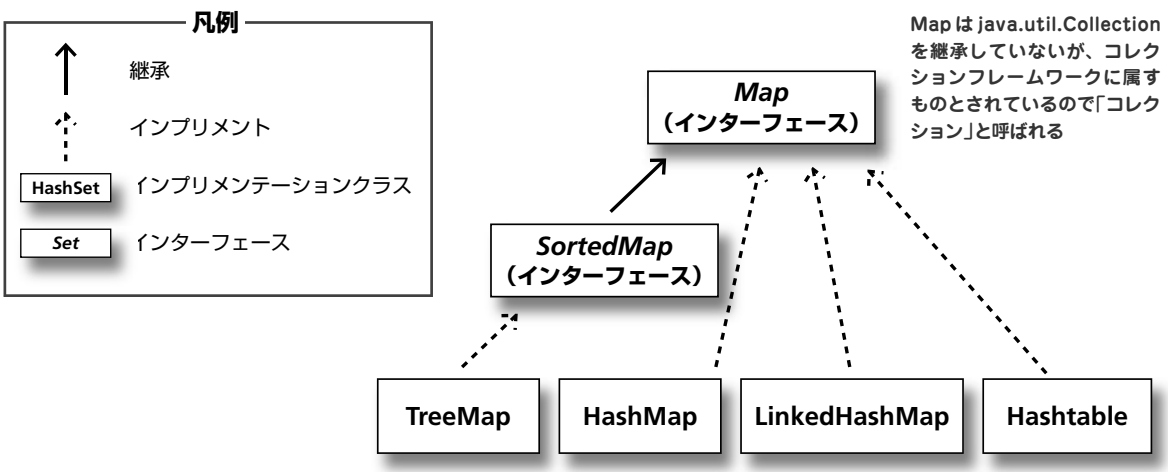
This PDF is prepared for tools@maruhouse.org, personal use only.

Collection API(一部)

Mapインターフェースは正確にはCollectionインターフェースを継承しているわけではありません。しかし、それでも、Mapは「コレクションフレームワーク(Collection API)」に属しているとされています。ですからMapを「コレクション」と呼んでも差し支えはないということです。

(注意:ここに示したのは、Collection APIの全容ではありません。他にもCollection APIに属するクラスやインターフェースはありますが、ここに示したものが最も重要であると言えます)。





Mapはjava.util.Collectionを継承していないが、コレクションフレームワークに属すものとされているので「コレクション」と呼ばれる

This PDF is prepared for tools@maruhouse.org, personal use only.

HashSetを使用する

重複の問題を解決するため、HashSetを使用して以下のようにコードを改造しました（注意：変更のない部分に関しては省略をしているので、使う際には改造前のバージョンからコピーするようにしてください。出力が読みやすいよう、SongクラスのtoString()メソッドは前のバージョンのものに戻し、曲のタイトルとアーチスト名ではなく、曲のタイトルだけを出力するようにしました）

```
import java.util.*;
import java.io.*;

public class Jukebox6 {
  ArrayList<Song> songList = new ArrayList<Song>();
  // main メソッドなど

  public void go() {
    getSongs();
    System.out.println(songList);
    Collections.sort(songList);
    System.out.println(songList);
    HashSet<Song> songSet = new HashSet<Song>();
    songSet.addAll(songList);
    System.out.println(songSet);
  }
  // getSongs() メソッドと addSong() メソッド
}
```

getSongs()は変更なし。引き続きSongオブジェクトをArrayListに入れる

型をSongに指定してHashSetを作成

HashSetには、他のコレクションを引数に指定すると、その全要素をHashSetに入れるaddAll()という簡単なメソッドがある。個々のSongを1つずつ追加していくのと結局は同じ（ただしずっと簡単にできる）

```
File  Edit  Window  Help  GetBetterMusic
%java Jukebox6

[Pink Moon, Somersault, Shiva Moon, Circles, Deep Channel,
Passenger, Listen, Listen, Listen, Circles]

[Circles, Circles, Deep Channel, Listen, Listen, Listen,
Passenger, Pink Moon, Shiva Moon, Somersault]

[Pink Moon, Listen, Shiva Moon, Circles, Listen, Deep Channel,
Passenger, Circles, Listen, Somersault]
```

ArrayListのソート前

ArrayListを（曲のタイトルによって）ソート後

要素をHashSetに入れた後、HashSetを出力（sort()を再度呼び出してはいない）

HashSetに入れただけでは重複は解消されていない

（しかも、HashSetに要素を入れた際、ソートした順序も狂ってしまっている。その点については後述……）

This PDF is prepared for tools@maruhouse.org, personal use only.

オブジェクトを「同等」とみなす条件は？

ここで問題にしているのは、「要素の重複」ですが、Songオブジェクトへの参照が2つある時、一体どのような条件が満たされれば、その2つは重複している、とみなせばいいのでしょうか。重複しているとみなすには、2つが「同等」である必要があります。では、同等であるとは、2つが「同じオブジェクトに対応していること」を指すのでしょうか。それとも「曲のタイトルが一致していること」を指すのでしょうか。

ここで覚えておくべきなのは「同等」にも、「参照の同等」と「オブジェクトの同等」があるということです。

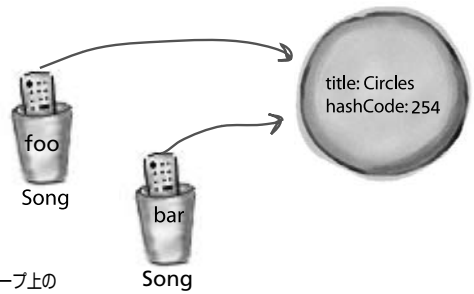
2つのオブジェクト、foo、barを同等なものとして扱うには、foo.equals(bar)の戻り値はtrueにしなければならず、hashCode()の戻すハッシュコードも両者同じにしなければならない。Setを使ってオブジェクトの重複を解消するには、Objectから継承されたhashCode()メソッドとequals()メソッドをオーバーライドし、2つの（本来は異なる）オブジェクトを同等とみなせるようにする必要がある。

▶参照の同等

参照は2つだが、ヒープ上にオブジェクトは1つ

「参照の同等」は、2つの参照が、ヒープ上の同じオブジェクトに対応している状態を言います。この場合、2つの参照を使ってhashCode()メソッドを呼び出すと、どちらも同じ結果が得られます。hashCode()メソッドをオーバーライドしていなければ（hashCode()はすでに見てきたとおりObjectクラスから継承されるメソッド）、各オブジェクトに固有のハッシュコード（整数）が戻されることになります（ほぼどのバージョンのJavaでも、ハッシュコードは、オブジェクトのヒープ上でのメモリアドレスに基づいて割り当てられる。つまり、2つのオブジェクトのハッシュコードが一致することは決してない）。

2つの参照が本当に同じオブジェクトに対応しているかを確かめるには、演算子"=="を使用します。これによって、（すでに述べたとおり）2つの変数のビットパターンが比較されます。両者が同じオブジェクトに対応していれば、ビットパターンは同じになるはずです。



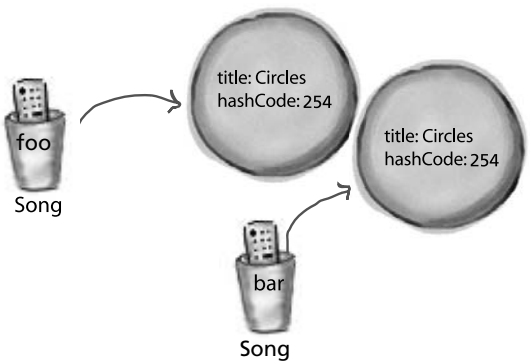

```
if (foo == bar) {
    // 2つの参照はヒープ上の
    // 同じオブジェクトに対応している
}
```

▶オブジェクトの同等

2つの参照があって、ヒープ上に2つのオブジェクトがあるが、2つのオブジェクトは実質上同等であるとみなせる、という状態を言います。

2つの違ったオブジェクトを実質上同等のものとして扱うには、Objectクラスから継承されるhashCode()、equals()といったメソッドをオーバーライドする必要があります。何を条件に「同等」とみなしたいかによって、コードを変えます。たとえば「曲のタイトルが同じなら2つのSongオブジェクトは同じとみなすようにしたい」というのであれば、そのようにコードを書きます。

上で述べているとおり、hashCode()をオーバーライドしなければ、各オブジェクトの固有のハッシュコードが戻されます。これは、Objectクラスから継承されたデフォルトの機能です。2つのオブジェクトを同等なものにするには、両者が同じハッシュコードを戻すようhashCode()のコードを書かなければなりません。さらに、equals()も、一方のオブジェクトを使い、もう一方のオブジェクトを引数にして呼び出した時にtrueが戻るようにコードが書かなくてはなりません。




```
if (foo.equals(bar) && foo.hashCode() == bar.hashCode()) {
    // 2つの参照はそれぞれ別のオブジェクトに対応しているが
    // 2つのオブジェクトは同等とみなす
}
```


This PDF is prepared for tools@maruhouse.org, personal use only.

HashSetの重複確認：hashCode()とequals()

Hashsetにオブジェクトを追加しようとすると、そのオブジェクトのハッシュコードによって、Set中のどこに保持されるかが決められます。また、同時に、追加されるオブジェクトのハッシュコードと、HashSet中の他のすべてのオブジェクトのハッシュコードが比較され、一致するものがまったくなければ、「重複はない」とみなされます。

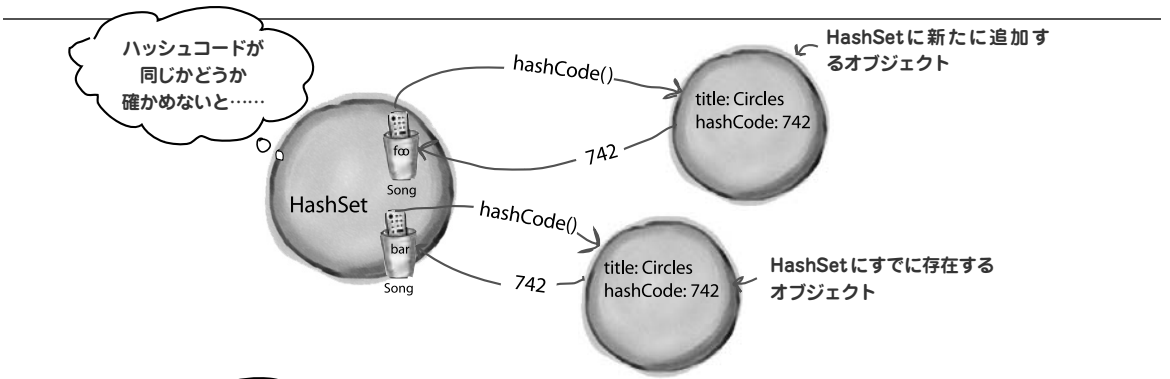
逆に言えば、ハッシュコードが異なれば、HashSetがオブジェクトを同等なものとみなすことは決してないということになります。

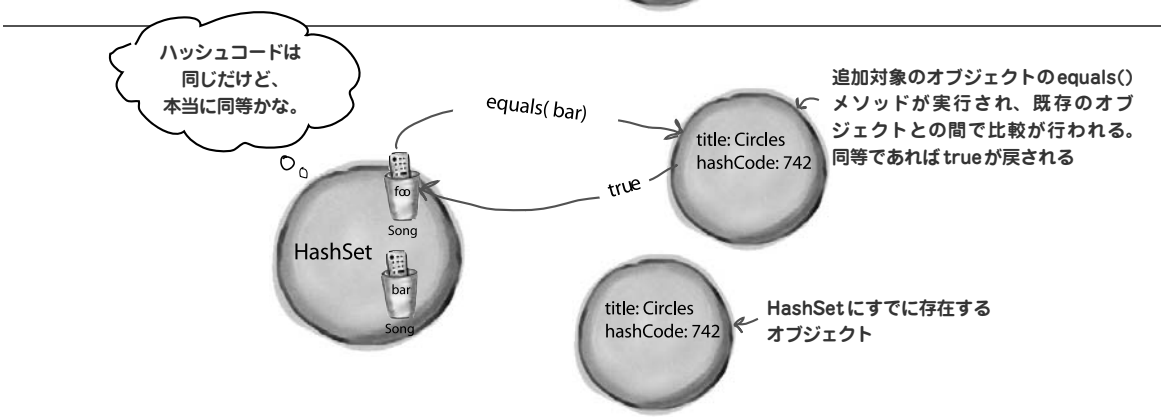
hashCode()は、同等とみなすべきオブジェクトに関しては同じハッシュコードを戻すようオーバーライドする必要があります。

ただハッシュコードがたとえ同じでも、2つのオブジェクトが同等であるとは限りません（詳細は後述）。そこで、追加しようとするオブジェクトと同じハッシュコードを持つオブジェクトがHashSet中に見付かった場合には、一方のオブジェクトのequals()メソッドを呼び出して両者が本当に同等であるかを確認します。

両者が同等であれば、追加しようとしたオブジェクトは、すでにHashSet中に存在するオブジェクトと重複するということになります。その場合には、追加は行われません。

その場合でも例外は発生しませんが、HashSetのadd()メソッドは、オブジェクトが追加されたかどうかを知らせるブール値を戻すので、気になる場合はそれで確認できます。add()メソッドがfalseを戻したら、追加しようとしたオブジェクトが、すでにHashSetに存在したオブジェクトのいずれかと重複したとみなすことができます。





This PDF is prepared for tools@maruhouse.org, personal use only.

hashCode()とequals()をオーバーライドしたSongクラス

```
class Song implements Comparable<Song>{
  String title;
  String artist;
  String rating;
  String bpm;

  public boolean equals(Object aSong) {
    Song s = (Song) aSong;
    return getTitle().equals(s.getTitle());
  }

  public int hashCode() {
    return title.hashCode();
  }

  public int compareTo(Song s) {
    return title.compareTo(s.getTitle());
  }

  Song(String t, String a, String r, String b) {
    title = t;
    artist = a;
    rating = r;
    bpm = b;
  }

  public String getTitle() {
    return title;
  }

  public String getArtist() {
    return artist;
  }

  public String getRating() {
    return rating;
  }

  public String getBpm() {
    return bpm;
  }

  public String toString() {
    return title;
  }
}
```

このメソッドを呼び出す際は、必ず、呼び出しに使用するSongオブジェクトとは別のSongオブジェクトを引数として渡す

幸い、曲のタイトルはStringであり、Stringにはオーバーライドされたequals()メソッドがあらかじめ用意されている。一方のSongオブジェクトのタイトル(Stringオブジェクト)を使ってequals()メソッドを呼び出す。引数には、もう一方のSongオブジェクトのタイトルを指定する

これも同様……、Stringクラスには、オーバーライドされたhashCode()メソッドもある。そこで、一方のタイトルを使用してhashCode()を呼び出し、その結果を戻す。hashCode()とequals()が同じインスタンス変数を使用している点に注意

今度は成功。HashSetを出力すると重複がないことがわかる。ただ、sort()メソッドを呼び出していないため、ArrayListの内容をHashSetに移した時に順序が狂ったまま

```
File  Edit  Window  Help  RebootWindows
%java Jukebox6

[Pink Moon, Somersault, Shiva Moon, Circles,
Deep Channel, Passenger, Listen, Listen,
Listen, Circles]

[Circles, Circles, Deep Channel, Listen,
Listen, Listen, Passenger, Pink Moon, Shiva
Moon, Somersault]

[Pink Moon, Listen, Shiva Moon, Circles,
Deep Channel, Passenger, Somersault]
```


This PDF is prepared for tools@maruhouse.org, personal use only.

Object クラスの HashCode()、equals() についてのルール

Objectクラスのドキュメントを見ると、以下のようなルールがあることがわかります。

- **2つのオブジェクトが同等であれば、ハッシュコードは一致する**
- **2つのオブジェクトが同等であれば、その2つを使ってequals()を呼び出した結果はtrueになる。つまり、a.equals(b)、b.equals(a)ともにtrue。**
- **2つのオブジェクトのハッシュコードが一致したからといって、両者が同等であるとは限らない。しかし、2つのオブジェクトが同等なら、必ずハッシュコードは一致する。**
- **従って、equals()をオーバーライドするのであれば、同時にhashCode()もオーバーライドしなくてはならない。**
- **hashCode()はデフォルトでは、ヒープ上の各オブジェクトについて、固有のハッシュコード(整数)を生成する。そのため、hashCode()をオーバーライドしなければ、2つのオブジェクトが同等とみなされることは決してない。**
- **equals()はデフォルトでは、"==" と同様の比較を行う。これは、指定の2つの参照がヒープ上の同一のオブジェクトに対応しているかを確認するということ。そのため、equals()をオーバーライドしなければ、2つのオブジェクトが同等とみなされることは決してない。参照に対応するオブジェクトが異なれば、参照のビットパターンは同一にならないため。**

a.equals(b)は以下のように書き換えられる。
a.hashCode() == b.hashCode()

ただし、a.hashCode() == b.hashCode()は、必ずしもa.equals(b)に書き換えられるわけではない。

素朴な疑問に答えます

Q：オブジェクトが同等でなくてもハッシュコードが同じになる可能性がある、というのはどういうことですか。

A：HashSetでは、保持する要素へのアクセスの効率化のためにハッシュコードを利用します。たとえば、ArrayListにあるオブジェクトが含まれているかを確認するのに、(インデックスではなく)オブジェクトのコピーを渡した場合、ArrayListは逐一、各要素の全体を見て、渡されたコピーと一致しているか確かめなくてはなりません。しかし、HashSetは、それよりはるかに短時間でオブジェクトの検索ができます。ハッシュコードを利用するからです。ハッシュコードは、各要素が入った「バケツ」に貼られたラベルのようなものです。「これと同等のオブジェクトを要素の中から探せ」と指示された場合、HashSetはまず、そのオブジェクトのハッシュコードを取得します。仮にそのハッシュコードが「742」だったとすると、「742」というラベルの貼られた「バケツ」を探せば、即、目的のオブジェクトは見付かります(あるいはそのオブジェクトが要素の中にないことがわかる)。
厳密にはこれほど単純な話ではないのですが、HashSetを使いこなすにはこれだけわかっていればとりあえずは十分でしょう。ハッシュコードは、実際には、博士論文のテーマになるほどのものですが、本書でそこまでのことを論じるつもりはありません。
オブジェクトが同等でなくてもハッシュコードが同じになるのは、hashCodeメソッドで使用されている「ハッシュコードアルゴリズム」がそういうものになっているからです。このアルゴリズムでは、複数のオブジェクトについて、偶然同一のハッシュコードが得られる可能性があるのです。つまり、HashSet中の1つの「バケツ」に同時に複数のオブジェクトが入ることがある、ということですが(ハッシュコードは必ず1つの「バケツ」につき1つ割り当てられるため)、これはさほど大きな問題とは言えません。悪影響があったとしても、せいぜいHashSetの効率が少々低下するくらいのものです(保持する要素が多くなり過ぎてしまう恐れはありますが)。同一のバケツ内に複数のオブジェクトがある場合、HashSetは、equals()メソッドを使って、どれが本当に目的のオブジェクトと一致しているかを確かめます。ハッシュコードは、ただ検索の範囲を狭めるためのもの、と理解してもいいでしょう。完全に一致するものを見付けるには、ハッシュコードが一致した「バケツ」内のすべてのオブジェクトについて、equals()による確認を行わなくてはなりません。

This PDF is prepared for tools@maruhouse.org, personal use only.

ソートした要素の順序が狂わないためにはTreeSetを使う

TreeSetは、重複を防げるという点ではHashSetと同じです。ただ、TreeSetを使うと、HashSetの場合のように、ソートした要素の順序が狂ってしまうということを防げます。TreeSetのインスタンスを、引数のないコンストラクタを使って作成すると、sort()メソッドのような処理が行われます。要素となっている各オブジェクトのcompareTo()メソッドを使ってソートが行われるのです。コンストラクタの引数にComparatorを指定して、それを代わりに使わせるということも可能です。TreeSetを使うデメリットは、ソートの必要が特にない場合でもソートが行われ、それによって少しパフォーマンスが低下するということです。とはいえ、そのパフォーマンス低下は、通常、ほとんど認識できない程度のものです。

```
import java.util.*;
import java.io.*;
public class Jukebox8 {
  ArrayList<Song> songList = new ArrayList<Song>();
  int val;

  public static void main(String[] args) {
    new Jukebox8().go();
  }

  public void go() {
    getSongs();
    System.out.println(songList);
    Collections.sort(songList);
    System.out.println(songList);
    TreeSet<Song> songSet = new TreeSet<Song>();
    songSet.addAll(songList);
    System.out.println(songSet);
  }

  void getSongs() {
    try {
      File file = new File("SongListMore.txt");
      BufferedReader reader = new BufferedReader(new FileReader(file));
      String line = null;
      while ((line= reader.readLine()) != null) {
      addSong(line);
      }
    } catch(Exception ex) {
      ex.printStackTrace();
    }
  }

  void addSong(String lineToParse) {
    String[] tokens = lineToParse.split("/");
    Song nextSong = new Song(tokens[0], tokens[1], tokens[2], tokens[3]);
    songList.add(nextSong);
  }
}
```

HashSetではなくTreeSetをインスタンス化する。引数のないコンストラクタを使うと、SongオブジェクトのcompareTo()メソッドを使ってソートが行われる(Comparatorを引数として渡すことも可能)。

addAll()を使用してすべてのSongオブジェクトをTreeSetに(ArrayListの場合と同様、songSet.add()を使用してSongオブジェクトを逐一追加することもできる)

This PDF is prepared for tools@maruhouse.org, personal use only.

TreeSetについての注意点

TreeSetを使うことは一見、何も難しくなさそうですが、実際に使う時には注意しなくてはならないこともあります。それを学んでもらうため、以下のエクササイズを用意しました。自分なりの答えを出すまでは決して次のページを見ないでください。

自分で考えてみよう

右のコードをよく見て、下の質問に答えてください（注意：このコードには文法上の誤りなどはありません）。

```
import java.util.*;

public class TestTree {
  public static void main (String[] args) {
    new TestTree().go();
  }

  public void go() {
    Book b1 = new Book("How Cats Work");
    Book b2 = new Book("Remix your Body");
    Book b3 = new Book("Finding Emo");

    TreeSet<Book> tree = new TreeSet<Book>();
    tree.add(b1);
    tree.add(b2);
    tree.add(b3);
    System.out.println(tree);
  }
}

class Book {
  String title;
  public Book(String t) {
    title = t;
  }
}
```

1). このコードをコンパイルすると結果はどうなるでしょうか。

2). 正しくコンパイルできるとすれば、実行した時にどのような結果が得られますか。

3). コードに何か問題（コンパイルに関する問題、実行に関する問題）はありますか。あるとすれば、どうすれば解決できますか。

This PDF is prepared for tools@maruhouse.org, personal use only.

TreeSetの要素は「比較のできる」ものにする

TreeSetは、当然、プログラマの心を読めるわけではないので、オブジェクトをどのようにソートしてよいか、指示されない限りわかりません。

TreeSetを使用する際には、次のいずれかの条件を満たす必要があります。

▶Comparableをインプリメントするオブジェクトを要素にする

前頁のBookクラスは、Comparableをインプリメントするものではありません。したがって、実行時に機能しません。TreeSetが要素のソートをしようとしても、これではソートの仕方がわからないのです。ただし、TreeSetのadd()メソッドの引数は、Comparableでなくてもよいことになっているため、コンパイルはできます。このadd()メソッドは、TreeSet作成時に指定された型であれば、どのようなものにでも対応できます。new TreeSet<Book>()と書いたのであれば、add()メソッドは自動的にadd(Book)となるということです。BookクラスはComparableをインプリメントしていなくてはならない、というわけではありません。実行時には、TreeSetに2つ目の要素を追加する際に問題が生じます。この時はじめて、TreeSetは、要素のオブジェクトのcompareTo()メソッドを呼び出そうとするのですが、呼び出すことができないからです。

```
class Book implements Comparable {
  String title;
  public Book(String t) {
    title = t;
  }
  public int compareTo(Object b) {
    Book book = (Book) b;
    return (title.compareTo(book.title));
  }
}
```

▶Comparatorを引数とするオーバーロードバージョンのコンストラクタを使用する

TreeSetは、sort()メソッドと似たような機能を果たします。そのため、要素のオブジェクトのcompareTo()メソッドを使うか(この場合は要素がComparableインターフェースをインプリメントしている必要がある)、Comparatorをインプリメントしたクラス(このクラスによってソートの仕方を知らせる)を使う必要が出てきます。Comparatorをインプリメントしたクラスを使う場合は、TreeSetのコンストラクタで引数として指定します。

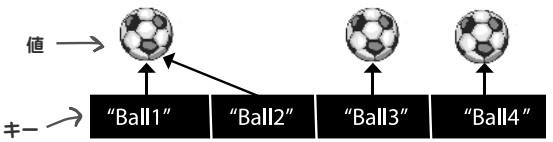
```
public class BookCompare implements Comparator<Book> {
  public int compare(Book one, Book two) {
    return (one.title.compareTo(two.title));
  }
}
class Test {
    public void go() {
    Book b1 = new Book("How Cats Work");
    Book b2 = new Book("Remix your Body");
    Book b3 = new Book("Finding Emo");
    BookCompare bCompare = new BookCompare();
    TreeSet<Book> tree = new TreeSet<Book>(bCompare);
    tree.add(new Book("How Cats Work");
    tree.add(new Book("Finding Emo");
    tree.add(new Book("Remix your Body");
    System.out.println(tree);
  }
}
```


This PDF is prepared for tools@maruhouse.org, personal use only.

List、Setの次はMap……

List、Setなどのコレクションはともに非常に便利なものです。しかし、時にはMapが最良の選択であるという場合があります（すでに述べたとおり、Mapは正確にはCollectionインターフェースを継承しているわけではないが「コレクションフレームワーク」に属しているとされる）。

一種の「プロパティリスト」のようなコレクションが必要になった場合にMapは力を発揮します。キーを指定すると、その名前に対応した値を戻してくれる、というようなコレクションです。キーは名前（文字列）であることが多いですが、Javaのオブジェクトであれば何でもキーにできます（自動ラッピングができるのでプリミティブもキーにすることが可能）。



Mapの個々の要素は2つのオブジェクト（キーと値）から成る
値の重複は許容されるが、キーの重複は許容されない

Mapの例

```
import java.util.*;

public class TestMap {

  public static void main(String[] args) {

    HashMap<String, Integer> scores = new HashMap<String, Integer>();

    scores.put("Kathy", 42);
    scores.put("Bert", 343);
    scores.put("Skyler", 420);

    System.out.println(scores);
    System.out.println(scores.get("Bert"));
  }
}
```

HashMapには2つの型パラメータが必要。一方はキー、もう一方は値に対応する

add()ではなくput()を使う。また、もちろん、引数は2つ（キーと値）

get()メソッドは、キーを指定すると値（ここでは整数）が戻される

```
File  Edit  Window  Help  WhereAml
%java TestMap
{Skyler=420, Bert=343, Kathy=42}
343
```

Mapの要素は「キー＝値」という形式で出力される。ListやSetと違い、[]ではなく{}が使用されるのも特徴

This PDF is prepared for tools@maruhouse.org, personal use only.

再びジェネリック型について

ジェネリック型の引数を持つメソッドに関してはこの章ですでに少し触れました。この種の引数は、「ポリモーフィズム」という点に関して、従来のJavaとは違った特徴があります。ここでは、何ページかにわたってその特徴について触れることにします。

まずは確認のために、例をあげて「配列」の引数がどのようなものだったかをポリモーフィズムの観点からもう一度見てみましょう。その後、同様の例を使ってジェネリック型について見ていきます。以下のコードを見てください。これはコンパイルも実行も問題なくできるコードです。

メソッドの引数がAnimalの配列であれば、Animalのサブクラスの配列も使用できる。

つまりメソッドの宣言が、

void foo(Animal[] a) { }

となっていて、DogがAnimalのサブクラスであれば、以下のように書ける。

foo(anAnimalArray);
foo(aDogArray);

通常の配列を使ったコード：

```
import java.util.*;

public class TestGenerics1 {
  public static void main(String[] args) {
    new TestGenerics1().go();
  }

  public void go() {
    Animal[] animals = {new Dog(), new Cat(), new Dog()};
    Dog[] dogs = {new Dog(), new Dog(), new Dog()};
    takeAnimals(animals);
    takeAnimals(dogs);
  }

  public void takeAnimals(Animal[] animals) {
    for(Animal a: animals) {
    a.eat();
    }
  }
}
```

Animalの配列の宣言、作成。DogもCatも要素にできる

Dogの配列の宣言、作成。要素はDogのみ（Catを入れるとコンパイルできない）

takeAnimals()の呼び出し。animals、dogsどちらの配列も引数にできる…

重要なのは、takeAnimals()メソッドの引数として、Animal[]、Dog[]の両方が使用できること。DogはAnimalのサブクラス（ポリモーフィズム）

ここではAnimalクラスのメソッドのみ呼びだせる点に注意。animalsパラメータはAnimalの配列であり、キャストは行っていない（キャストするとしたら何型？配列にはDogやCatが保持される可能性がある）

```
abstract class Animal {
  void eat() {
    System.out.println("animal eating");
  }
}
class Dog extends Animal {
  void bark() { }
}
class Cat extends Animal {
  void meow() { }
}
```

Animalクラスの階層構造

This PDF is prepared for tools@maruhouse.org, personal use only.

ポリモーフィズムとジェネリック型

ここまでは配列を使った例でしたが、同様のコードを、配列をArrayListに変えて書いたらどうなるでしょうか。特に問題はなさそうに思えますが果たしてどうでしょうか。

最初はAnimal型のArrayListだけを使ったコードを見てみましょう。配列をArrayListに変え、go()メソッドに少し手を加えてあります。

ArrayList<Animal>だけを使ったコード

Animal[]をArrayList<Animal>に変えるだけ

```
public void go() {
  ArrayList<Animal> animals = new ArrayList<Animal>();
  animals.add(new Dog());
  animals.add(new Cat());
  animals.add(new Dog());

  takeAnimals(animals);
}

public void takeAnimals(ArrayList<Animal> animals) {
  for(Animal a: animals) {
    a.eat();
  }
}
```

配列の作成のようなショートカットの構文がないので、要素を逐一追加

animals変数が配列ではなく、ArrayListに対応している以外は前ページと同じ

メソッドの引数は配列ではなくArrayListになっているが、他は同じ。forループのこの構文は配列にもコレクションにも有効

コンパイルも実行も問題なくできる

```
File  Edit  Window  Help  CatFoodIsBetter
%java TestGenerics2
animal eating
animal eating
animal eating
animal eating
animal eating
animal eating
```


This PDF is prepared for tools@maruhouse.org, personal use only.

ArrayList<Dog>ではどうか……

メソッドにAnimalの配列の引数があれば、Dogの配列を指定することも可能、というのがポリモーフィズムです。前ページの例で、ArrayList<Animal>の引数にArrayList<Animal>を指定できることは確かめました。そこで問題になるのは、ArrayList<Animal>の代わりにArrayList<Dog>を指定できるのかということです。配列で同様のことはできましたが、ArrayListの場合はどうでしょうか。

ArrayList<Dog>を引数に指定

```
public void go() {
  ArrayList<Animal> animals = new ArrayList<Animal>();
  animals.add(new Dog());
  animals.add(new Cat());
  animals.add(new Dog());
  takeAnimals(animals);

  ArrayList<Dog> dogs = new ArrayList<Dog>();
  dogs.add(new Dog());
  dogs.add(new Dog());
  takeAnimals(dogs);
}

public void takeAnimals(ArrayList<Animal> animals) {
  for(Animal a: animals) {
  a.eat();
  }
}
```

これが問題ないことは確認済み

DogのArrayListを作成してDogを入れる

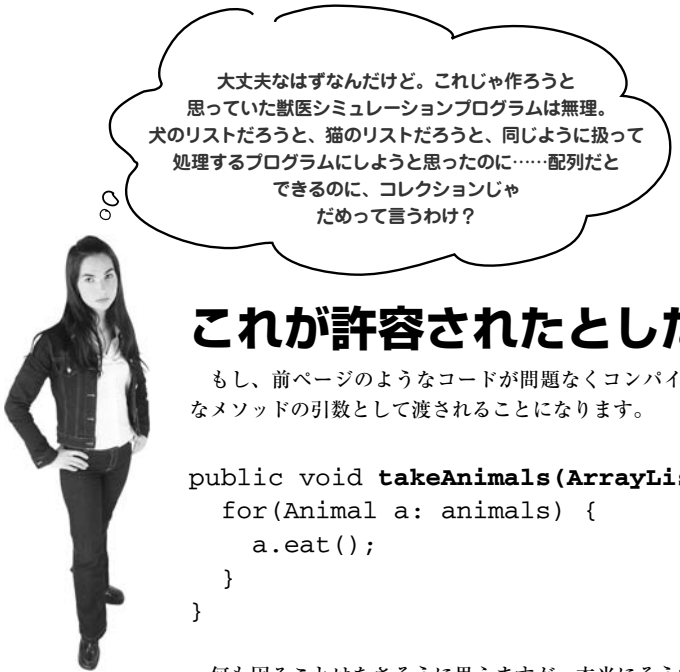
配列をArrayListに変えて問題は起きないか

コンパイルの結果：

```
File  Edit  Window  Help  CatsAreSmarter
%java TestGenerics3

TestGenerics3.java:21: takeAnimals(java.util.
ArrayList<Animal>) in TestGenerics3 cannot be applied to
(java.util.ArrayList<Dog>)
        takeAnimals(dogs);
        ^
1 error
```

コンパイルできるはず……と思ったが、エラーになる

This PDF is prepared for tools@maruhouse.org, personal use only.



これが許容されたとしたら……

もし、前ページのようなコードが問題なくコンパイルできるとしたら、ArrayList<Dog>が以下のようなメソッドの引数として渡されることになります。

```
public void takeAnimals(ArrayList<Animal> animals) {
  for(Animal a: animals) {
    a.eat();
  }
}
```

何も困ることはなさそうに思えますが、本当にそうでしょうか。ポリモーフィズムの基本は、「Animalが持っている機能（ここではeat()メソッド）はすべてDogも持っているということです。では、Dogへの参照を使ってeat()メソッドを呼び出すことに問題はあるのでしょうか。

何も問題はありません。

それ自体には問題はないのですが、以下のようなコードがあったとしたらどうでしょうか。

```
public void takeAnimals(ArrayList<Animal> animals) {
  animals.add(new Cat());
}
```

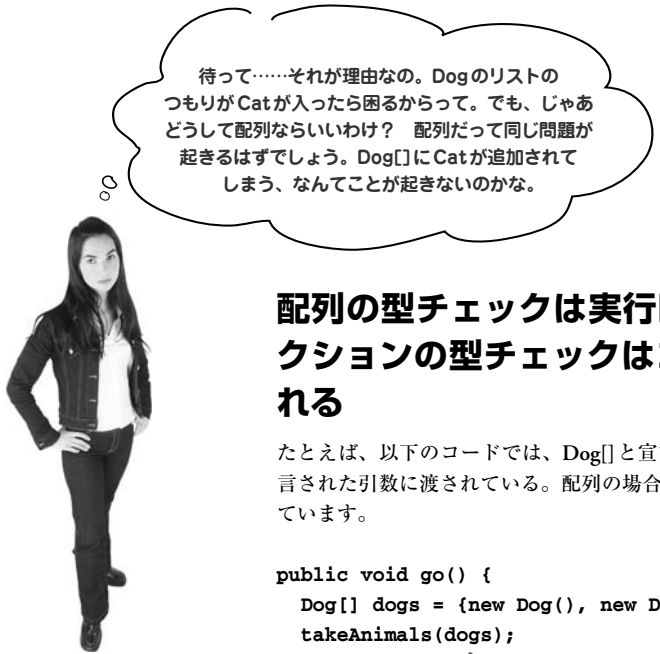
これは問題!! DogだけのArrayListにするはずがCatが追加されている

実はこれが問題なのです。もちろん、ArrayList<Animal>へのCatの追加を許容すること自体は良いのです。Animalのようなサブクラスを持つクラスのArrayListを作る意図はそこにあるのかもしれません。Animalのサブクラスであればすべてを受け入れる、というArrayListを作りたい場合もあるでしょう。

しかし、皆がそうとは限りません。Dogのみを保持する目的でAnimalのArrayListを作り、引数にするということもあるでしょう。そのArrayListに突然Catが追加されてしまう、という事態も起こり得ます。コンパイル時にはそうでなくても、実行時にCatが追加される事態は起こり得るのです。起きないかもしれませんが、コンパイラはリスクを嫌ってエラーを発生させます。

ArrayList<Animal>と宣言された引数には、ArrayList<Animal>のみ指定でき、ArrayList<Dog>やArrayList<Cat>は指定できないと覚えておいてください。

This PDF is prepared for tools@maruhouse.org, personal use only.



配列の型チェックは実行時にも行われるが、コレクションの型チェックはコンパイル時にのみ行われる

たとえば、以下のコードでは、Dog[]と宣言された配列（この配列が、Animal[]と宣言された引数に渡されている。配列の場合はこれが許容されている）にCatを追加しています。

```
public void go() {
  Dog[] dogs = {new Dog(), new Dog(), new Dog()};
  takeAnimals(dogs);
}

public void takeAnimals(Animal[] animals) {
  animals[0] = new Cat();
}
```

Dogの配列にCatを入れている。コンパイラはこれを許容する。メソッドの引数の型はAnimal []なので、Catの配列もAnimalの配列も渡される可能性があるため。コンパイラはこれを問題とはみなさない。

コンパイルはできるが、実行すると以下のような結果になる

```
File  Edit  Window  Help  CatsAreSmarter
%java TestGenerics1
Exception in thread "main" java.lang.ArrayStoreException:
Cat
        at TestGenerics1.takeAnimals(TestGenerics1.java:16)
        at TestGenerics1.go(TestGenerics1.java:12)
        at TestGenerics1.main(TestGenerics1.java:5)
```

コンパイラは許容してもJVMは許容しない

This PDF is prepared for tools@maruhouse.org, personal use only.

This PDF is prepared for tools@maruhouse.org, personal use only.

ワイルドカードを使えば……

実際にはそういうことはあまりしませんが、メソッドの引数を、Animalのあらゆるサブクラスに対応するものにする、ということは可能です。最も簡単なのは、まさにそのためにJavaに新たに導入された「ワイルドカード」と呼ばれる機能を使うことです。

```
public void takeAnimals(ArrayList<? extends Animal> animals) {
  for(Animal a: animals) {
    a.eat();
  }
}
```

ここでのキーワード "extends" はすでに述べたとおり、「継承」と「インプリメント」の両方を意味する。どちらの意味になるかは右に何が指定されるかによって変わる。「Petインタフェースをインプリメントする型のArrayList」という指定にしたければ、次のように書く。
ArrayList<? extends Pet>

「一体何が違うのか」と疑問に思っている人もいるでしょう。ワイルドカードにしても問題はなくならないのではないか、と感じる人もいるはずです。上記のようなコードでは問題は起きないにしても、すでに見てきたように、ArrayList<Dog>にCatを追加してしまうようなことが起きないと言えるのでしょうか。実行時のチェックが行われない限り、ワイルドカードを使っても使わなくても変わらないのではないか、そう思った人も多いでしょう。

疑問に思うのは無理もありませんが、ワイルドカードを使えば使わない場合と異なるのは間違いありません。宣言にワイルドカード(<?>)を使った場合、リストに新たな要素を追加するような処理はできなくなるからです。そうした処理を行おうとするとコンパイルエラーになります。

メソッドの引数にワイルドカードを使用した場合、引数に渡されたリストに問題を引き起こすような処理が行われそうになると、コンパイラがそれを防ぐ。リスト中の要素を使用してメソッドを呼び出すことはできるが、リストに要素を追加することはできない。
リストの既存の要素を操作することはできても要素の追加はできないということ。問題になりそうなコードはコンパイル時にすべて排除されるため、実行時に問題が起きることはない。

正しいコード(takeAnimals()メソッド中のコード):

```
for(Animal a: animals) {
  a.eat();
}
```

誤った(コンパイルできない)コード:

```
animals.add(new Cat());
```


This PDF is prepared for tools@maruhouse.org, personal use only.

別の書き方

sort() メソッドのドキュメントの内容はすでに本書の中で見てきました。ドキュメントを見ると、sort() メソッドでジェネリック型を使うことはすぐにわかります。また、戻り値の前に変わった形式の型パラメータの宣言があったことを覚えている人もいるでしょう。この宣言を使えば、前ページのものと同様のコードを違った形式で書けます。

これと、

```
public <T extends Animal> void takeThing(ArrayList<T> list)
```

これはどちらも同じ結果が得られる。

```
public void takeThing(ArrayList<? extends Animal> list)
```

素朴な疑問に答えます

Q：どちらも同じなら一方だけを使えばいいということですか。

A：どちらを選ぶかは、"T" を 2 箇所以上で使うかどうかで決めればいいでしょう。たとえば、2 つの引数を持つメソッドを作る場合、引数がどちらも Animal を継承するものであれば、

```
public <T extends Animal> void takeThing(ArrayList<T> one, ArrayList<T> two)
```

というように、"T" を使って型パラメータを一度だけ宣言する方が、

```
public void takeThing(ArrayList<? extends Animal> one, ArrayList<? extends Animal> two)
```

というように書くより効率的です。

This PDF is prepared for tools@maruhouse.org, personal use only.

エクササイズ

コンパイラになってみよう
（上級者向け）

下に示したコードをコンパイラになったつもりで見て、それぞれコンパイル可能かどうかを答えてください。この章で学んだことだけで即わかる、というものばかりではないかもしれませんが、その場合も、これまでに本書で学んだことを応用して、じっくり考えれば、おそらく正しい答えは得られるはずです。

（注意：どのコードも文法的に正しいクラス、メソッドに属しているものと考えてください）

コンパイル可能かを答えてください。

- ☐ `ArrayList<Dog> dogs1 = new ArrayList<Animal>();`
- ☐ `ArrayList<Animal> animals1 = new ArrayList<Dog>();`
- ☐ `List<Animal> list = new ArrayList<Animal>();`
- ☐ `ArrayList<Dog> dogs = new ArrayList<Dog>();`
- ☐ `ArrayList<Animal> animals = dogs;`
- ☐ `List<Dog> dogList = dogs;`
- ☐ `ArrayList<Object> objects = new ArrayList<Object>();`
- ☐ `List<Object> objList = objects;`
- ☐ `ArrayList<Object> objs = new ArrayList<Dog>();`

This PDF is prepared for tools@maruhouse.org, personal use only.

自分で考えてみよう

「リバースエンジニアリング」の答え

```
import java.util.*;
public class SortMountains {
  LinkedList<Mountain> mtn = new LinkedList<Mountain>();

  class NameCompare implements Comparator <Mountain> {
    public int compare(Mountain one, Mountain two) {
      return one.name.compareTo(two.name);
    }
  }
  class HeightCompare implements Comparator <Mountain> {
    public int compare(Mountain one, Mountain two) {
      return (two.height - one.height);
    }
  }
  public static void main(String [] args) {
    new SortMountains().go();
  }
  public void go() {
    mtn.add(new Mountain("Longs", 14255));
    mtn.add(new Mountain("Elbert", 14433));
    mtn.add(new Mountain("Maroon", 14156));
    mtn.add(new Mountain("Castle", 14265));

    System.out.println("as entered:\n" + mtn);
    NameCompare nc = new NameCompare();
    Collections.sort(mtn, nc);
    System.out.println("by name:\n" + mtn);
    HeightCompare hc = new HeightCompare();
    Collections.sort(mtn, hc);
    System.out.println("by height:\n" + mtn);
  }
}
class Mountain {
  String name;
  int height;

  Mountain(String n, int h) {
    name = n;
    height = h;
  }
  public String toString( ) {
    return name + " " + height;
}
```

高さを基準にしたソートは降順に

出力：

```
File  Edit  Window  Help  ThisOne'sForBob
%java SortMountains
as entered:
[Longs 14255, Elbert 14433, Maroon 14156, Castle 14265]
by name:
[Castle 14265, Elbert 14433, Longs 14255, Maroon 14156]
by height:
[Elbert 14433, Castle 14265, Longs 14255, Maroon 14156]
```


This PDF is prepared for tools@maruhouse.org, personal use only.

エクササイズの答え

答えは以下から選ぶ：

Comparator
Comparable
compareTo()
compare()
yes
no

1 ～ 3 の質問には、下のようなコード（これはコンパイル可能な正しいコードです）が存在するという前提で答えてください。

```
Collections.sort(myArrayList);
```

1. myArrayList に保持されるオブジェクトのクラスがインプリメントしていなくてはならないインターフェースは？ `Comparable`

2. myArrayList に保持されるオブジェクトのクラスがインプリメントしていなくてはならないメソッドは？ `compareTo()`

3. myArrayList に保持されるオブジェクトのクラスが同時に Comparator と Comparable をインプリメントすることはできるか。 `yes`

4 ～ 9 の質問には、下のようなコード（これはコンパイル可能な正しいコードです）が存在するという前提で答えてください。

```
Collections.sort(myArrayList, myCompare);
```

4. myArrayList に保持されるオブジェクトのクラスが Comparable をインプリメントすることはできるか。 `yes`

5. myArrayList に保持されるオブジェクトのクラスが Comparator をインプリメントすることはできるか。 `yes`

6. myArrayList に保持されるオブジェクトのクラスは Comparable をインプリメントしなくてはならないか。 `no`

7. myArrayList に保持されるオブジェクトのクラスは Comparator をインプリメントしなくてはならないか。 `no`

8. myCompare オブジェクトのクラスがインプリメントしなくてはならないインターフェースは？ `Comparator`

9. myCompare オブジェクトのクラスがインプリメントしなくてはならないメソッドは？ `compare()`

This PDF is prepared for tools@maruhouse.org, personal use only.

「コンパイラになってみよう」の答え

コンパイル可能かを答えてください。

- [ ] `ArrayList<Dog> dogs1 = new ArrayList<Animal>();`
- [ ] `ArrayList<Animal> animals1 = new ArrayList<Dog>();`
- [x] `List<Animal> list = new ArrayList<Animal>();`
- [ ] `ArrayList<Dog> dogs = new ArrayList<Dog>();`
- [ ] `ArrayList<Animal> animals = dogs;`
- [x] `List<Dog> dogList = dogs;`
- [x] `ArrayList<Object> objects = new ArrayList<Object>();`
- [x] `List<Object> objList = objects;`
- [ ] `ArrayList<Object> objs = new ArrayList<Dog>();`

This PDF is prepared for tools@maruhouse.org, personal use only.

This PDF is prepared for tools@maruhouse.org, personal use only.

17章　プログラムの提供

プログラムが完成したら……

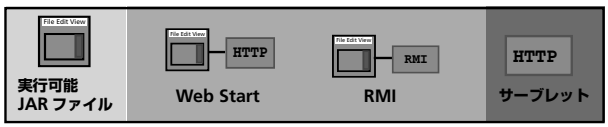
何度も修正をし、さまざまな苦労を重ねて（途中、「もう、やめたい」などと言ったりしながら）素晴らしいプログラムを完成させても、人に使ってもらえなければ意味がありません。しかし、どうすれば使ってもらえるのでしょうか。ユーザーにコードを配るにしても、誰がユーザになってくれるのかさえわからない時もあります。17章と18章では、作ったJavaプログラムを人に使ってもらう方法について解説していきます。プログラムの「パッケージング」などについて触れる他、プログラムの提供方法の種類（ローカル、リモート、その中間、など）などにも触れていきます。この章（17章）では多くのページをパッケージングの説明に割いています。パッケージングは、プログラムの提供の仕方を問わず必要になる作業だからです。次の18章で学ぶのは、Javaプログラミングの中でも特に高度なテクニックですが、じっくり読めば「わからない」ということはないので安心してください。注意して欲しいのは、プログラマの仕事は、プログラムを作ってユーザーに提供したらそれで終わり、というわけではないということです。その後は「保守」という大切な仕事が待っているからです。

This PDF is prepared for tools@maruhouse.org, personal use only.

Javaプログラムの提供

Javaプログラムが完成したら、それをユーザーに使ってもらうことになるのですが、ユーザーにプログラムを使ってもらうためには、何をどのように提供すればよいのでしょうか。第一に問題なのは、ユーザーはプログラムの開発者と同じシステムを持っているわけではない、ということです。そして当然のことながら、開発者が何らかのかたちで提供しない限り、ユーザーはプログラムのコードを手に入れることはできません。この章と次の章では、どうすればプログラムをユーザーに使ってもらえるか、ということに触れます。まず、この章では、すべてのコードをユーザーのシステム上に置いてプログラムを利用してもらう、という方法と、一部のコードはユーザーのシステム上に置くが、部分的にはネットワーク経由でコードを利用してもらう、という方法について解説します。前者の形態でプログラムを利用するユーザーを「ローカルユーザー」、後者の形態で利用するユーザーを「コンビネーションユーザー」と呼びます。コードをユーザーのシステム上に置いてもらうためには、「実行可能JARファイル」という形式にまとめる必要があります。また、コンビネーションユーザー向けのプログラムコードの提供には「Java Web Start」という技術を使用します。コードを主に(あるいは完全に)ネットワーク経由で利用する「リモートユーザー」への対応については次章で触れます。これにはRMI、サーブレットといった技術を使用します。

プログラムの提供方法の種類



① ローカル
ローカルユーザーは、実行可能JARファイル(JARについての詳細は後述)によってプログラムコードの提供を受け、すべてのコードを自らのシステム上で動かす。

② ローカル、リモートの中間(コンビネーション)
ユーザーは、プログラムのクライアント部分のコードを提供され、それを自らのシステム上で動かす。クライアントコードは、プログラムのサーバ部分とネットワーク経由で接続して使うことになる。

③ リモート
Javaプログラムのすべてのコードがサーバシステム上で動作する。通常、ユーザーは、このプログラムを自らのシステム上のWebブラウザで利用することになる。

プログラムは、ただ単純にクラスファイルを渡せばユーザーに使ってもらえるというものではありません。実際のプログラムの提供には、それ以外の作業が必要になるのですが、この章ではまず、それが必要な理由を説明します。

Javaプログラムはクラスファイルの集合であり、開発作業の成果物としては当然クラスファイルが得られる「クラスファイルがすべて完成した後、何をすべきか」がこの章のテーマ

頭の体操

すべてのコードをユーザーのシステム上に置いてプログラムを利用してもらう(ユーザーにローカルユーザーとしてプログラムを利用してもらう)、という方法にはどのようなメリット、デメリットがあるでしょうか。

また、コードをWebブラウザなどによりネットワーク経由で利用してもらう(プログラムをサーブレットとして提供する)、という方法にはどのようなメリット、デメリットがあるでしょうか。

This PDF is prepared for tools@maruhouse.org, personal use only.

プログラムができた。その後は……

Bobはついに素晴らしいJavaプログラムを完成させました。「あと一息」というところで意外に手間取り、大変でしたが、とうとうすべての開発作業が終わったのです。

Bobの作ったプログラムはGUIも備えた高度なもので、いくつものSwingコンポーネントを使っており、プログラムを構成するクラスの数も9つに上っていました。

プログラムができたからにはユーザーの手に渡さねばなりません。Bobは自ら作った9つのクラスのクラスファイルをユーザーのコンピュータにコピーすればよい、と単純に考えていました。プログラムの中ではAPIも利用していますが、ユーザーのコンピュータにも自分のコンピュータと同様にAPIがインストールされているものと思っていたからです。そこで、彼はまず、プログラムの開発作業に使用したファイルをすべて保存するのに使用していたワーキングディレクトリの内容を確かめることにしました。

Bobは、「ソースファイルが9つと、コンパイルによってできたクラスファイルが9つで、ディレクトリには合計18個のファイルがあるはず……」と思っていたのですが、実際には31個ものファイルがディレクトリ上に存在したのです。しかも、その多くには次のような奇妙な名前が付けられていました。

Account$FileListener.class
Chart$SaveListener.class

Bobはコンパイル時には内部クラス（プログラムにはGUIコンポーネントのためのイベントリスナークラスが内部クラスとして含まれていた）のクラスファイルも作られるのだ、ということをすっかり忘れていました。奇妙なファイルの正体は、内部クラスのクラスファイルだったのです。

ユーザーにプログラムを提供するために、Bobは必要なクラスファイルを注意深くディレクトリから抽出しなければなりませんでした。ユーザーに渡したクラスファイルに1つでも抜けがあれば、プログラムは動かないからです。また、Bobはユーザーにソースファイルを渡したくなかったので、その点についても注意が必要でした。すべてのファイルを1つのディレクトリにまとめたのは失敗だったと言えるでしょう。

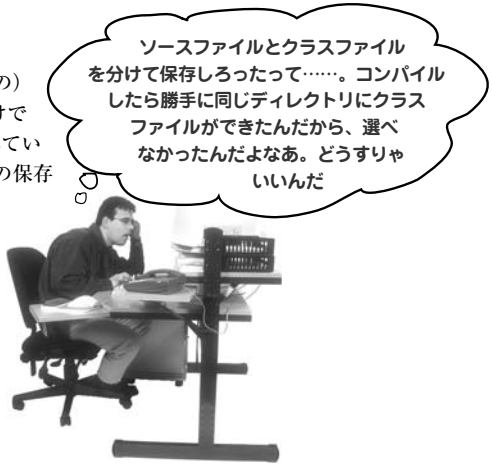
This PDF is prepared for tools@maruhouse.org, personal use only.

ソースファイルとクラスファイルを分けて保存する

前のページで述べたとおり、ソースファイルとクラスファイルを同じディレクトリに保存するというのは得策とは言えません。Bobははじめからソースファイルとクラスファイルを別のディレクトリに分けて保存すべきだったのです。これは、ソースファイルのコンパイル時に、クラスファイルがソースファイルとは別のディレクトリにできるようにすべき、ということを意味します。

それには、コンパイラの「-d」というオプションを使用します。

ソースファイル、クラスファイルをどのように、どんな（あるいは、いくつの）ディレクトリに振り分けるか、ということについては一定の決まりがあるわけではありません。自分でそれを決めることもあれば、働いている会社で決められているということもあるでしょう。本書では、ごく一般的、と思われるファイルの保存方法を紹介しておきます。



一般に、何らかのプログラムを開発する際は、まず「開発プロジェクトのためのディレクトリ」を1つ作ります。そして、その中に、ソースファイルのためのディレクトリ（“source”という名前にする）とクラスファイルのためのディレクトリ（“classes”という名前にする）を作ります。プログラムの作成は、sourceディレクトリに、.javaという拡張子の付いたソースファイルを作るところから始めます。そのソースファイルをコンパイルする際には、.classという拡張子の付いたクラスファイルの作成先としてclassesディレクトリを（-dオプションを使って）指定します。

-d オプションを使ったコンパイル

```
% cd MyProject/source
% javac -d ../classes MyApp.java
```

これにより、クラスファイルの作成先をclassesディレクトリにすることができる。「..」は「1つ上の階層に上がる」という意味

ここにはやはりコンパイル対象となるソースファイルの名前を指定する

-dオプションを指定すると、クラスファイルの作成先となるディレクトリが指定できます。このオプションを指定しなければ、作成先はデフォルトでカレントディレクトリ（この場合はsourceディレクトリ）になります。すべてのソースファイルを一度にコンパイルし、クラスファイルをすべて同じディレクトリに作成するには、以下のようなコマンドを使用します。

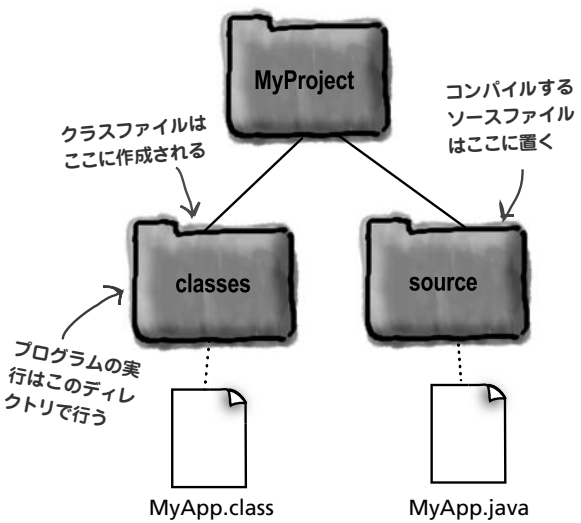
```
% javac -d ../classes *.java
```

「*.java」と書けば「カレントディレクトリのすべてのソースファイル」という意味になる

プログラムの実行

```
% cd MyProject/classes
% java MyApp
```

プログラムは、クラスファイルのディレクトリに移って実行する



注意：この章は、カレントディレクトリ（「.」で表される）がCLASSPATHに指定されているという前提で書かれています。読み進める前にCLASSPATHに「.」が指定されているかどうかを確認しておいてください）

This PDF is prepared for tools@maruhouse.org, personal use only.

プログラムに必要なJARファイルをファイルにまとめる

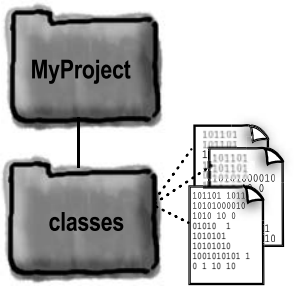
jar（Java ARchive）は、Javaプログラムの実行に必要なクラスファイルなどを1つにまとめるためのツールです。jarによって作るJARファイルのフォーマットは基本的にpkzipのファイルと同じです。多数のクラスファイルからなるプログラムでも、jarを使えば1つにまとめてユーザーに提供できるので非常に効率的です。jarツールのコマンドの使い方は、UNIXのtarとほぼ同じなので、UNIXのユーザーには馴染みやすいでしょう（本書では、ツールのjarは小文字、JARファイルは大文字で表記します）。

JARファイルを受け取ったユーザーはどうすればプログラムを実行できるのでしょうか。

実は、JARファイルは実行可能ファイルにすることができます。

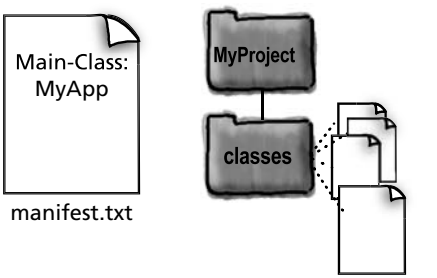
実行可能なJARファイル（実行可能JARファイル）を受け取ったユーザーは、JARファイルからクラスファイルを取り出さずにプログラムを実行できます。それを可能にするには、JARファイルに納める「マニフェストファイル（JARファイルの内容についての情報を書き込む特殊なファイル）」に、「main()メソッドはどのクラスにあるか」という情報を書き込む必要があります。

実行可能JARファイルの作成



① クラスファイルをすべてclassesディレクトリにまとめる。

この点に関しては後で詳しく説明しますが、今のところはともかくclassesディレクトリにまとめる、と覚えてください。

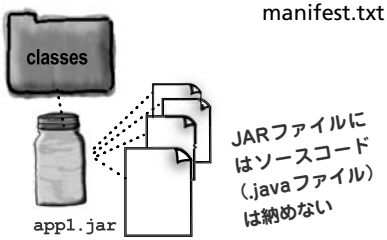


② manifest.txtというファイルを作り、main()メソッドを持つクラスの名前を書き込む。

具体的にはファイルの中に、以下のように書き込む。

```
Main-Class: MyApp
```
← .classは付けない

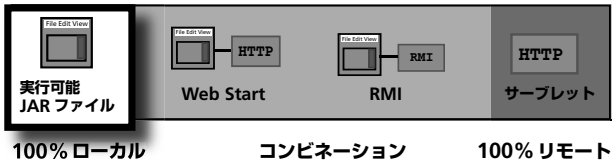
「MyApp」まで入力ができたら、必ずリターンキーを押してください。行末に改行文字を入れなければマニフェストファイルは正しく機能しません。マニフェストファイルもclassesディレクトリに配置します。



③ jarツールを起動してclassesディレクトリ中のファイル（マニフェストファイルを含む）をすべて納めたJARファイルを作成する。

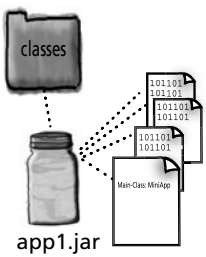
```
% cd MiniProject/classes
% jar -cvmf manifest.txt app1.jar *.class
```
（あるいは `% jar -cvmf manifest.txt app1.jar MyApp.class`）

This PDF is prepared for tools@maruhouse.org, personal use only.



JARファイルの実行

JVMは、JARファイルから、マニフェストファイルに指定されたクラスのデータをロードし、そのmain()メソッドを呼び出してプログラムを実行することができます。プログラムの実行のために、JARファイルからファイルを取り出す必要はありません。main()メソッドを呼び出した後、JVMは、JARファイルを含め、アクセスでき得るあらゆる箇所から必要なクラスデータを探し出してロードし、プログラムの実行を続けていきます(JVMにとって、必要なデータは見付かりさえすれば、どこに存在しても違いはない)。JVMがJARファイルにアクセスするためには、CLASSPATHにJARファイルの置かれているディレクトリが指定されている必要があります。




```
% cd MyProject/classes

%java -jar app1.jar
```

JARファイルはJVMのアクセスできる場所、つまりCLASSPATHに指定されたディレクトリに置く必要がある。最も簡単なのは、カレントディレクトリにJARファイルを置くこと

-jarオプションを付けると、後にクラスファイル名ではなく、JARファイル名を指定できる

JVMはJARファイル中のマニフェストファイルでmain()メソッドがどのクラスにあるかを確認する。main()メソッドについての情報が得られなければ、例外(非チェック例外)が発生する

OSの環境によっては、JARファイルのアイコンをダブルクリックするだけでプログラムを起動できる場合もあります。OSがWindowsやMac OS Xであれば、通常はそれが可能でしょう。また、JARファイルのアイコンを選んで「開く」メニュー(メニューの名前は環境によって異なることがありますが……)を選択する、という方法でプログラムを起動する場合もあります。

素朴な疑問に答えます

Q：JARファイルに、クラスファイルの納められたディレクトリをディレクトリごと入れるというのは不可能なのですか。

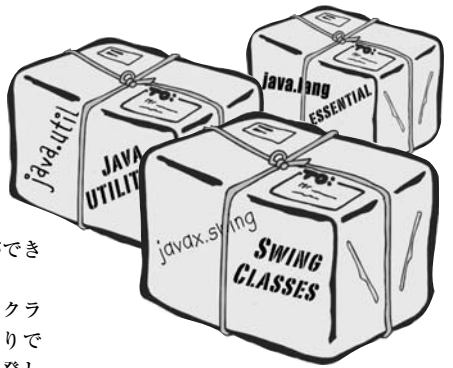
A：JVMは、JARファイルの中に、クラスファイルがファイルとして納められるという前提で検索をします。ディレクトリが納められていても、その中を見るということはしないのです。ただし、何らかのパッケージに含まれているクラスに関してはその限りではありません。JVMは、packageステートメントに記述されているパッケージに対応するディレクトリは検索するからです。

Q：すみません。よくわからないので、もう少し詳しく説明してください。

A：つまり、適当なディレクトリにクラスファイルを入れておいて、そのディレクトリをJARファイルに納めても、JVMは必要なクラスを見つけられないということです。しかし、packageステートメントに記述されているパッケージに対応するディレクトリ(あるいはディレクトリの階層構造)であれば、JVMは検索できるのです。これでもわからないかもしれませんが、次のページで詳しく説明するので安心してください。

This PDF is prepared for tools@maruhouse.org, personal use only.

クラスをパッケージにまとめる



Java APIに属するパッケージの例

java.text.NumberFormat
java.util.ArrayList
java.awt.FlowLayout
java.awt.event.ActionEvent
java.net.Socket

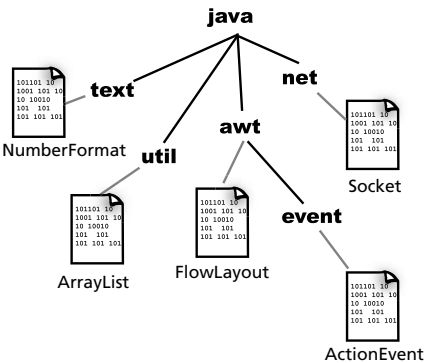
せっかく他の人が利用しやすいようにクラスを作ったとしても、ただ作っただけでは誰も使ってくれません。そこで、たとえば、会社の中で、「新しいクラスができたら必ずここに入れる」というような社内独自のライブラリを作り、その利用を社員全員に周知徹底した、とします。これで一見問題なさそうですが、この方法だと、「2つ以上のクラスの名前が重複する」という大変な問題が発生する恐れがあるのです。これでは、利用したいクラスがあっても、必要なクラスを明確に指定することができません。

この問題は、「パッケージ」を利用することで解決できます。Java APIのクラスがパッケージにまとめられている、ということはすでに見てきたとおりです。しかし、実は、Java APIのクラスだけでなく、プログラマが独自に開発したクラスも何らかのパッケージに納めるべきなのです。

クラスをパッケージにまとめる、ということは、各パッケージに対応するディレクトリを作り、そこにクラスファイルを保存するということでもあります。そして、JARファイルには、パッケージの階層構造に対応するディレクトリの階層構造をそのまま入れるのです。パッケージの階層構造とディレクトリの階層構造との間に少しでも違いがあると、プログラムのコンパイル、実行が正しく行えないので、細心の注意が必要です。

名前の重複を防ぐパッケージ

クラスをパッケージにまとめる目的はそれだけではありませんが、名前の重複を防ぐ、というのが重要な役割であることは間違いありません。たとえば、Customer、Account、ShoppingCartといった名前は、eコマースの開発者のおそらく半数には使われているはずですが、こうしたクラス名をパッケージングを行わずに使うのは危険です。オブジェクト指向言語であるJavaのクラスは、誰がどのようなクラスと組み合わせて使用するかわからないからです。無数にいるプログラマがどのような組み合わせでクラスを使用したとしても、決して名前の重複があってはなりません。見知らぬ他人が書いたクラス、将来書かれる可能性があるクラスとでさえ名前が重複しないようにする必要があるのです。



クラスを保存するディレクトリの階層構造は、パッケージの階層構造と一致させる

6章で述べたとおり、何らかのパッケージに属するクラスには、クラスそのものの名前以外に、属するパッケージの名前も含めた「フルネーム（完全名）」が付くことになります。たとえば、java.utilパッケージに属するArrayListクラスのフルネームは、「java.util.ArrayList」ということになります。また、javax.swingパッケージに属するJButtonクラスのフルネームは「javax.swing.JButton」、java.netパッケージに属するSocketクラスのフルネームは「java.net.Socket」です。ArrayListクラス、SocketクラスのフルネームはいずれもJava「java」で始まっていますが、これは、ArrayListクラス、Socketクラスの属するパッケージが、上の階層では共通のパッケージに属しているということを意味します。クラスファイルを保存するディレクトリの階層構造は、パッケージの階層構造と同じになるようにします。

This PDF is prepared for tools@maruhouse.org, personal use only.

パッケージングは名前の重複の防止に役立つが、パッケージ名自体が重複しないよう注意が必要。パッケージ名の重複を防ぐには、ドメイン名を逆にして使うという方法が一般的

パッケージ名の重複を避ける

クラスをパッケージにまとめれば、名前が重複する可能性はかなり減ります。しかし、パッケージ名が重複してしまっては何にもなりません。たとえば、名前が重複する可能性の高いAccountというクラスをshopping.customersというパッケージに所属させ、フルネームを以下のようにしたところで、重複の危険はあまり減るとは考えられないのです。

shopping.customers.Account

では、どうすればいいのでしょうか。Sunでは、ドメイン名を逆にしたものをパッケージ名として使用するよう推奨しています。確かに、ドメイン名は決して重複しないはずのものなので、この方法は有効と言えるでしょう。

```
com.headfirstbooks.Book
```

パッケージ名

クラス名

ドメイン名を逆にしてパッケージ名に使う

```
com.headfirstjava.projects.Chart
```

自分の属する組織、企業のドメイン名(この例ではheadfirstjava.com)の順序を逆にしてパッケージ名とする。その後に所属部署名などを続けることもある

「projects.Chart」という名前はごくありふれたものだが、「com.headfirstjava」という名前を付ければ、組織内での名前の重複だけに注意すればよくなる

クラス名の最初は大文字

This PDF is prepared for tools@maruhouse.org, personal use only.

クラスをパッケージに所属させる

クラスのソースファイル、クラスファイルを保存するディレクトリの階層構造は、パッケージの階層構造に一致させる

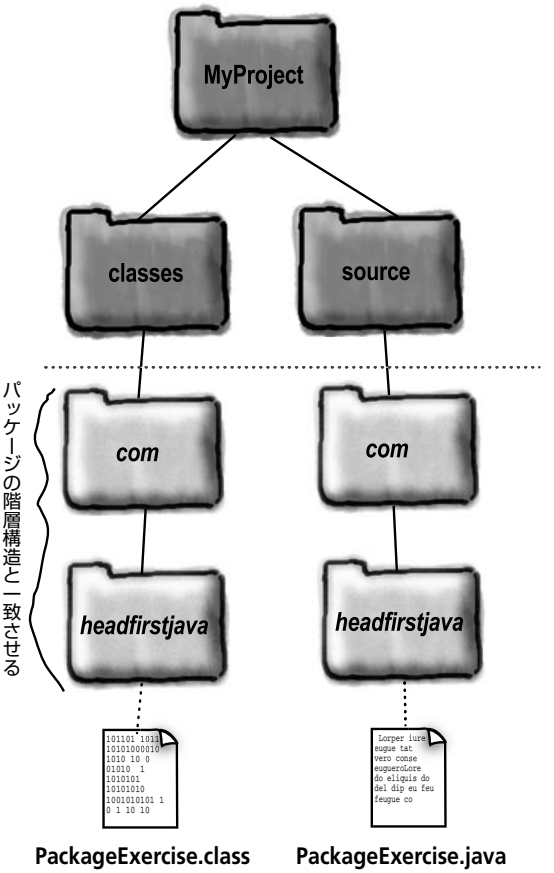
① **所属させるパッケージの名前を決める。**
ここでは、パッケージ名として「com.headfirstjava」を使用することにします。このパッケージに所属させるクラスの名前は「PackageExercise」です。つまり、クラスの完全名は以下のようになります。

```
com.headfirstjava.PackageExercise
```

② **packageステートメントをクラスのソースコードに入れる。**
packageステートメントは、クラスのソースコードの冒頭に入れます（特に、importステートメントが前にあってはならない）。1 つのソースコードの中にpackageステートメントを 2 つ以上入れることはできません。つまり、同じソースコードに記述されているクラスはすべて（当然、内部クラスも）同じパッケージに属することになります。

```
package com.headfirstjava;

import javax.swing.*;

public class PackageExercise {
  // クラスのコード
}
```

③ **パッケージの階層構造に合わせてディレクトリを作る。**
2 では、packageステートメントでクラスの所属するパッケージを指定しましたが、それだけでは十分ではありません。パッケージの階層構造に合わせ、ディレクトリ（の階層構造）も作らねばならないのです。「com.headfirstjava.PackageExercise」という完全名のクラスを作るには、sourceディレクトリの下と、classesディレクトリの下に、com、headfirstjavaというディレクトリを作る必要があります。PackageExerciseクラスのソースファイル、クラスファイルは、headfirstjavaディレクトリの中に保存するのです。
この作業をもし怠ったとしても、プログラムのコンパイルはできますが、その後に大変な苦労を強いられることになるので、やはり正しくディレクトリを作っておいた方がよいでしょう。



classes、sourceディレクトリの下に、パッケージの階層構造に対応するディレクトリの階層構造を作る

This PDF is prepared for tools@maruhouse.org, personal use only.

パッケージに属するクラスのコンパイル、実行

パッケージに属するクラスをコンパイル、実行する際には、若干、注意が必要です。まず、重要なのは、コンパイラとJVMが、使用するソースファイル、クラスファイルをすべて「見つける」ことができなければならない、ということです。クラスがコアライブラリ（コアAPI）に属するものであれば、その点はまったく問題になりません。コンパイラもJVMも、そうしたクラスに関しては必要なデータがどこにあるかを「知って」いるからです。しかし、プログラマが独自に作ったクラスの場合はそうはいきません。特に、コンパイルや実行の際に、ソースファイル、クラスファイルの保存されているディレクトリに移動すれば良いというように勘違いする人が多いので注意してください（これで偶然うまくいくこともあるが、確実ではない）。問題なくコンパイル、実行ができ、しかも手間が少なくて済む、という方法を以下に紹介しておきます。他に方法がないわけではないのですが、以下の方法が最も無難でしょう。

コンパイル（-dオプションを使用する）

```
% cd MyProject/source
```

コンパイルはsourceディレクトリで行う。.javaファイルの保存されているディレクトリに移動してはいけない

```
% javac -d ../classes com/headfirstjava/PackageExercise.java
```

classesディレクトリにクラスファイル（を納めたディレクトリの階層構造）ができるよう、このように指定する。

コンパイル対象のソースファイルのパスを指定する

com.headfirstjavaパッケージの.javaファイルをすべてコンパイルする場合は、コマンドを以下のようにします。

```
% javac -d ../classes com/headfirstjava/*.java
```

ディレクトリ中のすべての.javaファイルをコンパイル対象とするには、このように指定する

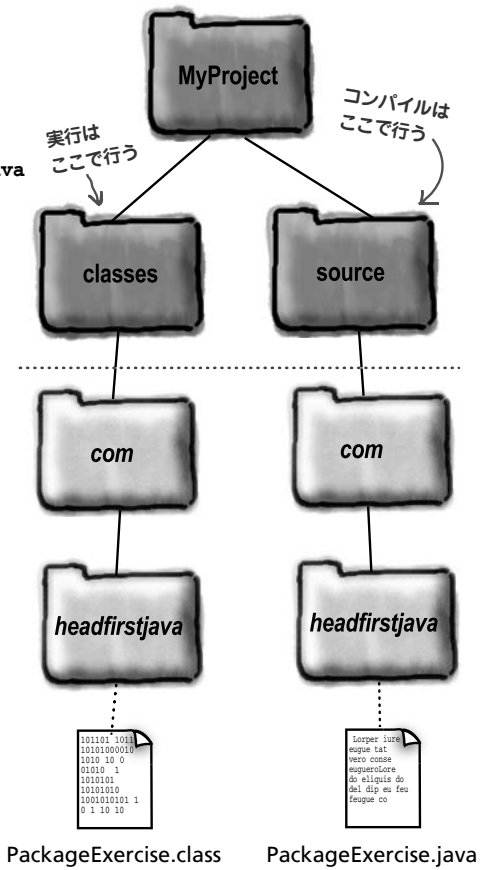
実行

```
% cd MyProject/classes
```

実行はclassesディレクトリで行う

```
% java com.headfirstjava.PackageExercise
```

クラスの完全名を指定する。JVMはまず、カレントディレクトリでcomディレクトリを探し、次にcomディレクトリでheadfirstjavaディレクトリを探す。そして次に、headfirstjavaディレクトリで実行すべきクラスファイルを探す。クラスファイルがcomディレクトリやclassesディレクトリに保存されていると、実行ができなくなる

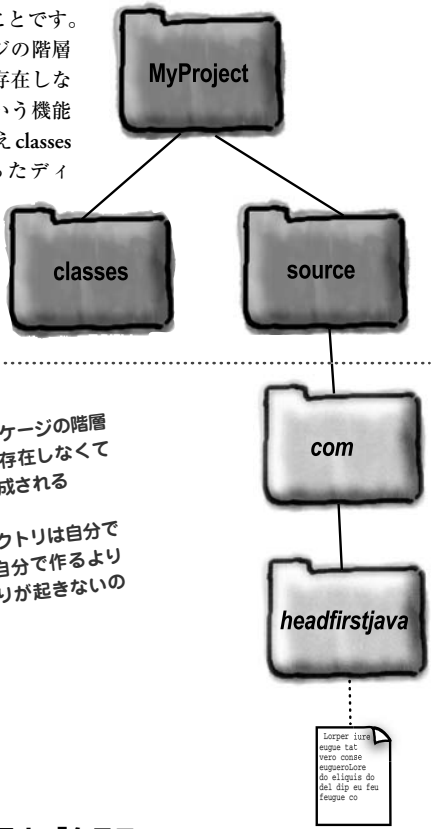


This PDF is prepared for tools@maruhouse.org, personal use only.

-dオプションの優れた機能

-dが、コンパイルによってできるクラスファイルの保存先を指定するためのオプションであることはすでに述べました。しかし、-dはただそれだけのオプションではないのです。まず、重要なのは、パッケージの階層構造を正しく認識して、「classesディレクトリの下のどのディレクトリにクラスファイルを保存すればよいか」を自動的に判断する機能を備えていることです。つまり、「classesの下のcomの下のheadfirstjava」というように細かく指定する必要がないということです。

また、その上、もしパッケージの階層構造に対応するディレクトリが存在しなければ、自動的に作成する、という機能も備えています。つまり、たとえclassesの下にcom、headfirstjavaといったディレクトリが存在しなくても、-dオプションを指定すれば自動的に作られるということです。



-dオプションを指定すると、「クラスファイルを、パッケージの階層構造から見て適切なディレクトリに作成せよ」とコンパイラに指示することができる。パッケージの階層構造に対応するディレクトリが存在しなければ自動的に作成される

素朴な疑問に答えます

Q：クラスファイルの置かれているディレクトリにcdで移動して実行しようとしたのですが、「クラスファイルが見付からない」というエラーになります。確かにそのディレクトリにクラスファイルは存在しているのですが……。なぜ、このようなことが起きるのですか。

A：クラスがパッケージに属している場合は、クラス名だけでなく、パッケージ名も含む完全名を指定する必要があるのです。また、ディレクトリの階層構造が、パッケージの階層構造と一致していることも必要になります。以下のようなコマンドを実行した場合、

```
% java com.foo.Book
```

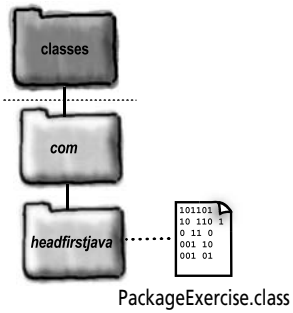
JVMはまず、カレントディレクトリ（あるいはCLASSPATHに指定されたディレクトリ）で、「com」というディレクトリを探します。「com」が見付かったら、その中で「foo」というディレクトリを探します。「com」と「foo」が見付かるまでは、「Book」というクラスのクラスファイルを探すことはないのです。「com」の中の「foo」の中にあるBookのクラスファイルのみを、正当なBookのクラスファイルであると認めるわけです。たとえ他の場所にBookのクラスファイルがあっとしても、それは正当なものであるとはみなされません。また、JVMは、カレントディレクトリより上の階層で「com」や「foo」を探すことはしません。

This PDF is prepared for tools@maruhouse.org, personal use only.

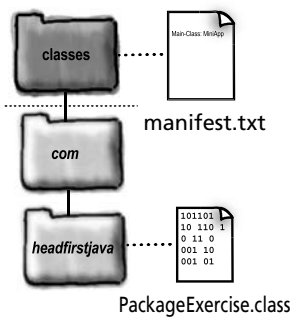
パッケージに属するクラスを実行可能JARファイルにまとめる

パッケージに属するクラスをJARファイルにまとめる場合は、クラスファイルそのものだけでなく、クラスファイルの保存されているディレクトリの階層構造ごとJARファイルに入れる必要があります。また、その際、パッケージに対応するディレクトリより上の階層のディレクトリをJARファイルに入れないよう注意しなければなりません。最上位のパッケージに対応するディレクトリが、JARファイル中での最上位ディレクトリにならなければならないのです。それより上の階層のディレクトリ（classesディレクトリなど）を誤って入れてしまうと、JARファイルは正しく機能しません。

実行可能JARファイルの作成



① **classesディレクトリの下に、パッケージの階層構造に対応するディレクトリの階層構造を作り、クラスファイルを適切なディレクトリに保存する。**

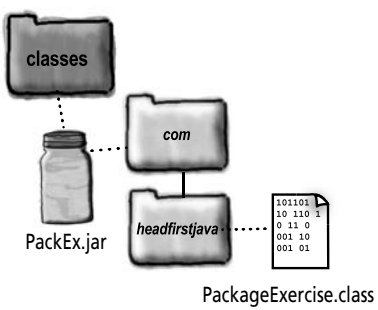


② **manifest.txtファイル（マニフェストファイル）を作成し、どのクラスにmain()メソッドがあるかを記述する。クラス名は必ず「完全名」にする。**

manifest.txtというテキストファイルを作り、以下の記述をする（途中で改行しないよう注意）。

```
Main-Class: com.headfirstjava.PackageExercise
```

このマニフェストファイルはclassesディレクトリに入れる。



③ **以下のようなコマンドでjarツールを起動して、上記のディレクトリ構造とマニフェストファイルを含むJARファイルを作成する。**

comディレクトリだけを引数に指定すれば、パッケージ全体（必要なクラスすべて）がJARファイルに入る。

```
% cd MyProject/classes
% jar -cvmf manifest.txt packEx.jar com
```

comディレクトリだけを指定すれば、ディレクトリ構造全体がJARファイルに納められる

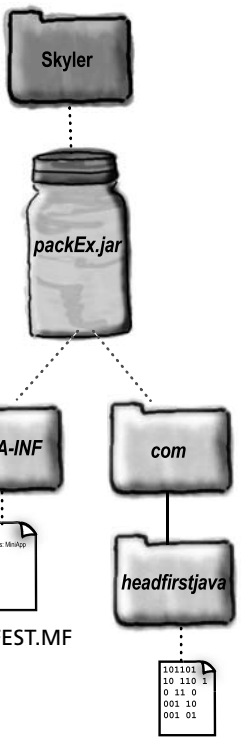
This PDF is prepared for tools@maruhouse.org, personal use only.

JARファイルの内容確認、内容の抽出

jarツールには、JARファイルの内容を確認する、JARファイルに納められているファイルを抽出する（圧縮ファイルを解凍するのとほぼ同じ）、といったことのためのオプションも用意されています。

packEx.jarというJARファイルがSkylerというディレクトリにおかれているとすると、内容確認や内容の抽出のために以下のような操作をすることになります。

JARファイルをSkylerディレクトリに置く



jarツールによって、自動的にMETA-INFディレクトリが作られ、そこにマニフェストファイルが置かれる

① JARの内容を確認する

```
% jar -tf packEx.jar
```

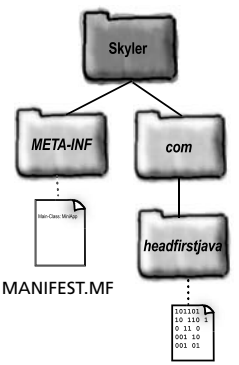
-tfオプションは、JARファイルの内容一覧を表示するオプション

```
File  Edit  Window  Help  Pickle
% cd Skyler
% jar -tf packEx.jar
META-INF/
META-INF/MANIFEST.MF
com/
com/headfirstjava/
com/headfirstjava/
PackageExercise.class
```

② JARファイルの内容を抽出する

```
% cd Skyler
% jar -xf packEx.jar
```

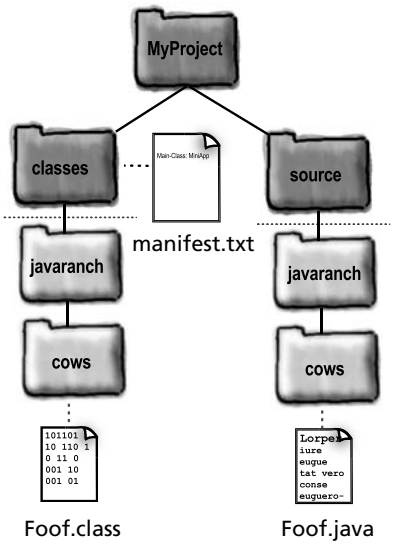
-xfは、圧縮ファイルの解凍のようなことをするためのオプション。packEx.jarの内容を-xfオプションを使って抽出すると、カレントディレクトリにMETA-INFディレクトリとcomディレクトリができる



jarツールはMETA-INFディレクトリとMANIFEST.MFファイルを作成する。このファイルには、元のマニフェストファイルと同じ情報が納められる。元のテキストファイルそのものではなく、テキストファイルの情報だけが利用される点に注意

This PDF is prepared for tools@maruhouse.org, personal use only.

自分で考えてみよう



左の図のようなディレクトリ構造（パッケージの階層構造を反映したもの）がある時、コンパイル、プログラムの実行、JARファイルの作成、JARファイルの実行を適切に行うにはどのようなコマンドを使用すればよいでしょうか。以下の空欄の部分を埋めてください。

コンパイル

```
% cd source
% javac _____
```

実行

```
% cd _____
% java _____
```

JARファイルの作成

```
% cd _____
% _____
```

JARファイルの実行

```
% cd _____
% _____
```

上の例では、パッケージ名の付け方に1つ問題がありますが、それは何でしょうか。

This PDF is prepared for tools@maruhouse.org, personal use only.

素朴な疑問に答えます

Q：実行可能JARファイル中のJavaプログラムを、JVMのないコンピュータ上で実行しようとするとどうなるのですか。

A：プログラムはまったく動きません。JARファイルがあってもJVMがなければJavaプログラムは実行できないのです。Javaプログラムを実行するには、必ずJVMをインストールしなくてはなりません。

Q：JARファイルを配布するユーザーのコンピュータに必ずJVMがインストールされているとは限らないわけですが、すべてのユーザーに確実にJVMをインストールしてもらうにはどうすればよいですか。

A：最もよいのは、インストーラを作り、プログラムとともに配布することです。それには、いくつもの企業から提供されているインストーラプログラムが利用できます。こうしたインストーラプログラムにはごく簡単なものから極めて高度な機能を備えたものまでさまざまな種類があります。中には、ユーザーのコンピュータ上に適切なバージョンのJVMが存在するかどうかを調べ、存在しなければインストールと環境設定を行う、といったことのできるものまであります。インストーラプログラムの例としては、Installshield、InstallAnywhere、DeployDirectorなどがあげられます。
インストーラプログラムを使えば、Javaの実行環境を配布するためのCD-ROMを作れる場合もあります。それぞれにインストールできるJVMのバージョンが異なる何種類ものインストーラを入れておけば、このCD-ROMだけで大多数のユーザーに対応することも可能でしょう。Solarisを利用しているユーザーはSolarisバージョン、Windowsを利用しているユーザーはWindowsバージョンをインストールするためのインストーラを使えばいいのです。ユーザーの利便性を考えれば、予算と時間が許す限り、この方法を採るべきでしょう。

重要ポイント

- ソースファイルとクラスファイルは同じディレクトリに置かない。
- 一般に、何らかのプログラムを開発する際は、まず「開発プロジェクトのためのディレクトリ」を1つ作る。そして、その中に、ソースファイルのためのディレクトリとクラスファイルのためのディレクトリを作る。
- クラスをパッケージにまとめると、クラス名の重複を防ぐのに役立つ。パッケージ名には、ドメイン名を逆にしたものを使うのが有効。
- クラスをパッケージに所属させるにはまず、ソースコードの冒頭に以下のようなpackageステートメントを入れる（特にimportステートメントより後にならないよう注意）。

```
package com.wickedlysmart;
```

- クラスをパッケージに所属させる場合、クラスのソースファイル、クラスファイルを保存するディレクトリの階層構造は、パッケージの階層構造に一致させる。たとえば、クラスの完全名が「com.wickedlysmart.Foo」である場合には、comディレクトリを作り、その中にwickedlysmartディレクトリを作り、その中にFooクラスのソースファイル、クラスファイルを納めなくてはならない。
- コンパイル時に-dオプションを使用すれば、クラスファイルがパッケージの階層構造からみて適切なディレクトリに保存される。

```
% cd source
% javac -d ../classes com/wickedlysmart/Foo.java
```

- クラスの実行の際には、classesディレクトリに移動し、完全名を指定する。

```
% cd classes
% java com.wickedlysmart.Foo
```

- クラスファイルはJARファイルにまとめることができる。JARファイルのフォーマットは基本的にpkzipのファイルと同じ。
- 実行可能JARファイルを作る際には、「どのクラスにmain()メソッドがあるか」を記述したマニフェストファイルを入れる。マニフェストファイルはテキストファイルで、中の記述は以下のようなものにする。

```
Main-Class: com.wickedlysmart.Foo
```

- マニフェストファイルの記述の行末ではリターンキーを押して改行文字を入れる。これを怠るとマニフェストファイルが正しく機能しない。
- JARファイルは以下のようなコマンドで作成する。

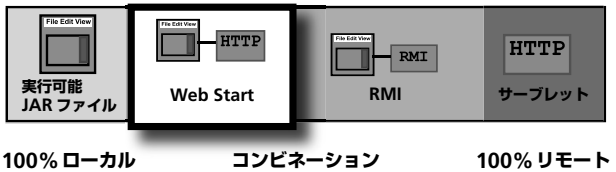
```
jar -cvfm manifest.txt MyJar.jar com
```

- パッケージに属するクラスをJARファイルにまとめる場合は、クラスファイルそのものだけでなく、クラスファイルの保存されているディレクトリの階層構造ごとJARファイルに入れる必要がある。その際、パッケージに対応するディレクトリより上の階層のディレクトリをJARファイルに入れないよう注意する。
- JARファイルに納めたプログラムを実行する際には以下のようなコマンドを使う。

```
java -jar MyJar.jar
```


This PDF is prepared for tools@maruhouse.org, personal use only.

This PDF is prepared for tools@maruhouse.org, personal use only.



Java Web Start

Java Web Start(JWS)を使用する場合、プログラムははじめ、ユーザのコンピュータ上のWebブラウザを使って起動することになります(起動時には、そのためのWebページ上のリンクをクリックする)。起動後、プログラムは、Webブラウザからは独立し、(ほぼ)スタンドアロンのプログラムとして動作します。初回実行時にプログラムはユーザのコンピュータにダウンロードされます。

Java Web Startは、クライアントのコンピュータ上に置かれ、Adobe Acrobat Readerなどと同様、ブラウザのプラグインとして機能するプログラム(JWS自身もプログラム)です。JWS(JWSヘルパーアプリケーションとも呼ばれる)は、Javaプログラムのダウンロード、アップデート、起動などの役割を担います。

JWSによってプログラムがダウンロードされる(実行可能JARファイルのかたちでダウンロードされる)際には、プログラムのmain()メソッドがJWSによって実行されます。2回目以降の実行には、ネットワーク接続、Webページ上のリンクのクリックは必要ありません。

その他、JWSには、サーバ上のプログラムに少しでも変更が加えられれば、それを検知する、という機能があります。その変更は、クライアント上のプログラムに自動的に反映されます。ユーザはそれについて関知する必要はありません。

ただ、問題は、Javaの実行環境とJWSをどのようにユーザに配布するか、ということです。JWSはダウンロード、プログラムの起動などを行うもので、プログラムの実行には、それとは別に実行環境(JVM)が必要なのです(しかも、JWS自身もJavaプログラム)。プログラムを実行しようとするユーザのコンピュータにJWSがインストールされているかどうかを確認し、インストールされていなければ、Sunのサイトからダウンロードするよう指示する、というようなことが自動的に行われるようにしておくべきでしょう。そして、たとえJWSがインストールされていても、Javaの実行環境が古い、というような場合には、Java 2 Standard Editionの最新版がインストールできるようにしておく必要があります。

このように問題はありますが、Webページを閲覧するのと同じような感覚で簡単にJavaプログラムを実行できるというのはやはり便利です。Webページ上にプログラムへのリンクを設けておけば、ユーザにプログラムを使ってもらえるというのも大きなメリットです。

注意すべきなのは、JWSを使った場合でも、プログラムはJARファイルのかたちでユーザに渡されるということです。この点はよく覚えておいてください。

JWSを使用する場合、ユーザははじめ、Webページ上のリンクをクリックしてプログラムを起動する。起動後、プログラムは、Webブラウザからは独立し、スタンドアロンのプログラムとして動作する。JWSを使用したプログラムも、実行可能JARファイルのかたちでユーザに配布される(ただし配布はWebを使って行われる)点に注意

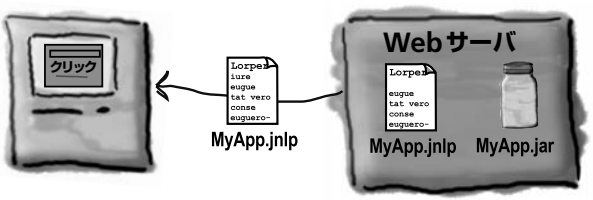
This PDF is prepared for tools@maruhouse.org, personal use only.

JWS(Java Web Start)を利用したJavaプログラムの実行

① **ユーザがWebページ上のリンク(Javaプログラムを実行するためのリンク)をクリックする。このリンクは、厳密には、「.jnlp」という拡張子の付いたファイルへのリンク。**

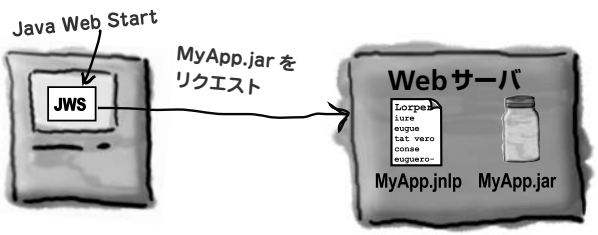
リンクを作るためのHTMLコードは以下のようになる。

```
<a href="MyApp.jnlp">Click</a>
```

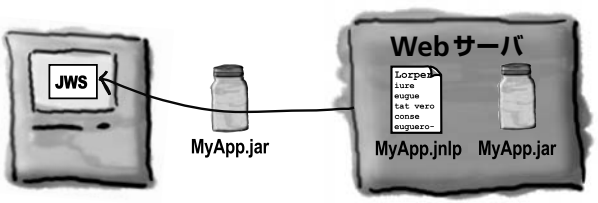


② **Web(HTTP)サーバが、ブラウザのリクエストに応え、.jnlpファイル(JARファイルではない)を送る。**

.jnlpファイルはXMLファイルの一種。プログラムの実行可能JARファイルの名前が記述されている。



③ **JWS(ブラウザのプラグイン)がブラウザによって起動される。JWSは.jnlpファイルの内容を読み取り、プログラムのJARファイルをサーバにリクエストする。**



④ **Webサーバが、リクエストされたJARファイルを提供する。**

⑤ **JWSはJARファイルを受け取ると、(通常のJARファイルの場合と同じように)指定のmain()メソッドを呼び出して、プログラムを起動する。**

このプログラムを次に実行する際には、ネットワーク接続を行う必要はない。JWSだけでプログラムを起動できる。

This PDF is prepared for tools@maruhouse.org, personal use only.

.jnlpファイル

JWSをプログラムの起動に利用する場合には、「.jnlp（Java Network Launch Protocol）ファイル」と呼ばれるファイルが必要です。.jnlpファイルはプログラムの実行可能JARファイルの名前が記述されたXMLファイルで、JWSは、このファイルを利用してプログラムのJARファイルを探します（.jnlpファイルには、JARファイルの名前以外にも、さまざまなことがらが記述できる）。.jnlpファイルの内容はたとえば以下のようになります。

```
<?xml  version="1.0" encoding="utf-8"?>
<jnlp  spec="0.2 1.0"
       codebase="http://127.0.0.1/~kathy"
       href="MyApp.jnlp">

  <information>
       <title>kathy App</title>
       <vendor>Wickedly Smart</vendor>
       <homepage href="index.html"/>
       <description>Head First WebStart demo</description>
       <icon href="kathys.gif"/>
       <offline-allowed/>
  </information>

  <resources>
       <j2se version="1.3+"/>
       <jar href="MyApp.jnlp"/>
  </resources>

  <application-desc main-class="HelloWebStart"/>
</jnlp>
```

codebaseタグには、「.jnlpファイルをどのサーバのどのディレクトリに置くか」を指定する。ここでは、サーバのアドレスとして「127.0.0.1」という「ローカルループバックアドレス」を指定しているので、ローカルホストに.jnlpファイルが置かれることになる。もし、.jnlpファイルを著者のWebサーバに置くとすれば、"http://www.wickedlysmart.com" という指定をすることになる

ここには、.jnlpファイルの名前、相対パス（codebaseに指定された保存場所を基準とする）を指定する。ここでは、"MyApp.jnlp" という指定をしているので、.jnlpファイルを、ローカルホスト上のルートディレクトリ（最上位のディレクトリ）に置くことになる

この例のとおりに.jnlpファイルを作る場合は、タグの抜けがないかをよく確認すること。1つでも抜けがあると、正しく機能しない恐れがある。informationタグは、主に、ユーザがダウンロード済みのプログラムが起動しようとした際、それに対応する画面を表示するのに使用される

このタグによって、ダウンロード済みプログラムが、インターネット接続を行わずに実行できるようになる。ただ、インターネット接続を行わなければ、プログラムの自動アップデートは行われない

このプログラムにはJavaのバージョン1.3以降が必要、という意味

実行可能JARファイルの名前を指定する。JARファイルは複数作る場合もある。また、JARファイルに、クラスファイルだけでなく、サウンドファイルや画像ファイルを納める場合もある

マニフェストファイルと同じようにmain()メソッドを持つクラスの名前を指定する

This PDF is prepared for tools@maruhouse.org, personal use only.

JWSを利用するプログラムを提供する手順

① **実行可能JARファイルを作成する。**

MyApp.jar

② **.jnlpファイルを作成する。**

MyApp.jnlp

③ **JARファイルと.jnlpファイルをWebサーバ上に配置する。**

④ **Webサーバで、.jnlpファイルに対応するMIMEタイプの設定を行う。.jnlpファイルのMIMEタイプは以下のとおり（MIMEタイプの設定の仕方はWebサーバごとに異なる）。**

```
application/x-java-jnlp-file
```

MIMEタイプを設定すると、ブラウザに送られる.jnlpファイルに適切なヘッダが付けられる。これによってブラウザは、そのファイルが.jnlpファイルであること、JWSを利用してプログラムを起動するのに利用できることを認識できる。

⑤ **.jnlpファイルへのリンクを備えたWebページを作成する。**

MyJWSApp.html

```
<HTML>
  <BODY>
    <a href="MyApp2.jnlp">Launch My Application</a>
  </BODY>
</HTML>
```


This PDF is prepared for tools@maruhouse.org, personal use only.

エクササイズ

ニワトリが先かタマゴが先か……

下に示したカードには、それぞれ、JWSを利用してJavaプログラムを実行する際に行われることが書かれています。正しい順序に並べ替えてください。

1.

2.

3.

4.

5.

6.

7.

素朴な疑問に答えます

Q：Java Web Startを使ったプログラムとアプレットとの違いは何ですか。

A：アプレットはあくまで「Webページの一部」としてサーバからダウンロードされるもので、Webブラウザから独立して実行されるわけではありません。アプレットは、ブラウザにとって、いわばJPEG画像などと同じようなものなのです。アプレットの実行には、Javaプラグインや、ブラウザにビルトインされたJava実行環境（最近ではあまり利用されなくなった）が使用されます。JWSを使用するプログラムと異なり、アプレットは必ずブラウザから起動しなければなりません。また、「自動アップデート」という点で、JWSプログラムはアプレットより優れていると言えるでしょう。JWSプログラムはいったんWebサーバからダウンロードされれば、それ以降ブラウザを使わず、ネットワークにも接続することなく起動できるようになります。

Q：JWSプログラムにセキュリティ上の制約はないのですか。

A：JWSプログラムには、ユーザのコンピュータ上のハードディスクの読み書きなどに関していくつかの制約があります。ただ、ダイアログボックスを表示して、ユーザに許可を求め、許可が得られた場合にのみ、ハードディスク上でファイルの読み書きを行う、ということはできます（そのためのAPIが用意されている）。その場合も、他から隔離された特別の領域のみを利用します。

重要ポイント

- Java Web Start（JWS）を利用すれば、スタンドアロンのプログラムをWebを使って配布できる。
- JWSは、ヘルパーアプリケーション（プラグイン）であり、ユーザーのコンピュータ上にJavaの実行環境とともにインストールする必要がある。
- JWSプログラムには、実行可能JARファイル、.jnlpファイルという2種類のファイルが必要。
- .jnlpファイルは、プログラムに関する情報を納めたXMLファイル。JARファイルの名前、保存場所や、main()メソッドを持つクラスの名前といった情報が提供される。
- ユーザーがWebページ上のリンクをクリックすると、ブラウザは、サーバから.jnlpファイルを受け取り、JWSを起動する。
- JWSは.jnlpファイルの内容を読み取り、WebサーバにJARファイルをリクエストする。
- JWSは、サーバからJARファイルを受け取ると、プログラムのmain()メソッドを呼び出す（どのクラスにmain()メソッドがあるかは.jnlpファイルに指定される）。

This PDF is prepared for tools@maruhouse.org, personal use only.

エクササイズ

○か×か

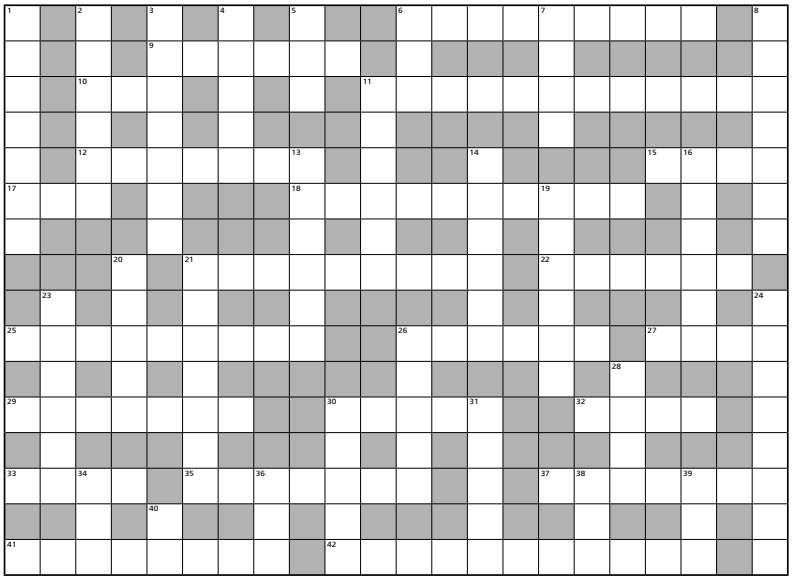
次の文は、いずれもこの章で学んだことがらについて述べたものです。正しいものに○を、誤っているものに×をつけてください。

1. Javaコンパイラの-dオプションは、コンパイルによってできるクラスファイルの作成先の指定に使用する。
2. .classファイルは、通常、「JAR」という名前のディレクトリに保存する。
3. jarツールによって作成するファイルの拡張子は「.mf」にする。
4. マニフェストファイルには、「どのクラスにmain()メソッドがあるか」という情報が納められる。
5. JARファイルに納めたプログラムを実行するには、ファイルを解凍しなくてはならない。
6. JARファイルのプログラムを実行するには、-archオプションを使用する。
7. クラスをパッケージに所属させる場合は、クラスを保存するディレクトリの階層構造を、パッケージの階層構造と一致させる必要がある。
8. パッケージの名前にドメイン名を利用するというのはあまり良い方法とは言えない。
9. 1つのソースファイルの中に複数のクラスがある時は、個々のクラスを別々のパッケージに属させるということができる。
10. パッケージに属するクラスをコンパイルする際には、できる限り-pオプションを使用すべき。
11. パッケージに属するクラスをコンパイルする場合は、sourceディレクトリに移動し、コンパイル対象のソースファイルのパスを指定する。
12. -dオプションをうまく使うと、クラスファイルを保存するディレクトリの階層構造を誤りなく作るのに役立つ。
13. JARファイルを解凍すると、META-INFというディレクトリができる。
14. JARファイルを解凍すると、MANIFEST.MFというファイルができる。
15. JWSは常にブラウザとともに動作する。
16. JWSプログラムは、.nlp（Network Launch Protocol）ファイルを必要とする。
17. JWSによって起動されるプログラムのmain()メソッドがどのクラスに属するかは、JARファイルの中で指定される。

This PDF is prepared for tools@maruhouse.org, personal use only.

Java に関する用語のクロスワードです（注：このクロスワードには英語で、すべて大文字で答えてください）。

Java クロスワード



ヨコのカギ

6. シリアライズの対象から外す
9. synchronized 宣言
10. プログラムの配布
11. ロック
12. ストリームを組み合わせる
15. シリアライズ
17. インスタンス変数の値の取得
18. ____JAR ファイル
21. インターフェースを____する
22. チェーンストリームは____ストリームとも呼ばれる
25. フォルダ
26. ネットワーク経由
27. プログラムの____
29. クラスをまとめる
30. プログラムの構成単位
32. 繰り返し
33. 8 ビット
35. 元に戻す
37. XP = ____ Programming
41. main() メソッドはどこ？
42. データの保護

タテのカギ

1. ____ボックス
2. インスタンス
3. 不要のオブジェクト
4. String____izer
5. 2 つの引数のうち小さい方を戻す
6. catch
7. 並べ替え
8. コンポーネント
11. I/O
13. プログラム開発後の作業
14. クラスに 1 つ
16. 利用する
19. 一時的に蓄える
20. ループを途中で抜ける
21. ラッパ
23. 2 進数
24. プリミティブ型の値を格納するオブジェクト
26. インスタンス変数
28. ソフトウェアを区別する
30. ストリームを使い終わったら____する
31. スレッドを一時的に停止する
34. タンジェントを求めるメソッド
36. インスタンス変数の値の設定
38. .jnlp ファイルは____ファイル
39. 実行形式ファイル
40. 分岐

ヒント

原子のように分割できない = atomic
実行可能 = executable
再生 = restore
カプセル化 = encapsulate
プログラムの提供 = deploy

This PDF is prepared for tools@maruhouse.org, personal use only.

エクササイズの答え

ニワトリが先か タマゴが先か……

1. ユーザがWebページ上のリンクをクリックする
2. ブラウザがWebサーバに.jnlpファイルをリクエストする
3. Webサーバが.jnlpファイルをブラウザに送る
4. WebブラウザがJWSを起動する
5. JWSがJARファイルをリクエストする
6. WebサーバがJARファイルをJWSに送る
7. JWSがJARファイルのmain()メソッドを呼び出す

○か×か

1. Javaコンパイラの-dオプションは、コンパイルによってできるクラスファイルの作成先の指定に使用する。 ○
2. .classファイルは、通常、「JAR」という名前のディレクトリに保存する。 ×
3. jarツールによって作成するファイルの拡張子は「.mf」にする。 ×
4. マニフェストファイルには、「どのクラスにmain()メソッドがあるか」という情報が納められる。 ○
5. JARファイルに納めたプログラムを実行するには、ファイルを解凍しなくてはならない。 ×
6. JARファイルのプログラムを実行するには、-archオプションを使用する。 ×
7. クラスをパッケージに所属させる場合は、クラスを保存するディレクトリの階層構造を、パッケージの階層構造と一致させる必要がある。 ○
8. パッケージの名前にドメイン名を利用するというのはあまり良い方法とは言えない。 ×
9. 1つのソースファイルの中に複数のクラスがある時は、個々のクラスを別々のパッケージに属させるということができる。 ×
10. パッケージに属するクラスをコンパイルする際には、できる限り-pオプションを使用すべき。 ×
11. パッケージに属するクラスをコンパイルする場合は、sourceディレクトリに移動し、コンパイル対象のソースファイルのパスを指定する。 ○
12. -dオプションをうまく使うと、クラスファイルを保存するディレクトリの階層構造を誤りなく作るのに役立つ。 ○
13. JARファイルを解凍すると、META-INFというディレクトリができる。 ○
14. JARファイルを解凍すると、MANIFEST.MFというファイルができる。 ○
15. JWSは常にブラウザとともに動作する。 ×
16. JWSプログラムは、.nlp(Network Launch Protocol)ファイルを必要とする。 ×
17. JWSによって起動されるプログラムのmain()メソッドがどのクラスに属するかは、JARファイルの中で指定される。 ×

This PDF is prepared for tools@maruhouse.org, personal use only.

Java クロスワード

1 D		2 O		3 G		4 T		5 M			6 T	R	A	N	7 S	I	E	N	T		8 W
I		B		9 A	T	O	M	I	C		R				O						I
A		10 J	A	R		K		N		11 S	Y	N	C	H	R	O	N	I	Z	E	D
L		E		B		E				T					T						G
O		12 C	H	A	I	N	E	13 D		R			14 S					15 S	16 A	V	E
17 G	E	T		G				18 E	X	E	C	U	T	A	19 B	L	E		C		T
S				E				P		A			A		U				C		S
			20 B		21 I	M	P	L	E	M	E	N	T		22 F	I	L	T	E	R	
	23 B		R		N			O					I		F				S		24 W
25 D	I	R	E	C	T	O	R	Y			26 S	O	C	K	E	T		27 U	S	E	R
	N		A		E						T				R		28 P				A
29 P	A	C	K	A	G	E			30 C	L	A	S	31 S			32 L	O	O	P		P
	R				E				L		T		L				R				P
33 B	Y	34 T	E		35 R	E	36 S	T	O	R	E		E		37 E	38 X	T	R	39 E	M	E
		A		40 I			E		S				E			M			X		R
41 M	A	N	I	F	E	S	T		42 E	N	C	A	P	S	U	L	A	T	E		S

This PDF is prepared for tools@maruhouse.org, personal use only.

This PDF is prepared for tools@maruhouse.org, personal use only.

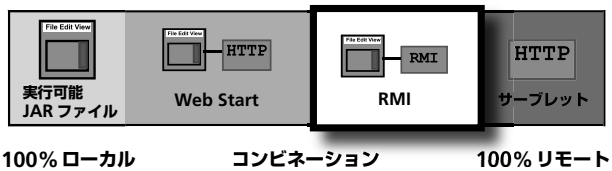
18章　RMIによるプログラムの提供

分散コンピューティング

遠距離恋愛は難しいって言うけど、RMIがあればそうでもないのよ。どんなに離れていても、一緒にいるのと変わらないんだもの

プログラムの構成要素は常に一か所にまとまっていなくてはならない、というわけではありません。もちろん、すべてが同じコンピュータ上の同じヒープにあり、使用するJVMも1つで済めば、これほど楽なことはありません。しかし、それがいつでも可能とは限らないのです。また、場合によってはそうすべきでないこともあります。たとえば、プログラムは大変な処理能力を必要とするものなのに、ユーザーが携帯用の小型デバイスしか持っていないとしたらどうでしょうか。プログラムを動かすのにデータベースからデータを取得する必要があるが、セキュリティ上の理由から、データベースアクセスをサーバ上のコードにのみ許可したい、ということもあるでしょう。大規模なeコマースバックエンドなどは、どうしてもトランザクション管理機能を備えたシステムで動作させねばならないということもあります。さまざまな理由から、一部はサーバマシン上で動作させ、残りはクライアントマシンで動作させるべきというプログラムもあり得ます。この章では、主に、こうしたことを可能にするRMI(Remote Method Invocation)という技術について解説します。その他、サーブレットや、RMIを基礎とするEJB(Enterprise Java Beans)、Jiniといった技術にも簡単に触れます。章の最後では、RMIの技術を応用した「ユニバーサルサービスブラウザ」という特殊なプログラムの作り方を紹介します。

This PDF is prepared for tools@maruhouse.org, personal use only.

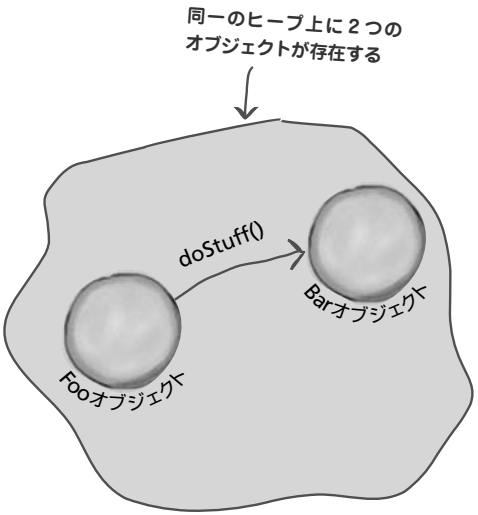


メソッドの呼び出しは通常、同一のヒープ上のオブジェクト間で行われる

本書でここまでに取りあげたプログラム例では、メソッド呼び出しはすべて、同一のJVMが動かすオブジェクト間で行われていました。これは、メソッドを呼び出す側のオブジェクトと呼び出される側のオブジェクトが同じヒープ上に存在するということです。

```
class Foo {
  void go() {
    Bar b = new Bar();
    b.doStuff();
  }
  public static void main (String [] args) {
    Foo f = new Foo();
    f.go();
  }
}
```

上記のコード例の場合、変数fに参照が格納されたFooオブジェクトと、変数bに参照が格納されたBarオブジェクトは同一のヒープ上に置かれ、同じJVMによって動作します。先述のとおり、参照型の変数に実際に格納される値（アドレス）がどのような数値になるのかは、JVMが決めます。JVMは、自らが管理下に置いているすべてのオブジェクトのヒープ上での位置を把握しており、いつでもすべてのオブジェクトにアクセスが可能です。しかし、JVMが把握できるのは、自らのヒープ上にあるオブジェクトについてだけです。つまり、たとえば、あるマシン上で動作しているJVMが、他のマシンのJVMのヒープ（上のオブジェクト）について知ることは通常できないということです。それどころか、JVMは、同じマシン上で動作する他のJVMのヒープ（上のオブジェクト）について知ることもできません。オブジェクトがたとえ物理的に同じマシンに存在していようと、管理するJVM（存在するヒープ）が異なれば、メソッドの呼び出しはできないのです。



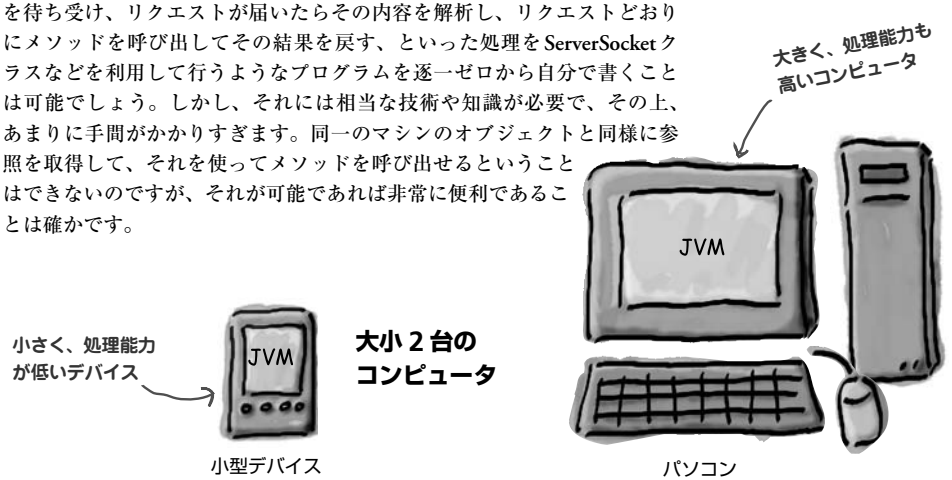
通常、メソッドの呼び出しが行われる場合、呼び出す側のオブジェクトと呼び出される側のオブジェクトは同じJVMが管理する同じヒープ上に存在する

This PDF is prepared for tools@maruhouse.org, personal use only.

他のマシン上のオブジェクトのメソッドを呼び出すには？

他のマシンと単にデータをやりとりするということは、すでに説明したとおり、ソケット接続、ストリームを利用すれば可能です。接続を確立し、ストリームを使ってデータを読み書きすればよいのです。

しかし、他のマシン上に存在し、他のJVMによって動作しているオブジェクトのメソッドを呼び出すということは、先述のとおり、基本的にはできません。もちろん、他のマシンからのメソッド呼び出しのリクエストを待ち受け、リクエストが届いたらその内容を解析し、リクエストどおりにメソッドを呼び出してその結果を戻す、といった処理をServerSocketクラスなどを利用して行うようなプログラムを逐一ゼロから自分で書くことは可能でしょう。しかし、それには相当な技術や知識が必要で、その上、あまりに手間がかかりすぎます。同一のマシンのオブジェクトと同様に参照を取得して、それを使ってメソッドを呼び出せるということはできないのですが、それが可能であれば非常に便利であることは確かです。



パソコンには小型デバイスにはない高い処理能力がある。

小型デバイスからデータを送り、
パソコンでそれを処理するという仕組みを作りたい。

小型デバイスで、パソコン上で実行される以下のような
メソッドを呼び出し、メソッドの処理結果だけを
受け取れるようにしたい。

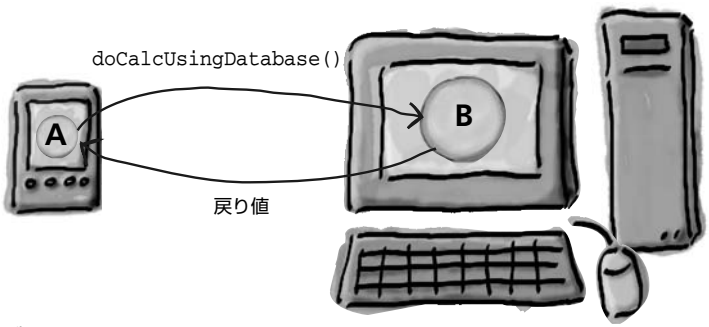
```
double doCalcUsingDatabase(CalcNumbers numbers)
```

パソコン上のオブジェクトへの参照を
小型デバイスで取得するということができれば便利。

This PDF is prepared for tools@maruhouse.org, personal use only.

小型デバイス上のオブジェクトAから パソコン上のオブジェクトBの メソッドを呼び出すには？

小型デバイス上のオブジェクトからパソコン上のオブジェクトへの参照を取得し、メソッドを呼び出せれば便利なのですが、それはできません。



RMIを利用すれば……

以下のようなコードを書いて、変数dにオブジェクトへの参照を格納する場合、そのオブジェクトは同一のヒープ上のオブジェクトでなければなりません。

```
Dog d = ???
```

他のヒープ（マシン）上のオブジェクトのメソッドを呼び出す、ということも絶対に不可能というわけではないのですが、それには、ソケット接続を利用した複雑な手続きが必要です。とはいえ、同一のマシンのオブジェクトを利用する場合とほぼ同じような簡単なコードを書くだけで、裏でそうした複雑な手続きが行われる、という仕組みを作ることはできるでしょう。つまり、同一マシン（ローカル）のオブジェクトのメソッドを呼び出しているのとほぼ変わらない操作で、他のマシン、あるいはヒープ（リモート）のオブジェクトのメソッドが呼び出せる（複雑な手続きは自動化してしまう）、という仕組みを作ればよいのです。

実は、Javaには、そのような「仕組み」があらかじめ用意されています。それが、RMI（Remote Method Invocation）です。RMIを利用すれば、ローカルオブジェクトのメソッドを呼び出すような感覚でリモートオブジェクトのメソッドが呼び出せます。プログラムを書いた本人がそれと意識しなくても、本来必要になるはずの複雑な手続きが裏で行われるのです。

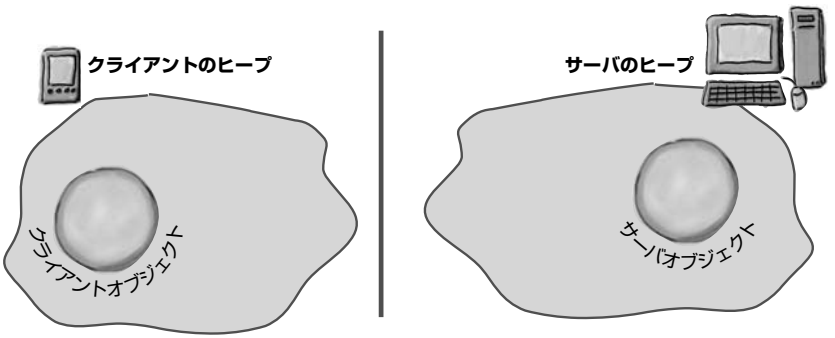
RMIについてはこの後詳しく説明しますが、その前に、RMIのような仕組みを自分で作るとしたらどうすればよいか、ということを考えてみましょう。それがわかれば、RMIについても理解しやすくなるはずです。

This PDF is prepared for tools@maruhouse.org, personal use only.

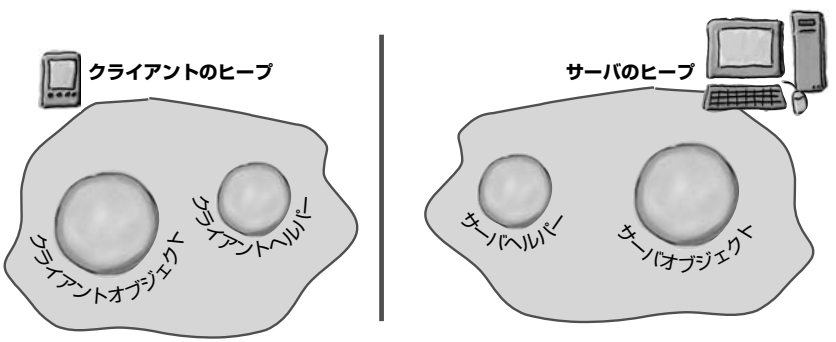
RMIを設計してみる

サーバ、クライアント、サーバヘルパー、クライアントヘルパーを作成する。

1. クライアントオブジェクト、サーバオブジェクトを作成する。サーバオブジェクトとは、メソッドを「呼び出される」側のオブジェクトのこと。



2. クライアントヘルパー、サーバヘルパーを作成する。いずれも、ネットワーク接続、I/Oに関連するこまごまとした処理を行うオブジェクト。この2つのオブジェクトのはたらきにより、クライアントオブジェクト、サーバオブジェクトは、まるで同一のヒープ上に存在するかのように動作できる。



This PDF is prepared for tools@maruhouse.org, personal use only.

ヘルパーの役割

クライアントオブジェクト、サーバオブジェクト間のやりとりを両者に代わって行うのがヘルパーの仕事です。ヘルパーのはたらきによって、クライアントオブジェクトは、サーバオブジェクトのメソッドを、ローカルオブジェクトと同じように呼び出せます。厳密には、クライアントオブジェクトが実際に呼び出すのはクライアントヘルパーのメソッドなのですが、クライアントオブジェクトからはヘルパーがサーバオブジェクトであるかのように見えるのです。クライアントヘルパーはサーバオブジェクトの代理を務めるのだと言ってもいいでしょう。

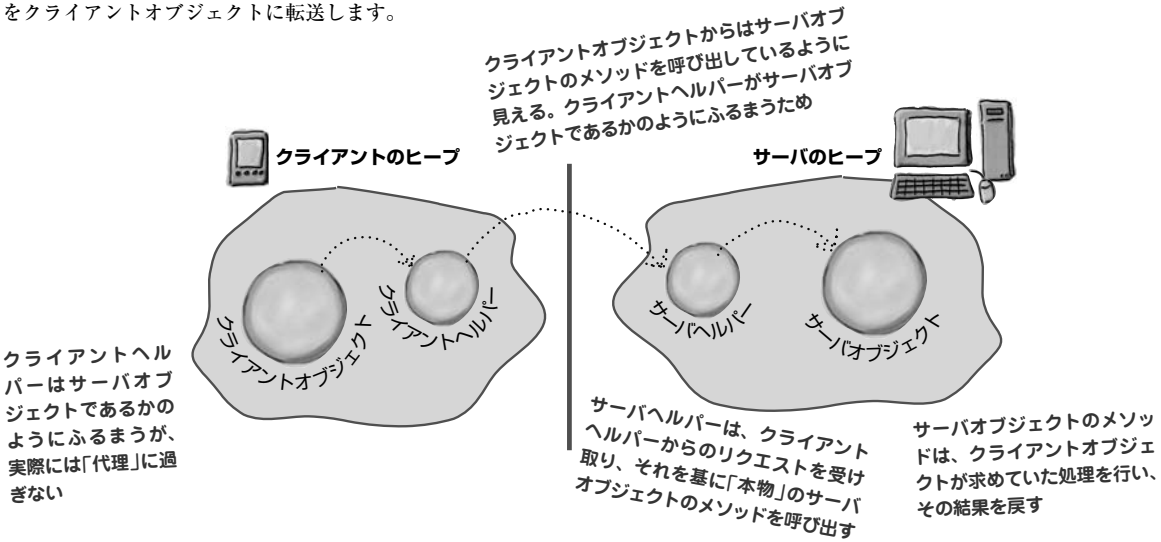
クライアントヘルパーはクライアントオブジェクトと同一のヒープ上にあるので、そのメソッドを呼び出すことには問題はありません。クライアントオブジェクトからは、そのメソッドが完全にサーバオブジェクトのものに見えるので、クライアントオブジェクトにとっては事実上、目的は達せられていることになります。

もちろん、クライアントヘルパーは実際にはサーバオブジェクトではありません。サーバオブジェクトと同じ名前のメソッドを持っていて、サーバオブジェクトの「ふり」をするだけです。そのメソッドには、サーバオブジェクトのものと同じ機能が備わっているわけではありません。そこで、クライアントヘルパーは、サーバオブジェクトに「メソッドを実行して欲しい」というリクエストを送り、呼び出されたメソッドの名前、指定された引数といった情報を伝えて、後はサーバオブジェクトからの返答を待つことになります。

サーバの側では、サーバヘルパーがクライアントヘルパーからのリクエストを（ソケット接続経由で）受け取り、リクエストとともに送られた情報を基に、「本物の」サーバオブジェクトのメソッドを呼び出します。サーバヘルパーにとって、サーバオブジェクトはローカルのオブジェクトなので、そのメソッドを呼び出すことには何の問題もありません。

サーバヘルパーはサーバオブジェクトからメソッドの戻り値を受け取ると、それをソケット接続経由でクライアントヘルパーに送ります。クライアントヘルパーはその戻り値をクライアントオブジェクトに転送します。

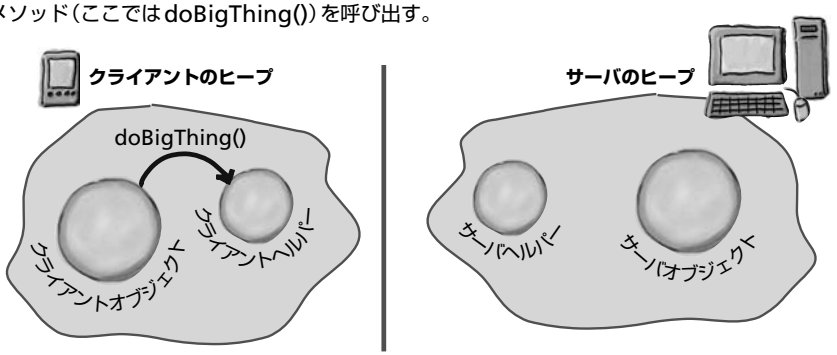
クライアントオブジェクトはサーバオブジェクトのメソッドを呼び出しているつもりだが、実際には同じヒープ上のクライアントヘルパーのメソッドを呼び出している。クライアントヘルパーは、ソケット接続、ストリームを使ったI/Oなどのこまごまとした処理を行う



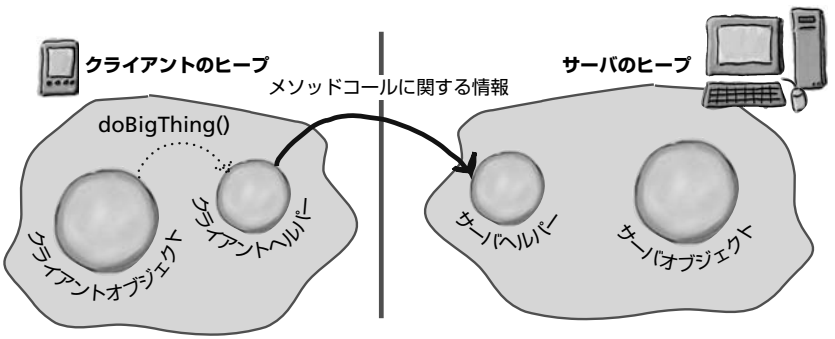
This PDF is prepared for tools@maruhouse.org, personal use only.

サーバメソッドを呼び出す

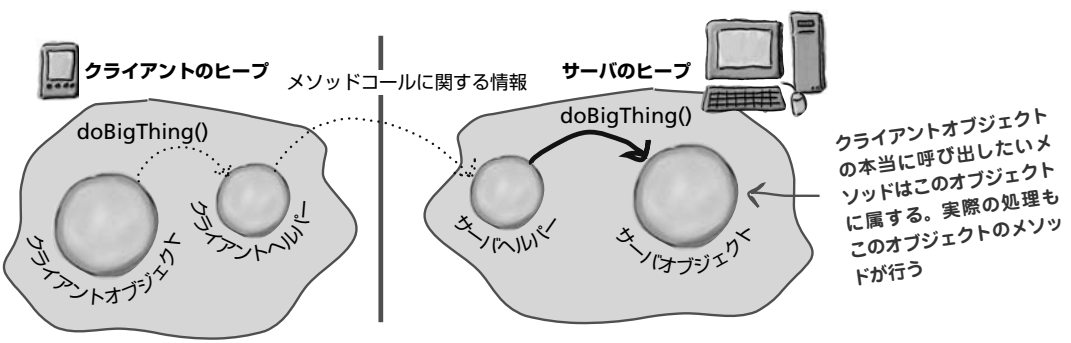
① クライアントオブジェクトが、クライアントヘルパーのメソッド（ここではdoBigThing()）を呼び出す。



② クライアントヘルパーが、メソッドコールに関する情報（メソッドの名前、引数など）をネットワーク経由でサーバヘルパーに送る。



③ サーバヘルパーはクライアントヘルパーからの情報を基にどのメソッドを呼び出させばよいか判断し、サーバオブジェクトの該当メソッドを呼び出す。



This PDF is prepared for tools@maruhouse.org, personal use only.

RMIのクライアントヘルパー、サーバヘルパー

RMI(Java RMIともいう)では、実際にクライアントヘルパー、サーバヘルパーを利用します。また、クライアントヘルパー、サーバヘルパーの作成は自動化されています。クライアントヘルパー、サーバヘルパーをどのように作ればよいかをプログラマが考える必要はないのです。クライアントヘルパーには、プログラマが何も考えなくても、必要なメソッド(クライアントオブジェクトが呼び出すサーバオブジェクトのメソッドと同じ名前のメソッド)が用意されます。

その他、RMIには、便利な機能がいくつも用意されていて、そうした機能を利用すれば、クライアントオブジェクトがプログラム実行時にクライアントヘルパーを探す、といったこともできます。RMIを利用すれば、ネットワーク、I/Oに関連するコードをプログラマが自分で書く必要はなくなります。クライアントオブジェクトからのリモートメソッド(サーバオブジェクトのメソッド)の呼び出しは、ローカルメソッド(同じヒープ上のオブジェクトのメソッド)の呼び出しとほぼ同じような方法で行えます。

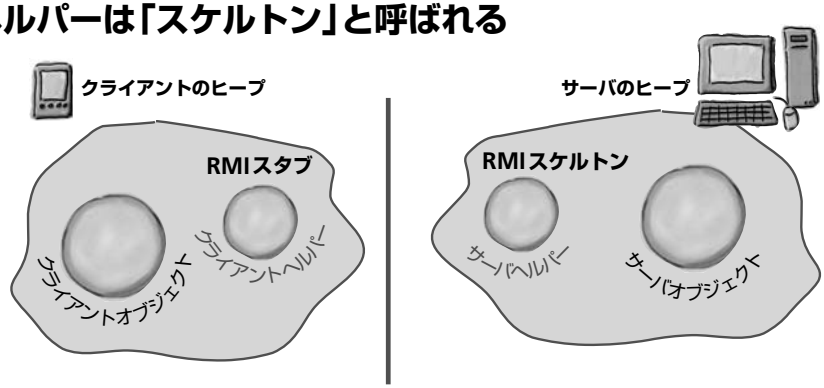
RMIを使ってリモートメソッドを呼び出す場合、クライアントオブジェクトが直接呼び出すのは実はクライアントヘルパーのメソッドです。しかし、メソッドコールに関する情報がクライアントヘルパーからネットワーク経由でサーバオブジェクトにまで転送され、最終的にはサーバオブジェクトのメソッドが呼び出されるので、見かけ上、直接サーバオブジェクトのメソッドを呼び出しているように見えるのです。裏では、ネットワーク、I/O関連の処理が行われていますが、それについては意識せずに済みます。

ただ、ネットワークやI/Oに関連する処理では例外が発生する恐れがあるという点には注意が必要です。リモートメソッドを呼び出すためのコードは、例外が発生するという前提で書かなければならないということです。直接呼び出すのがクライアントヘルパーの(ローカルの)メソッドであるとは言っても、その結果、ソケット接続やストリームを利用した処理が行われることになるからです。また、リモートメソッドを呼び出すというのは、別のJVMによって動作しているオブジェクトのメソッドを呼び出すということなので、JVMからJVMにメソッドコールに関する情報を転送するという必要が出てきます。その情報転送を具体的にどのような方法で行うか、というのはヘルパーオブジェクトの使用するプロトコルによって代わってきます。

RMIの場合、そのプロトコルとしてはJRMPかIIOPが選択できます。JRMPというのはRMI独自のプロトコルで、Javaプログラム間でのリモートメソッド呼び出しのために開発されたものです。一方のIIOPは、元々はCORBA(Common Object Request Broker Architecture)のためのプロトコルで、Java以外の言語のプログラムにも対応できます。一般に、CORBAを利用した場合の方がプログラミングに手間がかかります。やりとりをするプログラムの言語が異なれば、さまざまな「変換処理」が必要になるからです。

本書では、Javaプログラムどうしのやりとりに絞って話を進め、使用するプロトコルもJRMPに限定するため、紹介するコード例がさほど複雑になることはありません。

RMIでは、クライアントヘルパーは「スタブ」、サーバヘルパーは「スケルトン」と呼ばれる



This PDF is prepared for tools@maruhouse.org, personal use only.

RMIアプリケーション（サーバ部分）の作成手順

以下にRMIアプリケーションを作成するための手順を簡単にまとめておきます。個々の作業については後で詳しく説明します。

ステップ1：リモートインターフェースを作成する。

リモートインターフェースというのは、クライアントがリモート（RMIアプリケーションのクライアント部分）で呼び出せるメソッドを宣言したインターフェースのことです。このインターフェースは、「サーバオブジェクト（厳密にはスタブオブジェクト）への参照」を格納する変数の型として使用できます。スタブとサーバクラスは、このインターフェースをインプリメントします。

クライアントが呼び出すリモートメソッドを宣言するインターフェース

MyService.java

ステップ2：リモートインターフェースのインプリメンテーション（サーバクラス）を作成する。

クライアントが呼び出すリモートメソッドを含む「本物」のサーバクラスを作成します。このクラスでは、リモートインターフェースのメソッドをインプリメントします。クライアントが求める処理は最終的にはこのクラスのメソッドによってなされることになります。

「本物」のサーバクラス。クライアントの求める処理を実際に行うメソッドを含む。リモートインターフェースをインプリメントして作る

MyServiceImpl.java

ステップ3：スタブとスケルトンをrmicツールを使用して作成する。

スタブ（クライアントヘルパー）とスケルトン（サーバヘルパー）は、プログラマが自力で作成する必要はありません（それどころか、スタブ、スケルトンを構成するクラスのソースコードを見る必要さえ一切ありません）。いずれも、rmicというツールを利用することで自動的に作成されるからです。rmicツールは、Javaの開発環境にあらかじめ用意されています。

サーバクラスを引数に指定してrmicツールを実行する……

```
File  Edit  Window  Help  Eat
% rmic MyServiceImpl
```

すると、ヘルパーとなるクラスファイルが2つ作成される

MyServiceImpl_Stub.class

MyServiceImpl_Skel.class

ステップ4：RMIレジストリ（rmiregistry）を起動する。

RMIレジストリはいわば「電話帳」のようなものです。スタブ、スケルトンを検索するのに使用します。

```
File  Edit  Window  Help  Drink
% rmiregistry
```

RMIレジストリは専用のターミナルウィンドウ（コマンドプロンプト）で起動する

ステップ5：サーバオブジェクトを起動する。

サーバクラスのクラスファイルを実行すると、サーバクラスがインスタンス化され、RMIレジストリに登録されます。RMIレジストリに登録されたサーバオブジェクトは、クライアントに利用できるようになります。

```
File  Edit  Window  Help  BeMerry
% java MyServiceImpl
```


This PDF is prepared for tools@maruhouse.org, personal use only.

ステップ 1：リモートインターフェースを作成する

MyRemote.java

① java.rmi.Remoteを継承する。

リモートインターフェースを作成するには、java.rmi.Remoteを継承する必要があります。remoteは、「マーカーインターフェース」と呼ばれる、インプリメントすべきメソッドが定義されていない特殊なインターフェースです。たとえメソッドが定義されていなくても、このインターフェースを継承することは重要な意味を持ちます。インターフェースを継承して新たなインターフェースを作れる、という点に注意してください。

```
public interface MyRemote extends Remote {
```

Remoteインターフェースを継承すると、「このインターフェースはRMIに利用するものである」という意味になる。このようにインターフェースがインターフェースを継承することがあるという点に注意

② リモートインターフェースのメソッドを作る時は、RemoteExceptionという例外を発生する可能性があることを宣言する。

リモートインターフェースは、クライアントが直接メソッドを呼び出す「スタブ」によってインプリメントされるインターフェースです。スタブはネットワークやI/Oといった例外を発生させる恐れのある処理を行います。したがって、スタブを利用するクライアントのコードは、try/catchブロックを使うなどして、例外の発生に対応できるよう書く必要があります。リモートインターフェースのメソッドの宣言に、「throws RemoteException」という記述を入れておけば、そのメソッドのインプリメンテーションを利用する際に、必ず例外の発生を考慮したコードを書かねばならなくなります。

```
import java.rmi.*;

public interface MyRemote extends Remote {
  public String sayHello() throws RemoteException;
}
```

Remoteインターフェースはjava.rmiパッケージに属する

リモートメソッドを呼び出すと、常に例外発生の危険にさらされることになる。メソッドの定義でRemoteExceptionの発生を宣言しておけば、メソッドを利用するクライアントでは、それに対応したコードを書かざるを得なくなる

③ リモートインターフェースのメソッドの引数、戻り値は、プリミティブ型の値か、シリアライズ可能なオブジェクト（への参照）にする。

リモートインターフェースのメソッドの引数、戻り値は、プリミティブ型の値か、シリアライズ可能なオブジェクトへの参照にします。リモートメソッドを呼び出す際に指定する引数は、ネットワーク経由で転送されることになるのですが、その際にはシリアライゼーションが行われます。同様のことは戻り値にも言えます。Stringや配列など、APIに用意されているクラスの大半はシリアライズ可能になっているので問題はありませんが、独自に作ったクラスのオブジェクトを引数や戻り値に使う場合はシリアライズ可能になっているか（Serializableインターフェースをインプリメントしているか）必ず確認してください。

```
public String sayHello() throws RemoteException;
```

この戻り値はネットワーク経由でクライアントに転送されることになるので、シリアライズ可能でなくてはならない。引数に関しても同様のことが言える

This PDF is prepared for tools@maruhouse.org, personal use only.

ステップ2：リモートインターフェースのインプリメンテーションを作成する

MyRemoteImpl.java

① リモートインターフェースをインプリメントする。

クライアントが呼び出すリモートメソッドを含むサーバクラスは、リモートインターフェースをインプリメントして作る必要があります。

```
public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {
  public String sayHello() {
    return "Server says, 'Hey' ";
  }
  // その他のコード
}
```

コンパイラは、リモートインターフェースのメソッドがすべてインプリメントされているかを確認する。ここでは、インプリメントすべきメソッドは1つだけ

② UnicastRemoteObjectクラスを継承する。

サーバクラスは、リモートで利用できるものでなくてはなりません。クラスをリモートで利用できるものにする最も簡単な方法は、UnicastRemoteObjectクラス（java.rmi.serverパッケージのクラス）を継承することです。UnicastRemoteObjectを継承すれば、クラスは自動的にリモートで利用できるものになります。

```
public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {
```

③ 引数のないコンストラクタを作る（RemoteExceptionの発生を宣言する）。

UnicastRemoteObjectクラスには、「コンストラクタがRemoteExceptionを発生させる可能性がある」という問題があります。そのため、このクラスを継承する場合には、コンストラクタの宣言に「throws RemoteException」という記述を加える必要があります。クラスがインスタンス化される際には、スーパークラスのコンストラクタが必ず呼び出されます。つまり、そのコンストラクタが例外を発生させるのであれば、サブクラスのコンストラクタにも「throws RemoteException」という記述を入れざるを得ないのです。

```
public MyRemoteImpl() throws RemoteException {}
```

コンストラクタの「中身」は書かなくてよい。ただ、例外が発生する可能性があることだけを明記する

④ サーバオブジェクトをRMIレジストリに登録する。

サーバクラスができたら、それをクライアントから利用できるようにする必要があります。そのためには、クラスをインスタンス化し、RMIレジストリに登録しなければなりません（RMIレジストリにオブジェクトを登録するためのコードが実行された時点でRMIレジストリが起動されていなければ例外が発生する）。ただ、注意すべきなのは、表面上はサーバオブジェクトを登録しているようでも、実際にレジストリに登録されるのはスタブである、という点です。クライアントが直接使用するのはスタブだからです。RMIレジストリへの登録には、java.rmi.Namingクラスのrebind()メソッドを使用します。

```
try {
  MyRemote service = new MyRemoteImpl();
  Naming.rebind("Remote Hello", service);
} catch(Exception ex) {...}
```

RMIレジストリにサーバオブジェクトを登録する際には名前を付ける。ここでは"Remote Hello"という名前を付けている。クライアントはこの名前を使ってサーバオブジェクトをレジストリから検索できる。実際にRMIレジストリに登録されるのはスタブである点に注意

This PDF is prepared for tools@maruhouse.org, personal use only.

ステップ 3：スタブとスケルトンを作る

① サーバクラス（リモートインターフェースのインプリメンテーション）を引数に指定して、rmicを実行する。

rmicは、リモートインターフェースのインプリメンテーションを基に、スタブクラスとスケルトンクラスを作成するというツールです。このツールは、Javaの開発環境にあらかじめ用意されています。スタブクラス、スケルトンクラスの名前は元のクラス名に「_Stub」、「_Skel」を付けたものになります。使用するオプションによっては、スケルトンを作らずスタブだけを作ることや、クラスのソースコードがどのようなものになるかを確認することなどもできます。また、IIOPプロトコルに対応するためのオプションなども用意されています。ここで紹介しているのは、最も一般的なrmicツールの使い方です。スタブクラス、スケルトンクラスの作成先は、カレントディレクトリ（cdで最後に移動したディレクトリ）になります。ただ、このディレクトリは、リモートインターフェースのインプリメンテーションにアクセスできる場所でなければなりません。したがって、通常、rmicツールは、リモートインターフェースのインプリメンテーションが置かれているディレクトリで実行します（話が複雑になるので、ここではあえてパッケージについて触れるのを避けています。パッケージの階層構造やクラスの完全名などに関しては後で詳しく説明します）。

クラス名の後に“.class”を付けず、クラス名だけを指定している点に注意。

```
File  Edit  Window  Help  Whuffie
% rmic MyRemoteImpl
```

MyRemoteImpl_Stub.class

MyRemoteImpl_Skel.class

2つのヘルパーオブジェクトのクラスファイルが作られる

ステップ 4：RMIレジストリを起動する

① 新たにターミナルウィンドウ（コマンドプロンプト）を表示し、RMIレジストリを起動する。

RMIレジストリは、使用するクラスファイルへのアクセスが可能なディレクトリから起動する必要があります。最も簡単なのは、“classes”ディレクトリから起動することです。

```
File  Edit  Window  Help  Huh?
%rmiregistry
```

ステップ 5：サーバ（サーバオブジェクト）を起動する

① RMIレジストリとは別のターミナルウィンドウ（コマンドプロンプト）を表示し、サーバを起動する。

サーバオブジェクトの起動は、サーバクラスのmain()メソッドから行う場合もあれば、別のクラスから行う場合もあります。ここでは、わかりやすいよう、サーバクラスのクラスファイルそのものを実行し、サーバクラスのmain()メソッドから起動しています。このmain()メソッドの中でサーバクラスのインスタンスが作成され、RMIレジストリへの登録も行われます。

```
File  Edit  Window  Help  Huh?
% Java MyRemoteImpl
```


This PDF is prepared for tools@maruhouse.org, personal use only.

RMIアプリケーション（サーバ部分）のコード

サーバ

リモートインターフェース

```
import java.rmi.*;

public interface MyRemote extends Remote {

  public String sayHello() throws RemoteException;

}
```

RemoteExceptionクラス、Remoteインターフェースはjava.rmiパッケージに属する

java.rmi.Remoteを継承してインターフェースを作る

このインターフェースのメソッドはすべてRemoteExceptionを発生させる可能性がある

サーバクラス（リモートインターフェースのインプリメンテーション）

```
import java.rmi.*;
import java.rmi.server.*;

public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {

  public String sayHello() {
    return "Server says, 'Hey' ";
  }

  public MyRemoteImpl() throws RemoteException { }

  public static void main (String [] args) {

    try {
      MyRemote service = new MyRemoteImpl();
      Naming.rebind("Remote Hello", service);
    } catch(Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

UnicastRemoteObjectはjava.rmi.serverパッケージに属する

リモートアクセス可能なオブジェクトを作るには、UnicastRemoteObjectクラスを継承するのが最も簡単

リモートインターフェースのメソッドは当然すべてインプリメントする。ただ、「throws RemoteException」というコードはなくてよい

リモートインターフェースをインプリメントする

スーパークラスであるUnicastRemoteObjectのコンストラクタがRemoteExceptionを発生させる可能性があるため、このコードが必要。このコードによって、コンストラクタ（厳密にはスーパークラスのコンストラクタ）が例外を発生させる可能性があることが明確にわかるようになる

サーバクラスのインスタンスを作成し、RMIレジストリに登録する。登録にはスタティックメソッドであるNaming.rebind()を使用する。登録の際に指定した名前はクライアントがサーバオブジェクトを検索するのに使用する

This PDF is prepared for tools@maruhouse.org, personal use only.

クライアントがスタブを取得するには

クライアントがサーバオブジェクトのメソッドを呼び出すには、まずスタブオブジェクト（への参照）を取得しなくてはなりません。直接呼び出すのは、スタブオブジェクトのメソッドだからです。スタブオブジェクトの取得にはRMIレジストリを使用します。クライアントは、まるで電話帳のページから電話番号を見つけ出すように、RMIレジストリからスタブオブジェクトを探すのです。その際には、RMIレジストリへの登録時にサーバオブジェクトに付けられた名前が手掛かりになります。

lookup()メソッドはNamingクラスのスタティックメソッド

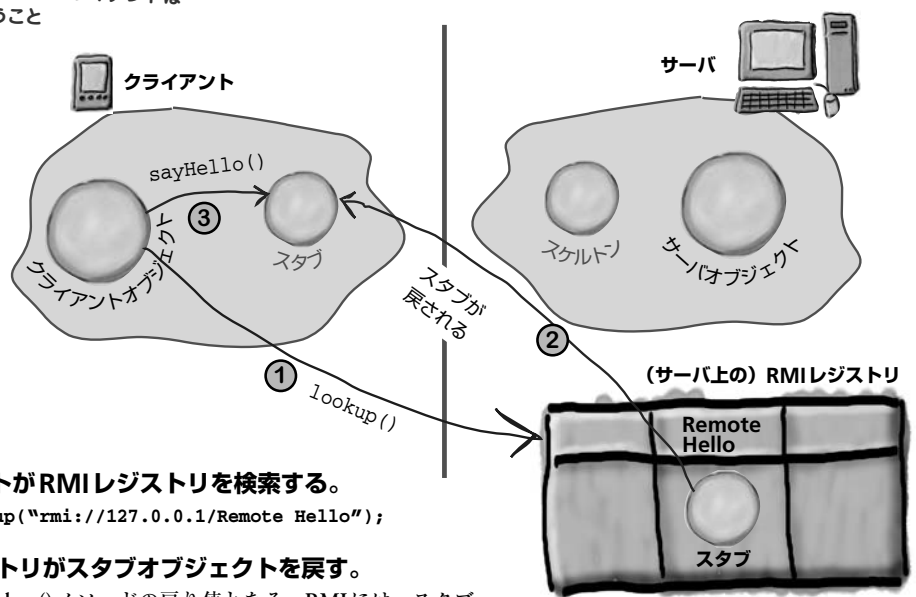
ここには、RMIレジストリ登録時にサーバオブジェクトに付けられた名前を指定する

```
MyRemote service = (MyRemote) Naming.lookup("rmi://127.0.0.1/Remote Hello");
```

スタブオブジェクトへの参照を格納する変数の型としては、必ずリモートインターフェースを使用する。これは、メソッドを呼び出すリモートオブジェクトのクラス名をクライアントは一切知る必要がないということ

lookup()メソッドによって得られる参照の型はObjectなので、キャストする必要がある。

サーバのホスト名、IPアドレスを指定する



① **クライアントがRMIレジストリを検索する。**

```
Naming.lookup("rmi://127.0.0.1/Remote Hello");
```

② **RMIレジストリがスタブオブジェクトを戻す。**

スタブはlookup()メソッドの戻り値となる。RMIには、スタブをデシリアライズする機能があらかじめ用意されている。ただし、クライアント上のスタブのクラスファイル（rmicによって作ることができる）がなければ、スタブのデシリアライズはできない。

③ **クライアントが、スタブのメソッドをサーバオブジェクトのメソッドであるかのように呼び出す。**

This PDF is prepared for tools@maruhouse.org, personal use only.

スタブのクラスファイルをクライアントに提供するには

前ページで触れたとおり、クライアントがスタブオブジェクトを取得するためには、スタブクラスのクラスファイルが必要で、このクラスファイルがないとデシリアライズが行えません。しかし、「RMIアプリケーションを利用するクライアントマシンにどのようにしてクラスファイルを配布すればよいのか」と疑問に思った人もいるのではないでしょうか。

クライアントの数がさほど多くなければ、逐一手作業で配布してもかまわないかもしれませんが、あまりいい方法とは言えません。本書の範囲を超えるため、詳しくは説明しませんが、「ダイナミッククラスダウンローディング」という便利な方法もあります。ダイナミッククラスダウンローディングでは、スタブオブジェクト（などのシリアライズされたオブジェクト）のデータをクライアントに送る際、「このオブジェクトのクラスファイルをどこで入手できるか」を知らせるURLを添付します。クライアント上のRMIシステムは、デシリアライズに際してクラスファイルがマシン上に見付からなければ、添付されたURL（とHTTPのGET命令）を使用してクラスファイルを入手するのです。ダイナミッククラスダウンローディングを行うためには、クラスファイルを提供できる簡単なWebサーバを用意し、クライアントのセキュリティ関連のパラメータをいくつか調整する必要があります。その他にも注意すべきことがらはあるのですが、ここでは「そういう方法がある」とおおまかに理解しておけば十分でしょう。

クライアントのコード

```
import java.rmi.*;

public class MyRemoteClient {
  public static void main (String [] args) {
    new MyRemoteClient().go();
  }

  public void go() {

    try {
      MyRemote service = (MyRemote) Naming.lookup("rmi://127.0.0.1/Remote Hello");

      String s = service.sayHello();

      System.out.println(s);
    } catch(Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

Namingクラス（RMIレジストリの検索に使用する）はjava.rmiパッケージに属する

戻り値はObject型になるので、キャストが必要

IPアドレス、あるいはホスト名を指定する

RMIレジストリへの登録の際にサーバオブジェクトに付けられた名前を指定する

RemoteExceptionの発生に対応できるコードを書かねばならないという点を除き、ごく普通のメソッド呼び出しと変わらない点に注意

This PDF is prepared for tools@maruhouse.org, personal use only.

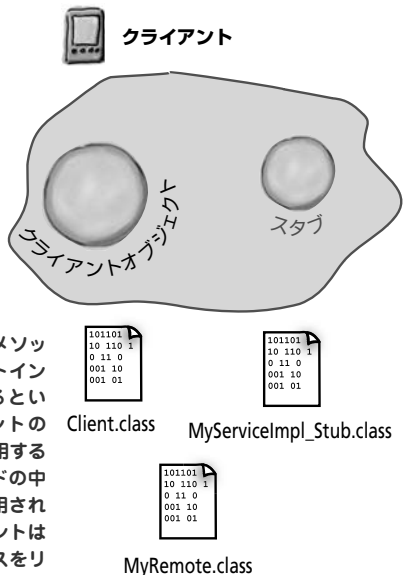
クライアントマシン、サーバマシンに必要なクラスファイル

RMIに関してプログラマが犯しやすいミスはだいたい以下の3つです。

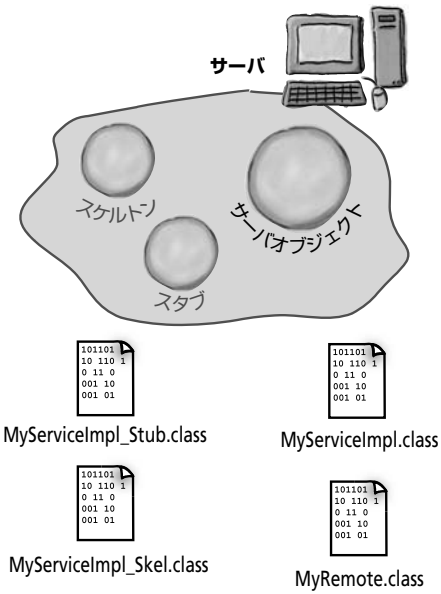
1) リモートサーバ(サーバクラス)をRMIレジストリよりも先に起動してしまう(Naming.rebind()メソッドによってサーバオブジェクトが登録される時にはRMIレジストリは動いている必要がある)。

2) シリアライズ可能でないオブジェクトを引数や戻り値にしてしまう(コンパイル時にこの誤りは検知されないので、実行時にはじめて気付くことになる)。

3) スタブのクラスファイルをクライアント上に置かない。



クライアントはスタブのメソッドを呼び出すのにリモートインターフェースを利用するという点に注意。クライアントのJVMはスタブクラスを利用するが、クライアントのコードの中で直接スタブクラスが使用されることはない。クライアントはリモートインターフェースをリモートオブジェクトであるかのように扱う



サーバはスタブクラスとスケルトンクラスの両方を必要とする。また、リモートインターフェースとそのインプリメンテーション(サーバクラス)も必要とする。スタブクラスが必要なのは、サーバオブジェクトをRMIレジストリに登録するため(この時、実際に登録されるのはスタブ)

This PDF is prepared for tools@maruhouse.org, personal use only.

自分で考えてみよう

ニワトリが先かタマゴが先か……

下に示したカードには、それぞれ、RMIアプリケーションが実行される際に行われることが書かれています。正しい順序に並べ替えてください。

1.

2.

3.

4.

5.

6.

7.

- クライアントがRMIレジストリからスタブを取得する
- スタブがメソッドコールをサーバに転送する
- クライアントがスタブのメソッドを呼び出す
- クライアントがRMIレジストリを検索する
- サーバオブジェクトがRMIレジストリに登録される
- RMIレジストリが起動される
- サーバクラス（リモートインターフェースのインプリメンテーション）がインスタンス化される

重要ポイント

- あるオブジェクトから、存在するヒープの異なる（実行に使用されるJVMの異なる）オブジェクトの参照を取得するということは基本的にできない。
- RMI（Remote Method Invocation）を利用した場合、コードの上ではリモートオブジェクト（実行するJVMの異なるオブジェクト）のメソッドを直接呼び出しているように見えるが、実際にはそうではない。
- クライアントがリモートオブジェクトのメソッドを呼び出しているように見える時、実際に呼び出されているのはスタブ（リモートオブジェクトの代理となるオブジェクト）のメソッド。
- スタブは「クライアントヘルパー」の一種で、ソケット接続、ストリーム、シリアライゼーションなどに関する細かい処理を行い、クライアントからのメソッドコールをサーバに転送する。
- RMIアプリケーション（のサーバ部分）を作るには、まずリモートインターフェースを作る必要がある。
- リモートインターフェースは、java.rmi.Remoteインターフェースを継承して作る。リモートインターフェースのメソッドの宣言には必ず「throws RemoteException」という記述を入れる。
- サーバクラスは、リモートインターフェースをインプリメントして作る。
- サーバクラスは、UnicastRemoteObjectを継承して作るべき（他にも方法はあるが、これが最も簡単）。
- サーバクラスには、「throws RemoteException」という宣言を含むコンストラクタが必要（スーパークラスのコンストラクタが例外を発生させる可能性があるため）。
- サーバクラスのオブジェクトを作成後は、RMIレジストリに登録する。
- サーバオブジェクトのRMIレジストリへの登録には、以下のようなスタティックメソッドを使用する。

```
Naming.rebind("Service Name", serviceInstance);
```

- RMIレジストリは、登録するサーバオブジェクトと同じマシン上で、サーバオブジェクトより先に起動する必要がある。
- クライアントがRMIレジストリでサーバオブジェクトを検索するには以下のようなスタティックメソッドを使用する。

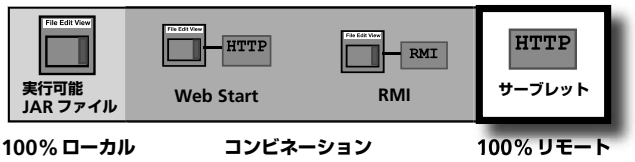
```
Naming.lookup("rmi://MyHostName/ServiceName");
```

- サーバオブジェクトのRMIレジストリへの登録、サーバオブジェクトの検索、スタブを使ったリモートメソッド呼び出しなど、RMI関連の処理の多くはRemoteException（チェック例外）を発生させる恐れがある。

This PDF is prepared for tools@maruhouse.org, personal use only.

RMIは実際にどのように応用されているのか？

This PDF is prepared for tools@maruhouse.org, personal use only.



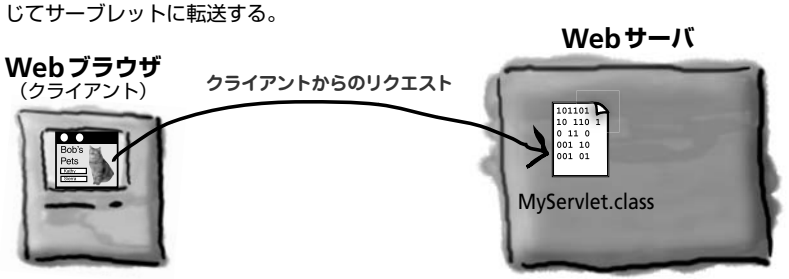
サーブレット

サーブレットはWebサーバ上で動作する特殊なJavaプログラムです。Webサーバは、ユーザの使用するWebブラウザからのリクエストの内容により、必要に応じてサーブレットを起動したり、利用したりします。つまり、サーブレットは、ユーザーのWebブラウザの操作に対応して、サーバ上で必要な処理を行うということです。Perlなどで作成されるCGIスクリプトと同じようなものと考えればわかりやすいでしょう。サーブレットは、Javaプログラムなので、基本的にJavaプログラムにできることは何でもできるのですが、一般的には、データベースとWebページの連携や、掲示板のWebページを作ることなどに使用されます。

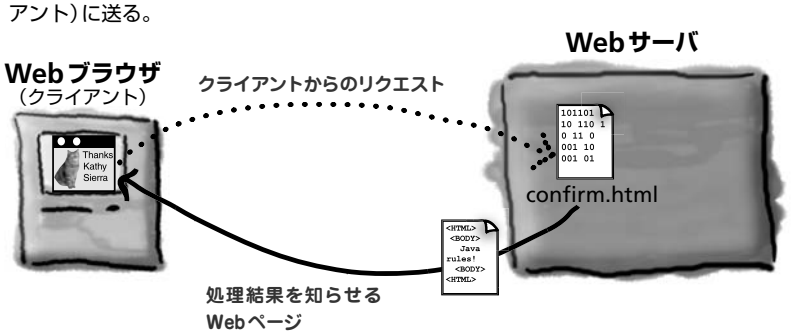
サーブレットはもちろん、RMIを利用したプログラムにすることもできます。

実際に、サーブレットをEJBのクライアントとして使用する場合などには、サーブレットとEJBのやりとりにRMIが応用されることが多くなっています（ただし、この場合、RMIの使用方法がここで解説したよりも少し複雑になるという点に注意）。

① ユーザーがWebページ上のフォームにデータを入力し、「送信」ボタンをクリックすると、Webサーバ（HTTPサーバ）にリクエストが送られる。Webサーバは、受け取ったリクエストを、内容に応じてサーブレットに転送する。

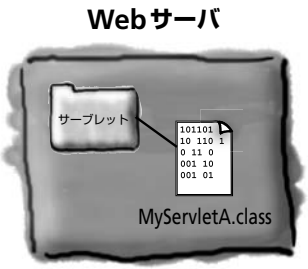


② サーブレットは、ユーザーの入力データをデータベースに追加し、処理結果を知らせるWebページを作ってWebブラウザ（クライアント）に送る。



This PDF is prepared for tools@maruhouse.org, personal use only.

サーブレットの作成・実行



① サーブレットをどこに配置すべきかを確認する。

ここでは、Webサーバがすでにコンピュータにインストールされ、起動もされており、しかも、サーブレットに対応するための環境設定もできているという前提で話を進めます。サーブレットを使う際に特に重要なのは、「サーブレットをどこに配置すればWebサーバがアクセスできるか」を確かめることです。WebサイトがISP(インターネットサービスプロバイダ)にホスティングされているものである場合は、サーブレットやCGIスクリプトの配置についてはISPに問い合わせることになります。

servlets.jar

② servlets.jarファイルを入手し、CLASSPATHに指定されたディレクトリに配置する。

サーブレットの作成に必要なクラスはJavaの標準のライブラリには用意されていません。サーブレットを使用するには、そうしたクラスの納められたservlets.jarファイルを入手する必要があります。このファイルはjava.sun.comや、Apache Tomcatのサイト(apache.org)などからダウンロードできます。servlets.jarに納められたクラスがなければサーブレットはコンパイルすらできません。

MyServletA.class

③ HttpServletクラスを継承してサーブレットのコードを書く。

サーブレットのクラスは、HttpServletクラス(javax.servlet.httpパッケージのクラス)を継承して作ります。他にもサーブレットのクラスを作る方法はあるのですが、通常はこの方法を採ります。

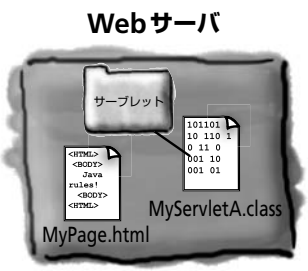
```
public class MyServletA extends HttpServlet {...}
```

MyPage.html

④ サーブレットを起動するHTMLページを作る。

サーブレットを起動できるようにするには、HTTPページ(Webページ)に以下のようなリンクを作ります。ユーザーがリンクをクリックすると、Webサーバは、該当するサーブレットを探し、発行されたHTTPコマンド(GET、POSTなど)に応じて適切なメソッドを呼び出します。

```
<a href="servlets/MyServletA">This is the most amazing servlet.</a>
```



⑤ サーブレット、HTMLページを、Webサーバからアクセスできるものにする。

この作業の具体的内容はWebサーバによって(使用するサーブレットのバージョンによって)大きく異なってきます。ISPからは、通常、「Webサイトのサーブレットディレクトリに置く」というような指示がなされることになります。しかし、Tomcatの最新バージョンを使用する場合などには、サーブレット(やHTMLページ)を正しく配置するだけでもかなりの(そのテーマで本が書けるくらいの)手間がかかります。

This PDF is prepared for tools@maruhouse.org, personal use only.

簡単なサーブレットの例

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
```

java.io以外にもサーブレット関連のパッケージを2つインポートする必要がある。いずれのパッケージもJavaの標準ライブラリに属するものではないことに注意。プログラマが自ら入手する必要がある

通常、サーブレットはHttpServletを継承し、メソッドを少なくとも1つオーバーライドして作成する

```
public class MyServletA extends HttpServlet {
```

HTTPのGET命令に対応するためのメソッドをオーバーライド

Webサーバは、ブラウザからのGET命令に対応してこのメソッドを呼び出す。引数には、クライアントからのリクエストを表すHttpServletRequestオブジェクトと、リクエストへの応答を表すHttpServletResponseオブジェクトを指定する

```
  public void doGet (HttpServletRequest request, HttpServletResponse response)
                                  throws ServletException, IOException {
```

これによって、サーブレットの処理の結果として得られたデータがどのような種類のものであるかをWebサーバ（あるいはブラウザ）に知らせる

```
    response.setContentType("text/html");
```

HttpServletResponseオブジェクトを利用して、Webサーバにデータを送るための出力ストリームを取得する

```
    PrintWriter out = response.getWriter();

    String message = "If you're reading this,it worked!";

    out.println("<HTML><BODY>");
    out.println("<H1>" + message + "</H1>");
    out.println("</BODY></HTML>");
    out.close();
  }
}
```

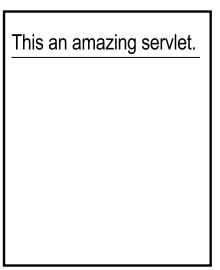
出力ストリームを使用して新たなHTMLページのデータを送る。このデータは、Webサーバによって（他のWebページのデータと同じように）ブラウザに送られる。このデータに対応する.htmlファイルはどこにも存在しないが、.htmlの存在するデータと同様に扱われる

サーブレットを起動するHTMLページ（Webページ）の画面は以下のようになる

サーブレットを起動するHTMLページ

```
<HTML>
  <BODY>
    <a href="servlets/MyServletA">This is an amazing servlet.</a>
  </BODY>
</HTML>
```

このリンクをクリックすると、サーブレットが起動される



This PDF is prepared for tools@maruhouse.org, personal use only.

重要ポイント

- サーブレットは、HTTP(Web)サーバ上で動作する特殊なJavaプログラム。
- サーブレットは、ユーザーのWebページの操作の結果として発行されるリクエストに応じて処理を行う。たとえば、ユーザーがWebページ上のフォームにデータを入力し、「送信」ボタンをクリックした場合に、サーブレットが送られてきたデータをデータベースに追加し、処理結果を知らせるWebページを新たに作ってブラウザに送り返す、というようなことができる。
- サーブレットプログラムをコンパイルするためには、サーブレットの作成に必要なクラスの納められたservlets.jarファイルを入手する必要がある。サーブレット関連のクラスはJavaの標準ライブラリには用意されていないため。servlets.jarファイルは、java.sun.comなどのWebサイトからダウンロードできる(注意:Servletライブラリは、Java 2 Enterprice Edition(J2EE)に含まれている)。
- サーブレットを実行するためには、サーブレットに対応したWebサーバが必要。そうしたサーバの例としては、apache.orgのTomcatなどがある。
- サーブレットは、Webサーバのアクセスできる場所に配置する必要がある。WebサイトがISPにホスティングされているものである場合は、サーブレットの配置についてISPに問い合わせることになる。
- サーブレットは、通常、HttpServletクラスを継承し、doGet()、doPost()といったメソッドをオーバーライドして作る。
- Webサーバは、クライアントのリクエストに応じて、サーブレットを起動したり、doGet()などのメソッドを呼び出す。
- サーブレットがリクエストへの応答を送信する際には、doGet()のパラメータであるHttpServletResponseオブジェクトを使用して出力ストリーム(PrintWriter)を取得する。
- サーブレットは、処理結果を知らせるWebページ(タグを含むHTMLデータ)を出力する。

素朴な疑問に答えます

Q:JSPとは何ですか。サーブレットとの関係は?

A: JSP(Java Server Pages)は、本質的にはサーブレットと同じものです。JSPのコードは最終的にサーブレットに変換されるからです。ただ、JSPを利用する場合とサーブレットを利用する場合では、プログラマが作成するものが異なってきます。サーブレットの場合、作成するのは、実行結果としてHTMLコードを出力する「Javaクラス」です(HTMLコードを出力しないサーブレットもあり得る)。しかし、JSPの場合、作成するものはそれと逆、つまりJavaコードを含む「HTMLコード」になるのです。JSPのメリットは、通常のHTMLページを作るのとほぼ同じ方法でダイナミックなHTMLページが作れるということです。唯一、通常HTMLページと異なるのは、ページ中にJavaのコードを埋め込むということだけです(Javaコードを埋め込むには特殊なタグを使用する)。埋め込まれたJavaコードは、ページが表示される際に実行され、ページの画面をさまざまに変えることになります。

サーブレットでは、実行結果を知らせるHTMLページを作成するためのコードを書くのにprintステートメントを使用しますが、HTMLページが複雑なものである場合は、これがかなり困難な作業になります。そうした負担が大幅に軽減されるのもJSPのメリットの1つです。ただ、実行結果が次々に変化するということでもない限り、サーブレットのコードを書くのは実際にはさほど困難にはならないため、JSPをどうしても使用しなければならないというような場面は少ないと言えます。そのため、Webサーバの中にはサーブレットには対応していても、JSPには対応していないというものもあります。

JSPには、Javaプログラマーの仕事とWebページデザイナーの仕事をかなり明確に分離できるというメリットもあります。ただ、それでも現実には、やはりJavaプログラマーがHTMLのことをまったく考えなくてよい、とかWebデザイナーがまったくJavaを知らなくてよい、ということはありません。ある程度両方の知識がないとJSPは使いこなせないのです。とはいえ、WebデザイナーがJSPのコードをゼロから書かなくて済むという便利なツールなどもあるので、そうしたツールを利用することで負担が軽減されることは確かでしょう。

Q:RMIに比べてサーブレットについての説明があまりに少ない気がしますが……。これでサーブレットについての説明は終わりなのでしょうか。

A:RMIはJavaの標準機能であり、必要なクラスも標準ライブラリにすべて納められていますが、一方、サーブレットやJSPはそうではない、ということから、本書での説明は簡潔に済ませました(サーブレットやJSPが広く使われていることも確かですが……)。RMIには、極端にバージョンの古いものは別にしても、通常、JVMだけで対応できます。しかし、サーブレットやJSPを使うためには、サーブレットコンテナを備えたWebサーバを用意し、適切な環境設定を行う必要があります。そういったことがらについて詳しく触れるのは本書の範囲を超えていると言えるでしょう。サーブレットやJSPの詳細については、『Javaサーブレットプログラミング 第2版』、『JavaServer Pages 第2版』(いずれもオライリー・ジャパン発行)といった書籍を参照してください。

This PDF is prepared for tools@maruhouse.org, personal use only.

PhraseOMaticプログラムを利用したサーブレットを作ってみる

サーブレットの基本についてはだいたいわかったと思いますので、ここで、1章で紹介した「PhraseOMaticプログラム」を応用したサーブレットを作ってみましょう。サーブレットと言えどもJavaプログラムには違いありません。したがってサーブレットから、PhraseOMaticクラスのメソッドを呼び出してもまったく問題はないということになります。ただ、注意すべきなのは、PhraseOMaticクラスのクラスファイルが、サーブレットと同じディレクトリに配置されていなければならないということだけです（PhraseOMaticクラスのコードは次のページに載せてあります）。

```
import java.io.*;

import javax.servlet.*;
import javax.servlet.http.*;

public class KathyServlet extends HttpServlet {
  public void doGet (HttpServletRequest request, HttpServletResponse response)
                                        throws ServletException,IOException {
    String title = "PhraseOMatic has generated the following phrase.";

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    out.println("<HTML><HEAD><TITLE>");
    out.println("PhraseOmatic");
    out.println("</TITLE></HEAD><BODY>");
    out.println("<H1>" + title + "</H1>");
    out.println("<P>" + PhraseOMatic.makePhrase());
    out.println("<P><a href=\"KathyServlet\">make another phrase</a></p>");
    out.println("</BODY></HTML>");

    out.close();
  }
}
```

このように、サーブレットから他のクラスのメソッドを呼び出すことができる。ここでは、PhraseOMaticクラスのスタティックメソッドであるmakePhrase()を呼び出している（次のページ参照）

This PDF is prepared for tools@maruhouse.org, personal use only.

サーブレットで利用しやすいよう改良したPhraseOMaticクラス

ここで使用するPhraseOMaticクラスのコードは、1章で紹介したものに若干改良を加えています。1章で紹介したコードでは、すべての処理をmain()メソッドで行うようになっていました。また、新しい言葉を作って画面に表示するのに逐一コマンドラインでプログラムを起動し直す必要がありました。それを、makePhrase()というStringを戻すスタティックメソッドで処理を行うように変えてあります。こうすれば、makePhrase()メソッドを他のコードから呼び出し、戻り値を得るという使い方ができます。この戻り値が無作為に選ばれた単語を組み合わせた言葉になるわけです。

以下のコードでは、Stringの配列に文字列を代入する際、紙面の都合で単語と単語の間に「改行」が入ってしまっている箇所がありますが、実際にプログラムを書く際には、改行は行わないでください。また、画面上、単語の途中で行末に到達したとしても自然にワードラッピングが行われるにまかせ、リターンキーを押しての改行はしないでください。

```
public class PhraseOMatic {
  public static String makePhrase() {

    // 選択する単語のリストを3つ作る
    String [] wordListOne = {"24/7","multi-Tier","30,000 foot","B-to-B","win-win","frontend",
    "web-based","pervasive","smart","six-sigma","criticalpath","dynamic"};

    String [] wordListTwo = {"empowered","sticky","valued-added","oriented","centric","distributed",
    "clustered","branded","outside-the-box","positioned","networked","focused","leveraged","aligned",
    "targeted","shared","cooperative","accelerated"};

    String [] wordListThree = {"process","tipping point","solution","architecture","core competency",
    "strategy","mindshare","portal","space","vision","paradigm","mission"};

    // 単語リストにいくつの単語が含まれているかを調べる
    int oneLength = wordListOne.length;
    int twoLength = wordListTwo.length;
    int threeLength = wordListThree.length;

    // 3つの数を無作為に作る。この数をリストから単語を選ぶのに使用する
    int rand1 = (int) (Math.random() * oneLength);
    int rand2 = (int) (Math.random() * twoLength);
    int rand3 = (int) (Math.random() * threeLength);

    // 単語を組み合わせて言葉を作る
    String phrase = wordListOne [rand1] + " " + wordListTwo [rand2] + " " + wordListThree [rand3];

    // 処理結果を戻す
    return ("What we need is a " + phrase);
  }
}
```


This PDF is prepared for tools@maruhouse.org, personal use only.

EJB(Enterprise JavaBeans)：大規模システムのためのRMI

RMIは、リモートでアクセスできるサーバプログラムを作り、実行するには確かに便利です。しかし、amazonやeBayのようなサービスを提供しようとしたら、RMIだけではとても無理です。大規模で本格的なエンタープライズアプリケーションの開発には、RMIにプラスアルファが必要なのです。まず、トランザクション処理のための仕組みが必要です。また、大量のアクセスが同時に行われた場合に対処できるようにしておかなくてはなりません。人気のある商品を扱っているeコマースサイトなどでは、一度に何億(は大げさですが……)という人が同時にサーバにアクセスすることもあります。セキュリティ対策(給与データベースなどに誰もが自由にアクセスできるようでは困ります)やデータ管理についても考慮しなくてはならないでしょう。こうした条件をすべて同時に満たせるもの、となるとエンタープライズアプリケーションサーバしかありません。

Javaをベースにしたエンタープライズアプリケーションサーバと言えば、通常はJ2EE(Java 2 Enterprise Edition)サーバを指します。J2EEサーバは、主にWebサーバとEJB(Enterprise JavaBeans)で構成され、サーブレット、あるいはEJBのかたちでユーザにアプリケーションを提供できます。サーブレットと同様、EJBに関しても、本書の範囲を超えるためここで詳しく触れることはせず、サンプルコードなども載せませんが、ひととおりその仕組みだけを説明しておきます。EJBの詳細については、『Enterprise JavaBeans 第3版』(Richard Monson-Haefel著、オライリー・ジャパン発行)を参照するといいでしょう(また、米国O'Reilly Mediaからは拙著『Head First EJB』も出版されています)。

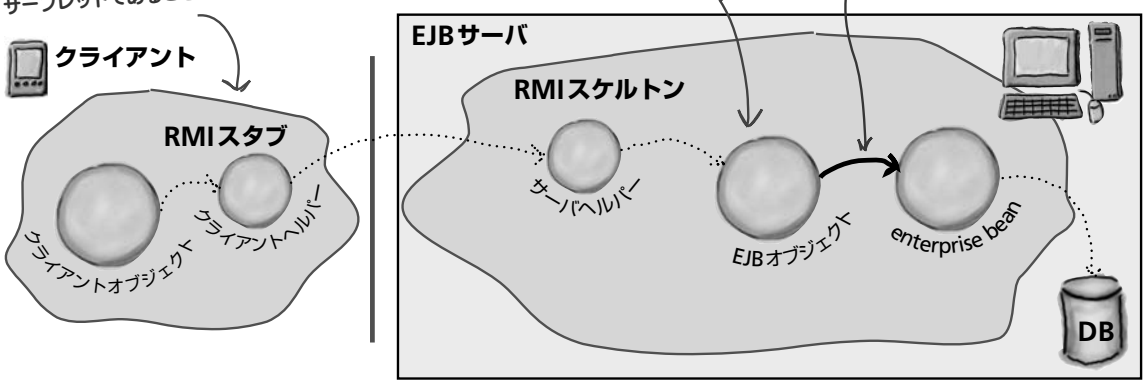
EJBサーバを利用すれば、トランザクション処理、セキュリティ、大量同時アクセスへの対処、データベース管理、ネットワーキング、といったRMIだけでは提供できないいくつものサービスが提供できる

EJBサーバは、クライアントとbeanオブジェクトとの間に入ってさまざまなサービスを提供する

EJBサーバが常に仲介をするため、beanオブジェクトにクライアントが直接にアクセスすることはない。beanオブジェクトにはEJBサーバのみが直接アクセスできる。不正なクライアントからのアクセスが試みられたとしても、EJBサーバがそれを阻止する。EJBサーバのサービスのほとんどは、クライアントからbeanオブジェクトへのアクセスに介入するかたちで提供される

EJBクライアントは、特に決まっているわけではないが、EJBサーバと同一のJ2EEサーバによって動作するサーブレットであることが多い

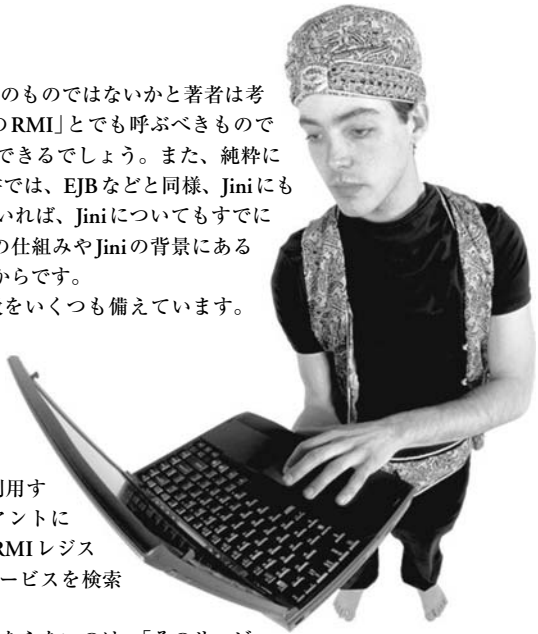
EJBサーバは、ビジネスロジックを保持するbeanオブジェクトへのクライアントからのコールを途中で傍受し、様々なサービス(セキュリティ関連のサービス、トランザクション処理など)を提供する



この図はかなり簡略化されたものである点に注意！

This PDF is prepared for tools@maruhouse.org, personal use only.

Jiniとは



Jiniは素晴らしい技術です。Java関連の技術の中でも最高のものではないかと著者は考えています。EJBは先述のとおり「大規模システムのためのRMI」とでも呼ぶべきものですが、Jiniは「RMIに翼が付いたようなもの」ということができるでしょう。また、純粋にJavaだけで成り立つもの、という点も特徴と言えます。本書では、EJBなどと同様、Jiniにも詳しく触れることはしませんが、RMIについて理解できていれば、Jiniについてもすでにかなりのことがわかっていると言っていいでしょう。Jiniの仕組みやJiniの背景にある考え方などは、RMIを知っていればスムーズに理解できるからです。

JiniはRMIを基礎とした技術ですが、RMIにはない特徴をいくつも備えています。その中でも特に重要なのは次の2つです。

・サービス(サーバ)の自動検索機能
・ルックアップサービスの自動修復機能

RMIの場合は、リモートサーバ(リモートサービス)を利用するのに、その名前と配置を知る必要があります。クライアントには、リモートサービスのIPアドレスやホスト名を指定し、RMIレジストリ登録時に指定されたサービス名を使って利用したいサービスを検索する、というコードが必要です。

しかし、Jiniの場合、サービスを利用するために知らねばならないのは、「そのサービスがどのようなインターフェースをインプリメントしているか」ということだけです。

Jiniでは、サービスの検索に、RMIレジストリよりもはるかに高度で柔軟性に富むルックアップサービスを利用します。Jiniのルックアップサービスの特徴はまず、自らの存在を積極的にネットワーク全体に知らせることができる、という点です。IPマルチキャストという手段によって、「ルックアップサービスが存在し、利用できる状態になっていますよ」というメッセージをネットワークに送ることができるのです。

また、クライアントの側も、ルックアップサービスの配置を尋ねるメッセージをネットワーク全体に送ることができます。

もちろん、クライアントが本当に探したいのは、ルックアップサービス自体ではなく、ルックアップサービスに登録されたさまざまなリモートサービスです。こうしたリモートサービスは、単なるソフトウェアの場合もあれば、プリンタ、デジタルカメラなどのデバイス、コーヒーメーカーなどの家電という場合もあり得ます。

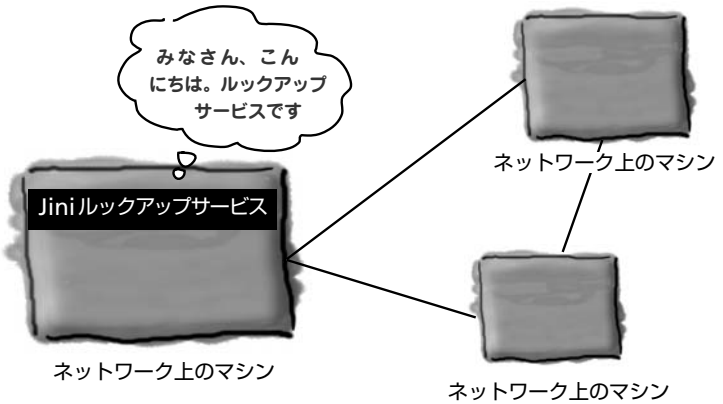
リモートサービス自身も、ネットワークに追加された時点でルックアップサービスを探し、自らを登録するということができます。ルックアップサービスへの登録時には、RMIにおけるスタブの役割を果たすオブジェクトをルックアップサービスにシリアライズして送ります。このオブジェクトはネットワーク関連の処理を行うなどして、クライアントとリモートサービスを仲介します。ただ、RMIとは異なり、ルックアップサービスへの登録時に「サービス名」が付けられることはなく、サービスの検索はサービス名ではなく「サービスがどのようなインターフェースインプリメントしているか」ということを手掛かりに行われます。

クライアントはルックアップサービスを見つけると、「ScientificCalculatorインターフェースをインプリメントするサービスはあるか」などと尋ねるメッセージを送ります。ルックアップサービスは、登録インターフェースのリストから該当するもの見つけた場合、クライアントにそのことを通知し、サービスからシリアライズされて送られたオブジェクトをクライアントに送信します。

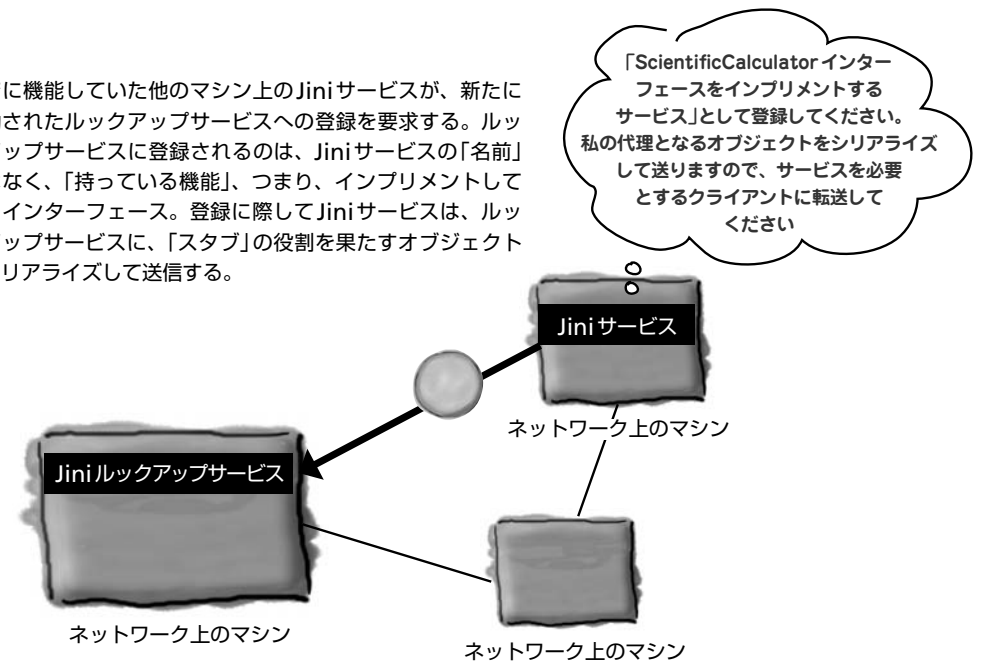
This PDF is prepared for tools@maruhouse.org, personal use only.

サービスの自動検索機能

① Jiniルックアップサービスがネットワーク上のどこかで起動され、IPマルチキャストによってその存在がネットワーク全体に通知される。



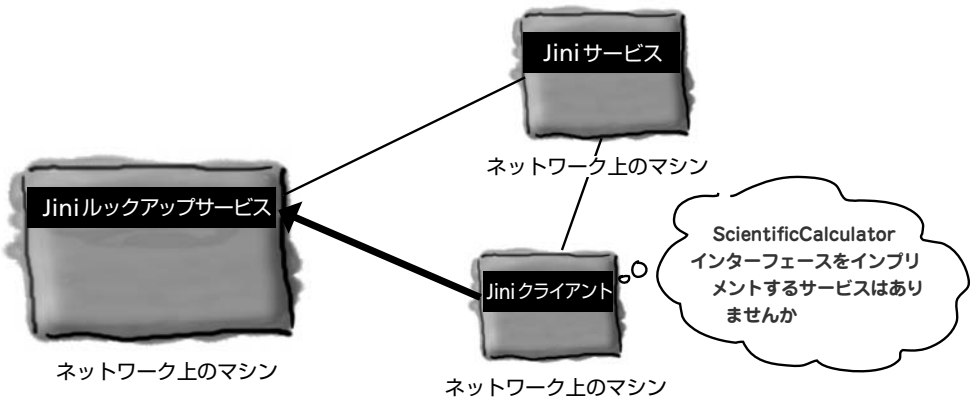
② すでに機能していた他のマシン上のJiniサービスが、新たに起動されたルックアップサービスへの登録を要求する。ルックアップサービスに登録されるのは、Jiniサービスの「名前」ではなく、「持っている機能」、つまり、インプリメントしているインターフェース。登録に際してJiniサービスは、ルックアップサービスに、「スタブ」の役割を果たすオブジェクトをシリアライズして送信する。



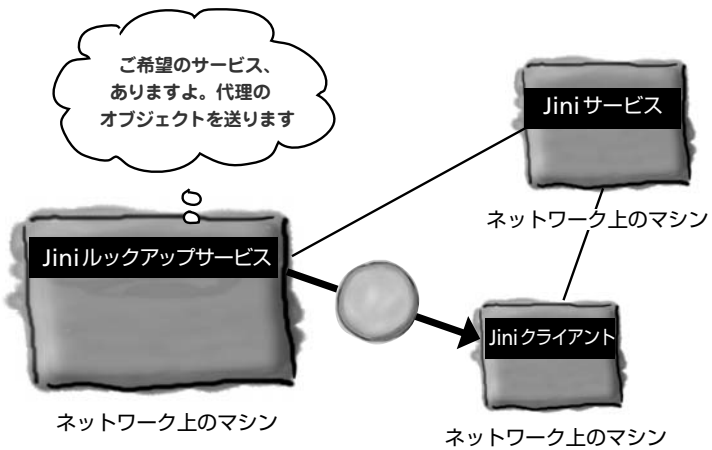
This PDF is prepared for tools@maruhouse.org, personal use only.

サービスの自動検索機能（続き）

③ ネットワーク上のクライアントが、「ScientificCalculatorインターフェースをインプリメントするサービスはないか」とルックアップサービスに問い合わせる。問い合わせるまで、クライアントには、そのようなサービスがどこに存在するかも、また、ネットワーク上に存在するのかどうかもまったくわからない。



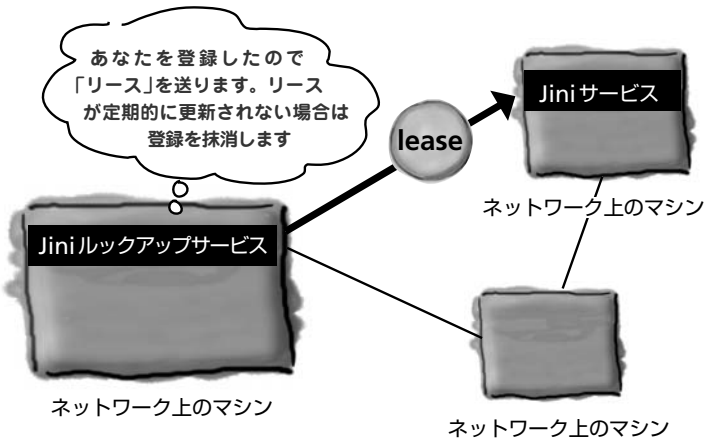
④ ルックアップサービスは問い合わせに応じてScientificCalculatorインターフェースをインプリメントするサービスを探し出し、クライアントに応答する。



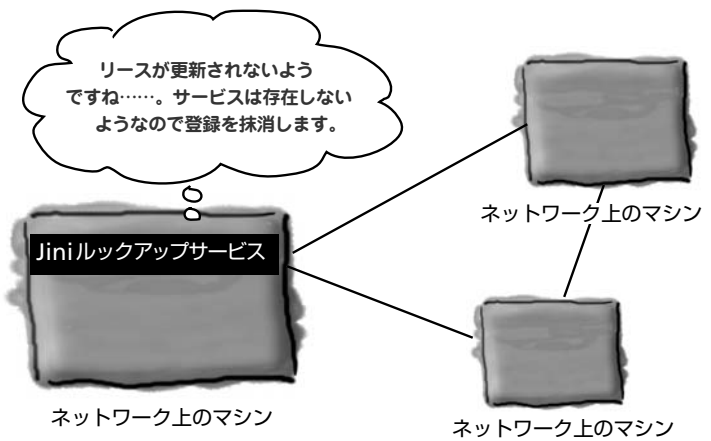
This PDF is prepared for tools@maruhouse.org, personal use only.

ルックアップサービスの自動修復機能

① Jiniサービスがルックアップサービスに登録を求めると、ルックアップサービスはサービスに「リース」と呼ばれるオブジェクトを渡す。登録されたサービスは、このリースを定期的に更新しなければならない。更新が滞ると、ルックアップサービスは、そのサービスがネットワーク上に存在しなくなったとみなす。この仕組みによって、ルックアップサービスは、常にネットワークの状況を正しく反映した情報を提供できる。



② サービスがネットワーク上から（マシンの電源が切られるなどして）消滅すると、リースが更新されなくなる。ルックアップサービスはリースが更新されなくなったサービスの登録を抹消する（電源が入れられ、サービスの提供が再開されれば、再び登録されることになる）。



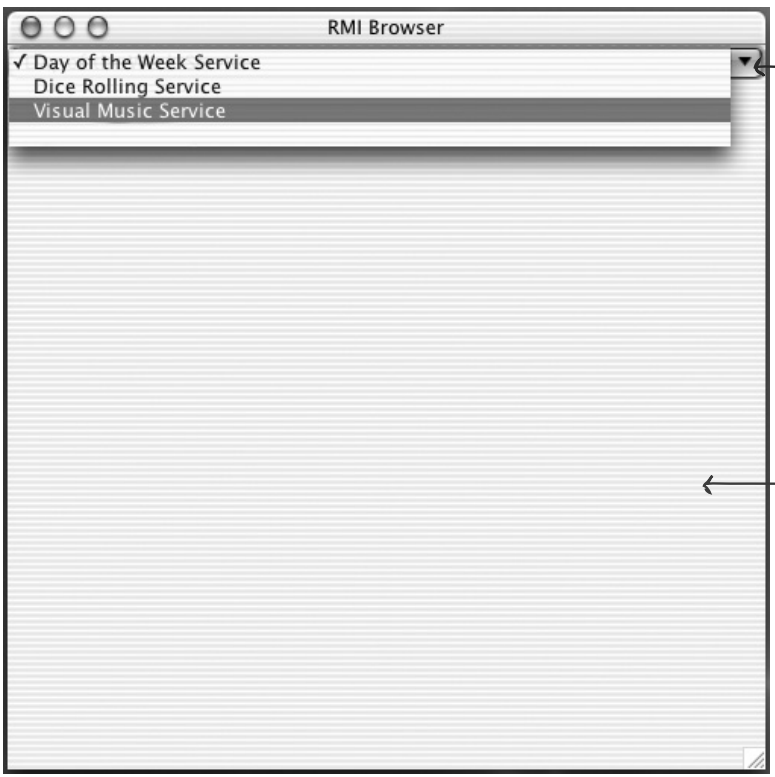
This PDF is prepared for tools@maruhouse.org, personal use only.

ユニバーサルサービスブラウザ

本書もそろそろ終わりに近づいてきました。最後に、「Jiniアプリケーションそのものではないが、それに近い体験ができる」というアプリケーションをRMIを利用して作ってみましょう。著者はこのアプリケーションに「ユニバーサルサービスブラウザ」という名前を付けました。「本物」のJiniアプリケーションと、ここで紹介するユニバーサルサービスブラウザの最大の違いは、「利用するサービスの見つけ方」です。サービスの検索には、ルックアップサービスではなく、RMIレジストリを使用します。ルックアップサービスは、その存在を自らネットワーク全体に知らせることができ、リモートサービスとは異なるマシン上に置くことも可能ですが、RMIレジストリは自らの存在を知らせるということができないので、リモートサービスと同一マシン上に置かねばなりません。

また、サービスの登録も、Jiniの場合のように自動的に行われるわけではないので、Naming.rebind()メソッドを使ってRMIレジストリに登録するコードを書く必要があります。しかし、クライアントがリモートサービスを見つけた後の動きは、Jiniアプリケーションとほぼ同じになります(存在しなくなったサービスの登録が自動的に抹消されるリースは使用しませんが……)。

ユニバーサルサービスブラウザは、表示するのがHTMLページではなく、JavaのGUIであるという点を除き、Webブラウザと似ています。このGUIは、インタラクティブなもので、リモートサービスにアクセスした際にダウンロードされます。



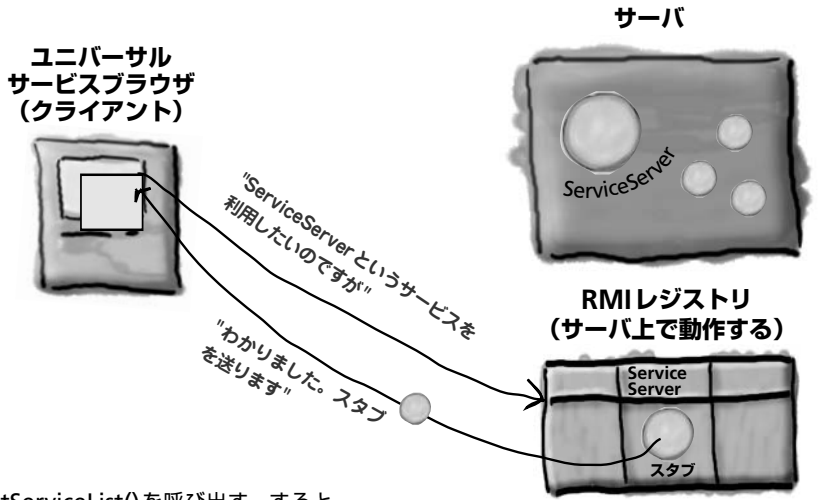
リストから利用するサービスを選べる。RMIのリモートサービス(リモートサーバ)は、サービスのリストを戻すgetServiceList()というメソッドを持つユーザーがサービスを選択すると、クライアントは、選択されたサービスに該当するサーバオブジェクトを送ってくれるよう要求を出す

サービスを選択すると、それに対応する画面が表示される

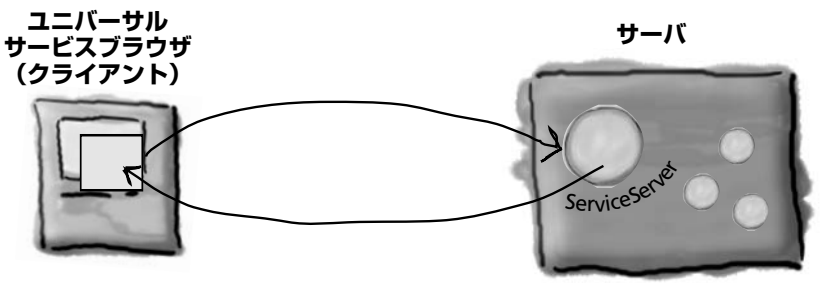
This PDF is prepared for tools@maruhouse.org, personal use only.

ユニバーサルサービスブラウザの動き

① クライアント（ユニバーサルサービスブラウザ）が起動し、RMIレジストリ で“ServiceServer”という名前のサービスを検索する。すると、サービスに対応するスタブが送られてくる。



② クライアントがスタブのgetServiceList()を呼び出す。するとServiceServerからサービスのリスト（配列）が送られてくる。

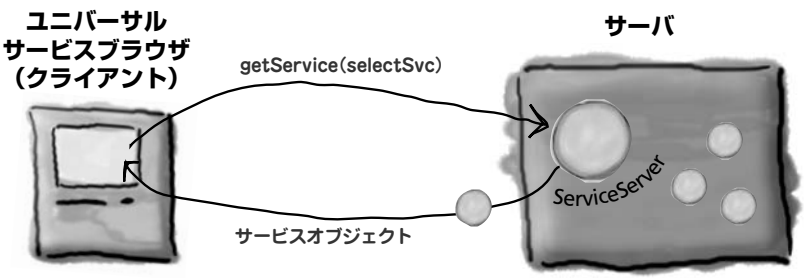


③ クライアントが画面上のGUIにサービスのリストを表示する。

This PDF is prepared for tools@maruhouse.org, personal use only.

ユニバーサルサービスブラウザの動き（続き）

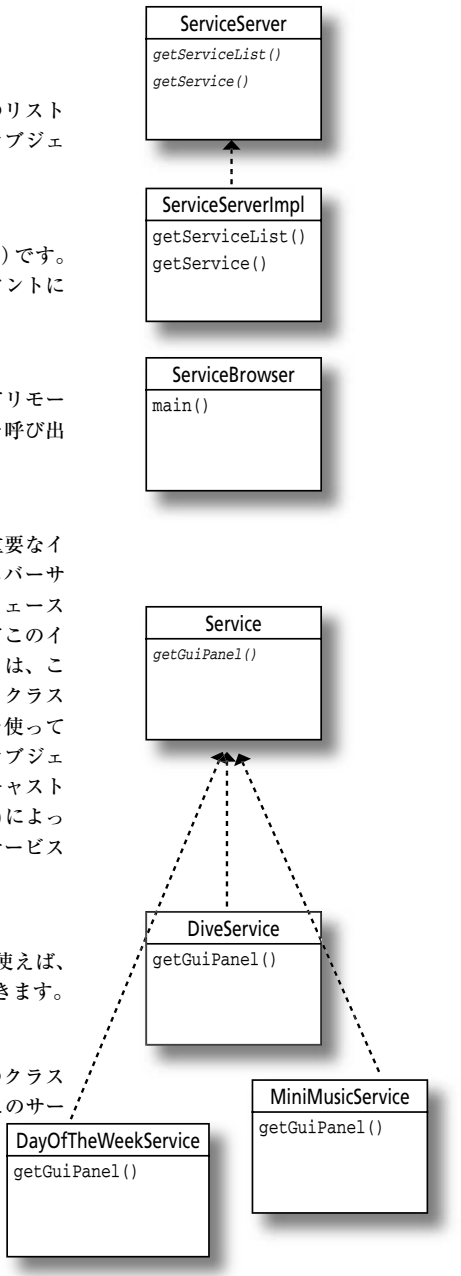
④ ユーザーがリストからサービスを選択すると、クライアントがリモートサービスのgetService()メソッドを呼び出す。すると、クライアント上で使用できるサービスオブジェクトがシリアライズされて送られてくる。



⑤ クライアントが、送られてきたサービスオブジェクトを使ってgetGuiPanel()メソッドを呼び出す。すると、サービスのGUIがブラウザの画面上に表示され、ユーザーがローカルで（サーバにアクセスしなくても）サービスを利用できる状態になる。ユーザーが他のサービスを選択するまではリモートサービスにアクセスすることはない。

This PDF is prepared for tools@maruhouse.org, personal use only.

ユニバーサルサービスブラウザに関連するクラス、インターフェース



① interface ServiceServer extends Remote

これは、ごく一般的なRMIのリモートインターフェースです（サービスのリストを取得するためのメソッドと、選択されたサービスに対応するサービスオブジェクトを戻すメソッドがあります）。

② class ServiceServerImpl implements ServiceServer

UnicastRemoteObjectを継承するRMIリモートサービス（リモートサーバ）です。自らをRMIレジストリに登録するコードを持っており、最終的にクライアントに送られるサービスオブジェクトを作成する機能を備えています。

③ class ServiceBrowser

非常に簡単なGUIを備えたクライアントです。RMIレジストリを検索してリモートサービスのスタブを取得し、次にスタブを利用してリモートメソッドを呼び出し、サービスのリストを取得します。リストはGUIに表示します。

④ interface Service

メソッドはgetGuiPanel()の1つだけという、非常に簡単なものですが、重要なインターフェースです。ユニバーサルサービスブラウザという名前に「ユニバーサル（万能の、普遍的な）」という言葉が使われている所以はこのインターフェースにあるからです。クライアントに送られるサービスオブジェクトはすべてこのインターフェースをインプリメントしている必要があります。クライアントは、このインターフェースをインプリメントしているサービスオブジェクトを、クラス名を知ることなく利用できます。クライアントがServiceServerのスタブを使ってgetService()メソッドを呼び出すと、選択されているサービスに対応するオブジェクトがシリアライズされて送られます。このオブジェクトはService型にキャストされ、getGuiPanel()メソッドの呼び出しに使用されるのです。getGuiPanel()によって戻されたJPanelオブジェクトが画面に表示されることで、ユーザーがサービスを利用するためのGUIができます。

⑤ class DiceService implements Service

「仮想サイコロ」サービスを提供するためのクラスです。このサービスを使えば、本物のサイコロと同じように1～6の範囲の無作為の数字を得ることができます。

⑥ class MiniMusicService implements Service

12章の「ミュージックビデオ」プログラムを覚えているでしょうか。このクラスは、それを応用し、「ミュージックビデオ」サービスを提供するものです。このサービスを利用すれば、何度でもミュージックビデオが見られます。

⑦ class DayOfTheWeekService implements Service

任意の年月日を入力すれば、その曜日を教えてくれるサービスを提供するためのクラスです。

This PDF is prepared for tools@maruhouse.org, personal use only.

interface ServiceServer(リモートインターフェース)

```
import java.rmi.*;
public interface ServiceServer extends Remote {
  Object [] getServiceList() throws RemoteException;
  Service getService (Object serviceKey) throws RemoteException;
}
```

一般的なRMIリモートインターフェース。リモートサービスに必要な2つのメソッドが宣言されている

interface Service(クライアントに送られるサービスオブジェクトがインプリメントするインターフェース)

```
import javax.swing.*;
import java.io.*;

public interface Service extends Serializable {
  public JPanel getGuiPanel();
}
```

ごく普通のインターフェース(リモートインターフェースではない)。メソッドはgetGuiPanel()の1つだけ。サービスオブジェクトはいずれもこのメソッドをインプリメントしなくてはならない。Serializableを継承しているので、このインターフェースをインプリメントしたオブジェクトは自動的にシリアライズ可能になる。サービスオブジェクトは、getService()メソッドが呼び出された時、サーバからクライアントに送られることになるので、シリアライズ可能にしておく必要がある

This PDF is prepared for tools@maruhouse.org, personal use only.

class ServiceServerImpl
（リモートインターフェースのインプリメンテーション）

```
import java.rmi.*;
import java.util.*;
import java.rmi.server.*;

public class ServiceServerImpl extends UnicastRemoteObject implements ServiceServer {

  HashMap serviceList;

  public ServiceServerImpl() throws RemoteException {
    setUpServices();
  }

  private void setUpServices() {
    serviceList = new HashMap();
    serviceList.put("Dice Rolling Service", new DiceService());
    serviceList.put("Day of the Week Service",new DayOfTheWeekService());
    serviceList.put("Visual Music Service",new MiniMusicService());
  }

  public Object [] getServiceList() {
    System.out.println("in remote");
    return serviceList.keySet().toArray();
  }

  public Service getService(Object serviceKey) throws RemoteException {
    Service theService = (Service) serviceList.get(serviceKey);
    return theService;
  }

  public static void main (String [] args) {
    try {
      Naming.rebind("ServiceServer", new ServiceServerImpl());
    } catch(Exception ex) { }
    System.out.println("Remote service is running");
  }
}
```

UnicastRemoteObjectを継承する

サービスオブジェクトを格納するHashMapコレクション。このコレクションには、サービスオブジェクトそのものだけでなく、その名前（String）も格納される。HashMapの詳細については付録Bを参照

コンストラクタが呼ばれると、サービスオブジェクト（DiceService、MiniMusicServiceなど）が作成される

サービスオブジェクトを作り、名前（String）とともにHashMapに格納

クライアントは、このメソッドを呼び出してサービスのリストを取得する。取得したリストはブラウザの画面に表示される（ユーザーはリストからサービスを選択できる）。このメソッドの実行結果として戻されるのは、HashMapに格納されたサービスオブジェクトの「名前」のリスト（Object型の配列）。オブジェクトそのものは戻されない。オブジェクトそのものが戻されるのはgetService()が呼び出された時

ユーザーが、getServiceList()メソッドによって得られたリストからサービスを選択すると、クライアントはこのgetService()メソッドを呼び出す。メソッドの引数はサービスの名前。指定の名前に対応するサービスがHashMapから取り出され、クライアントに渡される

This PDF is prepared for tools@maruhouse.org, personal use only.

class ServiceBrowser(クライアント)

```
import java.awt.*;
import javax.swing.*;
import java.rmi.*;
import java.awt.event.*;

public class ServiceBrowser {

  JPanel mainPanel;
  JComboBox serviceList;
  ServiceServer server;

  public void buildGUI() {
    JFrame frame = new JFrame("RMI Browser");
    mainPanel = new JPanel();
    frame.getContentPane().add(BorderLayout.CENTER, mainPanel);

    Object [] services = getServicesList();

    serviceList = new JComboBox(services);

    frame.getContentPane().add(BorderLayout.NORTH,serviceList);

    serviceList.addActionListener(new MyListListener());

    frame.setSize(500,500);
    frame.setVisible(true);

  }

  void loadService(Object serviceSelection) {
    try {
      Service svc = server.getService(serviceSelection);

      mainPanel.removeAll();
      mainPanel.add(svc.getGuiPanel());
      mainPanel.validate();
      mainPanel.repaint();
    } catch(Exception ex) {
      ex.printStackTrace();
    }
  }
```

このメソッドでは、RMIレジストリの検索を行い、スタブを取得して、そのgetServiceList()メソッドを呼び出す(メソッドのコードは次のページ参照)

取得したサービスの名前(Objectの配列)をJComboBoxに組み込む。これで、コンボボックスにサービス名のリストが表示される

ユーザーがサービスを選択してから、サービスオブジェクトを取得し、そのサービスのGUIを表示する、という処理はこのメソッドで行う(このメソッドは、JComboBoxに対応するイベントリスナーによって呼び出される)。まず、サービスの名前(元はgetServiceList()メソッドによってサーバから取得したもの)を引数に指定して、リモートサービス(実はスタブ)のgetService()メソッドを呼び出す。すると、サーバからサービスに対応するオブジェクトがシリアライズされて送られてくる(RMIによってデシリアライズは自動的に行われる)ので、それを利用してgetGuiPanel()メソッドを呼び出す。これによって得られたJPanelオブジェクトをブラウザのメインパネルに組み込む

This PDF is prepared for tools@maruhouse.org, personal use only.

```
  Object [] getServicesList() {
    Object obj = null;
    Object [] services = null;

    try {

      obj = Naming.lookup("rmi://127.0.0.1/ServiceServer");

    }
    catch(Exception ex) {
      ex.printStackTrace();
    }
    server = (ServiceServer) obj;

    try {

      services = server.getServiceList();

    } catch(Exception ex) {
      ex.printStackTrace();
    }
    return services;

  }

  class MyListListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {

      Object selection = serviceList.getSelectedItem();
      loadService(selection);
    }
  }

  public static void main(String [] args) {
    new ServiceBrowser().buildGUI();
  }
}
```

RMIレジストリを検索し、スタブを取得

スタブはリモートインターフェースの型にキャストし、getServiceList()メソッドが呼び出せるようにする

getServiceList()メソッドを呼び出すと、Object型の配列が得られるので、それをJComboBoxに組み込んで画面に表示し、ユーザーがサービスを選択できるようにする

JComboBoxのリストでユーザーが選択をすると、このコードが実行され、選択されたサービスに対応するオブジェクトがサーバから送られる（loadService()メソッドによってgetService()メソッドが呼び出されるため）

This PDF is prepared for tools@maruhouse.org, personal use only.

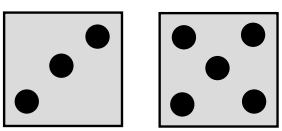
class DiceService
(「仮想サイコロ」サービスを提供するクラス)

```
import javax.swing.*;
import java.awt.event.*;
import java.io.*;

public class DiceService implements Service {

  JLabel label;
  JComboBox numOfDice;

  public JPanel getGuiPanel() {
    JPanel panel = new JPanel();
    JButton button = new JButton("Roll 'em!");
    String [] choices = {"1", "2", "3", "4", "5", "6"};
    numOfDice = new JComboBox(choices);
    label = new JLabel("dice values here");
    button.addActionListener(new RollEmListener());
    panel.add(numOfDice);
    panel.add(button);
    panel.add(label);
    return panel;
  }

  public class RollEmListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
      // サイコロを振る
      String diceOutput = "";
      String selection = (String) numOfDice.getSelectedItem();
      int numOfDiceToRoll = Integer.parseInt(selection);
      for (int i = 0; i < numOfDiceToRoll; i++) {
        int r = (int) ((Math.random() * 6) + 1);
        diceOutput += (" " + r);
      }
      label.setText(diceOutput);

    }
  }
}
```

重要なメソッド。Serviceインターフェースのメソッドのインプリメンテーション。ユーザーがサービスを選択すると、このメソッドが呼び出される。getGuiPanel()のコードは、JPanelを戻すという点さえ変えなければ、基本的にどのようなものにでもできる。たとえば、回転するサイコロの画像を画面に表示する、というようなことも可能

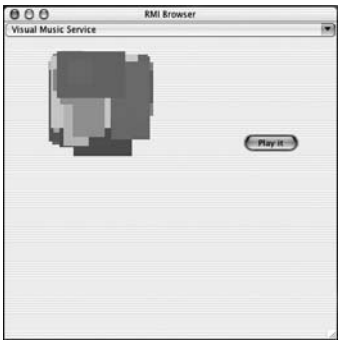
自分で考えてみよう

GUIに関して本書で学んだことを基に、回転するサイコロの画像を画面に表示できるよう、DiceServiceクラスを改良してみてください。まず、長方形を描き、その中に円を描く、という方法が簡単でしょう。円の数は、出た目によって変わることになります。



This PDF is prepared for tools@maruhouse.org, personal use only.

class MiniMusicService
(「ミュージックビデオ」サービスを提供するクラス)




```
import javax.sound.midi.*;
import java.io.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class MiniMusicService implements Service {

  MyDrawPanel myPanel;

  public JPanel getGuiPanel() {
    JPanel mainPanel = new JPanel();
    myPanel = new MyDrawPanel();
    JButton playItButton = new JButton("Play it");
    playItButton.addActionListener(new PlayItListener());
    mainPanel.add(myPanel);
    mainPanel.add(playItButton);
    return mainPanel;
  }

  public class PlayItListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {

      try {

        Sequencer sequencer = MidiSystem.getSequencer();
        sequencer.open();

        sequencer.addControllerEventListener(myPanel, new int [] {127});
        Sequence seq = new Sequence(Sequence.PPQ, 4);
        Track track = seq.createTrack();

        for (int i = 0; i < 100; i+= 4) {

          int rNum = (int) ((Math.random() * 50) + 1);
          if (rNum < 38) {//rNumの値が38より小さい場合（確率は75%）
            track.add(makeEvent(144,1,rNum,100,i));
            track.add(makeEvent(176,1,127,0,i));
            track.add(makeEvent(128,1,rNum,100,i + 2));
          }
        } // ループの終了

        sequencer.setSequence(seq);
        sequencer.start();
        sequencer.setTempoInBPM(220);
      } catch (Exception ex) {ex.printStackTrace();}

    } // actionperformed()の終了
  } // 内部クラスの終了
```

ユーザーがサービスを選択すると、このメソッドが呼び出され、ブラウザの画面にボタンが画面に表示される。ボタンがクリックされるとミュージックビデオが表示される

このあたりのコードについては12章で解説済み

This PDF is prepared for tools@maruhouse.org, personal use only.

class MiniMusicService(続き)

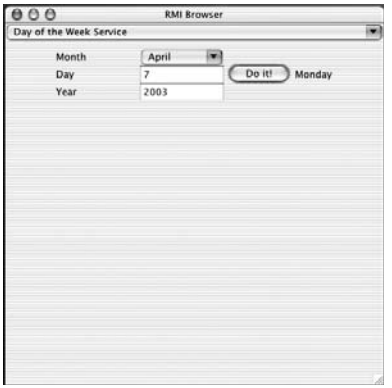
```
  public MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
    MidiEvent event = null;
      try {
        ShortMessage a = new ShortMessage();
        a.setMessage(comd, chan, one, two);
        event = new MidiEvent(a,tick);

      } catch(Exception e) { }
    return event;
  }

  class MyDrawPanel extends JPanel implements ControllerEventListener {

    // フラグはイベント発生時にのみ true になる
    boolean msg = false;

    public void controlChange(ShortMessage event) {
      msg = true;
      repaint();
    }

    public Dimension getPreferredSize() {
      return new Dimension(300,300);
    }

    public void paintComponent(Graphics g) {
      if (msg) {

        Graphics2D g2 = (Graphics2D) g;

        int r = (int) (Math.random() * 250);
        int gr = (int) (Math.random() * 250);
        int b =(int) (Math.random() * 250);

        g.setColor(new Color(r,gr,b));

        int ht = (int) ((Math.random() * 120) + 10);
        int width = (int) ((Math.random() * 120) + 10);

        int x = (int) ((Math.random() * 40) + 10);
        int y = (int) ((Math.random() * 40) + 10);

        g.fillRect(x,y,ht, width);
        msg = false;

      } // if 文の終了
    } // メソッドの終了
  } // 内部クラスの終了
} // クラスの終了
```

この部分のコードに関しても 12 章で解説済み。自分で注釈を付け、12 章のものと比較するのもよい練習になる

This PDF is prepared for tools@maruhouse.org, personal use only.

class DayOfTheWeekService

（任意の年月日を入力すれば、その曜日を教えてくれるというサービスを提供するためのクラス）




```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import java.io.*;
import java.util.*;
import java.text.*;

public class DayOfTheWeekService implements Service {

  JLabel outputLabel;
  JComboBox month;
  JTextField day;
  JTextField year;

  public JPanel getGuiPanel() {
    JPanel panel = new JPanel();
    JButton button = new JButton("Do it!");
    button.addActionListener(new DoItListener());
    outputLabel = new JLabel("date appears here");
    DateFormatSymbols dateStuff = new DateFormatSymbols();
    month = new JComboBox(dateStuff.getMonths());
    day = new JTextField(8);
    year = new JTextField(8);
    JPanel inputPanel = new JPanel(new GridLayout(3,2));
    inputPanel.add(new JLabel("Month"));
    inputPanel.add(month);
    inputPanel.add(new JLabel("Day"));
    inputPanel.add(day);
    inputPanel.add(new JLabel("Year"));
    inputPanel.add(year);
    panel.add(inputPanel);
    panel.add(button);
    panel.add(outputLabel);
    return panel;
  }

  public class DoItListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
      int monthNum = month.getSelectedIndex();
      int dayNum = Integer.parseInt(day.getText());
      int yearNum = Integer.parseInt(year.getText());
      Calendar c = Calendar.getInstance();
      c.set(Calendar.MONTH, monthNum);
      c.set(Calendar.DAY_OF_MONTH, dayNum);
      c.set(Calendar.YEAR, yearNum);
      Date date = c.getTime();
      String dayOfWeek = (new SimpleDateFormat("EEEE")).format(date);
      outputLabel.setText(dayOfWeek);
    }
  }
}
```

Serviceインターフェースのメソッドのインプリメンテーション。GUIを作成する

数値、日付のフォーマッティングに関しては10章を参照。ただ、ここではCalendarクラス、SimpleDateFormatクラスを使用している点に注意

This PDF is prepared for tools@maruhouse.org, personal use only.

おめでとうございます！

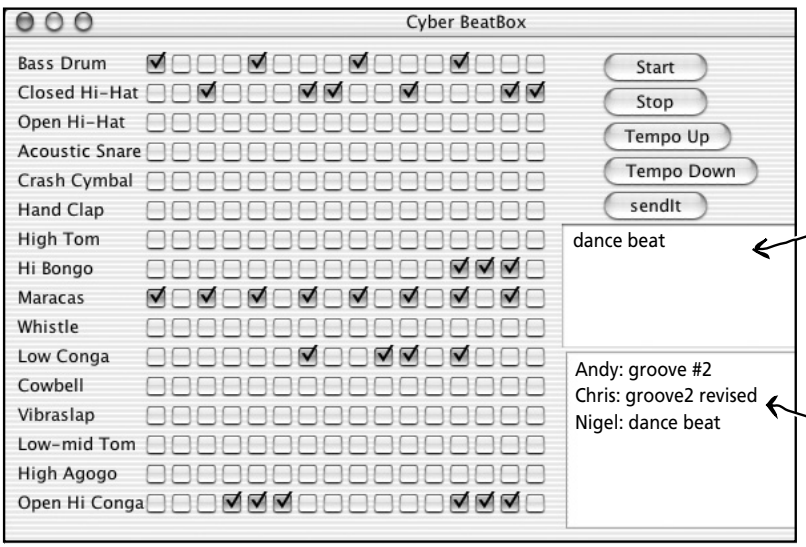
とうとう最後のページです

でも……、まだ付録が 2 つもあります。そちらにも必ず目を通してください。付録の内容もかなり充実していますので「おめでとう」はまだ早いかもしれません。

This PDF is prepared for tools@maruhouse.org, personal use only.

付録A

BeatBox プログラム完成版



作成したドラムパターンは「sendit」ボタンでサーバーに送れるが、送信する際には、ここに「このパターンはどのようなものであるか」というメッセージを書き込む

他のプログラムからドラムパターンとともにサーバーに送られてきたメッセージがここに表示される。メッセージをクリックすればパターンがロードされる。「Start」ボタンをクリックすればパターンが再生される

付録Aには、BeatBoxプログラムの完成版、そして、BeatBoxプログラムがドラムパターンを送受信するのに利用する簡単なサーバープログラムのコードを載せておきます。

This PDF is prepared for tools@maruhouse.org, personal use only.

BeatBoxプログラム完成版（クライアント）のコード

ここで紹介する「完成版」のコードには、本文中の「プログラム・キッチン」で紹介したものと重複する部分も多いので、同様の注釈を再度付ける、というようなことはしていません。「完成版」で追加されたのは主に次の部分です。

- GUI – コンポーネントを2つ追加。他のクライアントから送られてきたメッセージを表示するためのテキスト領域（スクロールリスト）と、メッセージを入力するためのテキストフィールド。
- ネットワーク機能 – 15章で紹介した簡単なチャットクライアントと同様、BeatBoxプログラムの完成版には、サーバと接続し、ストリームを利用してデータをやりとりする機能を持たせる。
- マルチスレッド – これも15章のチャットクライアントと同様。複数のスレッドを同時に利用する機能を持たせる。1つのスレッドでは、サーバから送られてくるメッセージとドラムパターンを待ち受け、読み取るという処理を行う。メッセージとドラムパターンは、StringとArrayListという2つのオブジェクトのかたちで（シリアライズされて）送られる。ArrayListには、ドラムパターン（全チェックボックスの状態）が格納される。

```
import java.awt.*;
import javax.swing.*;
import java.io.*;
import javax.sound.midi.*;
import java.util.*;
import java.awt.event.*;
import java.net.*;
import javax.swing.event.*;

public class BeatBoxFinal {

  JFrame theFrame;
  JPanel mainPanel;
  JList incomingList;
  JTextField userMessage;
  ArrayList<JCheckBox> checkboxList;
  int nextNum;
  Vector<String> listVector = new Vector<String>();
  String userName;
  ObjectOutputStream out;
  ObjectInputStream in;
  HashMap<String, boolean[]> otherSeqsMap = new HashMap<String, boolean[]>();

  Sequencer sequencer;
  Sequence sequence;
  Sequence mySequence = null;
  Track track;

  String[] instrumentNames = {"Bass Drum", "Closed Hi-Hat", "Open Hi-Hat", "Acoustic
  Snare", "Crash Cymbal", "Hand Clap", "High Tom", "Hi Bongo", "Maracas", "Whistle",
  "Low Conga", "Cowbell", "Vibraslap", "Low-mid Tom", "High Agogo", "Open Hi Conga"};

  int[] instruments = {35,42,46,38,49,39,50,60,70,72,64,56,58,47,67,63};
```


This PDF is prepared for tools@maruhouse.org, personal use only.

```
public static void main (String[] args) {
     new BeatBoxFinal().startUp(args[0]); // args[0] はユーザー ID/ スクリーンネーム
}

public void startUp(String name) {
     userName = name;
     // サーバへの接続
     try {
       Socket sock = new Socket("127.0.0.1", 4242);
       out = new ObjectOutputStream(sock.getOutputStream());
       in = new ObjectInputStream(sock.getInputStream());
       Thread remote = new Thread(new RemoteReader());
       remote.start();
     } catch(Exception ex) {
       System.out.println("couldn't connect - you'll have to play alone.");
     }
     setUpMidi();
     buildGUI();
} // startUp() の終了

public void buildGUI() {

     theFrame = new JFrame("Cyber BeatBox");
     BorderLayout layout = new BorderLayout();
     JPanel background = new JPanel(layout);
     background.setBorder(BorderFactory.createEmptyBorder(10,10,10,10));

     checkboxList = new ArrayList<JCheckBox>();

     Box buttonBox = new Box(BoxLayout.Y_AXIS);
     JButton start = new JButton("Start");
     start.addActionListener(new MyStartListener());
     buttonBox.add(start);

     JButton stop = new JButton("Stop");
     stop.addActionListener(new MyStopListener());
     buttonBox.add(stop);

     JButton upTempo = new JButton("Tempo Up");
     upTempo.addActionListener(new MyUpTempoListener());
     buttonBox.add(upTempo);

     JButton downTempo = new JButton("Tempo Down");
     downTempo.addActionListener(new MyDownTempoListener());
     buttonBox.add(downTempo);

     JButton sendIt = new JButton("sendIt");
     sendIt.addActionListener(new MySendListener());
     buttonBox.add(sendIt);

     userMessage = new JTextField();
```

コマンドライン引数（プログラムを実行する際にプログラム名の後に指定する引数）を受け付けるためのコード
コマンドライン引数の指定例
% java BeatBoxFinal theFlash

ネットワーク、I/O、スレッドなどに関連するコード。本文中で同様のコードについては説明済み

ごく一般的なGUI関連コード

This PDF is prepared for tools@maruhouse.org, personal use only.

```
    buttonBox.add(userMessage);

    incomingList = new JList();
    incomingList.addListSelectionListener(new MyListSelectionListener());
    incomingList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
    JScrollPane theList = new JScrollPane(incomingList);
    buttonBox.add(theList);
    incomingList.setListData(listVector);  // はじめはデータがない

    Box nameBox = new Box(BoxLayout.Y_AXIS);
    for (int i = 0; i < 16; i++) {
      nameBox.add(new Label(instrumentNames [i]));
    }

    background.add(BorderLayout.EAST, buttonBox);
    background.add(BorderLayout.WEST, nameBox);

    theFrame.getContentPane().add(background);
    GridLayout grid = new GridLayout(16,16);
    grid.setVgap(1);
    grid.setHgap(2);
    mainPanel = new JPanel(grid);
    background.add(BorderLayout.CENTER,mainPanel);

    for (int i = 0; i < 256; i++) {
      JCheckBox c = new JCheckBox();
      c.setSelected(false);
      checkboxList.add(c);
      mainPanel.add(c);
    } // ループの終了

    theFrame.setBounds(50,50,300,300);
    theFrame.pack();
    theFrame.setVisible(true);
  } // buildGUI() メソッドの終了

public void setUpMidi() {
  try {
    sequencer = MidiSystem.getSequencer();
    sequencer.open();
    sequence = new Sequence(Sequence.PPQ,4);
    track = sequence.createTrack();
    sequencer.setTempoInBPM(120);

  } catch(Exception e) {e.printStackTrace();}
} // setUpMidi() メソッドの終了
```

JListは本文中では紹介していないコンポーネント。サーバから送られるメッセージはこのコンポーネントに表示される。通常のチャットプログラムとは異なり、このメッセージはただ見るだけではなく、クリックすることで対応するドラムパターンのロード、演奏も行える

このページのその他のコードについてはすでに本文中で説明済み

Sequencerオブジェクトを取得し、Sequenceオブジェクト、Trackオブジェクトを作成する

This PDF is prepared for tools@maruhouse.org, personal use only.

```
public void buildTrackAndStart() {
  ArrayList<Integer> trackList = null; // 各楽器に対応する
  sequence.deleteTrack(track);
  track = sequence.createTrack();

  for (int i = 0; i < 16; i++) {

    trackList = new ArrayList<Integer>();

    for (int j = 0; j < 16; j++) {
      JCheckBox jc = (JCheckBox)checkboxList.get(j + (16*i));
      if (jc.isSelected()) {
        int key = instruments [i];
        trackList.add(new Integer(key));
      } else {
        trackList.add(null); // スロットは空に
      }
    } // 内側のループの終了
    makeTracks(trackList);
  } // 外側のループの終了
  track.add(makeEvent(192,9,1,0,15)); // パターンは常に最後まで演奏する
  try {
    sequencer.SetSequence(sequence);
    sequencer.getLoopCount(Sequencer.LOOP_COUTINUOUSLY);
    sequencer.start();
    sequencer.setTempoInBPM(120);
  } catch(Exception e) {e.printStackTrace();}
} // メソッドの終了

public class MyStartListener implements ActionListener {
  public void actionPerformed(ActionEvent a) {
    buildTrackAndStart();
  } // actionPerformed() の終了
} // メソッドの終了

public class MyStopListener implements ActionListener {
  public void actionPerformed(ActionEvent a) {
    sequencer.stop();
  } // actionPerformed() の終了
} // メソッドの終了

public class MyUpTempoListener implements ActionListener {
   public void actionPerformed(ActionEvent a) {
     float tempoFactor = sequencer.getTempoFactor();
     sequencer.setTempoFactor((float)(tempoFactor * 1.03));
   } // actionPerformed() の終了
} // 内部クラスの終了
```

チェックボックスの状態を１つ１つ調べ、Trackオブジェクトを作成し、対応する「楽器」を指定する（また、対応するMidiEventオブジェクトも作成する）。非常に複雑な処理だが、本文中で説明済み。詳しくは本文を参照のこと

GUIリスナー。これまでに紹介したものと特に変わるところなし

This PDF is prepared for tools@maruhouse.org, personal use only.

```
public class MyDownTempoListener implements ActionListener {
    public void actionPerformed(ActionEvent a) {
      float tempoFactor = sequencer.getTempoFactor();
      sequencer.setTempoFactor((float)(tempoFactor * .97));
    }
}
public class MySendListener implements ActionListener {
  public void actionPerformed(ActionEvent a) {
    // チェックボックスの状態を保持したArrayListを作成
    boolean [] checkboxState = new boolean [256];
    for (int i = 0; i < 256; i++) {
      JCheckBox check = (JCheckBox) checkboxList.get(i);
      if (check.isSelected()) {
        checkboxState [i] = true;
      }
    } // ループの終了
    String messageToSend = null;
    try {
      out.writeObject(userName + nextNum++ +":" +userMessage.getText());
      out.writeObject(checkboxState);
    } catch(Exception ex) {
      System.out.println("Sorry dude. Could not send it to the server.");
    }
    userMessage.setText("");
  } // actionPerformed()の終了
} // 内部クラスの終了

public class MyListSelectionListener implements ListSelectionListener {
  public void valueChanged(ListSelectionEvent le) {
    if (!le.getValueIsAdjusting()) {
      String selected = (String) incomingList.getSelectedValue();
      if (selected != null) {
        boolean [] selectedState = (boolean [])otherSeqsMap.get(selected);
        changeSequence(selectedState);
        sequencer.stop();
        buildTrackAndStart();
      }
    }
  } // valueChanged()の終了
} // 内部クラスの終了
```

「完成版」で追加したコード。すでに紹介したチャットクライアントに似ているが、ただStringのメッセージを送るのではなく、2つのオブジェクト（メッセージとドラムパターン）をシリアライズし、ストリーム、ソケット接続を利用してサーバに送るという点が異なる

これも追加したコード。ListSelectionListenerをインプリメントしたリスナーは、ユーザがメッセージを画面上のリストで選択した時に機能する。ユーザがメッセージを選択すると、そのメッセージに対応するドラムパターン（"otherSeqsMap" という名前のHashMapに格納されている）がロードされ、演奏が開始される。送られてきたListSelectionEventオブジェクトに関してif文による確認が必要な点に注意

This PDF is prepared for tools@maruhouse.org, personal use only.

```
public class RemoteReader implements Runnable {
    boolean[] checkboxState = null;
    String nameToShow = null;
    Object obj = null;
    public void run() {
      try {
      while((obj=in.readObject()) != null) {
    System.out.println("got an object from server");
    System.out.println(obj.getClass());
    String nameToShow = (String) obj;
    checkboxState = (boolean[]) in.readObject();
    otherSeqsMap.put(nameToShow, checkboxState);
    listVector.add(nameToShow);
    incomingList.setListData(listVector);
        } // while文の終了
      } catch(Exception ex) {ex.printStackTrace();}
    } // run()の終了
} // 内部クラスの終了
```

スレッドの「ジョブ」にあたるコード。ここでは、シリアライズされたデータをサーバから読み取るためのもの。「データ」とは具体的には、2つのオブジェクト（メッセージとドラムパターンに対応する）を指す。

2つのオブジェクト（Stringと、チェックボックスの状態を表すBoolean値を保持したArrayList）はデシリアライズし、JListコンポーネントに組み込む。ただし、JListにオブジェクトを直接組み込むわけではなく、オブジェクトのリストを保持するベクタを作り、それを表示すべきデータのリストとしてJListに登録する、というような方法を採る

```
public class MyPlayMineListener implements ActionListener {
    public void actionPerformed(ActionEvent a) {
      if (mySequence != null) {
    sequence = mySequence; // オリジナルに
        }
    } // actionPerformed()の終了
} // 内部クラスの終了
```

このメソッドは、ユーザーがリストからメッセージを選択した時に呼び出される。このメソッドにより、ドラムパターンはメッセージに対応するものに変更される

```
public void changeSequence(boolean[] checkboxState) {
    for (int i = 0; i < 256; i++) {
      JCheckBox check = (JCheckBox) checkboxList.get(i);
        if (checkboxState[i]) {
          check.setSelected(true);
        } else {
          check.setSelected(false);
        }
    } // ループの終了
} // changeSequence()の終了

public void makeTracks(ArrayList list) {
    Iterator it = list.iterator();
    for (int i = 0; i < 16; i++) {
      Integer num = (Integer) it.next();
      if (num != null) {
        int numKey = num.intValue();
        track.add(makeEvent(144,9,numKey, 100, i));
        track.add(makeEvent(128,9,numKey,100, i + 1));
      }
    } // ループの終了
} // makeTracks()の終了
```

MIDI関連のコード。前のバージョンと特に変わるところなし

This PDF is prepared for tools@maruhouse.org, personal use only.

```
  public MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
     MidiEvent event = null;
     try {
       ShortMessage a = new ShortMessage();
       a.setMessage(comd, chan, one, two);
       event = new MidiEvent(a, tick);
     }catch(Exception e) { }
     return event;
  } // makeEvent() の終了
} // クラスの終了
```

前のバージョンと特に変わるところなし

自分で考えてみよう

BeatBoxプログラム完成版に何か改良の余地はないか考えてみましょう。
参考までにいくつか改良の例を示しておきましょう。

1) 現状のコードでは、ユーザーがリストからメッセージを選ぶと、画面上のドラムパターンはどのようなものであろうとただちに破棄され、メッセージに対応するドラムパターンに入れ替わることになります。これでは、懸命に作っていた（あるいは修正していた）パターンをみすみす消してしまうということにもなりかねません。画面上のパターンを消去する前にダイアログボックスが表示され、保存するかどうかを尋ねるように改良すればこの問題は解決できます。

2) 現状のコードでは、プログラム実行の際にユーザーがコマンドライン引数を指定し忘れると、正しく実行されず、ただ例外が発生して終わるということになります。main()メソッドに、コマンドライン引数が指定されたかどうかをチェックし、指定されていなければ、デフォルト値を自動的に指定するか、コマンドライン引数を指定して再度実行するよう指示するメッセージを表示するコードを追加するとよいでしょう。

3) ボタンをクリックするだけでドラムパターンが自動的に作成される機能を追加する、というのも1つのアイデアです。クリックする度にまったく異なるパターンが無作為に作られるというのでもいいですし、あらかじめ「ジャズ」、「ロック」、「レゲエ」といった名前のついたパターンを用意しておき、それをボタンクリックだけで呼び出せる、というのも便利です。

This PDF is prepared for tools@maruhouse.org, personal use only.

BeatBoxサーバプログラム完成版

サーバプログラムのコードも以下に示しておきます。このコードは15章で紹介した簡単なチャットサーバとほぼ同じものです。ただ、受信、転送するのが文字列ではなく、2つの（シリアライズされた）オブジェクトであるという点が異なるだけです（2つのオブジェクトのうちの一方はStringなので結局は文字列なのですが……）。

```
import java.io.*;
import java.net.*;
import java.util.*;

public class MusicServer {

  ArrayList<ObjectOutputStream> clientOutputStreams;

  public static void main (String [] args) {
    new MusicServer().go();
  }

  public class ClientHandler implements Runnable {

    ObjectInputStream in;
    Socket clientSocket;

    public ClientHandler(Socket socket) {
      try {
        clientSocket = socket;
          in = new ObjectInputStream(clientSocket.getInputStream());

      } catch(Exception ex) {ex.printStackTrace();}
    } // コンストラクタの終了

    public void run() {
      Object o2 = null;
      Object o1 = null;
      try {

        while ((o1 = in.readObject()) != null) {

          o2 = in.readObject();

          System.out.println("read two objects");
          tellEeveryone(o1, o2);
        } // while文の終了

      } catch(Exception ex) {ex.printStackTrace();}
    } // run()メソッドの終了
  } // 内部クラスの終了
```

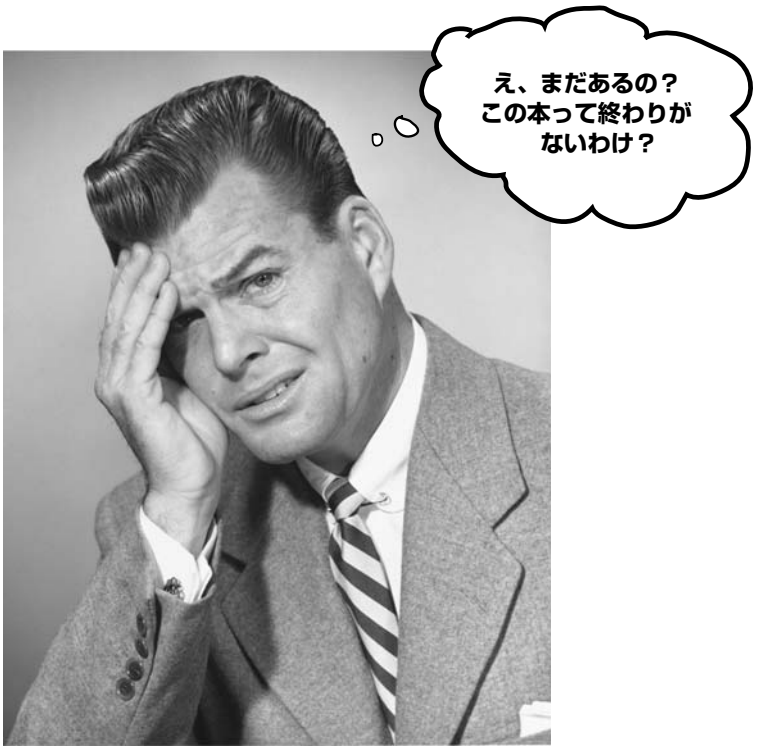

This PDF is prepared for tools@maruhouse.org, personal use only.

```
  public void go() {
    clientOutputStreams = new ArrayList<ObjectOutputStream>();

      try {
        ServerSocket serverSock = new ServerSocket(4242);

        while(true) {
          Socket clientSocket = serverSock.accept();
          ObjectOutputStream out = new ObjectOutputStream(clientSocket.getOutputStream());
          clientOutputStreams.add(out);

          Thread t = new Thread(new ClientHandler(clientSocket));
          t.start();

          System.out.println("got a connection");
        }
      } catch(Exception ex) {
        ex.printStackTrace();
      }
  } // go() メソッドの終了

  public void tellEveryone(Object one, Object two) {
    Iterator it = clientOutputStreams.iterator();
    while(it.hasNext()) {
      try {
        ObjectOutputStream out = (ObjectOutputStream) it.next();
        out.writeObject(one);
        out.writeObject(two);
      } catch(Exception ex) {ex.printStackTrace();}
    }
  } // tellEveryone() メソッドの終了

} // プログラムの終了
```


This PDF is prepared for tools@maruhouse.org, personal use only.

付録B

本文で触れなかった重要事項ベスト10



さて、いよいよ本当の終わりが近づいてきました。本書ではJavaについて実にさまざまなことを学んできましたが、もちろん、これで何もかもがわかったというわけではありません。まだまだ知っておいてもらいたいことはたくさんあります。ただ、著者の書きたいことをすべて書こうとすれば、残されたスペースではとても不可能です（文字を見えないくらいに小さくすればもしかすると可能かもしれませんが……）。そこで、中でも特に重要と思われることがらだけを10個選び、ヒットチャートのカウントダウンのようなかたちで取りあげることにしました。

この付録Bが終わればあとは索引を残すのみです（索引も、できれば是非読んでください）。

This PDF is prepared for tools@maruhouse.org, personal use only.

第⑩位　ビット操作

ビット操作の必要な理由

byte型のデータは8ビット、short型のデータは16ビットである、といったことは本文中で触れました。「ビット操作」というのは、この、データを構成する個々のビットの値（0か1）を意図的に変えることです。ビット操作は、あるビットの値が「1」ならば、機械のスイッチを「ON」にし、「0」ならば「OFF」にする、といったプログラムを書く際に応用ができます。ビット操作（あるいはビット演算）には次のような演算子（ビット演算子）が使えます（intは32ビットですが、読みやすいよう、コメントには末尾の8ビットだけを示してあります）。

NOT演算子（~）

全ビットの値を「反転」させる（0を1に、1を0にする）演算子。

```
byte x = 10; // 2進数で表すと00001010
x = ~x; // すべてのビットが反転して11110101になる
```

次に示す&、|、^という演算子は、いずれも2つの値（プリミティブ値）の個々のビットどうしを比較して、その結果を戻す、というものです。ここでは、例として以下の2つの値を使用します。

```
byte x = 10; // 2進数で表すと00001010
byte y = 6; // 2進数で表すと00000110
```

AND演算子（&）

2つの値の個々のビットどうしを比較し、両者が「1」になっているビットのみを「1」にした値を戻す。

```
byte a = x & y; // aには00000010が代入される
```

OR演算子（|）

2つの値の個々のビットどうしを比較し、少なくともどちらかが「1」になっているビットを「1」にした値を戻す。

```
byte a = x | y; // aには00001110が代入される
```

XOR（eXclusive OR）演算子（^）

2つの値の個々のビットどうしを比較し、どちらか一方のみが「1」になっているビットを「1」にした値を戻す。

```
byte a = x ^ y; // aには00001100が代入される
```

シフト演算子

シフト演算子は、ある値（プリミティブ値）のすべてのビットを一斉に特定の方向に「ずらす」という演算子です。数学の得意な人ならすぐにわかると思いますが、このシフト演算子をうまく使えば、数値を2倍、4倍、8倍……というように大きくすることや、逆に1/2倍、1/4倍、1/8倍……というように小さくしていくことが簡単にできます。

ここでは、例として以下のプリミティブ値を使用します。

```
byte x = -11; // 2進数で表すと11110101
```

負の数は、Javaでは、いわゆる「2の補数」を使って表します。正の数であるか、負の数であるかは、左端のビットを使って区別します。左端のビットが「1」であれば負の数、「0」であれば正の数ということです。また、正の数を負の数にするには、すべてのビットを反転した後、1を加える（つまりbyte型であれば00000001を加える）という処理が必要なので注意してください。byte型の「-11」は、「+11」が「00001011」なので「11110101」ということになります。

右シフト演算子（>>）

すべてのビットを指定された分だけ一斉に右にずらすという演算子です。左側のビットに空きができれば、その埋め合わせとして、元の値の左端のビットが1ならば1が、0ならば0が、自動的に入ります。右シフト演算で符号ビットが変化することはありません。

```
byte y = x >> 2; // yには11111101が代入される
```

符号なし右シフト演算子（>>>）

右シフト演算子とよく似ていますが、シフトによって左側のビットに空きができた場合に必ず「0」が入るという点が異なっています。この演算子を使った場合、符号ビットが変化することがあり得ます。

```
byte y = x >>> 2; // yには00111101が代入される（符号が変わる）
```

左シフト演算子（<<）

すべてのビットを指定された分だけ一斉に左にずらすという演算子です。右側のビットに空きができれば、その埋め合わせとして、自動的に「0」が入ります。符号ビットが変化することがあり得ます。

```
byte y = x << 2; // yには11010100が代入される
```


This PDF is prepared for tools@maruhouse.org, personal use only.

第9位　String、ラッパの不変性

Stringの不変性

Javaプログラムの規模が大きくなれば、必然的に使用するStringオブジェクトの数は増えます。Javaでは、セキュリティ上の理由から、またメモリの節約のために（Javaはメモリ容量の少ない携帯電話などにも利用されるという点に注意）、Stringオブジェクトには「不変性」という性質が持たされています。これがどのような意味を持つのかを次に例を使って説明してみましょう。

```
String s = "0";

for (int x = 1; x < 10; x++) {
  s = s + x;
}
```

上記のようなコードを実行すると、最終的に、10個のStringオブジェクト（それぞれが"0"、"01"、"012" ... "0123456789"という値を持つもの）ができることになります。変数sには、最終的に"0123456789"という値を持ったStringオブジェクトへの参照が格納されることになりますが、Stringオブジェクト自体は10個できるのです。

Stringオブジェクトが新たに作られると、JVMは必ずそれを、メモリ中の「Stringプール」と呼ばれる特殊な領域に格納します。String型の変数にはプール中のオブジェクトへの参照が納められます。もし、同じ値を持つStringオブジェクトに対応する変数が2つあったとしても、Stringオブジェクトは2つ作られるわけではなく、両方の変数に同じStringオブジェクトへの参照が納められるだけです。一方の変数を操作したとしても、もう一方の変数に影響を与えることはありません。変数の値に変更を加えても、対応するStringオブジェクトが変わるだけで（これは、たとえば、「0」という値を持ったStringオブジェクトに対応していたのが、「01」という値を持ったStringオブジェクトに対応するようになる、というようなことを意味する）、Stringオブジェクトの値そのものが変更されてしまうわけではないからです。

Stringプールには、「ガーベージコレクタが機能しない」という特徴もあります。いったん作られたStringオブジェクトは、たとえその後二度と使用しなくても、ずっとメモリ内にとどまるのです。

Stringの不変性がメモリの節約につながる理由

実は、プログラマが注意してプログラムを書かなければStringの不変性はメモリの節約にはつながりません。メモリの節約ができるのは、「不変性」の意味を十分に理解し、慎重にプログラムを書いた場合のみです。プログラム中で文字列を多用する場合には、StringよりもStringBufferというクラスを使用した方がメモリの節約につながることもあります。StringBufferについては後で詳しく触れます。

ラッパの不変性

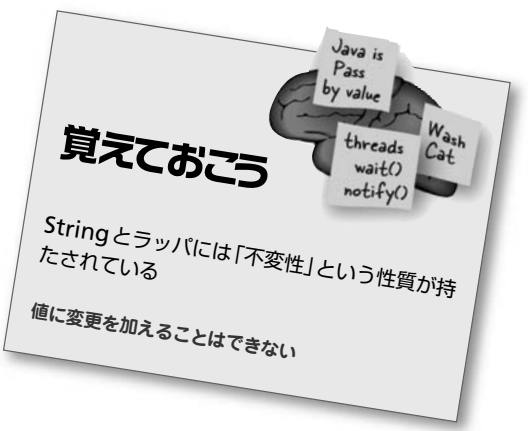
10章で述べたとおり、ラッパクラスには、主に次の2つの役割があります。

- プリミティブ型の値をオブジェクトとして扱えるようにする。
- 便利なユーティリティメソッド（スタティックメソッド）を提供する（Integer.parseInt()はそうしたユーティリティメソッドの例）。

ラッパオブジェクトは次のようなコードで作成します。

```
Integer iWrap = new Integer(42);
```

注意すべきなのは、ラッパオブジェクトに格納した値は後で変更できないということです。はじめに「42」を指定したら、後はずっと「42」のままなのです。ラッパオブジェクトにセッターメソッドはありません。もちろん、変数iWrapに別のラッパオブジェクトへの参照を格納するということはできますが、それでも元のラッパオブジェクトには何ら変更が加えられるわけではないのです。



This PDF is prepared for tools@maruhouse.org, personal use only.

第8位　assertステートメント

本書では、Javaプログラムの「デバッグ」についてはあまり触れませんでした。本書での学習には、Sunから提供されている開発環境「Java 2 SDK」を使用した人が多いと思います。今後は、IDE* と呼ばれる、さらに高度な開発環境を使用する人もいるでしょうが、そうした開発環境には、通常、独自のデバッグツールが用意されています。IDEを使用しないJavaプログラマのデバッグと言えば、以前はプログラム中のいたるところにSystem.out.println()というステートメントを入れる、という方法を採るのが一般的でした。このステートメントによって、変数の各地点における値や、「～を実行中（処理中）」といったメッセージなどが画面に出力されるようにし、「今、どの部分が実行されているのか」、「意図したとおりの流れで実行されているか」といったことがわかるようにしたのです（6章で紹介したGameHelperクラスのコードにはそうしたSystem.out.println()が入っていました）。プログラムが正しく動作することがわかれば、System.out.println()ステートメントは削除することになります。この作業はかなり面倒で、ミスも起きやすいものです。しかし、Java 1.4（5.0も）からは同様のことがかなり容易にできるようになっています。それは次に紹介するassertステートメントが使用できるためです。

assertステートメント

assertステートメントはSystem.out.println()ステートメントと同様のことを、もっと少ない手間でできるようにしたものです。プログラムのコードの中に埋め込むという点では、System.out.println()ステートメントとほぼ同じです。Java 5.0のコンパイラは、ソースファイルが5.0対応のものであるという前提でコンパイルをするので、"assert"というキーワードを使っただけで、オプションの指定などなしにデフォルトでassertステートメントであると解釈されます。

また、プログラム実行時にも、オプションを指定し、assertステートメントを使用していることをJVMに知らせる必要があります。これをしなかった場合、assertステートメントはJVMに無視されます（その分だけプログラムの実行は速くなる）。assertステートメントを使えば、他のコードにまったく手を加えることなくデバッグができます。また、デバッグ完了後も、オプションを省略すれば、assertステートメントは無視され、プログラムを普通に実行できるので、いちいちassertステートメントを削除する必要はありません。削除に伴うミスが発生しないだけでなく、プログラムが提供先でトラブルを起こした際にも、assertステートメントを使用するためのオプションを指定して実行し、その結果を見るだけで、即、デバッグができるので便利です。

assertステートメントの使い方

assertステートメントは、「これが成り立たなければプログラムは正しく動作しない」という条件が成り立っているかどうかを確認するために使用します。使用例を以下に示します。

```
assert (height > 0);
// height の値が 0 より大きいという条件が成り立てば（true であれば）プログラムは正しく動作する
// この条件が成り立たなければ、AssertionError という例外が発生する
```

次のように書くことで、スタックトレースに入る情報を増やすことも可能です。

```
assert (height > 0) : "height = " +
height + " weight = " + weight;
```

コロン（:）の後には、演算結果として得られる値がnullにならない、文法的に正しい式ならばどのようなものでも書くことができます。assertステートメントを作る際、注意すべきなのは、オブジェクトのステートを変えるようなものにはしない、ということです。そのようなassertステートメントを作ってしまうと、assertステートメントを有効にした場合としなかった場合とで、プログラムの動きが変わってしまうからです。

プログラムのコンパイル、実行

assertステートメントの入ったプログラムをコンパイルするには、以下のようなコマンドを使用します。

```
javac TestDriveGame.java
（オプションの指定が行われていない点に注意）
```

assertステートメントの入ったプログラムを、（assertステートメントを有効にして）実行するには、以下のようなコマンドを使用します。

```
java -ea TestDriveGame
```

* IDE（Integrated Development Environment）の代表例としては、Eclipse、BorlandのJBuilder、オープンソースのNetBeans（http://www.netbeans.org/で入手可能）などがある。

This PDF is prepared for tools@maruhouse.org, personal use only.

第7位　ブロック変数

ローカル変数は、9章で述べたとおり、宣言されたメソッドがスタック上に存在する間は有効です。しかし、変数の中には、もっと「寿命」の短いものもあります。メソッド中の{から}までの間（本文中では触れなかったが、これを「ブロック」と呼ぶ）のみ有効、という「ブロック変数」がそれです。つまり、ある（for、while文を使った）ループの処理が行われている間のみ有効な変数、特定の（if文の）条件が満たされた場合にのみ有効になる変数、などが作れるわけです。

ブロック変数の例を以下に示します。

```
void doStuff() {
  int x = 0;
  for(int y = 0; y < 5; y++) {

    x = x + y;
  }

  x = x * y;

}
```

- `void doStuff() {` ← メソッドの始まり
- `int x = 0;` ← doStuff()メソッド全体で有効なローカル変数
- `for(int y = 0; y < 5; y++) {` ← ループブロックの始まり。yはループブロック内だけで有効なブロック変数
- `x = x + y;` ← xもyもここでは有効
- `}` ← ループブロック終了
- `x = x * y;` ← yはここでは無効なので、このコードはコンパイルでエラーになる（Java以外の言語ではこのようなコードが許容されることもあるので注意）
- `}` ← メソッドの終了。これ以降、xも無効になる

上記の例では、yがブロック変数になっています。ループブロックの中で宣言されており、ループが終了すると無効になります。インスタンス変数よりはローカル変数、ローカル変数よりはブロック変数をできる限り使った方が、プログラムはデバッグしやすくなります。有効でない変数をうっかり使ってしまってもコンパイル時にエラーになるので、実行時にそれが発覚して大変な問題になるということはありません。

This PDF is prepared for tools@maruhouse.org, personal use only.

第❻位　リンク呼び出し

本書ではさまざまなことがらについて解説してきましたが、「ショートカット」と呼ばれる技法については触れてきませんでした。プログラムのコードが読みにくくなることを避けたかったからです。ただ、実際にプログラミングを始めると、自分ではショートカットを使わなくても、他人がショートカットを使って書いたコードを読まねばならない、ということがどうしても出てきます。ここで、ショートカットの中でも特によく使われる「リンク呼び出し」について、ひととおり説明しておきましょう。リンク呼び出しを使ったコードの例を以下に示します。

```
StringBuffer sb = new StringBuffer("spring");
sb = sb.delete(3,6).insert(2, "umme").deleteCharAt(1);
System.out.println("sb = " + sb);
// 実行結果は「sb = summer」になる
```

「リンク呼び出し」が使われているのは2行目です。あくまで例として作ったコードですが、リンク呼び出しについてはひととおりわかるはずです。上記のリンク呼び出しは以下のように実行されます。

1— 左のメソッドから順に実行される

2— sb.delete(3,6)が実行されると、その結果としてStringBufferオブジェクトが戻される（詳しくはAPIのドキュメントのStringBufferクラスのページを参照）。ここで戻されるStringBufferオブジェクトは「spr」という値を保持することになる。

3— 次に、delete(3,6)の実行結果として得られたStringBufferオブジェクト（「spr」という値を保持するもの）を使って、insert(2, "umme")メソッドが呼び出される。insert()の戻り値もStringBufferオブジェクトになる（ここでは同じ型の値を戻すメソッドだけをリンクしているが、戻り値の型の異なるメソッドもリンクできるので注意）。このStringBufferオブジェクトは次のメソッド呼び出しに利用される。1つのリンク呼び出しで呼び出せるメソッドの数に制限はない（ただし、4つ以上のメソッドが呼び出されることはまずない）。上記のリンク呼び出しを普通のコードに書き直すと以下のようになる。

```
sb = sb.delete(3,6);
sb = sb.insert(2, "umme");
sb = sb.deleteCharAt(1);
```

実のところ、上記の例のようなリンク呼び出しの使い方はあまり一般的とは言えません。次に、より一般的な使い方の例を示しておきましょう。すでに同様のテクニックは本書の中で使用していますが、ここで改めて確認しておいて下さい。下の例の場合、main()メソッドの中で、同じクラスに属するインスタンスメソッドを呼び出していますが、この場合、インスタンスへの参照を変数に格納するということが必要ありません。インスタンスへの参照を作らず、インスタンスのみ作成すればインスタンスメソッドが呼び出せるということです。

```
class Foo {
  public static void main(String [] args) {
    new Foo().go();
  }
  void go(){
    // メソッドのコード
  }
}
```

go()を呼び出すのに、Fooのインスタンスへの参照を変数に格納するということは必要ない

This PDF is prepared for tools@maruhouse.org, personal use only.

第5位　匿名ネストクラスとスタティックネストクラス

多様なネストクラス

ネストクラス（内部クラス）については、本文中でGUIやイベント処理について解説した際に少し触れました。リスナーインターフェースをインプリメントする手段の1つとして取りあげたのです。単に、あるクラスの{}の中にもう1つのクラスが入れ子になっている（ネストされている）という形態のものでしたが、ネストクラスの使用形態としては、それが最も一般的、実用的で、コードも読みやすいものになると言えます。この場合、注意しなくてはならないのは、ネストクラスのインスタンスを得るには、まず、外部クラスのインスタンスを作らねばならないということです。ネストクラスは、外部クラスの「メンバ」だからです。

ネストクラスには、それ以外に「スタティックネストクラス」、「匿名ネストクラス」などと呼ばれるものがあります。そうしたネストクラスについて、ここで詳しく解説することはしませんが、他人の書いたコードの中に含まれていても戸惑わないよう、簡単に触れておきます。実はJavaの中でも「匿名ネストクラス」ほどコードをわかりにくくするものは少ないのですが、まずはどちらかと言えばやさしいスタティックネストクラスから見てみましょう。

スタティックネストクラス

ここまで読んできた人であれば「スタティック」の意味はわかるでしょう。インスタンスごとに存在するのではなく、同じクラスのオブジェクトすべてによって共有されるという意味です。スタティックネストクラスは、これまでに紹介してきたイベントリスナーのような「スタティックでない」クラスと、見た目にはほとんど同じです。"static"というキーワードが使われている以外、違いがあるようには見えません。

```
public class FooOuter {
  static class BarInner {
    void sayIt() {
      System.out.println("method of a static inner class");
    }
  }
}

class Test {
  public static void main (String[] args) {
    FooOuter.BarInner foo = new FooOuter.BarInner();
    foo.sayIt();
  }
}
```

スタティックネストクラスは、他のクラスにネストされたクラス。staticというキーワードが使われる

スタティックネストクラスは、名前のとおり「スタティック」なもので、外部クラスのインスタンスがなくても使用できる。スタティックメソッドを呼び出す場合や、スタティック変数にアクセスする場合と同様、クラスの名前を使用すればよい

スタティックネストクラスは、一般のネストクラスのように、オブジェクトが外部クラスのオブジェクトと緊密に結び付くわけではありません。その点では、通常の（ネストクラスでない）クラスに似ていると言えます。とはいえ、外部クラスのprivateメンバにアクセスが可能という点で、やはり通常のクラスとは異なってきます。スタティックネストクラス自体が外部クラスのメンバだからです。ただし、アクセス可能なのは、スタティックなメンバに限ります。スタティックネストクラスは、外部クラスのインスタンスとの結び付きがなく、スタティックでない変数（インスタンス変数）やメソッドへのアクセスができないのです。

This PDF is prepared for tools@maruhouse.org, personal use only.

第5位　匿名ネストクラスとスタティックネストクラス（続き）

ネストクラスと内部クラスの違い

「ネストクラス」というのは、他のクラスのスコープ内で定義されているクラスのことです。一般のネストクラスだけでなく、匿名ネストクラス、スタティックネストクラスもそうです。ともかく、他のクラスの中にあれば、ネストクラスであると言えます。「内部クラス」と呼ばれるのは、ネストクラスの中でも「スタティックでない」ものです。この種のクラスについては本文ですでに触れました。簡単にまとめれば、内部クラスはすべてネストクラスだが、ネストクラスは内部クラスであるとは限らない、ということになります。

匿名ネスト（内部）クラス

GUIのコードを書いていて突然、ActionListenerをインプリメントするクラスのインスタンスが必要だと気付いたとしましょう。しかし、そんなインスタンスはなく、そもそもActionListenerをインプリメントするクラスすら書いていない。そんな場合の対応策としては以下の2つが考えられます。

1) 本書でGUIについて解説した時にすでに紹介したような内部クラスを書き、それをインスタンス化して、リスナー登録用メソッド（addActionListener()）に引数として渡す。

2) 匿名ネスト（内部）クラスを作成して即、インスタンス化する。つまり、リスナーオブジェクトが必要になった時点でクラスを作ってインスタンス化し、それを利用するということ。以下の例を見るとわかるが、これは、通常であればインスタンスのみを指定するメソッドの引数に、クラスのコードすべてを指定するということ。

```
import java.awt.event.*;
import javax.swing.*;
public class TestAnon {
  public static void main (String[] args) {

    JFrame frame = new JFrame();
    JButton button = new JButton( "click" );
    frame.getContentPane().add(button);
    // button.addActionListener(quitListener);
```

フレームを作成してボタンを追加している。その後は、ボタンに対応するリスナーの登録が必要だが、ActionListenerインターフェースをインプリメントするクラスはまだ作成していない

通常であれば、内部クラスのインスタンスへの参照を引数に指定し、このようなコードを書くことになる。内部クラスはActionListener（そしてactionPerformed()メソッド）をインプリメントするもの

このステートメントは、

```
    button.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent ev) {
        System.exit(0);
    }
    });
  }
}
```

ここまで続いている

ここで、"new ActionListener()"と書いている点に注意。ActionListenerはインターフェースであり、正確にはそのインスタンスを作ることはできない。このコードは、「ActionListenerインターフェースをインプリメントする新たなクラス（名前はない）を作る。インターフェースのメソッド、actionPerformed()のインプリメンテーションもここに書く」ということを意味する

しかし、ここでは、オブジェクトへの参照ではなく、クラスのコード全体を引数に指定している。引数の中に、ActionListenerをインプリメントするクラスのコードすべてを書いている。そして、そのクラスのインスタンスを作るコードも書いている

This PDF is prepared for tools@maruhouse.org, personal use only.

第4位　アクセスレベルとアクセス修飾子

Javaには4つのアクセスレベルと3つのアクセス修飾子があります。アクセス修飾子が3つしかないのは、アクセスレベル「default」には修飾子が必要ないからです。

4つのアクセスレベル（下へいくほどアクセスの制限が厳しくなります）

public ← アクセスレベルがpublicのクラス、変数、メソッド、コンストラクタなどには、どのコードからでもアクセスが可能

protected ← ほとんどdefaultと同じ（同じパッケージのコードからしかアクセスできない）だが、他のパッケージのサブクラスによって継承できる点が異なる）

default ← アクセスレベルdefaultのクラス、変数、メソッド、コンストラクタなどには、同じパッケージのクラスのコードからしかアクセスできない

private ← アクセスレベルprivateのクラス、変数、メソッド、コンストラクタなどには、同じクラス内のコードからしかアクセスできない。クラスが同じであれば、インスタンスが異なってもかまわないという点に注意。つまり、Dogオブジェクトから、他のDogオブジェクトのprivateメンバにアクセスすることは可能

3つのアクセス修飾子

```
public
protected
private
```

一般的に使用されるアクセス修飾子はpublicとprivateの2つです。

public

この修飾子は、あらゆるコードから使用できるようにしたいクラス、定数（static final変数）、メソッド（セッターメソッド、ゲッターメソッドなど）に利用します。コンストラクタのほとんどにはこの修飾子が使用されます。

private

この修飾子は、インスタンス変数のほぼすべて、そして、別のクラスのコード（他のクラスのpublicメソッド）から呼び出させたくないメソッドに使用します。

protected、defaultというアクセスレベルを使うことはまずありませんが、自分が使わなくても、他の人が書いたコードに使われることもあるので、一応どのようなものであるか知っておく必要があります。2つのアクセスレベルについては次で触れます。

This PDF is prepared for tools@maruhouse.org, personal use only.

第❹位　アクセスレベルとアクセス修飾子（続き）

default と protected

default

defaultとprotectedという2つのアクセスレベルは、いずれも「パッケージ」との関連が深いという点で共通しています。特に、defaultの方は、「同じパッケージに属するクラスのコードからのみアクセスできる」ということなので非常にわかりやすいと言えます。つまり、たとえば、アクセスレベルがdefaultのクラス（宣言に修飾子がついていないクラス）には、その同じクラスと同じパッケージに属するクラスのコードからしかアクセスできないということになります。

「クラスにアクセスできない」というのは、具体的には、まず、そのクラスをインスタンス化できないということを意味します。また、変数や引数、戻り値の型としてもそのクラスを使うことはできません。そのクラスを使うことはどのようなかたちであれ禁じられるのです。少しでも使用してしまうとコンパイル時にエラーになります。

これは、たとえpublic宣言されたメソッドであっても、アクセスレベルがdefaultのクラスに属している場合は、厳密にはpublicメソッドと言えないということを意味します。クラスにアクセスできなければ、それに属するメソッドにもアクセスできないからです。

アクセスできるコードを同じパッケージに属するものに限定するというのはどのような理由からでしょうか。通常、パッケージは、関係の深い、連携して機能することの多いクラスを集めて作られます。したがって、同じパッケージに属するクラスどうしのアクセスができるというのは理にかなっています。また、他のパッケージに属するクラスからアクセスできるクラスやメソッドの数を限定するというのも妥当なことと言えるでしょう。

これで「default」というアクセスレベルについては理解できたでしょうか。アクセス修飾子がまったく使われていなければ、そのクラス、変数、メソッド、内部クラスなどには、同じパッケージに属するクラスのコードからしかアクセスできないということです。

次に、「protected」というアクセスレベルについて見てみましょう。

protected

protectedはdefaultとほぼ同じなのですが、同じパッケージに属するクラスのサブクラスのコードからでもアクセスができる、という点が異なっています。サブクラス自体は属するパッケージが異なってもかまわないのです。protected宣言されたクラス、メソッド、コンストラクタ、変数などには、同じパッケージに属するクラスだけでなく、それを継承するサブクラス（他のパッケージに属するものも含む）からでもアクセスできるということです。

protectedを使用しなければならないという場面はほとんどないのですが、プログラムの設計によっては、protectedでなければならないということもまれにあります。注意すべきなのは、サブクラスからの「アクセス」は、「継承」というかたちに限定されるということです。他のパッケージに属するサブクラスで、スーパークラスのインスタンスへの参照を使い、protected宣言されたメソッド等にアクセスすることはできないのです。protected宣言されたメソッド等を直接利用するのではなく、それを継承するというだけのことです。つまり、厳密には、サブクラスはprotected宣言されたメソッド等にアクセスできる、というより、それを継承できるだけ、と言うべきでしょう。

This PDF is prepared for tools@maruhouse.org, personal use only.

第❸位　StringとStringBuffer/StringBuilderのメソッド

StringとStringBufferは、Java APIのクラスの中でも最もよく使用されるものです。

第9位「String、ラッパの不変性」でも述べたとおり、プログラム中で文字列を多用する場合にはStringよりもStringBuffer（あるいはStringBuilder）を使用した方がメモリの節約につながることもあります。Java 5.0では、文字列操作を「StringBufferよりもStringBuilderを使用した方がいいでしょう。スレッドセーフな文字列操作が必要になる場合はそう多くありません。以下にString、StringBuffer、StringBuilderの主要なメソッドの概要を簡単にまとめておきます。

StringとStringBuffer/StringBuilderに共通するメソッド

```
char charAt(int index);                    // 指定のインデックスに対応する文字を戻す
int length();                              // 文字列の長さを戻す
String substring(int start, int end);      // 文字列の一部を取り出す
String toString();                         // String、StringBufferに対応する文字列を戻す
```

文字列の連結に使用するメソッド

```
String concat(string);                     // Stringクラスのメソッド
String append(String);                     // StringBufferあるいはStringBuilderクラスのメソッド
```

Stringクラスのメソッド

```
String replace(char old, char new);        // 指定の文字列をすべて置換する
String substring(int begin, int end);      // 文字列の一部を取り出す
char [] toCharArray();                     // charの配列に変換する
String toLowerCase();                      // すべての文字を小文字に変換する
String toUpperCase();                      // すべての文字を大文字に変換する
String trim();                             // 先頭、末尾の空白を削除する
String valueOf(char [])                    // charの配列をStringに変換する
String valueOf(int i)                      // プリミティブ型データをStringに変換する
                                           // int以外のプリミティブ型もサポートされている
```

StringBuffer/StringBuilderクラスのメソッド

```
StringBxxxx delete(int start, int end);                    // 文字列の一部を削除する
StringBxxxx insert(int offset, any primitive or a char []); // 文字列に新たに文字を挿入する
StringBxxxx replace(int start, int end, String s);         // 文字列の一部を指定どおりに置換する
StringBxxxx reverse();                                     // 文字の並び順を反転させる
void setCharAt(int index, char ch);                        // 指定の文字を置換する
```

注意：StringBxxxxは、StringBuffer、StringBuilderの両方を指します。

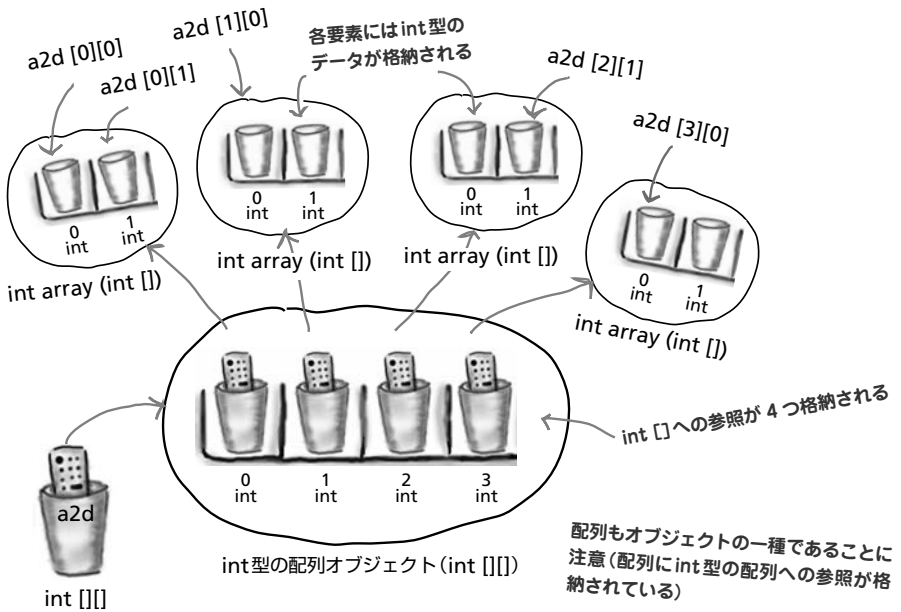
This PDF is prepared for tools@maruhouse.org, personal use only.

第❷位　多次元配列

通常、多次元配列というと、頭に長方形を思い浮かべる人が多いでしょう。たとえば、2 × 4 の 2 次元配列なら、合計 8 つの要素が並んでいる状態を思い浮かべるわけです。しかし、Java の場合は少し事情が異なり、2 × 4 の 2 次元配列は、5 つの配列から構成されます。Java では、2 次元配列が、「配列の配列」という構造になるためです（3 次元配列なら「配列の配列の配列」になります。4 次元配列、5 次元配列などがどうなるかは考えてみてください）。例を以下に示します。

```
int [][] a2d = new int [4][2];
```

この場合、JVM は、要素数 4 つの配列を作ります。4 つの要素はいずれも、要素数 2 つの配列への参照です。



多次元配列の使い方

- 3 番目の配列の 2 番目の要素にアクセスするためのコード：
  `int x = a2d [2][1]; // 要素の番号が 0 から始まる点に注意`
- 4 つの配列のうちの 1 つへの参照を変数に代入する：`int [] copy = a2d [1];`
- 2 × 3 の 2 次元配列の宣言、初期化を同時に行う：`int [][] x = { { 2,3,4 }, { 7,8,9 } };`
- 以下のようなコードも書ける：

```
int [][] y = new int [2][]; // 配列への参照を 2 つ格納する配列を作る
y [0] = new int [3]; // 1 つ目の参照に対応する要素数 3 つの配列を作る
y [1] = new int [5]; // 2 つ目の参照に対応する要素数 5 つの配列を作る
```


This PDF is prepared for tools@maruhouse.org, personal use only.

第❶位　列挙型（あるいはenum）

JFrame.EXIT_ON_CLOSEなど、APIにあらかじめ定義されている定数についてはすでに触れました。static finalを使用して変数を宣言すれば自ら定数を作ることも可能です。しかし、時には、「あらかじめ決めておいた複数の値に対応できる定数を作りたい」という場合もあります。この種の定数を「列挙型の定数」と呼びます。以前のJavaでは、列挙型の定数を完全なかたちで作ることはできませんでしたが、Java 5.0になってそれができるようになりました。

バンドのメンバー

気に入っているバンドがあって、そのバンドのWebサイトを作るとします。バンドのメンバーの名前を選択することで、サイトの機能が変わるようにしたい、という場合にはどうすればいいでしょうか。

Java 5.0より前のJavaを使った場合：

```
public static final int JERRY = 1;
public static final int BOBBY = 2;
public static final int PHIL = 3;
// 中略 ...
if (selectedBandMember == JERRY) {
  // JERRYに対応する処理を行うコード
}
```

この時点で"selectedBandMember"の値が「妥当」なものであるとは限らない

このコードは読みやすいところが長所と言えます。また、用意しておいた定数の値が決して変わらないのも長所と言えるでしょう。たとえば、JERRYは常に"1"です。ただ、問題は、selectedBandMemberの値が"1"、"2"、"3"のうちのいずれかである保証はなく、それを確かめるのも簡単ではないということです。仮にselectedBandMemberの値が知らない間に"812"になっていたとしたら、コードは正しく機能しません。

This PDF is prepared for tools@maruhouse.org, personal use only.

第❶位　列挙型（続き）

次に、Java 5.0 の enum を使用して同様のコードを書いてみましょう。非常に簡単なコードですが、列挙型を使用したコードの大半は、ほぼこれと同じくらい簡単なものです。

Java 5.0 の列挙型を使った場合：

このコードは簡単なクラス定義のようにも見える。実際、enum は特殊なクラスであると言える。このコードでは、"Members" という名前の新たな列挙型を作っていることになる

```
public enum Members { JERRY, BOBBY, PHIL };
public Members selectedBandMember;

// 中略 ...

if (selectedBandMember == Members.JERRY) {
  // JERRY に対応する処理を行うコード
}
```

この変数の値が妥当なものにならない心配はない

"selectedBandMember" は Members 型の変数であり、値は "JERRY"、"BOBBY"、"PHIL" のいずれかにしかならない

列挙型のインスタンスを使用するコード

列挙型は java.lang.Enum を継承するクラス

列挙型を作ると、自動的に Java.lang.Enum を継承することになります。列挙型は、他から独立した単独のクラスとして宣言することも、他のクラスのメンバとして宣言することもできます。

if 文、switch 文とともに列挙型を使用する

if 文や switch 文を使用した分岐のコードに列挙型を使ってみましょう。列挙型のインスタンスの比較を、== と .equals() メソッドの両方で行っている点に注意してください。通常は == を使用するのが望ましいとされます。

列挙型のインスタンスを変数に代入

```
Members n = Members.BOBBY;
if (n.equals(Members.JERRY)) System.out.println("Jerrrry!");
if (n == Members.BOBBY) System.out.println("Rat Dog");

Members ifName = Members.PHIL;
switch (ifName) {
  case JERRY: System.out.print("make it sing");
  case PHIL: System.out.print("go deep");
  case BOBBY: System.out.println("Cassidy!");
}
```

どちらも正しい "Rat Dog" と出力される

問題：出力はどうなるか？

答え："go deep Cassidy!"

This PDF is prepared for tools@maruhouse.org, personal use only.

第❶位　列挙型（続き）

より複雑な列挙型

列挙型には、通常のクラスのコンストラクタ、メソッド、変数のようなコードを加えることができ、また個々の値に固有のコードを付け加えることもできます。あまり一般的ではありませんが、他人の書いたプログラムにそのような列挙型が出てくることもあるので覚えておいた方がいいでしょう。

```
public class HfjEnum {
  enum Names {
    JERRY("lead guitar") { public String sings() {
                           return "plaintively"; }
                         },
    BOBBY("rhythm guitar") { public String sings() {
                           return "hoarsely"; }
                         },
    PHIL("bass");

    private String instrument;

    Names(String instrument) {
    this.instrument = instrument;
    }
    public String getInstrument() {
    return this.instrument;
    }
    public String sings() {
    return "occasionally";
    }
  }

  public static void main(String [] args) {
      for (Names n : Names.values()) {
      System.out.print(n);
      System.out.print(", instrument: "+ n.getInstrument());
      System.out.println(", sings: " + n.sings());
    }
  }
}
```

後で宣言されるコンストラクタに渡される引数

値に固有のコード。列挙型の基本形のメソッド（ここではsing()）のオーバーライドバージョンと考えることもできる。sing()の機能は、JERRYを使って呼び出された場合と、BOBBYを使って呼び出された場合とで変わってくる

列挙型のコンストラクタ。値が宣言される度に実行される（ここでは3度実行されることになる）

main()から呼び出されるメソッド

列挙型には必ずvalues()メソッドがある。以下のようにforループに使用されるのが普通

```
File  Edit  Window  Help  Bootleg
%java HfjEnum
JERRY, instrument: lead guitar, sings: plaintively
BOBBY, instrument: rhythm guitar, sings: hoarsely
PHIL, instrument: bass, sings: occasionally
%
```

基本形のsing()メソッドは、固有のコードがない値が使用された場合にのみ呼び出される

This PDF is prepared for tools@maruhouse.org, personal use only.

地球への遠い道

宇宙船「Traverser」のByte船長は、司令部から極秘の緊急指令を受けた。指令のメッセージには、Traverserが敵の領内を通過して無事帰還するために必要な30の航行コードが含まれ、厳重に暗号化されていた。ただ、問題は、近隣の銀河からやって来る敵、Hackarian星人が、Traverserの1つしかない航行コンピュータのヒープ上に偽物のオブジェクトを作る妨害電波を発生させる装置を開発していることだ。また、彼らには、変数を操作し、変数に格納される参照を、偽物のオブジェクトに対応するものに変えてしまうということもできる。それを防ぐには、Java 1.4で作られたTraverserの最新鋭航行プログラムに組み込んだインラインウイルスチェッカーを走らせるしかない。

Byte船長は、乗組員のEnsign Smithに、航行コードを処理するためのプログラム作成を命じ、以下のような指示を与えた。

「まず、最初の5つのコードは、ParsecKey型の配列に入れて欲しい。残りの25個のコードは、QuadrantKey型の、5×5の2次元配列に入れるんだ。2つの配列はどちらもpublic finalクラス“ShipNavigation”のplotCourse()メソッドに渡すようにする。コースオブジェクトが戻ってきたら、インラインウイルスチェッカーを走らせてプログラム中のすべての参照型変数をチェックしてから、NavSimプログラムを実行して、その結果を知らせてくれ」

数分後、Ensign Smithは、NavSimプログラムの実行結果を持って戻ってきた。「船長、NavSimプログラムの実行結果、いつでも確認できます」とSmith。すると船長は「ご苦労さん、では、どんな風に作業をしたのかを聞かせてもらおうか」と作業経過を報告するよう促した。以下はSmithの報告である。「はじめに、“ParsecKey [] p = new ParsecKey [5];”というコードを書き、ParsecKey型の配列を作成しました。そして次に、“QuadrantKey [] [] q = new QuadrantKey [5] [5];”というコードで、QuadrantKey型の配列を作成しました。その後は、forループを使って最初の5つのコードをParsecKey型の配列に格納し、残り25個のコードも、ネストしたforループを使ってQuadrantKey型の配列に格納しています。ウイルスチェックは、32個の参照型変数すべてについて行いました。ParsecKey型の配列への参照が1個、各要素への参照が合計で5個、QuadrantKey型の配列への参照が1個、各要素への参照が合計で25個、という計算です。ウイルスはまったく検出されなかったので、NavSimプログラムを実行し、再度ウイルスチェッカーを走らせましたが、問題はないようです」

報告を聞いたByte船長は、しばらくの間、冷たい目でSmithをじっと見ていたが、やがて静かにこう告げた。「Smithはしばらく自室で謹慎だ。船の安全を脅かしたからな。Javaの正しい知識が身に付くまで君の顔をブリッジで見たくない。後はBoolean中尉が引き継いでくれ。今度は間違いなく頼むよ」

船長はなぜSmithを謹慎させたのだろうか。いったい彼にどのような落ち度があったというのか？

This PDF is prepared for tools@maruhouse.org, personal use only.

地球への遠い道

Byte船長は、Javaの2次元配列が厳密には「配列の配列」であることを知っていたのである。5 × 5の2次元配列qのすべての要素にアクセスするためには、25個ではなく、以下のように合計31個の参照が必要になる。

配列そのものにアクセスするための参照 − 1つ

q [0] - q [4] にアクセスするための参照 − 5つ

q [0][0] - q [4][4] にアクセスするための参照 − 25個

Smithは、q [0] - q [4] にアクセスするための参照に関してチェックを怠っており、たとえHackerian星人によって不正に操作されていたとしても、彼のチェック方法では、それを発見できなかった。

This PDF is prepared for tools@maruhouse.org, personal use only.

This PDF is prepared for tools@maruhouse.org, personal use only.

索引

記号

!= 151
! 151
" 17
% 296
&& 151
& 150, 660
( ) 74
-- 115
, ...17, 76
. ....36, 62
.class 3
.jar 41
.java 3
.jnlp 598, 599
// 10
; 8, 10, 12, 105, 203
^ 660
{ } 7, 10, 12
| 660
|| 151
| 151
~ 660
+ 17
++ 105, 115
<?> 574
<< 660
< > 156
= 12, 52
== 11, 12, 86
>> 660
>>> 660

A

abstract....200, 229
ActionEvent.... 360
ActionListener.... 360
add()....133, 156
ArrayList 132-139, 156, 530-533
　int型の値を収める 288
　型パラメータ 543
assert 662

B

BeatBoxプログラム 419
　完成版 649
BorderLayout 403, 404, 412
BoxLayout.... 403, 411, 412
break 105, 109
BufferedWriter 453, 459, 478, 485

C

calenderオブジェクト 304, 305
calenderクラス 303
classesディレクトリ 585, 595
Collections.sort() 534, 535
Comparable.... 549
Comparator.... 552
CORBA 614

D

default....180

E

EJB(Enterprise JavaBeans) 631
else文 13
equals 86, 561, 563
Exception 331
extends 168, 180, 548

F

false....11
FileOutputStream....433, 445
FileWriter....453, 459
final....189
finallyブロック 327, 338
FlowLayout 403, 408, 412
format().... 295
forループ 95, 105
　拡張forループ 106, 116
　基礎知識....114
forループとwhileループの違い 115

G

getUserInput()....111
go() 77
Graphics2D....365, 368
GUI(Graphical User Interface)...353, 363, 368
　ウィジェット ..354, 362, 368, 370, 400, 412
　ウィンドウ 354, 368
　コンポーネント....400, 412
　描画パネル 364
　ボタン 355
　ルック&フィール.... 356

H

HAS-A関係.... 177
hashCode()....561, 563
HashMap 533
HashSet.... 533, 559, 561, 563

H

IDE(Integrated Development Environment)
　.... 662
if文 13
IIOP.... 614
implements....224, 229
import....155
increment()....514
indexOf() 133, 156

This PDF is prepared for tools@maruhouse.org, personal use only.

integer.parseInt() 105, 109, 117
interface 224, 229
IPマルチキャスト 632
IS-A関係 177-179, 181
isEmpty() 133, 156

J

J2EE(Java 2 Enterprise Edition) 631
JARファイル 585, 586, 593, 595
　実行可能JARファイル 582, 592
Java
　歴史 4
Java 2 SE(Standard Edition) 132
Java 5.0 ..... xxviii, 5
　ArrayListクラス 137
　拡張forループ 106, 116
　ジェネリック型..... 539, 540, 541, 568
　スタティックインポート..... 307
　プリミティブのラッピング 287-292
　列挙型の定数 671
Java API 125, 132, 154-160
Java Web Start 582, 597, 600
java.io.Fileクラス 452, 459
java.net..... 474
javacコマンド 3
JavaSound API 317
Java仮想マシン .....→JVM
JcheckBox 416
Jframe 354, 368
Jini 15, 632
JList 417
Jpanel..... 364
JPEG画像 365
JRMP..... 614
JSP(Java Server Pages)..... 628
JTextArea 414
JTextField..... 413
JVM(Java Virtual Machine) 2, 18, 236

L

LinkedHashMap..... 533
LinkedList 533
List..... 557

M

mainメソッド 8-10, 12
　役割 38
makeEvent() 389
manifest.txt 585
Map 557
Mathクラス.....274, 286
MIDI(Musical Instrumental Digital
　Interface) 317
MIDIデータ 317, 340, 343
MIDIファイル.....317, 340

N

new 241, 378
NIO(New I/O) 459
null値 56, 62

O

ObjectOutputStream 433, 445
Objectクラス 208-214

P

pack() 412
paintComponent().....364, 365
parseDouble() 292
parseInt() 292
PhraseOMaticプログラム 17
　サーブレットバージョン 629
PrintWriter.....479, 485
private 81,180, 189
protected 180
public 81, 180
publicfinal 41
publicstatic 41

R

ramdom() 17, 111
remove()メソッド 133, 156
RMI(Remote Method Invocation)
　..... 607, 610, 611, 623
　スケルトン 614, 618, 623
　スタブ 614, 618, 620, 621, 623
rmicツール 615
RMIアプリケーション 615
RMIレジストリ 615, 618
run() 500
Runnable.....494, 500

S

SDK(Software Development Kit).....xxviii
Sequencerオブジェクト.....318
Serializable 437, 445
Set..... 557
setMessage() 344
setSize() 412
size() 133, 156
sleep() 502, 517
Socketクラス 475
sort() 547
Split() 458, 459
start() 500
static final..... 282-284
String 669
StringBuffer 669
StringBuilder..... 669
Stringの不変性..... 661
Stringプール 661
stringをintに変換 105, 109, 117
super 180
Swing 354, 368, 399
Swingコンポーネント
　JcheckBox 416
　JList..... 417
　JTextArea..... 414
　JTextField 413
synchronized 510, 511, 514, 517, 522
System.out.print 13
System.out.printIn.....13, 662

T

TCPポート番号.....476, 485
Threadクラス 490, 493, 500
this() 258

This PDF is prepared for tools@maruhouse.org, personal use only.

throws 320, 325
transient 439, 440, 445
TreeSet 533, 534, 564-566
true 11
try/catch ブロック 321, 333, 338
catch ブロックを複数作る 332

U

UML xxix

V

void 75, 78

W

while 文 11,12

X

XP(エクストリームプログラミング) 101

あ行

アクセサ 79
アクセス修飾子 8, 81, 667
アクセスレベル 180, 667
アニメーション 382-385
アルファベット順 530
暗記カードプログラム 448
暗黙拡大変換 61, 78
イベント 356, 368, 369
イベント処理 357
イベントソース 359, 361, 368
イベントリスナー 358, 359, 361, 368, 374, 387
インクリメント 105
インクリメント演算子 109
インクルード 157
インスタンス変数 34, 41, 72, 73, 435
宣言と初期化 84
インスタンス変数とローカル変数の違い 85
インスタンス変数の値をプレーンテキストに書き込む 430, 431, 447
インターフェース 197, 224-227
インプリメント 224
インポート 155, 157
ウィジェット 354, 362, 370, 400, 412
ウィンドウ 354, 368
ウェルノウンポート 476, 485
エクストリームプログラミング 101
演算子
!= 151
! 151
&& 151
& 151, 660
^ 660
| 151, 660
|| 151
~ 660
<< 660
>> 660
>>> 660
演算子の優先順位 151
オーバーライド 32, 167, 172, 181
ルール 190
オブジェクト 235, 236
オブジェクトの参照 ....→オブジェクトリファレンス
共有 506
作成 241
寿命 258
保存 429, 431
ロック 509, 511, 522
シリアライズ 432
同等と見なす条件 560, 563
内部オブジェクト 378, 380
オブジェクト指向 27, 165
オブジェクト指向言語 12, 33
オブジェクトリファレンス 54, 55, 62, 215-217, 229
消滅 260-263
音楽演奏プログラム 316, 340
ガーベージコレクション 40, 235

か行

改行文字 13
カプセル化 36, 80-82
仮コード 99, 100, 108, 109
関数 28
疑似コード 99, 100
キャスト 78, 117
競合 504
クライアント 472, 483, 611, 623
クライアントプログラム 480
クラス 7, 8, 27, 28, 34, 78
Math クラス 274, 286
Object クラス 208-214
インスタンス化 200
具象クラス 200, 202, 204
作成 36, 99
サブクラス 31, 166, 168, 181
シリアライズの機能 437
スーパークラス 31, 166, 168, 181, 208
スーパークラスとサブクラスのコンストラクタ 250
スーパークラスのコンストラクタ 251-255
スタティックネストクラス 665
抽象クラス 200, 201, 202, 229
匿名ネストクラス 665
内部クラス 376, 380, 382, 665
名前付けのルール 53
パッケージを所属させる 589
クラスファイル 3, 582, 584, 595
クラスとオブジェクトの違い 35
クラスのコード
仮コード 99, 100, 108, 109
疑似コード 99
テストコード 99, 101, 102, 109
リアルコード 99, 109
グラデーション 367
グローバルな変数、メソッド 41
継承 31, 166, 168, 169, 171-176, 181
ダイアモンド継承 223, 229
多重継承 223, 229
ゲッターメソッド 79, 84
コアライブラリ 125
コメント 10
コンストラクタ 191, 241-256

This PDF is prepared for tools@maruhouse.org, personal use only.

コンストラクタのオーバーロード..... 247
デフォルトコンストラクタ 241, 246
コンテンツペイン 412
コンパイラ 2, 18
コンパイル 3
-dオプション..... 584, 590, 591, 595
コンパイルエラー 324
コンポーネント 400, 412

さ行

サーバ 472, 480, 611
サーバプログラム 483
サービスの自動検索機能..... 633
サーブレット 625-628
参照..... 239
寿命 258
ジェネリック型 539, 540, 541, 568
ジェネリックメソッド 544
実行可能JARファイル 582, 592, 595
寿命
オブジェクト 258
参照 258
参照型変数 260
ローカル変数 259
条件判定..... 11, 114, 151
状態 429
ショートカット..... 664
ジョブ 493
シリアライゼーション
.....429, 430, 431, 434, 438, 444, 445
シリアルバージョンID..... 460
数字
カンマを入れる 294
数値を数字に変換する 293
数値 273
フォーマッティング 294-300
図形の描画 369, 391
スケルトン 614, 618, 623
スコープ 259
スタック 236
スタティック..... 273
スタティックイニシャライザ 282, 284
スタティックインポート 307
スタティックネストクラス..... 665
スタブ 614, 618, 620, 621, 623
ステート 71, 73, 78, 429
ステートメント 7
ストリーム 433, 459, 478
接続ストリーム 433
チェーンストリーム 433
スレッド 471, 490, 492, 500
オブジェクトの共有 506
競合 504
実行可能状態 492
状態の変化 495
スリープ 501
プライオリティ 522
ブロック状態..... 492
マルチスレッド 489, 491, 498, 500
スレッドスケジューラ 497, 500
セッター 79, 80
ソース 2, 3
ソースファイル.....7, 584, 595
ソート..... 529
ソケット接続 475, 485

た行

ダイナミッククラスダウンローディング..... 621
ダイナミックバインディング..... 19
タイプセーフ 540
タイムスタンプ 303
タグインターフェース 437
チャット 472
チャットクライアントプログラム 486, 518
チャットサーバプログラム..... 520
抽象 166
抽象クラス 197, 201, 202
抽象メソッド 203, 204
ディレクトリを作成 452
テキストエディタ xxviii
テキストファイルからデータを読み込む 454
デザインパターン 181
デシリアライゼーション 441-443, 445
テストコード.....99, 101, 102, 109
手続き指向 27
デッドロック.....516
デバッグ 662
動物シミュレーションプログラム 170
匿名ネストクラス..... 665
トランザクション 516, 631

な行

内部オブジェクト.....378, 380
内部クラス.....189, 376, 380, 382, 665
名前付けのルール 53
ネットワーク接続.....471, 475
値渡し 77, 78, 87
ノンブロッキングIO..... 459

は行

バイトコード.....2, 18
配列..... 17, 59, 60, 62, 83
ArrayListクラスとの比較 135, 136, 137
多次元配列 670
バグ.....126-128
パッケージ.....154, 155, 587, 588, 595
拡張パッケージ 156
バッファ 453
バトルシップゲームプログラム 96, 126
パラメータ 74, 78, 79
ヒープ 40, 236, 608
引数.....8, 49, 74, 77, 78
日付のフォーマッティング 301-306
ビット操作..... 660
ビットパターン 86
描画パネル 364
標準出力..... 8
プッシュ 237
プリデクリメント演算子.....109
フルネーム..... 587
プログラム
実行 3
設計 97
提供 582
ブロック変数 663
プロトコル 183, 476
分岐 10, 13
ヘルパー 612, 623
変数 12
boolean 11, 51
byte 51

This PDF is prepared for tools@maruhouse.org, personal use only.

char 51
double..... 51
Exception..... 322
float..... 51
int.....12, 51
long 51
short 51
インスタンス変数 49, 236, 239, 243, 247, 264, 308, 435
カウンタ変数.....114
型 50
参照型 49, 50, 51, 62, 54, 55, 56
参照型の寿命 260
数値型 51
スタック変数..... 236
スタティック変数 279-282, 284, 308
宣言 50
名前付けのルール 53
配列型 59
プリミティブ型..... 49, 50, 51, 62
プリミティブ型のキャスト 117
プリミティブ型のラッピング..... 287-292
ブロック変数..... 663
変数の比較 86
ローカル変数 49, 236, 238, 264
ローカル変数の寿命とスコープ 259
ポインタ 54, 56
ポストインクリメント演算子 109
ポストデクリメント演算子 109
ポリモーフィズム
.....183-187, 200, 206, 215, 225, 330
ポリモーフィズムとジェネリック型..... 569

ま行

マーカーインターフェース 437
マニフェスト 41
マニフェストファイル.....585, 595
マルチスレッド 489, 491, 498, 500
ミュージックビデオプログラム..... 386
ミューテータ 79
メソッド 7, 31, 34, 41, 71, 72, 78, 166
値を渡す 74
スタティックメソッド 275-278, 284
抽象メソッド.....203, 204, 229
名前付けのルール 53
複数のパラメータ..... 76
メソッドのオーバーライド.....191
メソッドの呼び出し..... 608
リモートメソッド..... 614
例外を発生させるメソッド 323
メモリ
スタック 236, 237, 238
ヒープ 236, 238
メンバ 168, 180
文字列をつなげる 17
戻り値 8, 49, 75, 79

や行

ユニバーサルサービスブラウザ..... 636
予約語 53

ら行

ライブラリ .....→Java API
ラッパ 287, 292
ラッパの不変性 661
ラッピング..... 287-292
リアルコード 99, 109
リース 636
リモートインターフェース.....616, 623
リンク呼び出し..... 664
ループ 10, 11, 95, 115
ルック&フィール 356
ルックアップサービスの自動修復機能.... 635
レイアウトマネージャ 399, 401-412, 416
BorderLayout.....403, 404, 412
BoxLayout 403, 411, 412
FlowLayout.....403, 408, 412
例外 320-339
ClassCastException 19
ClassCastException 216
InterruptedException 501
NullPointerException.....151
RemoteException..... 616
RuntimeException 324, 325
チェック例外..... 325
複数種の例外..... 329
例外処理の回避 335
例外とポリモーフィズム 330
列挙型の定数..... 671
ロールバック 516
ロック 509, 511, 522

わ行

ワイルドカード 574

This PDF is prepared for tools@maruhouse.org, personal use only.

訳者あとがき

「コンピュータの世界には、学ぶべきことをイチから体系立てて説明してくれる本はないんですか」社会人になりたてのころ、「教育係」としてついてくれた先輩にこんな質問をしたことがあります。先輩の答えは「そんなものねーよ！」という極めてあっさりしたものでした。なんて理不尽な……そう思ったものの、そう言われてしまえば返す言葉がありません。その時はおとなしく引き下がりました。コンピュータをまともに触ったこともない、という状態でソフトウェアの会社に就職したのですが、案の定まったく何もわかりません。そこで何冊もの本を必死で読んでみたのですが、断片的な知識だけは増えるものの、いくら読んでも全体像がつかめず、「わかった」という実感は得られませんでした。そこで、つい、愚痴のような感じでこういう質問をしてしまったというわけです。今思えば、社会人になって学生時代のように手取り足取り順番に教えてもらえると思う方が甘かったのです。今の私なら、新入社員に同じような質問をされれば、やはり「そんなものねーよ！」と答えるに違いありません。

ただ、過去の自分の甘さは素直に認め、反省はするものの、コンピュータ関係の解説書に対する不満は現在もなくなったわけではありません。最も困るのが、ほとんどの本が「読者不在」の作り方をされている、という点です。私が新人の頃から考えれば、「初心者向け」と銘打たれた、わかりやすさを売り物にする本は比較にならないほど増えました。しかし、その多くは、実際には「本当にわからせようとしているのか……」と首を傾げたくなるようなものなのです。

Javaの入門書に関しても状況は同じです。中には非常に優れた本もあるのですが、大半は、ただ著者の知っていることを著者の書きやすいように書いた、という感じのものです。事実を正確に書いてはいても、説明をしよう、わからせようという意図はあまり感じられません。「初心者向け」、「わかりやすい」という点を特に強調したものでも、たいていは、上級者が読むような文章を要約して短くしてあるか、絵や図を多用してカラフルにしてあるかのどちらかです。難しいことを難しいまま手みじかに説明されても、余計にわからなくなるだけです。絵や図がやたらに多いのも、読みにくくしているだけに終わっているようです。絵や図を多用している本には、サンプルプログラムの作り方を懇切丁寧に説明してくれるものもあります。ただ言うとおりにしていればともかくプログラムが動く、というわけですが、これも功を奏しているとは言えないでしょう。確かに一応はプログラムが動かせるのですが、読者はなぜ自分がプログラムを動かせたかよくわかりません。たとえサンプルが作れても他に応用が利かないのです。

どうしてこのような本ばかりができるのか、確かな理由はわかりません。ただ一つ確かなことは、著者の多くが技術の専門家である、ということです。彼らはおそらく、初心者が何をわかっていないか、どういうところでつまづいてしまうか、想像もつかないのではないでしょうか。かと言って、素人が解説書を作ればいいというものではありません。十分な知識のない人には、そもそも本を作ることすらできないはずです。理想は、「素人の

This PDF is prepared for tools@maruhouse.org, personal use only.

気持ちがわかる専門家が書く」ということですが、なかなかそうはいかないものです。

本書『Head First Java』の著者はその点では理想的な人物と言えます。技術者として確かな知識を持っているだけでなく、まったくの初心者を含む多数の人（著者紹介によれば数百人）に実際にJavaを教えてきた経験があるからです。たくさんの人を教育する中で培われてきたであろうノウハウは本書に惜しげもなく投入されているようです。「素人の気持ちを実によく知っている」と感じられる工夫が随所にこらされています。

この本の最大の特徴は、「ユーザ目線で作られている」というところだと思います。これは、「まずJavaという言語を道具として使いこなせるようになってもらう」ということを主眼に書かれているということです。Java言語の構造や設計思想といった技術的背景についてももちろん詳しく説明しているのですが、その説明のタイミングが絶妙なのです。読者が「使いこなし方」をある程度理解して、「そういえば、中の仕組みはどうなっているんだろう……」と興味を持つであろう頃にすかさず説明をするので、ごく自然に頭に入ってきます。概念説明を事前に長々として実践に入る、というようなこともなく、使い方の説明と概念説明を同時に行ってどこが要点かわからなくなる、というようなこともありません。

たとえば、テレビを使う時に、最初に「どういう仕組みで映るのか」を気にする人はそう多くありません。まず、どう使えば番組が見られるのかを知りたいはずです。だいたい使いこなせるようになれば、中身について考える余裕も出てきます（ずっと考えない人もいますが）。それと同じではないでしょうか。普通、技術に詳しい人は、よくわかっているだけに、どうしても中の仕組みや構造のことが気になり、「あれもこれも」と説明したくなります。こういう仕組みだからこう使わなくてはならない、というような説明の仕方をしがちなのです。それも重要なのですが、一度にあれこれ言われても読み手は消化しきれません。テレビを使ったことのない人に「テレビ局の発射する電波はそれぞれ周波数が異なっているから、見る番組を選ぶには、適切な周波数の電波を受信できるよう調整する」というような説明をしても多分、わかりません。まずは「見る番組を選ぶには、1から12までのチャンネルボタンを使う」という説明で十分でしょう。本書ではまさにこのような説明がなされているのです。

「読者に自分の頭で考えさせる」というのも本書の大きな特徴です。Java言語の使い方を説明するといってもはじめから何でも親切に教えてくれるわけではありません。「こういうプログラムを作るには、果たしてどうすればいいでしょうか」という具合にまず読者に問いかけ、問題意識を持たせるようにしています。そして、読者が十分に考えた頃を見計らって解答を示すのです。プログラムが少しずつできあがっていく過程に読者も参加できるようにしているわけです。

自らの頭で考え、サンプルプログラムの作成に積極的に参加すれば、他のプログラムを作る時にも応用のきく知識、能力が身に付きます。決して「できた、動いた、ああよかっ

This PDF is prepared for tools@maruhouse.org, personal use only.

た」で終わりにはなりません。また、本書にはどのくらい応用力がついたかを確認できるユニークな練習問題も多数用意されています。問題を解いた結果を基に、理解の足りないところを復習すれば、さらに力をつけることができるでしょう。

本書は、著者も繰り返し言っているとおり、何でも載っている「リファレンスブック」ではありません。そして、漫然と受け身の姿勢で読んでいればいい、というような本でもありません。黙っていても手取り足取り教えてくれるような本ではないのです。頭と手を使い、能動的に取り組まなければ、何も得られません。本来、プログラミング言語はそうして学習するものでしょう。そう言うと何やら大変そうですが、本書は、無理をせずとも、自然に能動的な学習ができるように作られているので安心です。

正直に言って、これから勉強をする人をとてもうらやましく思います。私が新人の頃にこんな本があったら、先輩に間の抜けた質問せずとも済んだのに……と少し悔しく思っています。もちろん、そもそも取り組み方が甘かったのですから、この本があっても私に関しては結果は同じだったかもしれませんが……。しかし、真剣に勉強をしようという意志のある方には、必ず本書が役立つと確信しています。本書によって一人でも多くの方がJavaを使いこなせるようになれば、これ以上の喜びはありません。

最後になりましたが、本書の製作作業がスムーズに進むよう様々な尽力をいただいた株式会社オライリー・ジャパンの田村英男氏、多忙にもかかわらず非常に的確な監修をしていただいた島田秋雄氏、高坂一城氏、神戸博之氏、また複雑な組版作業を見事にこなしていただいた有限会社ビートップの方々に感謝の意を表したいと思います。ありがとうございました。

第2版のあとがき

内容には自信を持って送り出した第1版でしたが、不安もありました。何より、すでに大量のJava本が出回っている状況で、さらにもう1冊買ってもらうことができるのか、それが心配でした。「面白そうだけど、もうJava本はいいや」そう思われても何の不思議もないでしょう。

しかし、心配は無用だったようです。発売以来、第1版には予想以上の反響をいただきました。数字に現れる結果はもちろんですが、とにかく「面白かった」「はじめてJavaがよくわかった」「今まで読んできた本は何だったのかと思う」……、そういう声に安堵しました。自分が原書に感じた価値を日本の読者に伝えることができたと思えたからです。

おかげさまでこうして第2版を出すことができました。Java 5.0に対応し、さらにパワーアップした『Head First Java』。楽しんでいただければ幸いです。

2006年2月
夏目 大

This PDF is prepared for tools@maruhouse.org, personal use only.

●監修者紹介

島田 秋雄（しまだ あきお）
1990年神戸大学大学院理学研究科数学専攻を修了。同年、日本サン・マイクロシステムズ株式会社に入社。研究開発部ネットワークグループに配属。サン・マイクロシステムズ株式会社、技術推進統括本部システム開発センターにおいて、JavaによるSIやセキュリティ全般に従事。2004年から2005年まであるベンチャーキャピタルにおいて技術評価などに従事。現在フリーとなり、ITアーキテクトとして活動。主な訳書に、『Javaセキュリティ』（オライリー・ジャパン）、『Java2プラットフォームセキュリティ』（ピアソン・エデュケーション）などがある。

高坂 一城（こうさか かずき）
1995年法政大学大学院工学研究科システム工学専攻を修了。同年、サン・マイクロシステムズ株式会社に入社。現在、ソリューションサービスJavaコンサルティング部において、J2SEとJ2EEを中心としたJavaの技術サポートに従事。
著書に『J2MEプログラミングガイド』（共著、IDGジャパン、2003年）、訳書に『Java2 Platform Micro Edition プログラミング』（共訳、ピアソンエデュケーション、2002年）、監訳書に『Java 2 J2SE 1.4』（共同監訳、IDGジャパン、2003年）などがある。

神戸 博之（ごうど ひろゆき）
1995年、信州大学大学院 情報工学専攻修了。同年サン・マイクロシステムズ株式会社入社。現在Javaエンジニアリングサービスにて J2MEの技術開発に従事する。著書に『J2MEプログラミングガイド』（共著、IDGジャパン、2003年）、訳書に『Java2 Platform Micro Edition プログラミング』（共訳、ピアソンエデュケーション、2002年）、監訳書に『Java 2 J2SE 1.4』（共同監訳、IDGジャパン、2003年）などがある。

●訳者紹介

夏目 大（なつめ だい）
1966年大阪府生まれ。翻訳家、フリーライター。訳書に『スパマーを追いかけろ』、『初めてのPython 第2版』、『Mind Hacks』（いずれもオライリー・ジャパン）、『シティバンク 勝利の複雑系』、『15時間集中講座Java2』（コンピュータ・エージ社）、『シェーキーの子どもたち』（翔泳社）、著書に『よくわかるJava』（翔泳社）などがある。翻訳学校「フェロー・アカデミー」講師。

URL：`http://www.bitpranks.com/garage/natsume/`
Blog：`http://dnatsume.cocolog-nifty.com/natsume/`

This PDF is prepared for tools@maruhouse.org, personal use only.

Head First Java 第2版 —— 頭とからだで覚えるJavaの基本

2006年3月27日　初版第1刷発行
2010年5月6日　初版第6刷発行

著　者　Kathy Sierra（キャシー・シエラ）
Bert Bates（バート・ベイツ）
監　修　島田 秋雄（しまだ あきお）
高坂 一城（こうさか かずき）
神戸 博之（ごうど ひろゆき）
訳　者　夏目 大（なつめ だい）
発行人　ティム・オライリー
制　作　ビーンズ・ネットワークス
印　刷　株式会社平河工業社
発行所　株式会社オライリー・ジャパン
〒160-0002　東京都新宿区坂町26番地27　インテリジェントビル1F
Tel　(03) 3356-5227
Fax　(03) 3356-5263
電子メール　japan@oreilly.co.jp
発売元　株式会社オーム社
〒101-8460　東京都千代田区神田錦町3-1
Tel　(03) 3233-0641（代表）
Fax　(03) 3233-3440

Printed in Japan（ISBN4-87311-279-6）
乱丁本、落丁本はお取り替えいたします。

本書は著作権上の保護を受けています。本書の一部あるいは全部について、株式会社オライリー・ジャパンから文書による承諾を得ずに、いかなる方法においても無断で複写、複製することは禁じられています。

This PDF is prepared for tools@maruhouse.org, personal use only.