Java入門

文法，各種ライブラリ，
GUIアプリケーション構築

第0.65版

Copyright © 2019, Katsunori Nakamura

中村勝則

2019年12月15日

免責事項

本書の内容は参考的資料であり，掲載したプログラムリストは全て試作品である．本書の使用に伴って発生した不利益，損害の一切の責任を筆者は負わない．

目 次

1 序章 **1**
1.1 プログラム開発の基本的な手順 . . . 1
1.2 プログラム開発の具体的な作業 . . . 1
1.2.1 ソースプログラムの記述や入出力における文字コードの指定 . . . 2
1.2.2 jar 形式のライブラリを利用する方法 . . . 3

2 Java 言語 **4**
2.1 初めて学ぶ際の留意事項 . . . 4
2.2 変数と値に関する基礎事項 . . . 4
2.2.1 値の表現 . . . 5
2.2.2 算術演算 . . . 5
2.2.3 複数の文の記述 . . . 6
2.2.4 コメントの記述 . . . 6
2.2.5 変数の型と演算に関する例 . . . 6
2.2.6 メソッドと引数 . . . 7
2.2.7 イコール「=」による代入に関して . . . 7
2.2.8 文字と文字列の扱い . . . 7
2.2.8.1 文字列の部分を取り出す . . . 7
2.2.8.2 エスケープシーケンス . . . 8
2.2.9 値の型の変換 . . . 9
2.2.9.1 数値と文字列の間での表現の変換 . . . 10
2.2.10 基数の変換 . . . 10
2.2.10.1 10 進数→ n 進数 . . . 10
2.2.10.2 n 進数→ 10 進数 . . . 11
2.2.11 ビット演算 . . . 11
2.2.11.1 シフト演算 . . . 11
2.2.11.2 ビット毎の論理演算 . . . 12
2.3 制御構造 . . . 14
2.3.1 条件分岐 (1)： if 文 . . . 14
2.3.1.1 条件式の記述と boolean 型変数 . . . 15
2.3.1.2 文字列の比較 . . . 16
2.3.1.3 条件判定のための 3 項演算子 . . . 16
2.3.2 条件分岐 (2)： switch 文 . . . 17
2.3.3 反復 (1)： while 文 . . . 18
2.3.3.1 再帰代入による変数の値の変更 . . . 18
2.3.4 反復 (2)： for 文 . . . 20
2.4 配列 . . . 21
2.4.1 配列への記憶の割り当て . . . 22
2.4.2 配列宣言時の初期化 . . . 22
2.4.3 配列の要素の整列 . . . 23
2.4.4 拡張 for 文による要素へのアクセス . . . 24
2.5 オブジェクト指向プログラミング . . . 25
2.5.1 概念的な導入 . . . 25
2.5.1.1 オブジェクト指向における部品の考え方 . . . 25
2.5.1.2 オブジェクト指向プログラミングの概要 . . . 25

2.5.2 サンプルプログラムに沿った説明 . . . 25
2.5.2.1 クラス，インスタンス，コンストラクタ」の考え方 . . . 28
2.5.2.2 静的メンバ／メソッドについて . . . 30
2.5.2.3 static イニシャライザ . . . 30
2.5.3 クラスの拡張（継承） . . . 31
2.5.4 抽象クラスとインターフェースについて . . . 32
2.5.4.1 匿名クラスについて . . . 35
2.5.5 アノテーション . . . 36
2.5.6 総称型（Generics）について . . . 36
2.5.7 オブジェクトのクラスに関する情報 . . . 37
2.6 入出力 . . . 39
2.6.1 標準入出力 . . . 39
2.6.1.1 文字列生成への書式編集の応用 . . . 40
2.6.1.2 標準エラー出力 . . . 41
2.6.2 コマンド引数 . . . 42
2.6.3 ファイルへの出力 . . . 43
2.6.3.1 バイトストリームへの出力： FileOutputStream . . . 43
2.6.3.2 例外処理 (1)： try . . . 44
2.6.3.3 例外処理 (2)： throws . . . 45
2.6.3.4 PrintWriter による出力 . . . 45
2.6.4 ファイルからの入力 . . . 46
2.6.4.1 テキストの行単位の読込み . . . 46
2.6.4.2 Scanner クラスを用いた読込み . . . 48
2.6.5 バイナリファイルの入出力 . . . 49
2.7 final 修飾子 . . . 51
2.8 プログラムの終了 . . . 52

3 標準的なクラスライブラリ 53
3.1 データ構造 . . . 53
3.1.1 List（リスト）： ArrayList / LinkedList . . . 53
3.1.1.1 リスト生成時の初期化 . . . 55
3.1.1.2 要素の整列： sort . . . 55
3.1.1.3 要素の順序の反転： reverse . . . 55
3.1.1.4 拡張 for 文による List の要素へのアクセス . . . 56
3.1.1.5 Comparator を用いた整列 . . . 56
3.1.2 Set（セット，データ集合）： HashSet / TreeSet . . . 59
3.1.2.1 Iterator による要素への順次アクセス . . . 61
3.1.3 Map（連想配列，辞書）： HashMap / TreeMap . . . 61
3.1.3.1 登録されているマッピングに順次アクセスする方法 . . . 63
3.1.4 データ構造の変換 . . . 65
3.1.4.1 リスト，セットを配列に変換する方法 . . . 65
3.2 編集可能な文字列のクラス： StringBuilder，StringBuffer . . . 66
3.2.1 コンストラクタ . . . 66
3.2.2 文字列編集のためのメソッド . . . 66
3.2.3 サンプルプログラム . . . 66
3.2.4 byte 配列,String/StringBuilder/StringBuffer オブジェクトとの間の変換 . . . 67
3.3 Math クラス . . . 69

3.3.1 BigInteger，BigDecimal . . . 69
3.3.1.1 オブジェクトの生成と値のセット . . . 70
3.3.1.2 演算 . . . 70
3.3.1.3 MathContext について . . . 70
3.3.1.4 数学関数の多倍長演算について . . . 71
3.4 日付，時刻の扱い . . . 73
3.4.1 Date，Calender クラス . . . 73
3.4.2 時間の計測 . . . 75
3.4.3 LocalDateTime，LocalDate，LocalTime クラス . . . 76
3.4.3.1 基本的な機能 . . . 77
3.4.3.2 和暦の扱い . . . 80
3.5 ファイル，パス，URI . . . 82
3.5.1 パスの作成，パスの分解 . . . 82
3.5.2 URI . . . 83
3.5.3 File クラスのメソッド . . . 84
3.5.3.1 ファイル，ディレクトリの作成 . . . 85
3.5.3.2 ディレクトリ内容の取得，ファイルの存在検査 . . . 86
3.5.3.3 ファイル名の変更（パスの変更） . . . 86
3.5.3.4 ファイル，ディレクトリの削除 . . . 87

4 GUI と 2D グラフィックス 89
4.1 GUI プログラミングの基本的な概念 . . . 89
4.2 GUI アプリケーション構築の概略 . . . 89
4.2.1 GUI ツールキット . . . 90
4.3 Swing . . . 91
4.3.1 GUI の構築 . . . 91
4.3.1.1 GUI 構築の例 . . . 91
4.3.1.2 イベント処理 . . . 94
4.3.1.3 キーボードイベントからのキーの取り出し . . . 97
4.3.1.4 JFrame クラスとしてのアプリケーションの構築 . . . 97
4.3.1.5 より安全な実装 . . . 99
4.3.1.6 レイアウトマネージャ . . . 99
4.3.2 Swing でのグラフィックス作成の概要 . . . 101
4.3.2.1 グラフィックス作成の流れ . . . 101
4.3.2.2 プログラムの例 . . . 102
4.3.2.3 paintComponent の引数について . . . 103
4.3.3 座標系について . . . 103
4.3.4 グラフィックス作成のための基本的なメソッド . . . 103
4.3.4.1 描画色の指定 . . . 103
4.3.4.2 直線の描画： drawLine メソッド . . . 104
4.3.4.3 折れ線の描画： drawPolyline メソッド . . . 104
4.3.4.4 線の属性の指定 . . . 104
4.3.4.5 長方形の描画 (1) 枠： drawRect メソッド . . . 107
4.3.4.6 長方形の描画 (2) 塗りつぶし： fillRect メソッド . . . 107
4.3.4.7 楕円の描画 (1) 枠： drawOval メソッド . . . 107
4.3.4.8 楕円の描画 (2) 塗りつぶし： fillOval メソッド . . . 107
4.3.4.9 多角形の描画 (1) 枠： drawPolygon メソッド . . . 107

4.3.4.10 多角形の描画 (2) 塗りつぶし： fillPolygon メソッド . . . 107
4.3.4.11 文字の描画 . . . 107
4.3.4.12 階調色の指定 . . . 109
4.3.5 BufferedImage の利用 . . . 110
4.3.5.1 BufferedImage に対する描画 . . . 110
4.3.5.2 BufferedImage オブジェクトを画像データとして出力する方法 . . . 111
4.3.5.3 画像ファイルの読み込みと表示 . . . 113
4.3.5.4 画素に対する操作 . . . 115
4.3.6 ファイル選択ダイアログ . . . 117
4.4 SWT . . . 120
4.4.1 GUI アプリケーション構築の流れ . . . 120
4.4.2 GUI コンポーネントの生成と配置 . . . 121
4.4.3 イベント処理の登録 . . . 123
4.4.4 サンプルプログラム . . . 123
4.4.5 キーボードイベントからのキーの取り出し . . . 127
4.4.6 SWT でのグラフィックスの作成の概要 . . . 127
4.4.6.1 GC（Graphic Context） . . . 127
4.4.6.2 グラフィックス作成の流れ . . . 127
4.4.7 座標系について . . . 128
4.4.8 色の指定 . . . 128
4.4.9 描画のための基本的なメソッド . . . 129
4.4.9.1 直線の描画： drawLine メソッド . . . 129
4.4.9.2 折れ線の描画： drawPolyline メソッド . . . 129
4.4.9.3 線の属性の設定 . . . 129
4.4.9.4 長方形の描画 (1) 枠： drawRectangle メソッド . . . 130
4.4.9.5 長方形の描画 (2) 塗りつぶし： fillRectangle メソッド . . . 130
4.4.9.6 楕円の描画 (1) 枠： drawOval メソッド . . . 130
4.4.9.7 楕円の描画 (2) 塗りつぶし： fillOval メソッド . . . 130
4.4.9.8 多角形の描画 (1) 枠： drawPolygon メソッド . . . 130
4.4.9.9 多角形の描画 (2) 塗りつぶし： fillPolygon メソッド . . . 130
4.4.10 文字の描画 . . . 130
4.4.11 画素に対する操作 . . . 131
4.4.11.1 画素の着色処理 . . . 133
4.4.12 画像ファイルの入出力 . . . 133
4.4.12.1 画像ファイルの読み込み . . . 133
4.4.12.2 画像ファイルへの出力 . . . 133
4.4.13 サンプルプログラム . . . 134
4.4.14 ファイル選択ダイアログ . . . 137

5 スレッドとプロセス 140
5.1 マルチスレッドプログラミング . . . 140
5.1.1 Thread クラス . . . 140
5.1.2 Runnable インターフェース . . . 142
5.1.3 スレッドの同期 . . . 143
5.1.3.1 他のスレッドの終了を待機する方法： join メソッド . . . 143
5.1.3.2 synchronized 修飾子 . . . 144
5.1.3.3 スレッドの待機／実行の再開 . . . 147

5.1.4 スレッドの状態について . . . 149
5.2 concurrent ライブラリ . . . 150
5.2.1 Executor と ExecutorService . . . 150
5.2.2 Callable インターフェースと Future インターフェース . . . 152
5.3 外部プロセスの起動 . . . 155
5.3.1 Java プログラムと外部プロセスの関係 . . . 155
5.3.2 使用方法 . . . 155
5.3.3 サンプルプログラム . . . 156

6 ラムダ式，Stream API，メソッド参照 159
6.1 関数型インターフェース . . . 159
6.2 Stream インターフェース . . . 160
6.2.1 Stream オブジェクトの生成 . . . 161
6.2.2 forEach メソッド . . . 161
6.2.3 filter メソッド . . . 161
6.2.4 map メソッド . . . 162
6.2.5 sorted メソッド . . . 162
6.2.5.1 Stream のクローズ（終了） . . . 162
6.2.6 collect メソッド . . . 162
6.3 メソッド参照 . . . 163

7 正規表現 165
7.1 文字列探索： find メソッド . . . 165
7.1.1 パターンの検出位置 . . . 166
7.1.2 パターン検出の続行 . . . 166
7.2 置換処理： replaceFirst，replaceAll . . . 167
7.3 パターンの記述方法 . . . 167
7.4 パターンマッチのグループ . . . 169
7.5 文字列の分割： split メソッド . . . 170

8 TCP/IP 通信 171
8.1 ソケットの考え方 . . . 171
8.2 ソケットに関するクラス . . . 172
8.2.1 サーバソケット . . . 172
8.2.2 ソケット . . . 172
8.3 データの送受信 . . . 173
8.4 サンプルプログラム . . . 174

9 サウンドデータの再生 177
9.1 最も素朴な方法 . . . 177
9.2 サイズの大きなサウンドデータの再生 . . . 179
9.3 実用的なサウンド再生 . . . 181

10 ライブラリ，パッケージに関すること 184
10.1 ライブラリの作成 . . . 184
10.1.1 ライブラリファイル（クラスファイル）の保存場所 . . . 184
10.1.2 異なるパッケージが同名のクラスを提供する場合の扱い . . . 185
10.1.3 jar ファイルの作成 . . . 185

11 データベースへの接続 **186**
11.1 データベースについての基本的な考え方 . . . 186
11.1.1 データベースに対する基本的な操作 . . . 186
11.2 本書で取り扱うソフトウェア . . . 187
11.2.1 SQLite . . . 187
11.2.2 JDBC . . . 187
11.3 トランザクション . . . 187
11.3.1 データベースへの接続 . . . 187
11.3.2 Statement オブジェクトの取得 . . . 188
11.3.3 SQL 文の実行 . . . 188
11.3.3.1 検索結果の取り出し . . . 188
11.3.4 トランザクションの終了 . . . 188
11.4 SQL . . . 188
11.4.1 データベースの作成 . . . 189
11.4.2 テーブルの作成 . . . 189
11.4.2.1 カラムのデータ型 . . . 189
11.4.2.2 サンプルプログラム (1) . . . 190
11.4.3 テーブルの結合 . . . 192
11.4.3.1 サンプルプログラム (2) . . . 193

A Swing 簡易リファレンス **195**
A.1 多くのコンポーネントで共通のメソッド . . . 195
A.2 各種のコンポーネント . . . 195
A.3 主な GUI コンポーネントのクラス階層 . . . 200
A.4 代表的なイベント . . . 201

B SWT 簡易リファレンス **202**
B.1 多くのコントロールで共通のメソッド . . . 202
B.2 各種のコントロール . . . 202
B.3 代表的なイベント . . . 207

C java.nio ライブラリ **208**
C.1 Path インターフェースと Paths クラス . . . 208
C.2 Files クラス . . . 208

D Javadoc **212**
D.1 ソースプログラムの文字コード（エンコーディング）の指定 . . . 213
D.2 コメントの記述形式とタグ . . . 214

E サードパーティのライブラリ **215**
E.1 Apfloat . . . 215

F サンプルプログラム **217**
F.1 ArrayList，LinkedList の性能比較 . . . 217

1　序章

Java は 1990 年代前半に，米サン・マイクロシステムズ社（後に Oracle 社に吸収合併）により開発されたプログラミング言語とその実行環境である．Java は多くのシステム（PC，携帯情報端末，組込み制御系，大型計算機システム等）に実装されて広く普及した汎用性の高い言語処理系である．Java 言語は「オブジェクト指向プログラミング」の形を基礎とし，プログラムの開発と保守を容易にしている．

Java で作成されたプログラムは **Java 仮想マシン**（JVM）の上で実行される．JVM 自体も 1 つのコンピュータプログラムであり，JVM 用の API 群と併せて **Java 実行環境**（JRE）として配布されている．Java のプログラムを開発するにはコンパイラを含む **Java 開発キット**（JDK）が必要となる．JRE や JDK，関連の情報などは Oracle 社のインターネットサイト [1] から入手することができる．

参考）JRE，JDK は Oracle 社製のもの以外に IBM 製のものや，オープンソースの **OpenJDK** [2] がある．

1.1　プログラム開発の基本的な手順

Java によるプログラム開発の流れは基本的に下記のようなものである．

(1) ソースプログラムの作成
Java 言語の文法に則って記述されたプログラム（ソースプログラム）を作成する作業．ソースプログラムは**テキストファイル** [3]（拡張子は “.java”）として作成する．

(2) コンパイル作業
ソースプログラムを JRE で実行できる**中間言語**（**バイトコード**）に変換する作業．この作業により，“.class” の拡張子を持つバイトコードのファイルが作成される．

(3) バイトコードの実行
上の処理で作成されたバイトコードを JVM 上で実行する．

1.2　プログラム開発の具体的な作業

ここでは，ソースプログラムを作成してコンパイル作業を行い，得られたバイトコードを実行する流れの例を示す．まず次のようなサンプルプログラム test01.java をテキストエディタで作成する．

プログラム：test01.java

```
1 class test01 {
2     public static void main( String argv[] ) {
3         System.out.println("Hello, world.");
4     }
5 }
```

プログラムのコンパイルと実行は，コンピュータの最も基本的な操作環境である**コマンド**環境（CLI 環境 [4]）で行う．CLI 環境は各種 OS に備わっており，Microsoft Windows では**コマンドプロンプトウィンドウ**（図 1）や **Windows PowerShell** がそれに当たる．また Apple 社の macOS や Linux では**ターミナルウィンドウ**が CLI 環境である．

ソースプログラムのコンパイルには javac コマンドを用いる．先に示したサンプルプログラム test01.java をコンパイルするには，CLI 環境で，ソースプログラムと同一のディレクトリにおいて，

`javac test01.java` Enter

とコマンドを投入する．これによりバイトコードのファイル test01.class が作成される．

[1] https://www.oracle.com/
[2] http://openjdk.java.net/
[3] 可読な文字のみから構成されるファイル．
[4] **CLI**（Command Line Interface）は **CUI**（Character-based User Interface ／ Console User Interface）とも呼ばれる．

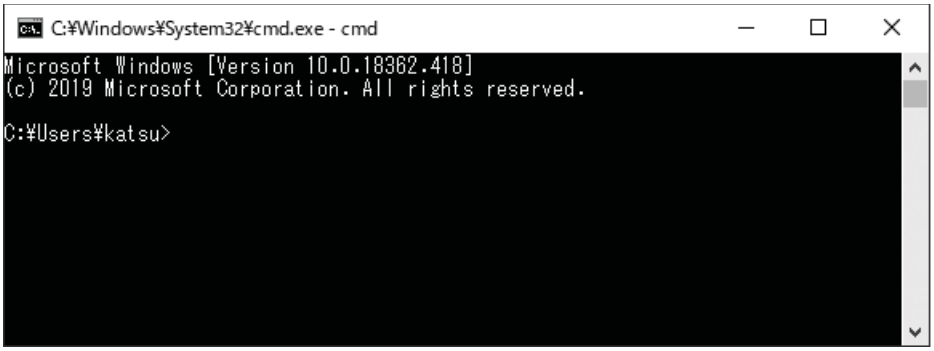


図 1: Windows のコマンドプロンプトウィンドウ

このバイトコードを実行するには java コマンドを

```
java test01  Enter
```

として投入する．その結果バイトコードが JVM 上で実行され，

```
Hello, world.
```

と表示される．

注意）javac，java コマンドを CLI 環境で実行するには，それらコマンドが当該 CLI 環境の**コマンドサーチパス**に登録されている必要がある．

1.2.1 ソースプログラムの記述や入出力における文字コードの指定

ソースプログラムに日本語など多バイトの文字コードを含む場合は，使用している文字コードをコンパイル時に指定しなければならないことがある．例えば，Windows 環境ではソースプログラムに使用する日本語文字は MS932 [5] であることが想定されており，それ以外の文字コードがソースプログラムに含まれていると，コンパイル時にエラーが発生する．次の例は，utf-8 の文字コードを含むプログラム test01_utf8.java を javac コマンドでコンパイルしようとした例である．

プログラム：test01_utf8.java（日本語部分の文字コードは utf-8）

```
1 class test01_utf8 {
2     public static void main( String argv[] ) {
3         System.out.println("日本語のテスト");
4     }
5 }
```

例. MS932 を想定する javac コンパイラで上記プログラムをコンパイルする例

```
javac test01_utf8.java  Enter      ←コンパイルの実行
test01_utf8.java:3:  エラー:  この文字は、エンコーディング MS932 にマップできません
          System.out.println("謌譖隱槭?繝?繧繝?");
                              ^
test01_utf8.java:3:  エラー:  この文字は、エンコーディング MS932 にマップできません
          System.out.println("謌譖隱槭?繝?繧繝?");
                                ^
test01_utf8.java:3:  エラー:  この文字は、エンコーディング MS932 にマップできません
          System.out.println("謌譖隱槭?繝?繧繝?");
                                  ^
エラー 3 個
```

この例では文字コードが適切でない旨のエラーメッセージが表示されておりコンパイルが失敗している．このような場合，次のようにしてコンパイラにソースプログラムの文字コードを指定する．

```
javac -encoding utf-8 test01_utf8.java  Enter
```

[5] **シフト JIS** コード（SJIS，Shift_JIS）とほぼ同じコード体系

「-encoding 文字コード」は文字コード指定のコンパイルオプションである．

java コマンドによってバイトコードを実行する際も文字コードを適切に指定しなければならないことがある．Windows 環境の java コマンドでは，入出力に使用する日本語文字は MS932 であることが想定されており，これ以外の文字コードを扱う CLI 環境でバイトコードを実行すると多バイト文字が正しく表示されない．例えば，文字コードが utf-8 に設定された CLI 環境で先の例で作成したバイトコードを実行すると次のようになる．

例. utf-8 のコマンド環境で MS932 の文字を表示する例

```
java test01_utf8  [Enter]      ←バイトコードを実行
{ ^ eXg                        ←日本語が正しく表示されない（いわゆる「文字化け」）
```

バイトコードを実行する際の入出力の文字コードを utf-8 に設定するには，次のようにして java コマンドを実行する．

例. utf-8 の CLI 環境で実行時の入出力の文字コードを utf-8 にしてバイトコードを実行する例

```
java -Dfile.encoding=utf-8 test01_utf8  [Enter]      ←バイトコードの実行
日本語のテスト                          ←日本語が正しく表示されている．
```

「-Dfile.encoding=文字コード」は実行時の入出力の文字コードを指定するオプションである．

1.2.2 jar 形式のライブラリを利用する方法

Java で利用できるプログラムライブラリの多くが **jar 形式**のファイル（'〜.jar'）として配布[6] されている．jar ファイルを利用するには，Java のソースプログラムをコンパイルする際に，jar ファイルのパスを指定する．具体的には javac コマンド実行時にオプション

'**-cp jar ファイルのパス**'

を与える．例えば SWT ライブラリを使用する Java のソースプログラム `swtprog01.java` をコンパイルする際，SWT ライブラリ `swt.jar` がカレントディレクトリ「.」に存在する場合

```
javac -cp swt.jar swtprog01.java
```

というコマンド操作でコンパイル[7] する．複数のライブラリを使用する場合はセミコロン「;」で区切って記述する．

例. swt.jar と apfloat.jar を使用するためのコンパイル作業

```
javac -cp swt.jar;apfloat.jar Java のソースファイル
```

実行時に jar ファイルを使用するには java コマンドに同様のオプションを与える．例えば SWT ライブラリを使用する Java のプログラム `swtprog01.class` を実行する際，SWT ライブラリ `swt.jar` がカレントディレクトリ「.」に存在する場合

```
java -cp swt.jar;.  swtprog01
```

というコマンド操作で swtprog01 を実行する．`-cp` オプションにカレントディレクトリ「.」を加えているのは，プログラム本体 `swtprog01.class` のディレクトリを加えるためである．

[6]本書でもサードパーティーが開発した SWT や Apfloat，SQLite 用 JDBC ドライバといったソフトウェアライブラリの使用方法について解説する．

[7]あるいは環境変数 CLASSPATH にクラスライブラリのファイルのパスを設定する．

2　Java 言語

2.1　初めて学ぶ際の留意事項

この章では，Java 言語の文法をはじめとする基本的な事柄について説明する．Java 言語は後に説明する**オブジェクト指向**の言語であり，学びの初期からこれに関する記述が現れる．そのため，初めて Java 言語を学ぶ者にとっては理解に少なからず困難が生じる．例えば，単純なメッセージを 1 行表示するだけのサンプルプログラムを前章 1.2 で test01.java として示したが，その段階で既にオブジェクト指向に関する記述が現れている．Java 言語では開発するアプリケーションプログラムを 1 つの**クラス**として定義し，実際に動作する機能を**メソッド**と呼ばれるものとして記述する．Java 言語におけるクラス定義の形式を概略的に表現すると次のようになる．

【クラスの定義】

```
class　クラス名　{
        メンバの定義
            ⋮
        メソッドの定義
            ⋮
}
```

オブジェクト指向に関する用語や考え方については後の「2.5 オブジェクト指向プログラミング」（p.25）で解説するので，それまでは上に示した概略的な形式を受け入れて本書の内容を読み進めていただきたい．すなわち，

クラス名　→　「プログラム名」
メソッド　→　「手続き（プログラム）の記述」

と読み替え，**メインメソッド**の記述を開始する宣言である「 public static void main(String argv[ ]) 」や，表示出力のための「 System.out.println(…)」といった記述も当面は形式的なものとして受け入れていただきたい．

2.2　変数と値に関する基礎事項

変数は，各種の値を保持するためのものであり，プログラミングにおける最も基本的な要素である．変数はその使用に先立って宣言しておく必要があり，

型名　変数名

と記述する．記述の先頭には格納する値の種類である**型**の名前を記述（型宣言）する．複数の変数をまとめて宣言することもでき，その場合は，

型名　変数名 1, 変数名 2, … , 変数名 n

とコンマ「,」で区切って記述する．Java で使用する変数の代表的な型を表 1 に示す．

一度宣言した変数には，宣言した型以外の値を格納することはできない．このことを指して「Java は**静的な型付け**の言語処理系である」と表現する．

変数への値の格納は**イコール記号**「=」を使用する．例えば int 型で宣言された変数 a に 124 という値を格納するには，

a = 124

と記述する．この際に注意しなければならないこととして，

<u>この記述は「a が 124 に等しい」ことを意味する**数式ではない**</u>

ということがある．すなわち，イコール「=」の記述が意味するものは，

<u>右辺の値（あるいは式の値）を，左辺の**変数に格納する**</u>

という文である．

表 1: 代表的な変数の型

型宣言	意味	備考（値の範囲など）
byte	1 バイト符号付き整数	-128〜127
short	2 バイト符号付き整数	-32768〜32767
int	4 バイト符号付き整数	-2147483648〜2147483647
long	8 バイト符号付き整数	-9223372036854775808〜 9223372036854775807
float	4 バイト浮動小数点数	正値：1.401298E-45 〜 3.4028235E+38 負値：-3.4028235E+38 〜 -1.401298E-45
double	8 バイト浮動小数点数	正値:4.94065645841246544E-324〜 1.79769313486231570E+308 負値:-1.79769313486231570E+308〜 -4.94065645841246544E-324
boolean	真偽型（論理値）	true（真）または false（偽）
char	文字型	2 バイト UNICODE 文字を 1 文字格納する
String	文字列型	任意の長さの文字列 （厳密には変数ではなく，String クラスである）

変数への値の格納の記述は，その変数を最初に宣言する文の中でも可能である．すなわち，

```
int a = 124
```

という記述が可能である．

2.2.1 値の表現

変数に格納する値を表現する場合の表記法を表 2 に示す．

表 2: データ型ごとの値の表現（代入文を例に）

値の表現	解説
int a = 124 int a = 0xff　（=255） int a = 017　（=15）	通常の整数の表記で値を表現する． 0x で始まると 16 進表記となる． 0 で始まると 8 進表記となる．
long a = 123456789012345L	int 型の値の範囲を超える大きさの数には末尾に L を付ける．
double a = 3.1415 double b = 0.31415e1	通常の小数点表記 指数表現　$0.31415 \times 10^1 = 3.1415$
char a = 'A'	char 型の値（1 文字）は**シングルクオーテーション**「'」で括る．
String a = "日本語"	文字列は**タブルクオーテーション**「"」で括る．

2.2.2 算術演算

数値の**算術演算**には表 3 に示すものがあり，これらの演算を組み合わせて更に複雑な計算が実現できる．

表 3: 算術演算

書き方	意味	書き方	意味	書き方	意味
a + b	a ＋ b （和）	a - b	a － b （差）	a * b	a × b （積）
a / b	a ÷ b （商）	a % b	a ÷ b の余り（剰余）		

2.2.3 複数の文の記述

複数の文を実行する順番に記述して連続的に実行することができる．この場合，各文の末尾には**セミコロン**「;」を記述しなければならない．

2.2.4 コメントの記述

Java ではソースプログラムの中に**コメント**を記述することができる．コメントとは，プログラムとしては意味を持たない注釈であり，プログラマにとっての視認性を高めるなどの目的で任意のものを記述する．コメントの書き方としては，スラッシュ記号 2 つ「//」を記述する方法があり，これ以降は行末までがコメントとなる．

コメントの例.

```
int a = 0;     // 変数の宣言です
```

この例では「// 変数の宣言です」の部分がコメントとなる．コメントを記述する更に別の方法として '/*' と '*/' で括る方法がある．これらで括るものは複数の行に渡って記述することができる．(ただし入れ子の記述はできない)

コメントの例.

```
int a = 0;
/* ここからコメントです.
   この行もコメントです.
   ここまでコメントです. */
String s = "Java の文";
```

2.2.5 変数の型と演算に関する例

下記のプログラム Sample1.java は整数の和を求めるものである．

プログラム：Sample1.java

```
1 class Sample1 {
2     public static void main( String argv[] ) {
3         int  a, b, c;
4         a = 2;
5         b = 3;
6         c = a + b;
7         System.out.println(c);
8     }
9 }
```

演習課題. このプログラムをコンパイルして実行せよ．

演習課題. このプログラムの 6 行目を `c = a / b;` に書き換えて割り算を実行すると正しくない計算結果が得られる．原因について考えプログラムを修正せよ．

演習課題. 次のプログラム Sample1_2.java を作成しコンパイルして実行せよ．

プログラム：Sample1_2.java

```
1 class Sample1_2 {
2     public static void main( String argv[] ) {
3         String  a, b, c;
4         a = "12";
5         b = "34";
6         c = a + b;
7         System.out.println(c);
8     }
9 }
```

演習課題. このプログラムにおいて加算演算「+」はどのような処理をしているか考えよ．

2.2.6　メソッドと引数

これまで出力用に `System.out.println( ... )` という文を記述してきた．末尾にある `println( ... )` が実際に処理を行うための**メソッド**と呼ばれるものである．本書では多くのメソッドの使用方法について解説する．メソッドにはその実行に必要となる各種の値や変数などを括弧 '( )' で括って与える．これを**引数**と呼ぶ．

メソッドは基本的に**値を返す**．例えば平方根を求めるメソッド Math.sqrt で 2 の平方根を求める記述は

```
double a = Math.sqrt( 2.0 );
```

となる．このとき `Math.sqrt( 2.0 )` という記述が $\sqrt{2}$ の**値を返す**ものとして扱われ，返された値が変数 a に得られている．

メソッドの中には値を返さないものも存在する．例えば出力用のメソッドである `System.out.println` は値を返さない．(計算結果をもたらさない)

2.2.7　イコール「=」による代入に関して

イコール「=」による変数への値の代入は，その記述自体が値を返す．例えば，

```
int a, b;
b = 2*(a=3);
```

を実行すると a に 3 が，b に 6 が代入される．上記の式に (a=3) があるが，この部分が代入処理を行うと同時に 3 という値を返している．その結果として b には 6 が代入される．これを応用すると**代入の連鎖**が実現できる．

例. 代入の連鎖

```
int a, b, c;
a = b = c = 25;
```

この結果，3 つの変数 a，b，c 全てに 25 が代入される．これは代入の記述が値を返すことによる結果であり，

```
a = (b = (c = 25));
```

と記述したことに等しい．

2.2.8　文字と文字列の扱い

文字列（String オブジェクト）の長さを求めるには length メソッドを使用する．

例. 文字列の長さを取得する

```
String s = "ABCDE";          ←文字列を生成
int n = s.length();          ←文字列長さを n に取得（=5）
```

2.2.8.1　文字列の部分を取り出す

■ 文字列から指定した 1 文字を取り出す： charAt メソッド

文字列から n 番目の文字を 1 文字取り出すには charAt(n) を実行する．

例. "日本国" から '本' の文字を取り出す

```
String s = "日本国";          ←文字列を生成
char c = s.charAt(1);          ←1 番目の文字を 1 文字取り出す（='本'）
```

charAt の実行結果は char 型の値として得られる．

● 文字列の要素の位置

注意すべき点として「文字列の先頭位置は 0 である」ということがある．また文字列の最終位置は「長さ-1」である．

■ **文字列から部分文字列を取り出す：** substring メソッド

文字列から部分文字列を取り出すには substring メソッドを使用する．substring の引数には取り出しの「開始位置」と「終了位置+1」を与える．

例. 部分文字列の取り出し

```
String s = "Java によるプログラミングを学ぶ";          ←文字列を生成
String s1 = s.substring(7,14);          ← 7 番目から 13 番目までを取り出す（="プログラミング"）
```

取り出した部分文字列が s1 に得られる．

■ **文字列の連結**

加算記号「+」で文字列を連結することができる．

例. 文字列の連結 (1)

```
String s = "日本国" + "憲法";          ←"日本国憲法" となる
```

String.join メソッドで文字列を連結することもできる．

例. 文字列の連結 (2)

```
String s = String.join(":","a","b","c","d");     ← "a:b:c:d" となる
```

join メソッドは，指定した区切り文字を境界にして文字列を連結する．join の引数には文字列の配列（ String[ ] ）を与えることもできる．すなわち，

```
String s = String.join( 区切りの文字列，文字列の配列 );
```

として，配列の全要素を連結することができる.（配列に関しては p.21 「2.4 配列」で解説する）

■ **文字列の分割**

split メソッドを使用すると，指定した区切り文字を境にして文字列を分割し，それらを配列にして返す.（配列に関しては p.21 「2.4 配列」で解説する）

例. 文字列の分割

```
String s = "a:b:c:d:e";
String ar[] = s.split(":");
```

この処理により文字列 s がコロン「:」を境にして分割され，各要素が配列 ar に得られる．すなわち，`ar[0]` →`"a"`，`ar[1]` →`"b"`，`ar[2]` →`"c"`，`ar[3]` →`"d"`，`ar[4]` →`"e"` となる．

split メソッドは正規表現に基づく処理ができるが，それに関しては「7.5 文字列の分割：split メソッド」（p.170）で解説する．

2.2.8.2 エスケープシーケンス

エスケープシーケンスは文字列に各種の制御を組み込むものである．例えばエスケープシーケンスの 1 つに「¥n」というものがあり，出力時に改行処理を行う．

例. 出力時に改行処理される文字列

```
String s = "1 行目です．¥n2 行目です．";        ←「¥n」を含む文字列
System.out.println(s);                          ←これを出力する
```

これを実行すると，

```
1 行目です.
2 行目です.
```

と出力され，文字列の「¥n」の所で改行処理されていることがわかる.「¥n」以外にもエスケープシーケンスがあり，使用頻度の高いものを表 4 に示す．

表 4: 使用頻度の高いエスケープシーケンス

ESC	解説	ESC	解説
¥n	改行	¥t	タブ
¥’	シングルクオーテーション	¥”	ダブルクオーテーション
¥¥	¥記号	¥uHHHH	HHHH（16 進数）の **UNICODE** 文字

例. エスケープシーケンスで表現する UNICODE 文字

```
String s = "¥u604b¥u611b";        ← UNICODE で表現された文字列
System.out.println(s);             ←これを出力する
```

これを実行すると，

恋愛

と出力され，16 進数の文字コードで与えられた文字が出力されていることがわかる．

演習課題. ”計算” という文字列を UNICODE 表現のエスケープシーケンスとして記述せよ．

2 つの**文字列の比較**については後の「2.3.1.2 文字列の比較」（p.16）で説明する．

書式編集による文字列生成の方法については後の「2.6.1.1 文字列生成への書式編集の応用」（p.40）で説明する．

2.2.9 値の型の変換

ここでは，与えられた値を異なる型に変換する方法について説明する．まず次の例について考える．

例. 整数（int 型）の値を浮動小数点数（double 型）に変換する

```
int a = 3;              ← int 型の値の生成
double a2 = a;          ← double 型に変換
```

この処理は特に問題なく実行できる．続いて次の例について考える．

例. 浮動小数点数（double 型）の値を整数（int 型）に変換する

```
double b = 3.1415;              ← double 型の値の生成
int b2 = b;                     ← int 型に変換
```

このような文を持つプログラムをコンパイルしようとすると次のようなエラーが発生する．

```
エラー: 不適合な型: 精度が失われる可能性がある double から int への変換
        int b2 = b;
                 ^
エラー 1 個
```

これは値を異なる型に変換しようとして，元の数値の精度が失われる（小数点以下の情報が失われる）ことを通知するエラーである．これは深刻な問題であるのでコンパイラがエラーとして通知している例である．
このような状況でも，精度の低下を許諾し敢えて変換する場合は次のように記述を改める．

例. 浮動小数点数（double 型）の値を整数（int 型）に変換する：改

```
double b = 3.1415;          ← double 型の値の生成
int b2 = (int)b;            ← int 型に変換：改
```

double 型の変数 b の前に (int) と記述することで，b の値を強制的に int 型に変換することができる．このような記述を**キャスト**と呼ぶ．

性質のかけ離れた型の間ではキャストを使用することはできない．

例. キャストが使えない例

```
int a = (int)"日本語";
```
←文字列を int 型にキャストする試み

このような文を持つプログラムをコンパイルしようとすると次のようなエラーが発生する.

```
エラー: 不適合な型: String を int に変換できません:
        int a = (int)"日本語";
                     ^
エラー 1 個
```

"日本語" という文字列が数値として解釈できないことは理解に難くない．また，数値を表現する文字列を int や double といった型に変換する方法は別の形で用意されている.

2.2.9.1 数値と文字列の間での表現の変換

数値と文字列の間で表現を変換する方法を表 5 に示す.

表 5: 数値と文字列の間での表現の変換

文字列（String）から数値型への変換	
`i = Integer.parseInt(S)`	String 型のオブジェクト S の値を数値として解釈し，int 型の変数 i に格納する.
`d = Double.parseDouble(S)`	String 型のオブジェクト S の値を数値として解釈し，double 型の変数 d に格納する.
数値型から文字列（String）への変換	
`S = String.valueOf(d)`	数値型（double,int など）の変数 d の値を表現する文字列を生成し，String 型のオブジェクト S に格納する.

例. 文字列を数値として解釈する

```
int a = Integer.parseInt("3");
```
←文字列を int 型として解釈し a に格納

```
double b = Double.parseDouble("3.1415");
```
←文字列を double 型として解釈し b に格納

例. 数値を文字列として解釈する

```
String a = String.valueOf(3);
```
← int 型の値を String 型として解釈し a に格納

```
String b = String.valueOf(3.1415);
```
← double 型の値を String 型として解釈し b に格納

2.2.10 基数の変換

2.2.10.1 10 進数→ n 進数

整数の値を，指定した基数表現の文字列に変換する方法について説明する.

書き方： Integer.toString(整数値, 基数)

変換結果は String 型のオブジェクトである．第 2 引数の「基数」を省略すると 10 進数の文字列表現が得られる.

例. 10 進数から n 進数への変換

```
int a = 65535;
```
← int 型の値を用意

```
String a2 = Integer.toString(a,2);
```
← 2 進数表現の文字列に変換

```
String a8 = Integer.toString(a,8);
```
← 8 進数表現の文字列に変換

```
String a16 = Integer.toString(a,16);
```
← 16 進数表現の文字列に変換

```
System.out.println(a2+"¥n"+a8+"¥n"+a16);
```
←出力

これを実行すると次のような出力が得られる.

```
1111111111111111
177777
ffff
```

2.2.10.2　n 進数→ 10 進数

Integer.toString の逆の働きをする parseInt メソッドが使用できる．

書き方：　Integer.parseInt(文字列, 基数)

変換結果は int 型の値である．第 2 引数の「基数」を省略すると 10 進数として解釈される．

例. n 進数から 10 進数への変換（先の例の続き）

```
int b2 = Integer.parseInt(a2,2);          ←文字列を 2 進数として解釈し整数に変換
int b8 = Integer.parseInt(a8,8);          ←文字列を 8 進数として解釈し整数に変換
int b16 = Integer.parseInt(a16,16);        ←文字列を 16 進数として解釈し整数に変換
```

2.2.11　ビット演算

計算機上では記憶はバイト単位で格納されており，更に各バイトは 8 つのビットから構成（図 2）される．

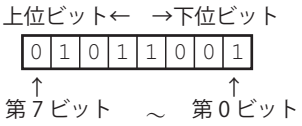


図 2: 1 バイトは 8 ビットで構成される

勿論，整数の値もそれを表す 2 進数のビット表現（図 3）[8] になっている．

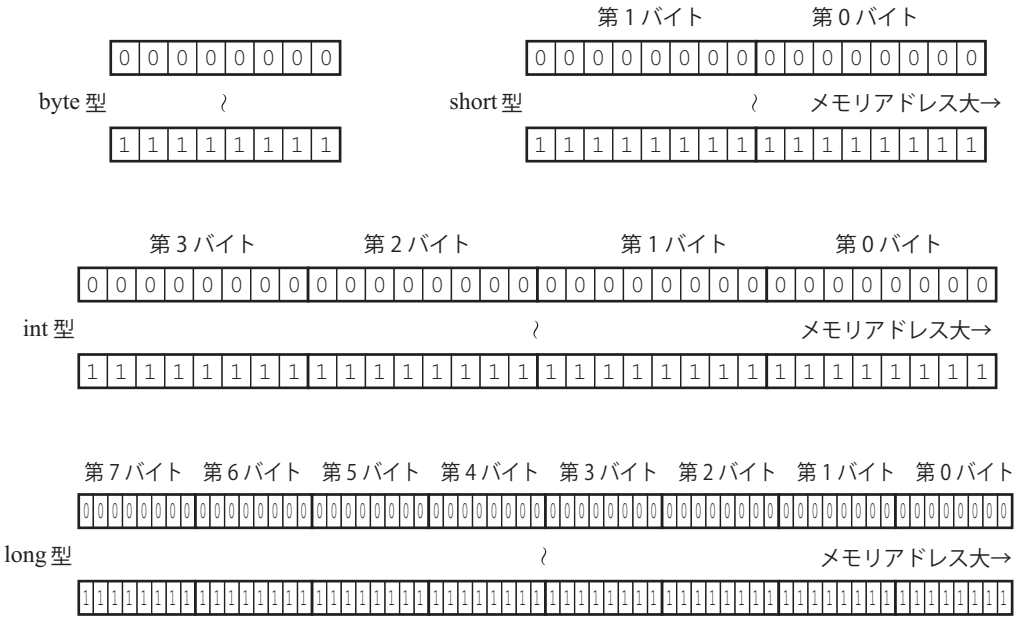


図 3: 整数のビット表現

2.2.11.1　シフト演算

整数は 2 進数で表現されているので，各種のビット操作により特徴的な演算ができる．ビット操作の中でも重要なものに**シフト演算**がある．シフト演算は記憶のビット列を右あるいは左にずらすもの（右シフト／左シフト）である．シフト演算は変数や値に対して行うことができ，左に n ビットシフトすると 2^n 倍され，右に n ビットシフトすると $\frac{1}{2^n}$ 倍される．(表 6)

[8] Java 以外の言語処理系では整数値を格納するバイト順が異なることがある．Java のような順番の格納形式を**ビッグエンディアン**といい，その逆を**リトルエンディアン**という．

表 6: シフト演算

演算	式		意味
左シフト	$v << n$	→	v の値を 2^n 倍する
右シフト	$v >> n$	→	v の値を $\frac{1}{2^n}$ 倍する

例. 左シフト演算

```
int a = 1 << 8;          ←1 を左に 8 ビット分シフト（2^8 倍）する演算
```

この結果，a の値は 256 となる．

演習課題. `1 << 31` はどのような値になるか確かめ，何故そうなるのかについて考えよ.（ヒント：「2 の補数表現」）

■ シフト演算の再帰代入

シフト演算（表 6）の結果，対象となった値は変化しない.（次の例）

例. シフト演算前後で値は変わらない

```
int a = 1;               ←a に 1 を代入
int b = a << 3;          ←a 左に 3 ビットシフトする
```

この結果，`b` の値は 8 となるが，`a` は 1 のままである．`a` の値を変化させる型でシフトするには

```
a = a << 3
```

とする．あるいは**再帰代入**の形 ‘`<<=`’ で

```
a <<= 3
```

と記述することもできる．同様に右シフトの再帰代入 ‘`>>=` ’ も可能である．

2.2.11.2 ビット毎の論理演算

整数値のビット毎の論理演算を行うための演算子を表 7 に示す.

表 7: 論理演算

演算	説明
$a \mid b$	a と b の各ビット同士の論理和（OR）をとった値
a & b	a と b の各ビット同士の論理積（AND）をとった値
a ^ b	a と b の各ビット同士の排他的論理和（XOR）をとった値
~a	a の各ビットを反転した値（**ビット反転**：1 の補数）

例. 整数値同士のビット積

```
int a = Integer.parseInt("1111",2);
int b = Integer.parseInt("1010",2);
int c = a & b;
String s = Integer.toString(c,2);
System.out.println(s);
```

これは 1111(2 進) と 1010(2 進) のビット積を求める例である．これを実行すると ‘`1010`’ と表示される．

■ ビット毎の論理演算の再帰代入

ビット毎の論理演算（表 7）の結果，対象となった値は変化しない．先の例に続いて

```
System.out.println(Integer.toString(a,2));
```

とすると ‘1111’ と出力され，a の値が変化していないことがわかる．
演算の結果を a に与えるには

```
a = a & b
```

とする．あるいは**再帰代入**の形 ‘&=’ で

```
a &= b
```

と記述することもできる．同様に

ビット毎の論理和の再帰代入	`|=`
ビット毎の排他的論理和の再帰代入	`^=`

も使用できる．

演習課題． int 型整数の指定したビットを 1 にするプログラムを実現せよ．（他のビットは変更しない）

演習課題． int 型整数の指定したビットを 0 にするプログラムを実現せよ．（他のビットは変更しない）

演習課題． int 型整数の指定したビットの値を取得するプログラムを実現せよ．

2.3 制御構造

2.3.1 条件分岐 (1)： if 文

if 文により条件判定と実行の分岐ができる.

【if による条件分岐】

・その 1

```
if ( 条件式 ) {
        「条件式」が真である場合に実行するプログラム
}
```

・その 2

```
if ( 条件式 ) {
        「条件式」が真である場合に実行するプログラム
} else {
        「条件式」が偽である場合に実行するプログラム
}
```

条件式が偽であった場合に，更に別の条件を判定するには次のように記述する.

【if による条件分岐】

・その 3

```
if ( 条件式 1 ) {
        「条件式 1」が真である場合に実行するプログラム
}  else if ( 条件式 2 ) {{
        「条件式 1」が偽の場合,「条件式 2」が真である場合に実行するプログラム
                ⋮
```

このように,「条件式 1」が偽の場合は「条件式 2」を検査し，更に偽の場合は… と条件の判定を連鎖することができる.

if 文による条件判定を応用したサンプルプログラム を示す.

プログラム：Bmi0.java

```
1  class Bmi0 {
2      public static void main( String argv[] ) {
3          double weight = 65.0;   // 体重
4          double height = 1.65;   // 身長
5          double bmi = weight / height / height;  // BMI値の計算
6          System.out.println("BMI:"+bmi);
7          if ( bmi < 18.5 ) {
8              System.out.println("痩せています. ");
9          } else if ( bmi < 25.0 ) {
10             System.out.println("普通です. ");
11         } else {
12             System.out.println("太っています. ");
13         }
14     }
15 }
```

これは「ボディマス指数」を体重と身長の値から算出して，その値で適正体重か否かを判定するプログラムである.

演習課題. 世界保健機構（WHO）では，ボディマス指数による適正体重を次の表のような基準で分類している．先のプログラム Bmi0.java をこの基準に基づいて判定する形に修正せよ．

判定	BMI 値
痩せすぎ	16.00 未満
痩せ	16.00 以上，16.99 以下
痩せぎみ	17.00 以上，18.49 以下
普通体重	18.50 以上，24.99 以下
前肥満	25.00 以上，29.99 以下
肥満（1 度）	30.00 以上，34.99 以下
肥満（2 度）	35.00 以上，39.99 以下
肥満（3 度）	40.00 以上

2.3.1.1 条件式の記述と boolean 型変数

条件判定のために記述する**条件式**は，関係を記述するもの（表 8）と条件の**論理的結合**など（表 9）からなる．

表 8: 値の関係（数値の関係）

関係の記述	意味	関係の記述	意味
値 1 == 値 2	値 1 と値 2 は等しい	値 1 != 値 2	値 1 と値 2 は等しくない
値 1 < 値 2	値 1 は値 2 より小さい	値 1 > 値 2	値 1 は値 2 より大きい
値 1 <= 値 2	値 1 は値 2 以下	値 1 >= 値 2	値 1 は値 2 以上

※文字列の比較の方法に関しては後述する．

表 9: 条件の論理的結合

関係の記述	意味
条件 1 && 条件 2 && … && 条件 n	条件 1〜条件 n は全て真
条件 1 || 条件 2 || … || 条件 n	条件 1〜条件 n の内，少なくとも 1 つは真
! 条件	条件は真でない

論理的結合は組み合わせることができる．例えば

「A または B」かつ「C または D」

という条件を記述するために，

(A || B) && (C || D)

という記述が可能である．

条件は値を持つ．これは理解し難い場合があるので，次のプログラム BooleanTest1.java を例に考える．

プログラム：BooleanTest1.java

```
1 class BooleanTest1 {
2     public static void main( String argv[] ) {
3         boolean b;
4         b = 3 < 5;
5         System.out.println(b);
6     }
7 }
```

3 行目で **boolean** 型変数 b を確保し，その変数に「3 < 5」という条件の式が代入されている．条件の判定結果は **true**（真）か **false** （偽）の値を取るので，b には true が格納される．このような真理値（true / false）は boolean

型の値であり，それを格納する変数は boolean 型で宣言する．

条件を判定するための各種のメソッドも存在し，それらが返す値は boolean 型である．boolean 型のメソッドは，条件の式として扱うことができる．

2.3.1.2　文字列の比較

文字列の比較には表 8 に挙げた記述を使うことができないので compareTo メソッドを使用する．これを用いたサンプルプログラムを StrComp01.java に示す．

プログラム：StrComp01.java

```
1  class StrComp01 {
2      public static void main( String argv[] ) {
3          String  s1 = "あいうえお",
4                  s2 = "かきくけこ",
5                  s3 = "あいうえお";
6          // 順序の判定
7          System.out.println( "\""+s1+"\"と\""+s2+"\"のcompareToによる比較:");
8          System.out.println( s1.compareTo(s2) );
9          System.out.println( "\""+s1+"\"と\""+s3+"\"のcompareToによる比較:");
10         System.out.println( s1.compareTo(s3) );
11         System.out.println( "\""+s2+"\"と\""+s3+"\"のcompareToによる比較:");
12         System.out.println( s2.compareTo(s3) );
13         // 等しいかどうかのみの判定
14         System.out.println( "\""+s1+"\"と\""+s2+"\"のequalsによる比較:");
15         System.out.println( s1.equals(s2) );
16         System.out.println( "\""+s1+"\"と\""+s3+"\"のequalsによる比較:");
17         System.out.println( s1.equals(s3) );
18     }
19 }
```

このプログラムを実行すると次のように表示される．

```
"あいうえお"と"かきくけこ"の compareTo による比較:
-9
"あいうえお"と"あいうえお"の compareTo による比較:
0
"かきくけこ"と"あいうえお"の compareTo による比較:
9
"あいうえお"と"かきくけこ"の equals による比較:
false
"あいうえお"と"あいうえお"の equals による比較:
true
```

文字列に対する compareTo メソッドは文字コードの順に従って文字列を比較する．すなわち，`s1.compareTo(s2)` の結果は次のような値となる．

s1 < s2	s1 と s2 が等しい	s1 > s2
負の値	0	正の値

2 つの文字列の内容が 同じかどうかのみを判定 する場合は equals メソッドを使用すると良い．すなわち String クラスの s1, s2 があるとき `s1.equals(s2)` を評価すると，内容が同じなら true ，異なれば false となる．

2.3.1.3　条件判定のための 3 項演算子

条件判定の結果によって返す値を選択する **3 項演算子**（下記）がある．

条件式　?　条件式が真の場合に返す値　:　偽の場合に返す値

例. 偶数／奇数を判定するプログラム

```
int n = 5;
String s = (n%2==0)?  "even" :  "odd";
System.out.println(s);
```

これを実行すると「odd」が表示される.

2.3.2 条件分岐 (2)： switch 文

if 文は，値の比較などの条件式を判定して実行するプログラムを選択するが，これとは別に，変数の値に応じて実行する処理を選択するための **switch** 文 がある.

【switch 文】変数の値に応じて実行する処理を選択する

```
switch (変数) {
    case 値 1:
        （処理 1）
        break;
    case 値 2:
        （処理 2）
        break;
            :
    default:
        （上記のいずれにも該当しない場合の処理）
}
```

switch 文は，複雑な条件を判定するためのものではなく，変数の値を個別に判定するものである．また，case の記述により選択した処理は，その部分の処理が終了しても，switch 構文の最後まで処理を続行（**フォールスルー**）する．したがって，所望の処理が終了するところで break を挿入する必要がある．
case で指定した値のどれにも該当しない場合の処理は default 以下に記述する.

switch 文を応用したサンプルプログラム Switch01.java を示す.

プログラム：Switch01.java

```
1  class Switch01 {
2      public static void main( String argv[] ) {
3          String s = "冬";
4          switch ( s ) {
5          case "春":
6              System.out.println("spring");
7              break;
8          case "夏":
9              System.out.println("summer");
10             break;
11         case "秋":
12             System.out.println("autumn");
13             break;
14         case "冬":
15             System.out.println("winter");
16             break;
17         default:
18             System.out.println("other");
19         }
20     }
21 }
```

2.3.3　反復 (1)： while 文

プログラムの反復実行のために **while** 文がある．

【while による反復】 -その 1 -

while　(条件式)　{
　　「条件式」が真である間，反復実行するプログラム
}

while 文を応用して，指定した値まで数を数えるプログラムを Rep01.java に示す．

プログラム：Rep01.java

```
1 class Rep01 {
2     public static void main( String argv[] ) {
3         int c = 0;              // 数を数えるための変数
4         while ( c < 10 ) {      // cが10未満の間数え続ける
5             System.out.print(c+",");
6             c = c + 1;          // 次の数の算出と更新
7         }
8     }
9 }
```

このプログラムは，はじめに c に 0 をセットし，c の値を 1 つずつ増加させながらそれを出力するものである．繰り返しの条件は `c < 10` としており「c が 10 未満である間」反復する．

2.3.3.1　再帰代入による変数の値の変更

プログラムの 6 行目に `c = c + 1` という記述があるが，これは右辺の評価結果の値を左辺に代入する，すなわち「現時点の c の値に 1 を足す」「その値を c に上書きする」という処理である．このような形で変数の値を変更する（直前の値から次の値を生み出す）処理は**再帰代入**と呼ばれ，実際のプログラミングにおいて頻繁に用いられる．

再帰代入のために更に簡略化した演算子（表 10）が用意されている．

表 10: 再帰代入のための演算（代表的なもの）

書き方	処理の意味	解説
`n += c`	`n = n + c`	現時点の n の値に c の値を加えたものを n に上書きする．
`n -= c`	`n = n - c`	現時点の n の値から c の値を引いたものを n に上書きする．
`n *= c`	`n = n * c`	現時点の n の値を c 倍した値を n に上書きする．
`n /= c`	`n = n / c`	現時点の n の値を c で割ったも値を n に上書きする．

更に，変数の値を 1 つ増やす（減らす）処理には表 11 のような演算子も用意されている．

表 11: インクリメント／デクリメント

書き方	処理の意味	解説
`n++`	`n = n + 1`	変数 n の値を 1 つ増やす.（**インクリメント**）
`n--`	`n = n - 1`	変数 n の値を 1 つ減らす.（**デクリメント**）

演習課題． 演算子 `++`，`--` は変数の前に記述することもできるが，変数の後ろに記述した場合との違いについて調べよ．

例題　1 から 100 までの整数の合計を求めるプログラムを書け．

「1+2+…+100」と全てを記述するのは現実的な方法ではない．Addall1.java に示すプログラムについて考える．

プログラム：Addall1.java

```
1  class Addall1 {
2      public static void main( String argv[] ) {
3          int c, n;
4          c = 1;            // 回数を数える変数
5          n = 0;            // 計算結果を保持する変数
6          while ( c <= 100 ) {
7              n += c;       // 加算（累積）
8              c++;          // 次の回数を算出
9          }
10         System.out.println(n);
11     }
12 }
```

考え方：

足すべき数 1,2,…,100 を変数 c に次々と作り，それを変数 n に累積してゆく．

プログラムの流れ：

整数を順番に生成するための変数として c が宣言されている．4 行目の「c=1」により，c にまず 1 がセットされる．合計を計算する変数として n が宣言されている．5 行目の「n=0」により，n が初期化され，何も加算されていない状態になる．

6 行目の while 文で c の値が調べられ，それが 100 以下の場合に限り 7,8 行目を実行する．7,8 行目では簡略化された再帰代入によって n に数を累積し c の値を 1 つ増やしている．6 行目に戻り，c の値が 100 を超えるまで同様の処理を繰り返す．

演習課題． Addall1.java を変更して 1〜10 億までの数を加算せよ．

ポイント． 正しくない加算結果が出た場合，その原因を考えて解決せよ．

【while による反復】 - その 2 -

do　{
　　反復実行するプログラム
} while　（ 条件式 ）

この構文による反復では「反復実行するプログラム」を実行した後で，次回実行するかどうかを「条件式」で判定する．すなわち「反復実行するプログラム」を必ず 1 回は実行する．while (...) {...} の構文と do {...} while(...) の構文による反復の違いを次の while01.java と while02.java で確かめる．

プログラム：while01.java

```
1  class while01 {
2      public static void main( String argv[] ) {
3          int c = 0;
4          String s = "abcdefgha";
5          while ( c < s.length() && s.charAt(c) != 'a' ) {
6              System.out.print(s.charAt(c)+",");
7              c++;
8          }
9          System.out.println("終了");
10     }
11 }
```

プログラム：while02.java

```
1  class while02 {
2      public static void main( String argv[] ) {
3          int c = 0;
```

```
4          String s = "abcdefgha";
5          do {
6              System.out.print(s.charAt(c)+",");
7              c++;
8          } while ( c < s.length() && s.charAt(c) != 'a' );
9          System.out.println("終了");
10     }
11 }
```

これら 2 つのプログラムは一見して似ているが実行結果が異なる．どちらも反復の終了条件が「文字列中に'a' を見つけた場合に反復終了」というものであるが，while02.java の do {...} while(...) 構文では 初回は必ず実行する という性質上，初回の'a' では反復を終了しない．プログラムの実行例を次に示す．

while01.java の実行：
　　終了　　←文字列先頭の 'a' の検出によって反復終了

while02.java の実行：
　　a,b,c,d,e,f,g,h, 終了　　←文字列末尾の 'a' の検出によって反復終了

2.3.4　反復 (2)： for 文

プログラムの繰り返しに際しては，

(1) 状態の初期化
(2) 処理の実行
(3) 次の処理のための準備

の 3 つの部分を記述することが多い．例えば前述のプログラム Addall1.java では 4,5 行目で変数 c,n を初期化しているがこれが (1) にあたる．そして 7 行目が (2) にあたり，8 行目で次の繰り返し処理に備えて c の値を 1 つ増やしており，これが (3) にあたる．

このようなスタイルの繰り返し処理をわかりやすい形で実現するための方法がある．Addall1.java と同じ計算を行う別のプログラム Addall2.java について考える．

プログラム：Addall2.java

```
1  class Addall2 {
2      public static void main( String argv[] ) {
3          int c, n;
4          n = 0;
5          for ( c = 1; c <= 100; c++ ) {
6              n += c;
7          }
8          System.out.println(n);
9      }
10 }
```

5〜7 行目の **for 文**で繰り返しを実現している．

【for による反復】

for　(初期化処理;　実行条件;　次の実行のための準備)　{
　　「実行条件」が真である間，反復実行するプログラム
}

for 文には**拡張 for 文**という構文もある．これに関しては「2.4.4 拡張 for 文による要素へのアクセス」(p.24) で解説する．

2.4 配列

先に解説した**変数**は値を保持するために不可欠な要素であるが，1 つの変数には 1 つの値しか格納できない．実際のプログラミングには多数の要素をもつ（あるいは多次元の構造をもつ）データ集合を保持することが必要になる．

例えば，数列の中の最大値を求めるプログラムについて考える．

【例】数列 $a_0, a_2, a_3, \cdots, a_{19}$ の最大値を求める手順

1. 数列の最初の項 a_0 を最大値の候補として max にセットする．
2. a_1 から a_{19} を順番に max と比較する．max よりも大きい項があれば，
   その項の値を max にセットする．
3. max の値を出力する．

これを Java 言語で実装したものがプログラム Getmax.java である．

プログラム：Getmax.java

```
1  class Getmax {
2      public static void main( String argv[] ) {
3          int[] a;            // 配列であることの宣言
4          int c, max;
5          a = new int[20];    // 整数20個分の配列を割り当てる
6          //--- 配列に乱数を作成 ---
7          for ( c = 0; c < 20; c++ ) {
8              a[c] = (int)(Math.random() * 100.0);    // 乱数生成
9              System.out.print(a[c]+",");
10         }
11         //--- 最大値を探す ---
12         max = a[0];         // 最大値の最初の候補
13         for ( c = 1; c < 20; c++ ) {
14             if ( max < a[c] ) {
15                 max = a[c];
16             }
17         }
18         System.out.print(" max="+max+", ("+a.length+" elements)");
19     }
20 }
```

解説：

添字を付けて複数のデータを格納する記憶構造が**配列**である．このプログラムの例では，20 個の整数を格納する配列を確保して，そこに数列の各項を格納している．配列変数の使用を宣言する場合は，型の直後に ‘[ ]’ を記述し，その後に配列変数の名前を記述する．

配列変数の宣言：　型名 [ ]　変数名

多次元の配列を宣言するには，　‘**型名** [ ][ ]…[ ]　**変数名**’ とする．

1 次元の整数型配列 a を宣言する例：　`int[ ]  a`

3 次元の整数型配列 b を宣言する例：　`int[ ][ ][ ]  b`

配列宣言のための ‘[ ]’ は変数名の直後に記述しても良い．

例． `int a[ ]`

今回のプログラムでは，整数の数列を格納するために，整数の配列の使用を宣言（3 行目）している．通常の変数と異なる点として注意すべきことがある．それは，

配列変数を宣言しただけでは，実際に値を格納する記憶域はまだ確保されていない

ということである．配列のための記憶域の確保には，配列のサイズなどを指定する必要があり，これを行うには ‘new’ を使用する．

2.4.1　配列への記憶の割り当て

プログラムの 5 行目にある ‘`a = new int[20]`’ は「20 個の長さの 1 次元整数型配列の記憶域を新たに確保して，それを変数名 a に割り当てる」ことを意味する．

（配列の添字に関する注意）

配列の要素には，配列名の後ろに ‘**[添字]**’ を付けてアクセスする．(「添字」を「**インデックス**」と呼ぶこともある）配列変数の添え字は 0 から始まる．今回のプログラムでは配列の要素の数は 20 であるが，添字は 0 から始まり 19 で終わる．

プログラムの 7〜10 行目で，数列の各項に乱数を設定している．乱数の発生には `Math.random()` メソッドを用いる．

> **乱数の発生：**　Math.random()
> 0 以上 1 未満の乱数（double 型）を生成する．

今回の例では，発生した乱数を 100 倍した後，型変換 ‘`(int)`’ により整数型に変換している．これにより 0〜99 の乱数を発生することができる．

プログラムの 12〜17 行目で最大値を探し，18 行目でそれを出力している．

Getmax.java をコンパイルして実行した例を次に示す．

```
83,3,70,14,25,69,87,20,89,48,79,91,0,53,68,79,20,89,9,68, max=91, (20 elements)
```

実行の度に異なる数列が得られる．

■ 配列のサイズの取得

プログラムの 18 行目に ‘`a.length`’ という記述がある．配列の後ろに ‘.length’ を付けると，その配列の要素の個数が得られる．このことは次のサンプルプログラム ChkArrayLength.java で確認できる．

プログラム：ChkArrayLength.java

```
 1  class ChkArrayLength {
 2      public static void main( String argv[] ) {
 3          int a[]      = new int[30];          // 1次元配列
 4          int b[][][] = new int[2][3][4];      // 3次元配列
 5          System.out.println("a.length:"+a.length);
 6          System.out.println("b.length:"+b.length);
 7          System.out.println("b[0].length:"+b[0].length);
 8          System.out.println("b[0][0].length:"+b[0][0].length);
 9      }
10  }
```

このプログラムを実行すると次のような出力が得られる．

```
a.length:30
b.length:2
b[0].length:3
b[0][0].length:4
```

2.4.2　配列宣言時の初期化

配列には宣言時に指定した値の並びを与えることができる．例えば int 型の配列 a を宣言する際に

```
int[] a = {1,2,3,4}
```

と記述すると，`a[0]=1`，`a[1]=2`，`a[2]=3`，`a[3]=4` という 4 つの要素を持つ配列 a ができる．

2.4.3 配列の要素の整列

Arrays クラス（java.util.Arrays）の sort メソッドを用いると，配列の要素の順序を整列することができる．

書き方： **Arrays.sort(配列)**

sort メソッドを用いたサンプルプログラム SortArray.java を示す．

プログラム：SortArray.java

```
import java.util.Arrays;

class SortArray {
    public static void main( String argv[] ) {
        // 配列の初期化
        int[] a = {9,8,6,3,5,4,2,7,1};
        int c;
        // 内容確認
        System.out.print("整列前：");
        for ( c = 0; c < a.length; c++ ) {
            System.out.print(a[c]+",");
        }
        System.out.println();
        // 整列処理
        Arrays.sort(a);
        System.out.print("整列後：");
        for ( c = 0; c < a.length; c++ ) {
            System.out.print(a[c]+",");
        }
    }
}
```

このプログラムを実行した例を次に示す．

```
整列前：9,8,6,3,5,4,2,7,1,
整列後：1,2,3,4,5,6,7,8,9,
```

演習課題． 配列の要素の順序を反転するプログラムを作れ．

Comparator を使用すると整列順序の基準を柔軟に設定することができる．例えば次のようなプログラムについて考える．

プログラム：SortArray2.java

```
import java.util.Arrays;
import java.util.Comparator;

// 文字列の3文字目以降から比較するComparator
class MyCmp implements Comparator<String> {
    public int compare( String s1, String s2 ) {
        String k1 = s1.substring(2,s1.length());
        String k2 = s2.substring(2,s2.length());
        return( k1.compareTo(k2) ); // keyを比較
    }
}

// メイン
class SortArray2 {
    public static void main( String argv[] ) {
        // 配列の初期化
        String[] a = { "A-orange", "B-grape", "C-banana", "D-apple" };
        int c;
        // 内容確認
        System.out.print("整列前： ");
        for ( c = 0; c < a.length; c++ ) {
            System.out.print(a[c]+",");
        }
        System.out.println();
```

```
25          // 整列処理(1)
26          Arrays.sort(a);
27          System.out.print("整列後1：");
28          for ( c = 0; c < a.length; c++ ) {
29              System.out.print(a[c]+",");
30          }
31          System.out.println();
32          // 整列処理(2)
33          Arrays.sort(a, new MyCmp() );
34          System.out.print("整列後2：");
35          for ( c = 0; c < a.length; c++ ) {
36              System.out.print(a[c]+",");
37          }
38      }
39  }
```

このプログラムでは文字列（String）の配列 a の要素を整列する．ただし，**Arrays.sort(配列)** の書き方では配列の要素の先頭の文字も含めた整列となる．

「各要素の 3 文字目以降で整列」という基準にするために Comparator **クラス** MyCmp を定義している.「クラス」の概念をはじめ，Comparator に関しては**オブジェクト指向**の概念が必要となるので，基本的な考え方を含め，後の「3.1.1.5 Comparator を用いた整列」（p.56）の所で詳しく説明する．

SortArray2.java をコンパイルして実行した例を次に示す．

```
整列前： A-orange,B-grape,C-banana,D-apple,
整列後 1：A-orange,B-grape,C-banana,D-apple,
整列後 2：D-apple,C-banana,B-grape,A-orange,
```

2.4.4 拡張 for 文による要素へのアクセス

拡張 for 文を用いると簡潔な方法で配列の要素に順次アクセスすることができる．

for (型名　変数名 : 配列名) { 処理内容 }

拡張 for 文を応用したサンプルプログラムを示す．

プログラム：ExFor01.java

```
1  class ExFor01 {
2      public static void main( String argv[] ) {
3          String[] seasons = {"spring","summer","autumn","winter"};
4          // 拡張for文
5          for ( String s : seasons ) {
6              System.out.println(s);
7          }
8      }
9  }
```

このプログラムの実行例を次に示す．

```
spring
summer
autumn
winter
```

2.5　オブジェクト指向プログラミング

オブジェクト指向プログラミング（以下 **OOP** と記す）の説明の前に，概念的な準備をする．

2.5.1　概念的な導入

各種システムの振る舞いを一般化したモデルとして**有限状態機械**（FSM:Finite State Machine）がある．有限状態機械は内部に変数などを持ち，それらの値からなる**状態**を**アクション**によって変化させる機械である．コンピュータのプログラムも，変数の状態を変化させながら動作する抽象的な機械と見ることができ，その意味では有限状態機械である．オブジェクト指向に根ざしたプログラミング言語は有限状態機械を実装するための簡便な方法[9]を与える．

2.5.1.1　オブジェクト指向における部品の考え方

実際の機械では，それを構成する部品は単なる変数という概念にとどまらない，ある役割を持った存在である．例えば室内空調機（エアコン）を例に考えてみると，それはヒーターと冷却器，更にそれらを制御するコントローラからなる装置とみなすことができる．もちろん抽象的には，室温という変数を上下させる機械であるが,「ヒーターの温度」,「冷却器の温度」,「部屋の気温」といった，装置を設計する者の解釈があり，オブジェクト指向では設計者の意図を具体化しやすい表現方法が与えられている．

2.5.1.2　オブジェクト指向プログラミングの概要

実際の機械を構成する部品は，限られた種類に分類することができる．例えば電子装置を構成する部品はおおよそ

“抵抗器，コンデンサ，トランス，コイル，トランジスタ，ダイオード，IC（集積回路），…”

といったものに分類することができる．すなわち，装置の部品には，機能や目的毎に，それが属する「種類」がある．オブジェクト指向プログラミングにもこれと同じ概念があり，作成しようとするプログラムは様々な部品から構成され，更にそれら各部品が属する**クラス**がある．従って，プログラムの開発は，実装しようとする機能を部分的な機能に分けて考え，それらが属するクラスを定義し，その部品の振る舞い（アクション）を**メソッド**として実装するという流れになる．

2.5.2　サンプルプログラムに沿った説明

ここでは 1 つのサンプルプログラムの作成を通してオブジェクト指向に関する具体的な記述の方法を示す．作成するプログラムは，文字列データを次々と記録，蓄積する**スタック**（図 4,5）を実現するものである．

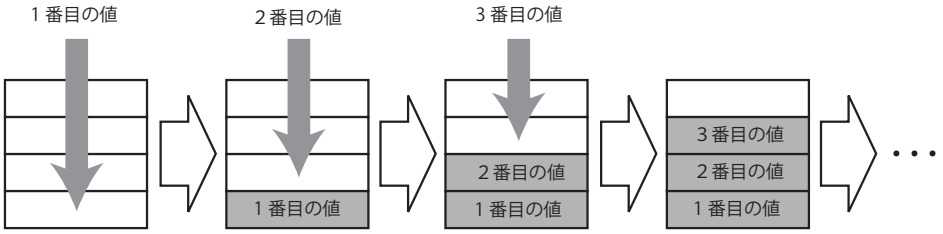


図 4: スタックに値を蓄える操作（push）

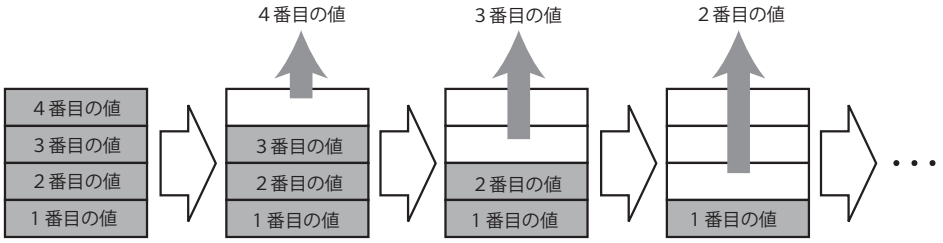


図 5: スタックから値を取り出す操作（pop）

[9]オブジェクト指向プログラミング（OOP）の考え方は FSM に由来するものではないが，FSM の考え方と親和性が高いため，OOP の導入解説のために FSM を紹介した．

(1) クラスの定義

class　クラス名　{ 定義 }

「定義」の部分に，そのクラスを構成するメンバ（変数など）やメソッドを記述する．

(2) メンバの定義

クラスを構成する要素を記述する．例えば，スタックの機能を実装する場合，データを格納するための記憶域と，スタックポインタ，それにスタックポインタの最大値を格納する変数が必要となる．

スタック 'MyStack' のクラスを構成する例：

```
class MyStack {
    String[ ]    memory;      // データを格納する配列
    int          p, max;      // スタックポインタと，その最大値
}
```

この記述により，必要なデータ構造を持った 'MyStack' というクラスが利用できるようになる．

(3) メソッドの定義

スタックはデータの蓄積（push），取り出し（pop）といった振る舞いをするが，このような実際の動作を実現するのが「メソッド」である．次のような記述によりメソッドを定義する．

戻り値の型　メソッド名 (引数の記述)　{ 処理内容 } 引数は「**型 1　引数名 1，　型 2　引数名 2，　…　**」の形で記述する．

メソッド定義の冒頭に記述する「引数の記述」のことを**仮引数**と呼ぶ．

(4) インスタンスの生成

クラスの定義ができたら，そのクラスの具体的なオブジェクトである**インスタンス**を生成し，そのインスタンスにデータを保持する．またメソッドもインスタンスに対して実行する．もちろん，同じクラスから複数のインスタンスを生成し，それぞれに独立したオブジェクトとして扱うことができる．

以上を踏まえて，スタック機構を実装するプログラム StackTest.java について具体的に考える．

プログラム：StackTest.java

```
1  //スタックの定義
2  class MyStack {
3      //記憶域，スタックポインタ，最大値
4      String[] memory;
5      int p, max;
6      //初期化
7      MyStack(int m) {
8          memory = new String[m];
9          max = m - 1;
10         p = -1;
11         System.out.print(m+"の大きさのスタックを確保し，");
12         System.out.println("スタックポインタを初期化しました．");
13     }
14     //push
15     public void push(String d) {
16         if ( p < max ) {
17             p++;
18             memory[p] = d;
19             System.out.println(p+"番目に"+d+"をpushしました．");
20         } else {
```

```
21                 System.out.println("スタックオーバーフロー！（失敗）");
22             }
23         }
24         //pop
25         public String pop() {
26             String r;
27             if ( p >= 0 ) {
28                 r = memory[p];
29                 System.out.println(p+"番目のデータをpopします．: "+r);
30                 p--;
31             } else {
32                 r = "";
33                 System.out.println("スタックは空です！（失敗）");
34             }
35             return(r);
36         }
37     }
38
39     //スタックのテスト
40     class StackTest {
41         public static void main( String argv[] ) {
42             MyStack stk;
43             stk = new MyStack(8);    // 8段のスタックを生成
44             stk.push("a");
45             stk.push("b");
46             stk.push("c");
47             stk.pop();
48             stk.pop();
49             stk.pop();
50             stk.push("d");
51             stk.push("e");
52             stk.push("f");
53             stk.pop();
54             stk.pop();
55             stk.pop();
56             stk.pop();
57         }
58     }
```

解説：

(1) スタック ‘MyStack’ のクラスの定義（2〜37 行）

4,5 行目でデータの記憶域，スタックポインタとその最大値のための変数を宣言している．

(2) スタックを初期化するメソッドの定義（7〜13 行）

スタックは初期状態ではデータは空であり，スタックポインタは-1 になっている．この初期状態を作る処理（初期化処理）を行うのがこの部分である．メソッドの名前として，クラス名と同じものが記述されているが，このような，インスタンス生成時の初期化処理のためのメソッドを**コンストラクタ**と呼ぶ．コンストラクタに関しては後で更に説明する．

(3) データを蓄積（push）するメソッドの定義（15〜23 行）

（詳細は後述）

(4) データを取り出す（pop）するメソッドの定義（25〜36 行）

（詳細は後述）

(5) アプリケーションの動作の記述（40〜58 行）

スタック機構を利用したプログラムの本体がクラス ‘StackTest’ である．すなわちこのクラスがシステムの全体像であり，この内部の ‘main’ メソッドがプログラムの主たる動作を実現している．

(5)-1. スタック機構の実体の生成（42,43 行目：インスタンスの生成）

先に定義した MyStack はいわばスタック機構の設計図であり，それに基づいて実際に動作するスタックの実体 ‘stk’ を宣言し，その内容を 43 行目の new により生成している．（インスタンスの生成）

2.5.2.1　クラス，インスタンス，コンストラクタ」の考え方

OOP では「クラス」はあくまで設計図であり，オブジェクトが所属する「分類枠」である．この分類枠に属する個別の実体を**インスタンス**といい，同じクラスに属するインスタンスを複数生成することができる．これを理解するための日常的な例えとして,「人類」に属する個体（個人）が複数存在している様子などが挙げられる．
更に詳細に考えると,「人類」の枠の配下に「日本人」という分類枠があり，インスタンスとしての個人はこの「日本人」に属する.（図 6）

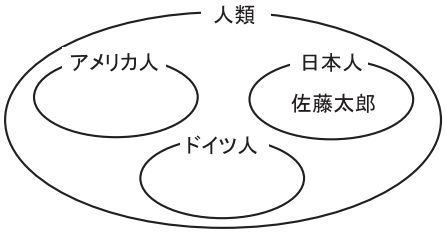


図 6: 分類の考え方：枠と個体

OOP においても，あるクラスの配下にそれのバリエーションとしてのクラス（**サブクラス**もしくは**拡張クラス**，**派生クラス**ともいう）を定義することができる．サブクラスから見た上位のクラスを**スーパークラス**と呼ぶ.

拡張クラスは上位のクラスの特性を**継承**する．この性質（継承：インヘリタンス）は OOP にとって基本的なものであり，拡張クラスは上位のクラスで定義されたメンバやメソッドの定義を自動的に引き継ぐ．すなわち OOP によるシステム開発は，既存のクラスを次々と拡張して新たな機能を加え（メンバやメソッドを増設し）てゆくというスタイルになる．

‘new’ により，そのクラスに属するインスタンスが生成されるが，生成の時点では通常，何らかの初期化処理をすることが多い．例えば今回のスタック機構の例では，スタック内部に配列を用意したり，スタックポインタの初期化とスタックポインタの最大値の登録を行なっているが，これが初期化処理に相当する．このように，インスタンスの生成時に初期化処理を行うメソッドが**コンストラクタ**である．コンストラクタは，そのクラスの名前と同じメソッド名を与えて記述（7〜13 行目）する．これにより，インスタンス生成時の初期化処理が行われる．

(5)-2. データの格納と取り出し（44 行目以降）

実体としてのスタック stk（MyStack のインスタンス）に対して push と pop の処理を実行している．メソッドには引数を渡すことができる．例えば，データを格納（push）するには，格納したいデータをスタックに引き渡す必要があり，**ドット「.」**を用いた ‘`stk.push("a")`’ のような書き方になる．

メソッドの実行：　オブジェクト. メソッド (引数の列)

今回の例では，3 つのデータを続けて push し，その後 3 つのデータを続けて pop するという処理を 2 サイクル繰り返している．

push 処理の詳細説明：（15〜23 行目）

スタックに格納できるデータの個数には上限があり，スタックポインタが取ることができる最大値がある．すなわち，スタックポインタが最大値になっておればデータの push は不可能であり，処理ができないことを通知（21 行目）する必要がある．push ができるのはスタックポインタの値がその最大値未満である場合に限られるので，push する前にそれを確認（16 行目）している．push 可能な場合は，スタックポインタを 1 つ上げ，それが示す配列の要素にデータを格納する.（17,18 行目）

スコープについて：

15 行目の ‘public’ は，そのメソッドを利用できる範囲（**スコープ**）を意味する．通常の場合，メソッドのスコープは ‘public’ として，当該クラスの外から実行できる形にする．今回の例でも，push や pop の処理は StackTest クラスの main メソッドの中（MyStack の外）から要求されるので public のスコープにしている．

メソッドのプログラムを作成する際，適切に分割してそれらをサブルーチン化し，別々のメソッドとして記述することがある．そのような場合，サブルーチンとしてのメソッドは当該クラス外から利用されないことが多く，その場合はスコープを ‘private’ や ‘protected’ として，当該クラスの外部からは見えなくすることができる．(表 12 参照)

表 12: メソッドのスコープ

スコープ	範　　囲
private	当該クラスからのみ利用可能（参照可能）
protected	当該クラスとそのサブクラスからのみ参照可能
public	どこからでも参照可能

pop 処理の詳細説明：（25〜35 行目）

スタックにデータが格納されていない場合（空の場合）はデータを返すことができない．すなわちスタックポインタの値が 0 以上の場合に限りデータの取り出しができる．この条件を判定しているのが 27 行目である．このメソッドでは，取り出したデータは変数 r に格納して，35 行目の return 文によりそれを呼び出し元に返している．また，pop 処理の後は 30 行目にあるようにスタックポインタの値を 1 つ減らしている．

スタックが空で pop できるデータが無い場合は，r に空の文字列をセットした後，pop ができない旨の通知（33 行目）をしている．

呼び出し元に値を返す方法：　return(値)

メソッドの型：

pop メソッドは文字列型の値を返すので，26 行目にあるように ‘String’ 型を指定している．このように，返す値によってメソッドの型が決まる．ただ，push メソッドは値を返さないので，15 行目にあるように ‘void’（型なし）を指定している．

プログラムの最上位のクラスとメソッドについて

今回のプログラムは，全体として ‘StackTest’ というクラスとして実装されており，最初に実行されるプログラムは，このクラスの main メソッドとして記述されている．このようにインスタンスを生成せずにメソッドを利用することもでき，その場合のメソッドは**静的メソッド**と呼ばれる．静的メソッドを定義する際，‘static’（静的）という**修飾子**を付ける．(static 宣言されたメソッドは**静的メソッド**と分類される．)

プログラム StackTest.java の実行結果を《実行結果：2.5.2.1》に示す．

《実行結果：2.5.2.1》

```
8 の大きさのスタックを確保し，スタックポインタを初期化しました．
0 番目に a を push しました．
1 番目に b を push しました．
2 番目に c を push しました．
2 番目のデータを pop します．：c
1 番目のデータを pop します．：b
0 番目のデータを pop します．：a
0 番目に d を push しました．
1 番目に e を push しました．
2 番目に f を push しました．
2 番目のデータを pop します．：f
1 番目のデータを pop します．：e
0 番目のデータを pop します．：d
スタックは空です！(失敗)
```

演習課題． キューの機構を実現するプログラム QueueTest.java を実装せよ．

2.5.2.2　静的メンバ／メソッドについて

静的メソッドを実現する例としてサンプルプログラム StaticTest01.java を示す．これは階乗の計算を行うプログラムである．

プログラム：StaticTest01.java

```
1  class Calc {
2      // 静的メンバ
3      static int M;
4      // 静的メソッド：階乗の計算
5      public static int fact( int n ) {
6          if ( n == 0 ) {
7              return( 1 );
8          } else {
9              return( n*fact(n-1) );
10         }
11     }
12 }
13 // メイン
14 class StaticTest01 {
15     public static void main( String argv[] ) {
16         // Calcクラスのインスタンスを生成せずに静的メソッドを実行
17         System.out.println( Calc.fact(10) );
18         // Calcクラスの静的メンバにアクセス
19         Calc.M = 93;
20         System.out.println( Calc.M );
21     }
22 }
```

Calc クラスに静的メソッド fact が定義されている．17 行目でこれを呼び出しているが，Calc クラスのインスタンスを生成せずに実行している．

このプログラムを実行すると 10! の値である 3628800 が表示される．Calc クラスの静的メンバ M もこのクラスのインスタンスを生成せずに使用できる [10]．

メインプログラムである main メソッドもまた StaticTest01 クラスの静的メソッドである ことが理解できる．

2.5.2.3　static イニシャライザ

static イニシャライザを記述すると，クラス内の静的メンバの初期化処理ができる．サンプルプログラム StIni01.java でそれを例示する．

プログラム：StIni01.java

```
1  class MyClass {
2      public static int num;      // 静的メンバ
3      // クラス初期化子（staticイニシャライザ）
4      static {
5          num = 12345;
6      }
7  }
8  // メイン
9  class StIni01 {
10     public static void main( String argv[] ) {
11         System.out.println("MyClass.num："+MyClass.num);
12     }
13 }
```

このプログラムの 4〜6 行目に static イニシャライザを記述している．これにより，`MyClass` クラスの静的メンバ `num` の値を初期化している．このプログラムを実行すると次のような出力が得られる．

```
MyClass.num：12345
```

[10] クラスの静的メンバは，他のクラスからアクセスできる共有データとしての応用が可能である．

もちろんこの程度の初期化であれば，メンバ宣言の時点で

```
public static int num = 12345;
```

と記述しても良いが，計算処理を伴う初期化が必要な場合に static イニシャライザが有効である．

2.5.3 クラスの拡張（継承）

既存のクラスを基にして新たに機能を追加した別のクラスを定義（クラスの拡張）することができる．

書き方： **class クラス名 extends スーパークラス { 定義 }**

既存のクラスである「スーパークラス」の定義を継承した新しい「クラス」を定義する．クラスの継承を例示するサンプルプログラム RndNum01.java を示す.（これは乱数発生機の例である）

プログラム：RndNum01.java

```
1  // 乱数発生器（スーパークラス）
2  class Rnd {
3      double  r;      // メンバ
4      // コンストラクタ
5      Rnd() {
6          r = Math.random();
7      }
8      // 乱数を返すメソッド
9      public double next() {
10         double n;
11         n = r;
12         r = Math.random();  // 新たな乱数を生成
13         return( n );
14     }
15 }
16 // 整数乱数発生器（拡張クラス）
17 class RndInt100 extends Rnd {
18     public int next100() {
19         return( (int)(next()*100.0) );
20     }
21 }
22 // メイン
23 class RndNum01 {
24     public static void main( String argv[] ) {
25         int c;
26         // 乱数発生器
27         System.out.println("** Rndオブジェクトによる実行 **");
28         Rnd rnd = new Rnd();
29         for ( c = 0; c < 5; c++ ) {
30             System.out.println( rnd.next() );
31         }
32         // 整数乱数発生器
33         System.out.println("** RndInt100オブジェクトによる実行 **");
34         RndInt100 rndint100 = new RndInt100();
35         for ( c = 0; c < 5; c++ ) {
36             System.out.println( rndint100.next100() );
37         }
38         // スーパークラスのメソッドも使える
39         for ( c = 0; c < 5; c++ ) {
40             System.out.println( rndint100.next() );
41         }
42     }
43 }
```

解説： 2〜15 行目に乱数発生器である Rnd クラスが定義されている．このクラスはメンバとして double 型の変数 r を持ちこれに乱数を保持する．コンストラクタでインスタンスを生成する際に最初の乱数（0 以上 1.0 未満）が r にセットされる．このクラスに定義されているメソッド next により r の値が呼び出し元に返され，新たな乱数が r に再設定される．

乱数生成： **Math.random()** （0 以上 1.0 未満の正の乱数）

17〜21 行目にクラス RndInt100 を定義しているが，これは Rnd クラスの定義を継承している．すなわち Rnd クラスのメンバ r やメソッド next が RndInt100 の定義として含まれる．従って RndInt100 クラスの定義は Rnd が持たない機能についてのみ記述すればよい．このクラスに定義されているメソッド next100 は Rnd クラスの next メソッドを利用して 100 未満の正の整数の乱数を返す．

このプログラムを実行した例を次にしめす．

```
** Rnd オブジェクトによる実行 **
0.30407580500298026
0.8977220295563769
0.5724417134299655
0.3967222997971721
0.7379435637816544
** RndInt100 オブジェクトによる実行 **
46                                  ←ここから next100 メソッドによる処理
9
37
32
46
0.08623833491387622                 ←ここから next メソッドによる処理
0.08737076155345824
0.10517693196869926
0.6210849340917248
0.645976086436407
```

RndInt100 クラスのオブジェクトは Rnd クラスのメソッドが使用できることがわかる．

2.5.4 抽象クラスとインターフェースについて

先にオブジェクト指向の考え方の比喩で「人類」,「日本人」といった分類枠について言及した．この例えでもわかるように，人種・国籍の全くない「人類」という枠に直接所属する個人は存在しない．OOP においてもこれに近い**抽象クラス**という概念がある．抽象クラスはインスタンスを生成できないクラスであり，拡張クラスの原型を定義する場合などに用いられる．抽象クラスを定義するには下記のように **abstract 修飾子**を付けてクラスを定義する．

抽象クラスの定義： **abstract class { … }**

サンプルプログラム AbClassTest1.java を示しながら抽象クラスの応用について説明する．

プログラム：AbClassTest1.java

```
1  //  抽象クラス
2  abstract class Base {
3      //  抽象メソッド：  名前の表示
4      abstract public void printName( String sei, String mei );
5
6      //  通常のメソッド
7      public void info( ) {
8          System.out.println("抽象クラスの通常のメソッドです．");
9      }
10 }
11
12 //  拡張クラス(1)：  日本語
13 class JName extends Base {
14     @Override
15     public void printName( String sei, String mei ) {
16         System.out.println(sei+" "+mei+"さん");
17     }
18 }
```

```
19 //  拡張クラス(2)：  英語
20 class EName extends Base {
21     @Override
22     public void printName( String sei, String mei ) {
23         System.out.println("Mr."+mei+" "+sei);
24     }
25 }
26
27 //  メイン
28 public class AbClassTest1 {
29     public static void main( String argv[] ) {
30         JName j = new JName( ); //  日本人
31         EName e = new EName( ); //  アメリカ人
32
33         j.printName( "佐藤", "太郎" );
34         e.printName( "Smith", "John" );
35
36         j.info( );
37         e.info( );
38     }
39 }
```

解説：

このプログラムでは抽象クラス Base から拡張クラス JName と EName を作っている．Base には**抽象メソッド** printName がある．

抽象メソッドとは処理内容の定義が記述されていないメソッドであり，実際の処理内容は，そのクラスを継承した拡張クラスの側で記述する．

抽象メソッドの宣言：　abstract　戻り値の型　メソッド名 (引数…);

拡張クラスの側で抽象メソッドの内容を記述する場合は**@Override アノテーション**[11] を付ける．

このサンプルプログラムは，人名の表示を国籍ごとに異なる形で行う例である．実行結果を次に示す．

《実行結果》

```
佐藤 太郎さん
Mr.John Smith
抽象クラスの通常のメソッドです.
抽象クラスの通常のメソッドです.
```

実際のプログラミングにおいて重要な概念に**インターフェース**がある．広い意味でのオブジェクト指向の概念には**多重継承**が含まれるが Java にはこの概念が無い．多重継承とは，拡張クラスを定義する際に複数のスーパークラス（上位クラス）の性質を引き継ぐことを指すが，Java では拡張クラスは 1 つのスーパークラスしか取ることができない．Java では，メンバやメソッドなどの特性をクラスの概念とは別に定義する仕組みが用意されており，これが**インターフェース**である．Java のクラスは複数のインターフェースを引き継ぐ（実装する）ことができ，実質的に多重継承に近い利便性を実現している．

インターフェースもまた，インスタンスを生成することができない．

インターフェースの定義

プログラミングをする上では，インターフェースの定義はクラスの定義に似た手続きをとる．（下記）

インターフェースの定義：

interface　インターフェース名　{ … }

インターフェースの定義を引き継ぐ（実装する）クラスの定義：

[11] **アノテーション**については「2.5.5 アノテーション」を参照のこと．

class　クラス名　implements　インターフェース名　{ … }
実装するインターフェースはコンマで区切って複数指定することができる．

インターフェースの定義の中では，メソッドは本体を持つことが許されず，

型名　メソッド名 ();

のように名前のみ列挙する．すなわち，メソッドの具体的な定義はそのインターフェースを実装するクラスの中で記述（オーバーライド）する．また，当然であるがインターフェースにはそれを拡張するサブクラスのような概念が無く，あくまで「別のあるクラスがインターフェースの機能を具体的に実装する」のが基本である．

インターフェースを応用したサンプルプログラムを次に示す．

プログラム：InfTest1.java

```
1  // インターフェース1
2  interface Inf1 {
3      public double triple( );
4  }
5
6  // インターフェース2
7  interface Inf2 {
8      public double half( );
9  }
10
11 // 実装クラス
12 class Calc implements Inf1, Inf2 {
13     // メンバ
14     double dat;
15
16     // コンストラクタ
17     Calc( double n ) {
18         dat = n;
19     }
20
21     // インターフェースのメソッドを実装
22     @Override
23     public double triple( ) {
24         return( 3.0 * dat );
25     }
26     @Override
27     public double half( ) {
28         return( dat / 2.0 );
29     }
30 }
31
32 public class InfTest1 {
33     public static void main( String argv[ ] ) {
34         Calc c = new Calc( 4.0 );
35         System.out.println( c.triple( ) );
36         System.out.println( c.half( ) );
37     }
38 }
```

解説：

このプログラムでは 2 つのインターフェース Inf1, Inf2 を持ち，それらを Calc というクラスで実装している．インターフェース内で定義されているメソッドに関しても，その具体的な処理内容は Calc 内で記述されている．

このプログラムを実行した結果を次に示す．

《実行結果》

```
12.0
2.0
```

2.5.4.1　匿名クラスについて

Java では**匿名クラス**（anonymous class）[12] というものが扱える．これを応用すると，インターフェースのオブジェクトを直接的に生成することができる．

> **■ インターフェースを匿名クラスとして扱う方法：**
>
> **インターフェース名　インスタンス名 = new インターフェース名 () {**
>
> **（インターフェースの実装に関する記述）**
>
> **};**

インターフェースは new によるインスタンスの生成ができないが，この形の記述によりインターフェースと同名の匿名クラスが自動的に定義され，new によるインスタンスの生成が可能となる．

これつにいてサンプルプログラム ItfDflt01.java を示して説明する．

プログラム：ItfDflt01.java

```
1  interface MyInterface {
2      public int square( int n );
3      // default修飾子によってメソッドの定義の記述が可能に
4      default public int cubic( int n ) {
5          return( n*n*n );
6      }
7  }
8  // メイン
9  class ItfDflt01 {
10     public static void main( String argv[] ) {
11         // 匿名クラスとしてinterfaceのクラスのインスタンスを生成
12         MyInterface itf = new MyInterface() {
13             // メソッドの記述
14             public int square( int n ) {
15                 return( n*n );
16             }
17         };
18         System.out.println( "itf.square(16)："+itf.square(16) );
19         System.out.println( "itf.cubic(4)："+itf.cubic(4) );
20         // クラス名を調べる
21         System.out.println( "Class name of itf："+itf.getClass().getName() );
22     }
23 }
```

このプログラムでは square，cubic という 2 つのメソッドを持つ MyInterface というインターフェースを定義している．プログラムの 12〜17 行目で MyInterface を匿名クラスとしてインスタンス itf を生成している．また，21 行目では itf のクラス名を表示している．
（オブジェクトのクラス名を取得する方法に関しては p.37「2.5.7 オブジェクトのクラスに関する情報」で解説する）

■ インターフェースの default 実装について

通常の場合，インターフェース内ではメソッドの処理内容は記述しない．ただし，**default 実装**の形を取ると，プログラム ItfDflt01.java の 4〜6 行目にあるように，メソッドの処理内容を記述することができる．

プログラム ItfDflt01.java の実行結果の例を次に示す．

```
itf.square(16)：256
itf.cubic(4)：64
Class name of itf：ItfDflt01$1
```

MyInterface の匿名クラスとして生成したオブジェクト itf のクラス名が ItfDflt01$1 となっていることがわかる．このクラス名は自動的に与えられたものである．

[12] **無名クラス**とも呼ばれる．

2.5.5 アノテーション

Java のアノテーションはクラスやメソッド，パッケージに対してメタデータとして付加情報を記入する機能である．アノテーションの記述そのものは，作成するプログラムの実行手順とは無関係であるが，記述するプログラムの内容に付加的な情報を与え，コンパイラや JVM の動作に影響を与えるという意味では単なるコメント以上の働きがある．

2.5.6 総称型（Generics）について

クラスは「クラス名」が与えられて定義される．また，オブジェクト生成時の宣言や，メソッドの仮引数の記述において，そのクラスを明記するために「クラス名」が記述される．例えば文字列クラス（String クラス）のインスタンス s を生成する場合次のように記述する．

```
String s = new String("これは文字列です．");
```

ここで，**総称型**（Generics）と呼ばれるものを用いると**未定のクラス名**を用いてメンバや仮引数の型を宣言することができる．例えば総称型の <T> を用いて未定のクラスのオブジェクト s を次のようにして宣言できる．

```
T s;
```

これはクラス定義における型宣言に関する柔軟性をもたらすものである．すなわち，総称型の記述を応用することで，クラスのメンバやメソッド，あるいはメソッドに渡す引数を宣言する際にそれらの型（クラス）を未定にしておき，実際にインスタンスを生成する際に未定にしておいたクラス名を確定するということが可能となる．これに関してサンプルプログラム Gnrcs01.java を例に挙げて説明する．

プログラム：Gnrcs01.java

```
1  // Genericsを用いたクラス宣言
2  class MyClass<T> {
3      T   value;      // メンバ
4      // コンストラクタ
5      MyClass( T v ) {
6          value = v;
7      }
8      // メソッド
9      T set( T v ) {
10         value = v;
11         System.out.println("値："+value);
12         return( value );
13     }
14 }
15
16 // メイン
17 class Gnrcs01 {
18     public static void main( String argv[] ) {
19         // インスタンス生成時に各部のクラスを確定する
20         MyClass<Integer> m1 = new MyClass<Integer>(3);
21         MyClass<String>  m2 = new MyClass<String>("a");
22         int n = 5;
23         m1.set(n);      // 整数（int型）の値を与えている
24         String s = new String("これは文字列です．");
25         m2.set(s);      // 文字列（Stringクラス）のオブジェクトを与えている
26     }
27 }
```

プログラム内に記述されているクラス MyClass は，定義の記述の段階ではメンバのクラス，メソッドの戻り値のクラス，メソッドへの引数のクラスが未定であり，T と宣言されている．未定のクラス名（総称型）を用いる際は，当該クラスの定義の冒頭でクラス名の直後に <T> などと <...> で括って仮のクラス名を示す記号を記述する．以降，クラス定義の記述の中でその記号（この例では T）をクラス名として用いる．この MyClass のインスタンスを生成する場合は，クラス名の直後に <...> で括って T に替わるクラス名を記述する．これにより，生成されるインスタンスの各部のクラスが確定される．

このプログラムを実行した例を次に示す．

```
値：5
値：これは文字列です．
```

この例からわかるように，1 つのクラスから多様なインスタンスが生成できる．特に，定義の基本構造が同じで，異なるクラスのデータ型を扱うクラスを記述する際に総称型（Generics）は高い利便性をもたらす．

■ 総称型にプリミティブ型の名前を与える場合の注意事項

総称型に**プリミティブ型**の型名（基本的な変数の型）を与えることはできない．プリミティブ型の型名を与える場合，当該プリミティブ型に対応するクラス名が定義されている（表 13）のでそれを用いる．上記プログラムでは int 型の代わりとなる Integer クラスを用いて `MyClass<Integer>` と記述している．

表 13: プリミティブ型に対応するクラス名

byte 型	→	Byte クラス	short 型	→	Short クラス	int 型	→	Integer クラス
long 型	→	Long クラス	float 型	→	Float クラス	double 型	→	Double クラス
boolean 型	→	Boolean クラス	char 型	→	Character クラス			

2.5.7 オブジェクトのクラスに関する情報

Java のオブジェクトに対して getClass メソッドを実行すると，そのオブジェクトが属するクラスに関する情報が得られる．

書き方： オブジェクト.getClass()

getClass メソッドの戻り値は Class クラス（java.lang.Class）の値である．またこの戻り値に対して getName メソッドを実行すると，そのクラス名が文字列（String）として得られる．

これらのことに関してサンプルプログラム ClassTest01.java を挙げて説明する．

プログラム：ClassTest01.java

```
1  class MyClass {     // クラスの定義
2      MyClass() { // コンストラクタ
3      }
4  }
5  // メイン
6  class ClassTest01 {
7      public static void main( String argv[] ) throws ClassNotFoundException {
8          MyClass mc = new MyClass();     // インスタンスの生成
9          Class cl = mc.getClass();       // オブジェクトからクラス情報を取得
10         String cname = cl.getName();    // クラス情報からクラス名を取得
11         System.out.println("MyClass mc のクラス名の取得(1)："+cname);
12         // クラス名（文字列）からクラス情報を取得する
13         cl = Class.forName("MyClass");
14         cname = cl.getName();
15         System.out.println("MyClass mc のクラス名の取得(2)："+cname);
16     }
17 }
```

このプログラムでは，`MyClass` クラスを定義しており，そのインスタンス `mc` を生成している．更に mc が属するクラスに関する情報を getClass メソッドで取得してオブジェクト `cl` としている．この cl に対して getName メソッドを実行してクラス名の文字列を `cname` に取得している．

Class クラスのメソッド forName を用いると，文字列で表現されたクラス名からクラス情報を取得する（13 行目）こともできる．

このプログラムを実行した結果の例を次に示す.

```
MyClass mc のクラス名の取得 (1)：MyClass
MyClass mc のクラス名の取得 (2)：MyClass
```

forName 実行の際，存在しないクラス名を引数に与えると ClassNotFoundException というエラーが発生するので，7 行目にあるように throws の記述によって**例外処理**をしている．例外処理に関しては後の「2.6.3.2 例外処理 (1)：try」（p.44）,「2.6.3.2 例外処理 (2)：throws」（p.45）で解説する．

2.6 入出力

2.6.1 標準入出力

コマンド環境（CLI 環境）ではキーボードから入力を受け付け，ディスプレイに文字列を出力する．これまでもコマンド環境に対して表示のための出力を ‘`System.out.println(` 出力内容 `)`’ という記述によって実現してきたが，これを厳密に解説すると以下のようになる．

記述	解説
System	システムの入出力ストリームや環境変数などを扱うクラス
out	PrintStream クラスのオブジェクトであり，System クラスの静的メンバの 1 つ．(**標準出力**)
println	PrintStream クラスのメソッドであり，引数に与えたものを出力し，末尾で改行する．

出力表示のためのディスプレイ（CLI 環境の出力）は**標準出力**と呼ばれ，Java では System.out オブジェクトがこれに対応する．すなわち，これに対して出力のためのメソッドを実行することで出力内容がディスプレイに表示される．出力のためのメソッドは println 以外にも print メソッドがあり，これによる出力では末尾での改行は行われない．また，書式を指定した出力のために printf メソッドがある．printf メソッドを使用したサンプルプログラムを に示す．

プログラム：StdIO01.java

```
1 class StdIO01 {
2     public static void main( String argv[] ) {
3         int a = 1234;                    // 整数
4         double pi = 3.141592653589793;   // 浮動小数点数
5         String s = "John Smith";         // 文字列
6         System.out.printf("%6d:%5.2f:%12s:\n",a,pi,s);  // 書式指定出力
7     }
8 }
```

このプログラムを実行すると ‘ `1234: 3.14: John Smith:`’ と表示される．printf メソッドの引数には**書式**に続いて出力対象の値（変数など）を記述（下記）する．

書き方：　printf(書式，値 1，値 2，…)

書式に従って出力並びが編集される様子を図 7 に示す．

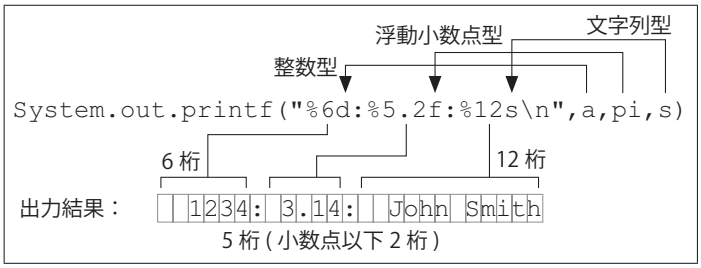


図 7: 書式付き出力の例

「書式」ではパーセント記号「%」に続いて，出力対象の値の「桁数」と「型」を記述する．書式に指定する型の代表的なものを表 14 に示す．

表 14: 書式に指定する型（代表的なもの）

型指定	意味	型指定	意味
d	整数（10 進数）	f	浮動小数点数
x	整数（16 進数）	o	整数（8 進数）
s	文字列	c	文字（UNICODE の 1 文字）

桁の長さの数値として負値（マイナスの値）を指定すると表示が左寄せになる．書式の中にはエスケープシーケンスを含めることができる．

2.6.1.1　文字列生成への書式編集の応用

String.format メソッドを用いると printf メソッドと同様の書式編集によって文字列（String）を生成することができる．

書き方：　String.format(書式，値１，値２，…)

このメソッドの実行によって書式編集された文字列（String）が生成される．これを応用したサンプルプログラム StringFormat01.java を示す．

プログラム：StringFormat01.java

```
1  class StringFormat01 {
2      public static void main( String argv[] ) {
3          int a = 1234;                    // 整数
4          double pi = 3.141592653589793;   // 浮動小数点数
5          String s = "John Smith";         // 文字列
6          // 書式指定による文字列生成
7          String f = String.format("%6d:%5.2f:%12s:\n",a,pi,s);
8          System.out.println(f);
9      }
10 }
```

このプログラムの 7 行目で String オブジェクト f に " 1234: 3.14: John Smith:" という文字列が生成される．

キーボードからの入力は**標準入力**と呼ばれ，これには System クラスの静的メンバ in が対応する．System.in は InputStream クラスのオブジェクトであり，これに対して入力用のメソッドを実行することで標準入力からの入力を取得することができる．

InputStream はバイト単位で値を取得する**バイトストリーム**であり，そこからバイト列を連続して取得し，更にそれらを文字列や数値などとして解釈するにはある種の変換作業が必要となる．Scanner クラスを用いた入力データの変換に関して次のサンプルプログラム StdIO01.java に沿って解説する．

プログラム：StdIO02.java

```
1  import java.util.Scanner;
2
3  class StdIO02 {
4      public static void main( String argv[] ) {
5          // 標準入力からの入力をScannerクラスの機能により文字列に変換する
6          Scanner scn = new Scanner( System.in );
7          String s;
8          while ( true ) {
9              System.out.print("入力してください> ");
10             s = scn.next();      // 標準入力から文字列を1つ取得する
11             System.out.println("取得したデータ: "+s);
12             if ( s.equals("end") ) {    // "end" なら終了する
13                 break;
14             }
15         }
16         System.out.println("終了します. ");
17     }
18 }
```

解説．プログラムの 1 行目にあるように，Scanner クラスを使用するには必要となる**ライブラリ**を予め読み込む必要がある．6 行目では System.in に接続された Scanner クラスのインスタンス scn を生成している．この scn に対して入力用のメソッドを実行することで実際に標準入力からデータが得られる．10 行目は，scn に対して next メソッド

を使用して 1 つの文字列を取得するものである．next による文字列の取得は**同期入力**であり，キーボードの Enter のタイミングで文字列を取得する．

このプログラムは文字列を入力して Enter を押すとそれを表示する．また ”end” が入力されると終了する．実行例を次に示す．

```
入力してください> abc Enter
取得したデータ： abc
入力してください> def Enter
取得したデータ： def
入力してください> end Enter
取得したデータ： end
終了します.
```

演習課題． Scanner クラスには next メソッド以外にも数値などを読み込むメソッドが用意されているが，どのようなメソッドがあるかを調べ，StdIO02.java を浮動小数点数を読み込む形に改造せよ．

2.6.1.2 標準エラー出力

CLI 環境の出力には，先に説明した標準出力以外に**標準エラー出力**と呼ばれるものがある．標準エラー出力は基本的には CLI 環境への出力であり，一見して標準出力と同じもののように見える．例えば次のサンプルプログラム StdIO03.java について考える．

プログラム：StdIO03.java

```
1 class StdIO03 {
2     public static void main( String argv[] ) {
3         System.out.println("標準出力への書き出しです．（１）");
4         System.err.println("標準エラー出力への書き出しです．（２）");
5         System.out.println("標準出力への書き出しです．（３）");
6         System.err.println("標準エラー出力への書き出しです．（４）");
7     }
8 }
```

これは標準出力と標準エラー出力に交互に文字列を出力するプログラムであり，実行すると次のように 4 行の文字列が表示される．

```
標準出力への書き出しです.（1）
標準エラー出力への書き出しです.（2）
標準出力への書き出しです.（3）
標準エラー出力への書き出しです.（4）
```

しかし，あくまで標準出力と標準エラー出力は異なるものであり，出力先を**リダイレクト**するとそれが明らかとなる．

標準出力は通常はディスプレイを指しており，System.out に対して出力したものはディスプレイに表示される．しかし，先のプログラムのバイトコードの実行において次のようにすると，標準出力に対する出力内容がファイルに出力される．

例． 標準出力のファイルへのリダイレクト

```
java StdIO03 > out1.txt  Enter
```

これにより，標準出力（System.out）への出力内容はファイル ‘`out1.txt`’ に出力され，ディスプレイには次のように表示される．

```
標準エラー出力への書き出しです.（2）
標準エラー出力への書き出しです.（4）
```

ファイル ‘`out1.txt`’ の内容を確認すると次のようなものとなっている．

```
標準出力への書き出しです.（1）
標準出力への書き出しです.（3）
```

CLI 環境でのプログラミングにおいては，利用者に対する注意喚起のメッセージや，プログラムの実行状態に関するログの出力などを，主たる出力とは区別する習慣があり，そのような特別な（付随的な）情報の出力先として標準エラー出力が用いられる．

以上のことを整理すると，CLI 環境におけるプログラミングにおいては，図に示すように出力が 2 系統，入力が 1 系統存在する．

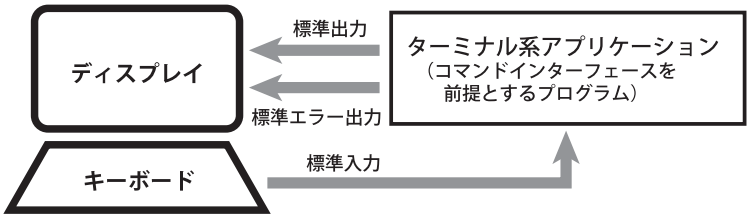


図 8: CLI 環境における入出力の外観

2.6.2 コマンド引数

CLI 環境でプログラムを起動する際，プログラムに複数の**コマンド引数**を与えることができる．次のようなサンプルプログラム CmdArgs01.java について考える．

プログラム：CmdArgs01.java

```
1 class CmdArgs01 {
2     public static void main( String argv[] ) {
3         int len = argv.length;       // コマンド引数の個数
4         System.out.println("コマンド引数の個数："+len);
5         for ( int c = 0; c < len; c++ ) {
6             System.out.println( (c+1)+"番目の引数："+argv[c] );
7         }
8     }
9 }
```

これは実行時にコマンド引数を受け付けるプログラムであり，次のようにして実行することで，コマンド引数の受理の様子が理解できる．

例. コマンド引数を付けて実行

```
java CmdArgs01 a b c  [Enter]      ←3 つの引数 a b c を与えて起動
コマンド引数の個数：3              ←コマンド引数の個数（プログラム 3 行目で取得）を表示
1 番目の引数：a                    ←コマンド引数を順番に全て表示している
2 番目の引数：b                              ：
3 番目の引数：c
```

メインプログラム（メインメソッド）である main は宣言時に

main(String argv[])

と仮引数を与えるが，これは実行時に与えられるコマンド引数を受け取るためのものである．すなわち，実行時のコマンド引数は配列 argv に受理される．また，配列の要素の個数はメンバ length の値として参照できる．

コマンド引数を受け付けるプログラムの例をもう 1 つ示す．

プログラム：CmdArgs02.java

```
1 class CmdArgs02 {
2     public static void main( String argv[] ) {
```

```
3          int a = Integer.parseInt(argv[0]);       // 最初の引数を整数に変換
4          int b = Integer.parseInt(argv[1]);       // 次の引数を整数に変換
5          // 加算結果を表示
6          System.out.println( argv[0]+"+"+argv[1]+"="+(a+b) );
7      }
8  }
```

これは，2つのコマンド引数を整数であると解釈して加算するプログラムである．これをコンパイルして実行すると次のようになる．

例. 与えられた2つの引数を加算する

```
java CmdArgs02 1 2  [Enter]    ←引数として　1　2　を与えている
1+2=3                          ←出力
```

演習課題. 任意の個数のコマンド引数を受け取り，それらを整数と解釈して全ての和を求めるプログラムを作成せよ．

2.6.3 ファイルへの出力

ここでは，ファイルに各種の値を出力する方法について解説する．

2.6.3.1 バイトストリームへの出力： FileOutputStream

ファイルの内容が表すものは**テキストデータ**[13]（文字列），画像，音声と様々な種類のものがあるが，それらは全てバイトデータの列として媒体（ディスクなど）に保存されている．従って，データの出力先としてファイルを最も素朴な形で捉えると，それは**バイトストリーム**と見做すことができる．出力先のファイルをバイトストリームとして扱うために FileOutputStream[14] クラスがある．

FileOutputStream クラスを用いてファイルにアスキー文字を書き込むサンプルプログラム FileOut01.java を例に挙げ，ファイルへの出力処理について説明する．

プログラム：FileOut01.java

```
1  import java.io.FileOutputStream;    // ファイル出力のためのクラス
2  import java.io.IOException;         // 例外処理のためのクラス
3
4  class FileOut01 {
5      public static void main( String argv[] ) {
6          try {       // 実行におけるエラー検出
7              FileOutputStream fo;
8              fo = new FileOutputStream("ascii01.txt");   // 出力ファイルを開く
9              for ( int c = 32; c < 127; c++ ) {
10                 fo.write( c );  // ファイルに値を出力
11             }
12             fo.close();         // 出力ファイルを閉じる
13             System.err.println("** ファイル出力が正常に処理できました **");
14         } catch ( IOException e ) {
15             System.err.println("** エラーが発生しました **\n"+e);
16         } finally {
17             System.err.println("** プログラムを終了します **");
18         }
19     }
20 }
```

FileOutputStream はライブラリ java.io.FileOutputStream として提供されており，使用に際して予め読み込んでおく必要がある．（プログラム1行目）　FileOutputStream クラスのインスタンスを生成する際に，コンストラクタの引数に対応する出力ファイルのパス（ファイル名）を与える．（プログラム8行目）　以後，生成したインスタンス

[13] 文字コードに対応するバイトデータのみで構成される（可読な文字のみで構成される）データ．

[14] バイトデータを1バイトずつファイルに送り出すための標準的な I/O ストリームのクラス．

を介してファイルへの出力処理ができる．このプログラムでは出力先のファイルを ”ascii01.txt” としており，これを FileOutputStream クラスのインスタンス fo として扱っている．

プログラムの 10 行目にあるように，fo に対して write メソッドを実行することで実際にファイルに値を出力する．ファイルへの出力処理が終了すれば fo に対して close メソッドを実行して ファイルを閉じる．

2.6.3.2　例外処理 (1)：　try

プログラム FileOut01.java の 6 行目から ‘try {…’ という記述があるが，これが**例外処理**である．プログラムから計算機環境の各種資源[15] にアクセスする際に，何らかの事情[16] により所望の処理が完遂しないことがある．その結果としてプログラムの実行ができない状況（**例外**）が発生し，例外に対する対応を取る必要がある．

具体的には，例外が発生し得るプログラムの部分を 6 行目から始まる記述のように

try { 例外が発生し得るプログラム } …

と記述する．

プログラムの実行において発生し得る例外事象にはさまざまなものがあるが，プログラム FileOut01.java においては入出力に関する例外である IOException を扱う形になっている．IOException はファイルを開く，あるいはファイルに対して入出力を行う際に発生し得るものであり，これが発生した際の対応を 14 行目以降で行っている．

IOException はライブラリ java.io.IOException にクラスとして定義されており，利用に先立って予め読み込んでおく必要がある．(2 行目)

例外処理は **try 構文**で記述する．

【例外処理】

```
try {
        例外が発生し得るプログラム
} catch ( 例外クラス　例外に関する情報を保持するオブジェクト ) {
        対応処理
} catch ( …
          :
} finally {
        try 構文の締めくくり処理
        （例外発生の有無に関わらず必ず実行される処理）
}
```

プログラム FileOut01.java の 14 行目では例外発生時の情報を IOException クラスのオブジェクト e に受け取る形になっている．

■ FileOut01.java の実行例

このプログラムをコンパイルして実行すると，次のような出力がファイル ”ascii01.txt” として作成される．

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMN
OPQRSTUVWXYZ[¥]^_`abcdefghijklmnopqrstuvwxyz{|}
```

また，正常にプログラムが終了したことにより，標準エラー出力に次のように出力される．

```
** ファイル出力が正常に処理できました **
** プログラムを終了します **
```

例外を発生させる例：

プログラムの実行の結果として作成されるファイル ”ascii01.txt” をテキストエディタなどで開いて確認している状態（ファイルをひらいたままの状態）で，再度プログラムを実行しようとすると，ファイルが書き込めない旨の

[15] 記憶資源，入出力デバイス，通信路など様々なものがある．

[16] 記憶資源の容量不足，アクセス権限の欠如，入出力装置の未準備など事情は様々である．

例外が発生し（テキストエディタが当該ファイルを使用中のため）次のようなメッセージが表示される．

```
** エラーが発生しました **
 java.io.FileNotFoundException:  ascii01.txt (プロセスはファイルにアクセスできません。
別のプロセスが使用中です。)
** プログラムを終了します **
```

2.6.3.3　例外処理 (2)：　throws

先に説明した try による例外処理は，例外発生時の柔軟な対処が可能であるが，throws の記述により，例外処理をライブラリに全て任せる形を取ると，プログラム全体を簡潔にすることができる．次に挙げるサンプルプログラム FileOut01_2.java は先の FileOut01.java とほぼ同じ機能を実現するものであるが，throws の記述により例外への対応を簡略化している．

プログラム：FileOut01_2.java

```
1  import java.io.FileOutputStream;    // ファイル出力のためのクラス
2  import java.io.IOException;         // 例外処理のためのクラス
3
4  class FileOut01_2 {
5      public static void main( String argv[] ) throws IOException {
6          FileOutputStream fo;
7          fo = new FileOutputStream("ascii01.txt");   // 出力ファイルを開く
8          for ( int c = 32; c < 127; c++ ) {
9              fo.write( c );  // ファイルに値を出力
10         }
11         fo.close();         // 出力ファイルを閉じる
12     }
13 }
```

例外を発生させる例：

FileOut01_2.java の実行の結果として作成されるファイル "ascii01.txt" をテキストエディタなどで開いて確認している状態（ファイルをひらいたままの状態）で，再度プログラムを実行しようとすると，ファイルが書き込めない旨の例外が発生し（テキストエディタが当該ファイルを使用中のため）次のようなメッセージが表示される．

```
Exception in thread "main" java.io.FileNotFoundException: ascii01.txt (プロセスはファイルにアクセスでき
ません。別のプロセスが使用中です。)
        at java.io.FileOutputStream.open0(Native Method)
        at java.io.FileOutputStream.open(Unknown Source)
        at java.io.FileOutputStream.¡init¿(Unknown Source)
        at java.io.FileOutputStream.¡init¿(Unknown Source)
        at FileOut01_2.main(FileOut01_2.java:7)
```

2.6.3.4　PrintWriter による出力

PrintWriter クラスを用いると，PrintStream の場合と同様に print，println，printf といったメソッドを使用してファイル出力ができる．すなわち，標準出力に対して出力するような方法でファイルに対して出力処理ができる．

PrintWriter を用いたサンプルプログラム FileOut02.java を示す．

プログラム：FileOut02.java

```
1  import java.io.PrintWriter;     // ファイル出力のためのクラス
2  import java.io.IOException;     // 例外処理のためのクラス
3
4  class FileOut02 {
5      public static void main( String argv[] ) throws IOException {
6          PrintWriter fo;
7          fo = new PrintWriter("ascii02.txt");    // 出力ファイルを開く
8          // 書式指定出力
9          int a = 1234;                    // 整数
10         double pi = 3.141592653589793;   // 浮動小数点数
11         String s = "John Smith";         // 文字列
12         fo.printf("%6d:%5.2f:%12s:\n",a,pi,s);  // 書式指定出力
```

```
13          fo.println("PrintWriterによるファイル出力処理");
14          fo.close();          // 出力ファイルを閉じる
15      }
16  }
```

このプログラムをコンパイルして実行すると，次のような出力がファイル "ascii02.txt" として作成される．

```
  1234:  3.14:  John Smith:
PrintWriter によるファイル出力処理
```

■ 出力時の文字コードの指定

テキストファイルを出力する際に文字コードを指定するには次のようなステップを踏むと良い．

(1) ファイルを FileOutputStream として開く

(2) 文字コード変換の機能を持つ OutputStreamWriter クラスに (1) を変換する

(3) 上で得たものを PrintWriter に変換する

(4) 上で得た PrintWriter に対して出力する

この手順を応用して出力時の文字コードを指定するサンプルプログラム FileOut03.java を示す．

プログラム：FileOut03.java

```
1   import java.io.FileOutputStream;    // 基本的な出力ストリーム
2   import java.io.OutputStreamWriter;  // 文字コードの指定ができる出力ストリーム
3   import java.io.PrintWriter;         // ファイル出力のためのクラス
4   import java.io.IOException;         // 例外処理のためのクラス
5
6   class FileOut03 {
7       public static void main( String argv[] ) throws IOException {
8           // 通常のバイトストリームとして出力ファイルを開く
9           FileOutputStream s = new FileOutputStream("ascii03.txt");
10          // 文字コード指定のためにクラスを変換する
11          OutputStreamWriter w = new OutputStreamWriter(s,"jis");
12          // println を使用するために PrintWriter に変換する
13          PrintWriter pw = new PrintWriter( w );
14          pw.println("日本語の文字列");
15          pw.println("文字コードを指定して出力しています");
16          pw.close();          // 出力ファイルを閉じる
17      }
18  }
```

このプログラムは日本語の文字列をファイル "ascii03.txt" に出力するものであり，出力の文字コードを "jis" としている．

2.6.4 ファイルからの入力

2.6.4.1 テキストの行単位の読込み

テキストファイルの内容を 1 行づつ読み込む方法について説明する．基本的な手順としては次のようなステップを踏む．

(1) 読込み用のストリームオブジェクトとしてファイルを開く

(2) 上で得たものをバッファあり読込みのためのオブジェクトに変換する

(3) 上で得たものから読み込む

この手順を応用してテキストファイルから 1 行ずつ読み込むサンプルプログラム FileIn01.java を示す．

プログラム：FileIn01.java

```
1   import java.io.FileReader;       // テキストファイルのリーダ
2   import java.io.BufferedReader;   // バッファあり読込み
3   import java.io.IOException;      // 例外処理のためのクラス
```

```
4
5  class FileIn01 {
6      public static void main( String argv[] ) throws IOException {
7          String s;
8          // ファイルリーダ（実際にテキストファイルから読み込むオブジェクト）
9          FileReader f = new FileReader("FileIn01_sample.txt");
10         // バッファありに変換（行単位で読み込むため）
11         BufferedReader fi = new BufferedReader(f);
12         while ( (s = fi.readLine()) != null ) {
13                 System.out.println(s);
14         }
15         fi.close();
16     }
17 }
```

テキストファイルを読み込むための基本的なクラスとして FileReader クラスがあり，それを 9 行目で生成している．それを 11 行目で BufferedReader クラスのオブジェクトに変換している．このオブジェクトに対して readLine メソッドを実行することでファイルから 1 行ずつ文字列を読み込むこと（12 行目）ができる．読み込み処理がファイルの末尾に到達すると，読み込むデータがなくなり，readLine メソッドは null を返すので，これを検出して読み込み処理を終了している．
このプログラムは入力ファイル "FileIn01_sample.txt" からデータを読み込んでいる．

テキストファイル：FileIn01_sample.txt

```
1  Java（ジャバ）は、狭義ではプログラミング言語のJavaを指し、広義では
2  Java言語を中心にしたコンピューティング・プラットフォームを意味する。
3  後者はJavaプラットフォームと呼ばれ、Javaの関連技術はJavaテクノロジと
4  総称されている。Java言語の構文はC++に類似したものであり、オブジェクト
5  指向が主要パラダイムとして導入されている。
6
7  出典: フリー百科事典『ウィキペディア（Wikipedia）』
```

ファイルの内容が上に示すものである場合のプログラムの実行結果の例を次に示す．

```
Java（ジャバ）は、狭義ではプログラミング言語の Java を指し、広義では
Java 言語を中心にしたコンピューティング・プラットフォームを意味する。
後者は Java プラットフォームと呼ばれ、Java の関連技術は Java テクノロジと
総称されている。Java 言語の構文は C++に類似したものであり、オブジェクト
指向が主要パラダイムとして導入されている。

出典: フリー百科事典『ウィキペディア（Wikipedia）』
```

■ 入力時の文字コードの指定

テキストファイルを入力する際に文字コードを指定するには次のようなステップを踏むと良い．

(1) ファイルを FileIntputStream（入力用バイトストリーム）として開く
(2) 文字コード変換の機能を持つ InputStreamReader クラスに (1) を変換する
(3) 上で得たものをバッファあり読込みのためのオブジェクトに変換する
(4) 上で得たものから読み込む

この手順を応用して入力時の文字コードを指定するサンプルプログラム FileIn02.java を示す．

プログラム：FileIn02.java

```
1  import java.io.FileInputStream;      // ファイル入力用バイトストリーム
2  import java.io.InputStreamReader;    // 文字コードの指定ができる入力ストリーム
3  import java.io.BufferedReader;       // バッファあり読込み
4  import java.io.IOException;          // 例外処理のためのクラス
5
6  class FileIn02 {
7      public static void main( String argv[] ) throws IOException {
8          String lin;
9          // ファイル入力用バイトストリーム（実際にファイルから読み込む）
10         FileInputStream s = new FileInputStream("ascii03.txt");
```

```
11          // 文字コード指定のためにクラスを変換
12          InputStreamReader f = new InputStreamReader(s,"jis");
13          // バッファありに変換（行単位で読み込むため）
14          BufferedReader fi = new BufferedReader(f);
15          while ( (lin = fi.readLine()) != null ) {
16                  System.out.println(lin);
17          }
18          fi.close();
19      }
20  }
```

このプログラムは，先に挙げたプログラム FileOut03.java で出力した jis コードのテキストファイル ”ascii03.txt” から 1 行ずつ読み込みながらそれを標準出力に出力するものである．実行した例を次に示す．

日本語の文字列
文字コードを指定して出力しています

2.6.4.2 Scanner クラスを用いた読込み

先に「2.6.1 標準入出力」(p.39) で Scanner クラスを用いた値の取得について説明した．Scanner を用いると，ファイルから値を読み込むこともできる．Scanner を用いてファイルから整数値を読み込むサンプルプログラムを FileIn03.java に示す．

プログラム：FileIn03.java

```
1   import java.io.FileReader;      // テキストファイルのリーダ
2   import java.util.Scanner;       // Scannerクラス
3   import java.io.IOException;     // 例外処理のためのクラス
4
5   class FileIn03 {
6       public static void main( String argv[] ) throws IOException {
7           int n = 0;
8           // ファイルリーダ（実際にテキストファイルから読み込むオブジェクト）
9           FileReader f = new FileReader("FileIn03_sample.txt");
10          // スキャナ
11          Scanner scn = new Scanner(f);
12          while ( scn.hasNextInt() ) {
13              n += scn.nextInt();
14          }
15          System.out.println("sum:"+n);
16          scn.close();
17      }
18  }
```

このプログラムでは，Scanner のコンストラクタの引数に FileReader を与えている．Scanner オブジェクトを介して整数値を読み込むには nextInt メソッドを使用（13 行目）する．

読み込むべき次の整数値が存在するかどうかを hasNextInt メソッドで検査することができ，次の整数値が存在すれば true を，存在しなければ false を返す．プログラムの 12 行目ではこれを用いて処理の終了を判定している．

このプログラムはファイル ”FileIn03_sample.txt” から値を取得するが，このファイルの内容が次に示すようなものである場合の動作につて考える．

テキストファイル：FileIn03_sample.txt

```
1 2 3 4 5 6 7 8 9 10
この行は整数ではありません．
```

このファイルの 1 行目は整数値が空白文字を区切りにして並んでいる．また，2 行目は文字列であり，整数値ではない．このようなファイルから nextInt メソッドで値を読み込むと，1 行目末尾の 10 を読み込んだ後は整数値が存在していないので hasNextInt メソッドが偽となり while ループは終了する．実行結果として ‘sum:55’ と表示される．

Scanner からの値の読み込みには，nextInt 以外にも表 15 に示すようなものがある．

表 15: 型を指定して読み込むメソッド（よく使うもの）

メソッド	機能
nextInt	整数値（int）の読み込み
nextLong	整数値（long）の読み込み
nextDouble	浮動小数点数（double）の読み込み
nextByte	バイト値（byte）の読み込み
nextBoolean	真理値（boolean）の読み込み
nextLine	1 つの行（String）の読み込み

■ 文字コードを指定した読み込み

nextLine メソッドを使用して日本語文字列の行を読み込む場合，対象のファイルの文字コードを指定するには，InputStreamReader から Scanner オブジェクトを生成すると良い．これを行うサンプルプログラム FileIn03_2.java を示す．

プログラム：FileIn03_2.java

```
1  import java.io.FileInputStream;      // ファイル入力用バイトストリーム
2  import java.io.InputStreamReader;    // 文字コードの指定ができる入力ストリーム
3  import java.util.Scanner;            // Scannerクラス
4  import java.io.IOException;          // 例外処理のためのクラス
5
6  class FileIn03_2 {
7      public static void main( String argv[] ) throws IOException {
8          String msg;
9          // ファイル入力用バイトストリーム（実際にファイルから読み込む）
10         FileInputStream s = new FileInputStream("FileIn03_sample.txt");
11         // 文字コード指定のためにクラスを変換
12         InputStreamReader f = new InputStreamReader(s,"utf-8");
13         // スキャナ
14         Scanner scn = new Scanner(f);
15         while ( scn.hasNextLine() ) {
16             msg = scn.nextLine();
17             System.out.println(msg);
18         }
19         scn.close();
20     }
21 }
```

このプログラムでは，入力ファイル FileIn03_sample.txt の文字コードが utf-8 である場合を想定している．入力は行単位で行われる．

このプログラムの実行結果の例を次に示す．

```
1 2 3 4 5 6 7 8 9 10
この行は整数ではありません.
```

2.6.5 バイナリファイルの入出力

コンピュータで扱うデータは，記憶資源上の実体としては全て**バイト列**であり，それを扱う際にそのバイト列が何を意味するものか（文字列，数値，etc.）を解釈して処理する．特に**文字コード**に対応するバイトデータのみから成るものを**テキストデータ**と呼ぶ．また，テキストデータでないものを**バイナリデータ**[17] と呼ぶ．（バイナリデータから成るファイルを**バイナリファイル**と呼ぶ）

バイナリファイルの入出力の考え方は単純であり，単にバイトデータを**バイトストリーム**に対して入出力するのみである．例えば，既に説明した FileOutputStream クラスを用いるとバイトデータをファイルに書き込むことができ，逆に，バイトデータをファイルから入力するには FileInputStream クラス（今回紹介する）を用いる．ただし，バイナ

[17] 厳密な意味では，テキストデータもバイナリデータの範疇に含まれる．

リデータの入出力は概して多量のデータを扱うことが多く，1 バイトずつの入出力では処理の効率が低いため,「バッファありの入出力」（**バッファド I/O**）と呼ばれる手法で入出力するのが一般的である．

■ バッファド I/O

バイトストリームに対する入出力は 1 バイトずつ行うのが基本であるが，システムが持つ**緩衝記憶領域（バッファ）**に入出力のバイトデータを蓄積してまとめて入出力を実行することで，入出力の効率を高めることができる．これを**バッファド I/O** と呼ぶ．

具体的には，バイトストリームのオブジェクト（FileInputStream，FileOutputStream など）をバッファド I/O のためのクラスのオブジェクトに変換して，それに対して入出力の処理を実行することでバッファド I/O が実現できる．

次に示すサンプルプログラム BinIO.java は，画像データファイルを読み込んで，同じ内容を別のファイルに保存する（データのコピーを行う）ものである．これを例に挙げてバイナリデータの入出力について解説する．

プログラム：BinIO.java

```
 1  import java.io.FileInputStream;          // 入力ファイルストリーム
 2  import java.io.FileOutputStream;         // 出力ファイルストリーム
 3  import java.io.BufferedInputStream;      // バッファあり入力ストリーム
 4  import java.io.BufferedOutputStream;     // バッファあり出力ストリーム
 5  import java.io.IOException;              // 入出力例外
 6
 7  class BinIO {
 8      public static void main( String argv[ ] ) throws IOException {
 9          byte[] buf = new byte[4096];     // 読み取り用バッファ
10          int lTotal = 0,                  // 総読み取りサイズ
11              lRead;                       // 実際に1回で読み取ったサイズ
12
13          //  読み取り用ファイル
14          FileInputStream fi = new FileInputStream("Earth.jpg");
15          BufferedInputStream bi = new BufferedInputStream(fi);
16
17          //  書き出し用ファイル
18          FileOutputStream fo = new FileOutputStream("Earth_new.jpg");
19          BufferedOutputStream bo = new BufferedOutputStream(fo);
20
21          //  「読み取って書き出し」の繰り返し
22          while ( (lRead = bi.read(buf)) > 0 ) {
23              bo.write(buf,0,lRead);
24              lTotal += lRead;
25          }
26
27          //  終わればファイルを閉じる
28          bi.close();
29          bo.close();
30          //  処理結果報告
31          System.out.println("Copy is finished.");
32          System.out.println("Total size : "+lTotal+" bytes");
33      }
34  }
```

解説：

入力用のファイルは FileInputStream として開き，それを BufferedInputStream のオブジェクトに変換（14,15 行目）している．同様に，出力用のファイルは FileOutputStream として開き，それを BufferedOutputStream のオブジェクトに変換（18,19 行目）している．実際の入出作業は BufferedInputStream の bi に対して，出力作業は BufferedOutputStream のオブジェクト bo に対して行う．

このプログラムは，画像ファイル "`Earth.jpg`" の内容を読み込み，同じ内容の "`Earth_new.jpg`" を作成する．データの読み込みは 9 行目で確保している byte 型の配列 `buf`（サイズは 4,096 バイト）に このサイズずつ 行い（22 行目の read メソッド），それを 23 行目の write メソッドで出力している．これら read，write メソッドはストリームオブジェクトに対して実行する．

書き方：　read(入力を受け取るためのデータオブジェクト)

ストリームからデータを読み込み「入力を受け取るためのデータオブジェクト」にセットする．読み込む長さはこのオブジェクトのサイズを前提とする．ただし，読み込みの最終段階では，このオブジェクトと同じ長さのデータが得られないことがある．read メソッドは，実際に読み込んだデータの長さを返す．

書き方：　write(出力するデータオブジェクト, 開始位置, 長さ)

「開始位置」は「出力するデータオブジェクト」の書き出し位置（オフセット）を指定し,「長さ」には書き出し位置から出力するデータの長さを指定する．

2.7　final 修飾子

変数，クラス，メソッドの宣言時の **final 修飾子**に関する基本的な事柄について説明する．

■ 変数宣言時の final 修飾子

変数の宣言時に final 修飾子を付けると，その変数への値の割り当ては最初の 1 度のみ可能となる．これは**定数**を宣言する場合などに応用できる．

プログラム：final01.java

```
 1 class final01 {
 2     public static void main( String argv[] ) {
 3         final int a;
 4         final String s;
 5         a = 2;
 6         s = "初期値";
 7         System.out.println("a="+a+",s="+s);
 8 //      a = 3;
 9 //      s = "次の値";
10     }
11 }
```

このプログラムの 8,9 行目のコメントを外す（値の変更を試みる）とコンパイル時にエラー（下記）が発生する．

```
final01.java:8:  エラー:  変数 a はすでに代入されている可能性があります
              a = 3;
              ^
final01.java:9:  エラー:  変数 s はすでに代入されている可能性があります
              s = "次の値";
              ^
エラー 2 個
```

■ クラス宣言時の final 修飾子

クラスの宣言時に final 修飾子を付けると，そのクラスは拡張できなくなる．すなわち，他のクラスに継承することができなくなる．

プログラム：final02.java

```
final class FinClass {      // 拡張不能な（継承させない）クラスの宣言
    int a;
    FinClass() {
        a = 0;
    }
}
/*
class MyClass extends FinClass {    // 継承の試み
    int b;
    MyClass() {
        b = 1;
    }
}
*/
class final02 {
    public static void main( String argv[] ) {
        FinClass fc = new FinClass();
        System.out.println("fc="+fc);
```

```
19      }
20  }
```

このプログラムの 7,14 行目のコメントを外す（クラス拡張を試みる）とコンパイル時にエラー（下記）が発生する．

```
final02.java:8:  エラー:  final FinClass からは継承できません
class MyClass extends FinClass {
                      ^
エラー 1 個
```

クラス内のメソッド宣言にも final 修飾子が使用可能で，その場合は，オーバーライドを禁止する．

2.8 プログラムの終了

System クラスの exit メソッドを呼び出すとプログラムを終了する．

書き方：　System.exit(終了コード)

「終了コード」[18] は，当該プログラムを起動したシェル環境（コマンドウィンドウ，ターミナルウィンドウ）に通知される．

例. Windows のコマンドプロンプトウィンドウでの終了コードの確認

```
echo %ERRORLEVEL%
```

例. bash（Linux，macOS）での終了コードの確認

```
echo $?
```

このメソッドを使用すると，プログラムを強制的に終了させることができる．

[18] **終了ステータス**とも言う．多くの場合，プログラムが正常終了する場合は終了コードは 0 とする．

3　標準的なクラスライブラリ

Java 言語処理系には多数の高機能なクラスライブラリが付属しており，作成するプログラムにそれらを読み込んで使用することができる．この章では特に使用頻度の高いものについて解説する．

3.1　データ構造

複数の要素を持つデータ構造として最も基本的なものに**配列**[19] がある．インデックス（添字）による配列の要素へのアクセスは高速であるが，

(1) サイズの変更が基本的には許されていない

(2) 要素の挿入と削除がやりにくい（処理に時間がかかる）

といった問題もあり，より柔軟なデータ構造が必要となる．ここでは，配列よりも柔軟で高機能なデータ構造について説明する．

3.1.1　List（リスト）：　ArrayList / LinkedList

1 次元配列に似たデータ構造に**リスト**がある．リストは要素の追加や挿入，削除が容易にできるデータ構造である．具体的には ArrayList，LinkedList というクラスが標準的に提供されており，それぞれ

```
import java.util.ArrayList
import java.util.LinkedList
```

をプログラムに読み込むことで利用できる．これらクラスのインスタンスを生成すると空のリストが作成され，それに対して要素の追加や挿入，削除の処理を行う．リストの要素には**インデックス**（〜番目）を指定してアクセスする．配列の場合と同じく，先頭要素のインデックスは 0 である．

リストのオブジェクトを生成する場合は，要素のクラスを<...>で括って指定する．

書き方：　ArrayList<クラス> インスタンス名 = new ArrayList<クラス名>()

書き方：　LinkedList<クラス> インスタンス名 = new LinkedList<クラス名>()

注意）

<クラス>の部分には int，double など変数の型を与えることはできない．変数の型はクラスとは異なる**プリミティブ型**と呼ばれるものであり，リストの要素の型指定に与える際は，プリミティブ型をクラスとして定義したもの（「2.5.6 総称型（Generics）について」（p.36）の表 13 参照）を指定しなければならない．

例. 整数値（int 型）の値を要素として持つリスト L の作成

```
ArrayList<Integer> L = new ArrayList<Integer>()
```

リストのオブジェクトに対する基本的なメソッドを表 16 に示す．

表 16: リストに対するメソッド（基本的なもの）

メソッド	解説
add(要素) add(n, 要素)	リストに対して新たに「要素」を追加する.（末尾に追加） リストの n 番目の位置に「要素」を挿入する.
get(n) set(n, 新要素)	リストの n 番目の要素を返す. リストの n 番目の要素を「新要素」で置き換える.
remove(n) clear( )	リストの n 番目の要素を削除する. リスト全要素を抹消する.
size( )	リストの要素数を int 型で返す.

リストを用いたサンプルプログラム List01.java を示す．

[19]「2.4 配列」（p.21）参照.

プログラム：List01.java

```
1  import java.util.ArrayList;
2  import java.util.LinkedList;
3
4  class List01 {
5      public static void main( String argv[] ) {
6          int c;
7          // リストの作成（要素はString）
8          ArrayList<String> lst = new ArrayList<String>();
9  //      LinkedList<String> lst = new LinkedList<String>();
10         // 要素の追加
11         lst.add("apple");
12         lst.add("orange");
13         lst.add("grape");
14         // 要素の参照
15         System.out.println("リストの内容："+lst+", 長さ："+lst.size() );
16         for ( c = 0; c < lst.size(); c++ ) {
17             System.out.println(c+" : "+lst.get(c));
18         }
19         // 要素の挿入
20         lst.add(1,"banana");
21         System.out.println("addで1番目に「baana」を挿入すると："+lst);
22         // 要素の書き換え
23         lst.set(2,"みかん");
24         System.out.println("setで2番目を「みかん」にすると："+lst);
25         System.out.println("indexOfでgrapeを探すと："+lst.indexOf("grape"));
26         // 要素の削除
27         lst.remove(2);
28         System.out.println("lst.remove(2)すると："+lst);
29         // 初期化
30         lst.clear();
31         System.out.println("lst.clear()すると："+lst);
32     }
33 }
```

このプログラムを実行すると次のような出力となる．

```
リストの内容：[apple, orange, grape], 長さ：3
0 :  apple
1 :  orange
2 :  grape
add で 1 番目に「baana」を挿入すると：[apple, banana, orange, grape]
set で 2 番目を「みかん」にすると：[apple, banana, みかん, grape]
indexOf で grape を探すと：3
lst.remove(2) すると：[apple, banana, grape]
lst.clear() すると：[]
```

このプログラムは ArrayList の動作試すものである．ArrayList と LinkedList は機能としてはほぼ同じであり，プログラムの 8 行目をコメントにし，9 行目のコメントを外すことで LinkedList の動作を試すことができる．(同じ結果となる）　しかし表 17 に示すように処理の種類によってその性能（速度）が異なるので使い分ける必要がある．

表 17: ArrayList と LinkedList の処理速度の比較

処理の種類	ArrayList	LinkedList
要素の参照，書き換え	早い	遅い
要素の挿入，削除	遅い	早い

「F.1 ArrayList，LinkedList の性能比較」（p.217）参照

■ ArrayList，LinkedList を使い分けるための 1 つの判断基準

リストへの要素の追加と削除が頻繁に発生する処理においては LinkedList を，それ以外の場合では ArrayList を使用するのが良い．

3.1.1.1　リスト生成時の初期化

リストオブジェクトを生成する際に，初期の値を与えることができる．例えば String オブジェクトを要素とする ArrayList を宣言する際に初期の要素を与えるには次のように記述する．

【ArrayList の初期化の例 1】

```
ArrayList<String> lst = new ArrayList<String>() {
    {
        add("orange");
        add("grape");
            :
    }
};
```

このように add メソッドを書き並べて初期化することができる．あるいは，リストのコンストラクタの引数に Arrays クラス（java.util.Arrays）の asList メソッドで生成した値を与える方法もある．（次の例参照）

【ArrayList の初期化の例 2】

```
ArrayList<String> lst = new ArrayList<String>(
    Arrays.asList("みかん", "ぶどう", "バナナ", "りんご"));
```

asList の引数には配列を与えることもできる．この方法を用いると配列をリストに変換することができる．

【ArrayList の初期化の例 3】

```
String ar[ ] = {"みかん","ぶどう","バナナ","りんご"};
ArrayList<String> lst = new ArrayList<String> ( Arrays.asList(ar) );
```

LinkedList に関しても同様の方法で初期化できる．

3.1.1.2　要素の整列：　sort

Collections クラス（java.util.Collections）の sort メソッドを使用するとリストの要素の順番を整列することができる．

書き方：　Collections.sort(リストオブジェクト)

「リストオブジェクト」の要素が**昇順**に整列される．

3.1.1.3　要素の順序の反転：　reverse

Collections クラス（java.util.Collections）の reverse メソッドを使用するとリストの要素の順番を反転することができる．

書き方：　Collections.reverse(リストオブジェクト)

「リストオブジェクト」の要素の順序が反転される．

sort，reverse メソッドを用いたサンプルプログラムを示す．

プログラム：List03.java

```
1  import java.util.ArrayList;
2  import java.util.Collections;
3  import java.util.Arrays;
4
5  class List03 {
6      public static void main( String argv[] ) {
7          // 初期化
8          ArrayList<String> lst = new ArrayList<String>(
9              Arrays.asList("orange", "grape", "banana", "apple"));
```

```
10          // 内容確認
11          System.out.println("整列前："+lst);
12          // 整列（ソート）
13          Collections.sort(lst);
14          System.out.println("整列後："+lst);
15          // 要素の順序の反転
16          Collections.reverse(lst);
17          System.out.println("反転後："+lst);
18      }
19  }
```

このプログラムを実行した例を次に示す．

```
整列前：[orange, grape, banana, apple]
整列後：[apple, banana, grape, orange]
反転後：[orange, grape, banana, apple]
```

3.1.1.4　拡張 for 文による List の要素へのアクセス

拡張 for 文を用いて List の要素にアクセスすることができる．サンプルプログラム ExFor02.java にそれを示す．

プログラム：ExFor02.java

```
1   import java.util.ArrayList;
2   import java.util.Arrays;
3
4   class ExFor02 {
5       public static void main( String argv[] ) {
6           // 初期化
7           ArrayList<String> lst = new ArrayList<String>(
8               Arrays.asList("orange", "grape", "banana", "apple"));
9           // 拡張for文
10          for ( String s : lst ) {
11              System.out.print(s+",");
12          }
13      }
14  }
```

このプログラムを実行した例を次に示す．

```
orange,grape,banana,apple,
```

3.1.1.5　Comparator を用いた整列

リストは単純な数値や文字列だけではなく，様々なクラスのオブジェクトを要素として持つ形で作成できる．例えば次のように定義されたクラスのオブジェクトを要素として持つリストについて考える．

```
class MyRecord {
    String key, value;
    MyRecord(String k, String v) {
        key = k;
        value = v;
    }
}
```

このようなクラスのオブジェクトを要素として持つリストを整列する場合，要素（MyRecord オブジェクト）のどの部分（どのメンバの値）を基準にして整列するかが問われる．このように，整列順序の基準を独自に定めるために Comparator インターフェース（java.util.Comparator）が利用できる．

■ 独自の基準でリストを整列するための手順

(1) Comparator インターフェースを実装したクラスを定義し，そのクラスでメソッド compare を定義する．
このメソッドは Comparator インターフェースに定義されたメソッドであり，これの定義内容を記述する．

(2) 上で定義したクラスのインスタンスを生成し，sort メソッドに与えて整列を実行する．

この手順を実現するサンプルプログラムを示す．

プログラム：CmprtrSort.java

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

// データレコード用のクラス
class MyRecord {
    // メンバ
    String key, value;
    // コンストラクタ
    MyRecord( String k, String v ) {
        key = k;
        value = v;
    }
    // 出力メソッド
    void printRec() {
        System.out.println("["+key+","+value+"]");
    }
}

// MyRecordのためのComparator
class MyCmp1 implements Comparator<MyRecord> {
    public int compare( MyRecord r1, MyRecord r2 ) {
        String k1 = r1.key, k2 = r2.key;
        return( k1.compareTo(k2) ); // keyを比較
    }
}
class MyCmp2 implements Comparator<MyRecord> {
    public int compare( MyRecord r1, MyRecord r2 ) {
        String v1 = r1.value, v2 = r2.value;
        return( v1.compareTo(v2) ); // valueを比較
    }
}

// メイン
class CmprtrSort {
    public static void main( String argv[] ) {
        // 配列の作成
        ArrayList<MyRecord> lst = new ArrayList<MyRecord>();
        lst.add( new MyRecord("orange","ミカン") );
        lst.add( new MyRecord("grape","ブドウ") );
        lst.add( new MyRecord("banana","バナナ") );
        lst.add( new MyRecord("apple","リンゴ") );
        // 内容確認
        int c;
        for ( c = 0; c < lst.size(); c++ ) {
            lst.get(c).printRec();
        }
        System.out.println();
        // 整列処理(keyで整列)
        Collections.sort( lst, new MyCmp1() );
        System.out.println("*** keyで整列後 ***");
        for ( c = 0; c < lst.size(); c++ ) {
            lst.get(c).printRec();
        }
        System.out.println();
        // 整列処理(valueで整列)
        Collections.sort( lst, new MyCmp2() );
        System.out.println("*** valueで整列後 ***");
        for ( c = 0; c < lst.size(); c++ ) {
            lst.get(c).printRec();
        }
    }
}
```

解説： このプログラムでは Comparator インターフェースを実装する 2 つのクラス MyCmp1，MyCmp2 が定義されている．これらクラスにはそれぞれ compare メソッドが 2 つの引数を取る形で記述されている．このメソッドは 2 つの引数の間の順序関係を判定するものである．判定結果は「2.3.1.2 文字列の比較」（p.16）で説明した compareTo メソッドと同様の値を返すように記述する．

実際に整列を実行するには sort メソッドの第 2 引数に MyCmp1 あるいは MyCmp2 のインスタンスを与える．

Comparator を実装するクラスは次のような形で定義する．

```
class　クラス名　implements　Comparator<要素のクラス>　{
        public int compare( 要素のクラス　引数 1, 要素のクラス　引数 2 ) {
                引数 1 と引数 2 の順序関係を整数（int 型）で返すプログラム
        }
}
```

このプログラムを実行した例を次に示す．

```
[orange, ミカン]
[grape, ブドウ]
[banana, バナナ]
[apple, リンゴ]
*** key で整列後 ***
[apple, リンゴ]
[banana, バナナ]
[grape, ブドウ]
[orange, ミカン]
*** value で整列後 ***
[banana, バナナ]
[grape, ブドウ]
[orange, ミカン]
[apple, リンゴ]
```

重要） Comparator はラムダ式（p.159「6 ラムダ式」で解説する）で記述することもでき，より簡潔にプログラムを表現することができる．（サンプルプログラム CmprtrSort2.java ）

プログラム： CmprtrSort2.java

```
1  import java.util.ArrayList;
2  import java.util.Collections;
3
4  // データレコード用のクラス
5  class MyRecord {
6      // メンバ
7      String key, value;
8      // コンストラクタ
9      MyRecord( String k, String v ) {
10         key = k;
11         value = v;
12     }
13     // 出力メソッド
14     void printRec() {
15         System.out.println("["+key+","+value+"]");
16     }
17 }
18
19 // メイン
20 class CmprtrSort2 {
21     public static void main( String argv[] ) {
22         // 配列の作成
23         ArrayList<MyRecord> lst = new ArrayList<MyRecord>();
24         lst.add( new MyRecord("orange","ミカン") );
25         lst.add( new MyRecord("grape","ブドウ") );
26         lst.add( new MyRecord("banana","バナナ") );
27         lst.add( new MyRecord("apple","リンゴ") );
```

```
28          // 内容確認
29          int c;
30          for ( c = 0; c < lst.size(); c++ ) {
31              lst.get(c).printRec();
32          }
33          System.out.println();
34          // 整列処理(keyで整列)
35          Collections.sort( lst, (k1,k2) -> (k1.key).compareTo(k2.key) );
36          System.out.println("*** keyで整列後 ***");
37          for ( c = 0; c < lst.size(); c++ ) {
38              lst.get(c).printRec();
39          }
40          System.out.println();
41          // 整列処理(valueで整列)
42          Collections.sort( lst, (k1,k2) -> (k1.value).compareTo(k2.value) );
43          System.out.println("*** valueで整列後 ***");
44          for ( c = 0; c < lst.size(); c++ ) {
45              lst.get(c).printRec();
46          }
47      }
48  }
```

このプログラムの 35,42 行目がラムダ式を用いた整列処理である．プログラム中では Comparator を実装するクラスの定義やそのメソッドの生成などを行っていない．このプログラムは先の CmprtrSort.java と同じ実行結果となる．

3.1.2 Set（セット，データ集合）： HashSet / TreeSet

重複のない複数の要素を保持するデータ構造として HashSet と TreeSet があり，それぞれライブラリ java.util.HashSet, java.util.TreeSet として提供されている．両者はほぼ同じ機能を持つが，特徴に若干の違いがある．HashSet は高速にアクセスできる Set であるが順序の概念がない．これに対して TreeSet はアクセス速度の面では HashSet に劣るが，要素の順序が自動的に整列されるのと，使用するメモリの量が少ないという特徴[20] がある．

Set は，重複のない大量の要素を保持し，要素の存在を高速に検索する処理に適している．Set オブジェクトは次のようにして生成する．

書き方： HashSet<要素のクラス> オブジェクト名 = new HashSet<要素のクラス>()
書き方： TreeSet<要素のクラス> オブジェクト名 = new TreeSet<要素のクラス>()

Set 生成時に要素を与えて初期化する方法として「3.1.1.1 リスト生成時の初期化」（p.55）で解説したのと類似の方法が使用できる．

Set オブジェクトに対する基本的なメソッドを表 18 に示す．

表 18: Set オブジェクトに対するメソッド（基本的なもの）

メソッド	解説
add(要素) remove(要素) clear()	Set に「要素」を追加する． Set から「要素」を削除する． Set の要素を全て抹消する．
contains(要素) isEmpty( ) size( )	Set が「要素」を持つ場合 true，それ以外の場合は false を返す． Set が 1 つも要素を持たない場合 true，それ以外の場合は false を返す． Set の要素の個数を返す．
equals(他のセット)	「他のセット」と同じ場合 true，それ以外の場合は false を返す．
addAll(他のセット) retainAll(他のセット) removeAll(他のセット)	「他のセット」の要素を全て追加する． 「他のセット」と重複する要素のみ残す． 「他のセット」と重複する要素を削除する．

[20] **平衡二分探索木**を応用したデータ構造であり，これが「Tree」という命名の由来である．

HashSet を用いて Set の機能を示すサンプルプログラム Set01.java を示す.

プログラム：Set01.java

```
import java.util.HashSet;
import java.util.Arrays;

class Set01 {
    public static void main( String argv[] ) {
        // Setの作成
        HashSet<String> set1 = new HashSet<String>(
            Arrays.asList("orange","grape","banana","apple"));
        HashSet<String> set2 = new HashSet<String>(
            Arrays.asList("peach","grape","banana","persimmon"));
        HashSet<String> set3 = new HashSet<String>(
            Arrays.asList("orange","grape","banana","apple"));
        HashSet<String> set4 = new HashSet<String>();
        System.out.println("set1："+set1);
        System.out.println("set2："+set2);
        System.out.println("set3："+set3);
        System.out.println("set4："+set4);
        // 要素数
        System.out.println("set1の要素数："+set1.size());
        // 空かどうか
        System.out.println("set1は空か："+set1.isEmpty());
        System.out.println("set4は空か："+set4.isEmpty());
        // 等しいか検査
        System.out.println("set1とset2が等しいか："+set1.equals(set2));
        System.out.println("set1とset3が等しいか："+set1.equals(set3));
        // 共通集合
        HashSet<String> set11 = new HashSet<String>(set1);  // set1の複製
        set11.retainAll(set2);
        System.out.println("set1とset2の共通集合：      "+set11);
        // 和集合
        HashSet<String> set12 = new HashSet<String>(set1);  // set1の複製
        set12.addAll(set2);
        System.out.println("set1とset2の和集合：        "+set12);
        // 差集合
        HashSet<String> set13 = new HashSet<String>(set1);  // set1の複製
        set13.removeAll(set2);
        System.out.println("set1からset2を除いた集合：  "+set13);
        // クリア
        set1.clear();
        System.out.println("set1.clear()の後：  "+set1);
    }
}
```

このプログラムの実行例を次に示す.

```
set1：[orange, banana, apple, grape]
set2：[banana, persimmon, peach, grape]
set3：[orange, banana, apple, grape]
set4：[]

set1 の要素数：4
set1 は空か：false
set4 は空か：true

set1 と set2 が等しいか：false
set1 と set3 が等しいか：true

set1 と set2 の共通集合：      [banana, grape]
set1 と set2 の和集合：        [orange, banana, apple, persimmon, peach, grape]
set1 から set2 を除いた集合：  [orange, apple]

set1.clear() の後：  []
```

演習課題. set1 ∪ set2 − set1 ∩ set2 を求める機能を Set01.java に追加せよ.

3.1.2.1 Iterator による要素への順次アクセス

Iterator インターフェース（java.util.Iterator）を用いることで，Set の要素を 1 つずつ順番に取り出すことができる．

【Iterator の生成】

Iterator<クラス名>　オブジェクト名 = Set オブジェクト.iterator()

「クラス名」には「Set オブジェクト」の要素のクラス名を指定する．これにより Iterator クラスのインスタンスである「オブジェクト」が生成される．Iterator オブジェクトは元の Set オブジェクトの要素に順番にアクセスするためのものであり，これに対して next メソッドを実行することで要素を 1 つずつ取り出すことができる．また取り出す要素が残っているかどうかを検査するには hasNext メソッドを用いる．

Iterator クラスを用いたサンプルプログラム Set02.java を示す．

プログラム：Set02.java

```
1  import java.util.HashSet;
2  import java.util.TreeSet;
3  import java.util.Arrays;
4  import java.util.Iterator;
5
6  class Set02 {
7      public static void main( String argv[] ) {
8          // Setの作成
9          HashSet<String> set1 = new HashSet<String>(
10             Arrays.asList("orange","grape","banana","apple"));
11         TreeSet<String> set2 = new TreeSet<String>(
12             Arrays.asList("orange","grape","banana","apple"));
13         System.out.println("set1："+set1);
14         System.out.println("set2："+set2);
15         // Iteratorを介して要素を1つずつ取り出す：set1
16         Iterator<String> it = set1.iterator();
17         System.out.print("set1の要素に1つずつアクセス：");
18         while ( it.hasNext() ) {
19             System.out.print( it.next()+"," );
20         }
21         System.out.println();
22         // 拡張for文で要素を1つずつ取り出す：set2
23         System.out.print("set2の要素に1つずつアクセス：");
24         for ( String s : set2 ) {
25             System.out.print( s+"," );
26         }
27     }
28 }
```

このプログラムを実行した例を次に示す．

```
set1：[orange, banana, apple, grape]
set2：[apple, banana, grape, orange]
set1 の要素に 1 つずつアクセス：orange,banana,apple,grape,
set2 の要素に 1 つずつアクセス：apple,banana,grape,orange,
```

Set の要素に 1 つずつ順番にアクセスできていることがわかる．また TreeSet では要素が整列されていることも確認できる．

24〜26 行目では**拡張 for 文**を用いて set2 の要素にアクセスしている．

3.1.3 Map（連想配列，辞書）： HashMap / TreeMap

Map は**キー**と**値**の対応（マッピング）を保持するものである．配列や List では，要素へのアクセスはインデックス（整数）が基本であるが，Map を用いると，整数以外の値にもデータを対応させることができ，主記憶域上に**辞書**のようなデータテーブルを実現してキーによる値の検索を実現することができる．登録されるマッピングにおけるキー

の値の重複は許されない．

Map には HashMap クラス（java.util.HashMap）と TreeMap クラス（java.util.TreeMap）があり，両者はほぼ同じ機能を持つが，特徴に若干の違いがある．HashMap は高速にアクセスできる Map であるが，マッピングの登録に順序の概念がない．これに対して TreeMap はアクセス速度の面では HashMap に劣るが，マッピングの登録の順序が自動的にキーで整列されるのと，使用するメモリの量が少ないという特徴がある．

Map オブジェクトは次のようにして生成する．

書き方： HashMap<キーのクラス><値のクラス>　オブジェクト名 =
new HashMap<キーのクラス><値のクラス>()

書き方： TreeMap<キーのクラス><値のクラス>　オブジェクト名 =
new TreeMap<キーのクラス><値のクラス>()

Map オブジェクトに対する基本的なメソッドを表 19 に示す．

表 19: Map に対するメソッド（基本的なもの）

メソッド	解説
put(キー, 値)	「キー」と「値」のマッピングを登録する．
get(キー)	「キー」に対する「値」を返す．
remove(キー)	「キー」の登録があれば，その値とともに削除する．
clear( )	全ての登録情報を抹消する．
containsKey(キー)	「キー」の登録があれば true を，なければ false を返す．
containsValue(値)	「値」を持つ登録があれば true を，なければ false を返す．
isEmpty( )	マッピングの登録がなければ true を，あれば false を返す．
size( )	登録されているマッピングの数を返す．
keySet( )	登録されている全てのキーの Setビュー を返す．
values( )	登録されている全ての値の Collectionビュー を返す．
entrySet( )	登録されている全てのマッピングの ビュー を返す．

ビュー：通常の Set や List ではなく，元のオブジェクトを参照するもの

Map の基本的な処理に関するサンプルプログラム Map01.java を示す．

プログラム：Map01.java

```
import java.util.HashMap;

class Map01 {
    public static void main( String argv[] ) {
        // Mapの作成
        HashMap<String,String> m = new HashMap<String,String>() {
            {
                put("orange","ミカン");
                put("grape", "ブドウ");
                put("banana","バナナ");
                put("apple", "リンゴ");
            }
        };
        System.out.println("m："+m);
        System.out.println("m.size()："+m.size());
        System.out.println("m.isEmpty()："+m.isEmpty());
        // マッピングの参照
        System.out.println("m.get(\"banana\")："+m.get("banana"));
        // キー，値の存在検査
        System.out.println("キーにgrapeがあるか："+m.containsKey("grape"));
        System.out.println("キーにpeachがあるか："+m.containsKey("peach"));
        System.out.println("値に「リンゴ」があるか："+m.containsValue("リンゴ"));
        System.out.println("値に「モモ」があるか：  "+m.containsValue("モモ"));
        // 削除
        m.remove("banana");
        System.out.println("banana消去後のm："+m);
        m.clear();
```

```
28          System.out.println("m.clear()の後："+m);
29          System.out.println("m.isEmpty()："+m.isEmpty());
30      }
31  }
```

このプログラムは HashMap にマッピングを登録し，基本的なメソッドの動作の例を示すものである．このプログラムの実行結果の例を次に示す．

```
m：{orange=ミカン，banana=バナナ，apple=リンゴ，grape=ブドウ }
m.size()：4
m.isEmpty()：false
m.get("banana")：バナナ
キーに grape があるか：true
キーに peach があるか：false
値に「リンゴ」があるか：true
値に「モモ」があるか：  false
banana 消去後の m：{orange=ミカン，apple=リンゴ，grape=ブドウ }
m.clear() の後：{}
m.isEmpty()：true
```

3.1.3.1　登録されているマッピングに順次アクセスする方法

■ キー，値を別々に取り出す

Map オブジェクトに対して keySet メソッドを実行すると，キーの Set の**ビュー**を取り出すことができる．これに対して iterator メソッドを実行すると反復処理のためのイテレータが得られる．また，Set のビューに対して**拡張 for 文**を使用することもできる．

Map オブジェクトに対して values メソッドを実行すると，値の Collection のビューを取り出すことができる．これに対して iterator メソッドを実行すると反復処理のためのイテレータが得られる．また，Collectiont のビューに対して拡張 for 文を使用することもできる．あるいは，Collectiont のビューから List を作成することもできる．

keySet，values メソッドを使用するサンプルプログラム Map02.java を示す．

プログラム：Map02.java

```
1   import java.util.HashMap;
2   import java.util.TreeMap;
3   import java.util.Iterator;
4   import java.util.ArrayList;
5
6   class Map02 {
7       public static void main( String argv[] ) {
8           // Mapの作成
9           HashMap<String,String> m1 = new HashMap<String,String>() {
10              {
11                  put("orange","ミカン");
12                  put("grape", "ブドウ");
13                  put("banana","バナナ");
14                  put("apple", "リンゴ");
15              }
16          };
17          // 上で作成したMapを元にTreeMapを作成
18          TreeMap<String,String> m2 = new TreeMap<String,String>(m1);
19          // キーの集合，値の列を別々に取得して出力（HashMap）
20          Iterator it = m1.keySet().iterator();
21          System.out.print("HashMapのキー：");
22          while ( it.hasNext() ) {                // Iteratorによる方法
23              System.out.print(it.next()+",");
24          }
25          System.out.println();
26          it = m1.values().iterator();
27          System.out.print("HashMapの値：  ");
28          while ( it.hasNext() ) {                // Iteratorによる方法
29              System.out.print(it.next()+",");
```

```
30          }
31          System.out.println();
32          // キーの集合，値の列を別々に取得して出力（TreeMap）
33          System.out.print("TreeMapのキー：");
34          for ( String k : m2.keySet() ) {        // 拡張for文による方法
35              System.out.print(k+",");
36          }
37          System.out.println();
38          // Collectionのビューからリストを生成することが可能
39          ArrayList<String> v2 = new ArrayList<String>(m2.values());
40          System.out.println("TreeMapの値：  "+v2);
41      }
42  }
```

このプログラムを実行した例を次に示す．

```
HashMap のキー：orange,banana,apple,grape,
HashMap の値：  ミカン, バナナ, リンゴ, ブドウ,
TreeMap のキー：apple,banana,grape,orange,
TreeMap の値：  [リンゴ，バナナ，ブドウ，ミカン]
```

TreeMap では，キーの順序が整列されていることが確認できる．

■ キーと値のペアを取り出す方法

Map オブジェクトに対して entrySet メソッドを実行すると，全ての**マッピング**の Set のビューを取り出すことができる．個々のマッピングは Entry である．

Entry は java.util.Map.Entry に定義されているインターフェースで，これに対して getKey メソッドを実行するとキーが得られる．同様に getValue メソッドを実行すると値が得られる．

entrySet，getKey，getValue を用いたサンプルプログラム Map03.java を示す．

プログラム：Map03.java

```
1   import java.util.HashMap;
2   import java.util.Map.Entry;
3
4   class Map03 {
5       public static void main( String argv[] ) {
6           // Mapの作成
7           HashMap<String,String> m = new HashMap<String,String>() {
8               {
9                   put("orange","ミカン");
10                  put("grape", "ブドウ");
11                  put("banana","バナナ");
12                  put("apple", "リンゴ");
13              }
14          };
15          // マッピングを個々に取り出す
16          for ( Entry<String,String> e : m.entrySet() ) {
17              System.out.println( e.getKey() + ':' + e.getValue() );
18          }
19      }
20  }
```

このプログラムを実行した例を次に示す．

```
orange:ミカン
banana:バナナ
apple:リンゴ
grape:ブドウ
```

3.1.4 データ構造の変換

3.1.4.1 リスト，セットを配列に変換する方法

リストやセットを配列に変換するには toArray メソッドを使用する．このメソッドの引数には，変換先の配列オブジェクトを与える．

例. ArrayList lst を String の配列 ar に変換する

```
String ar[] = lst.toArray( new String[lst.size()] )
```

配列からリスト，配列からセット，それらの逆の変換処理をサンプルプログラム List02_1.java に示す．

プログラム：List02_1.java

```
1  import java.util.ArrayList;
2  import java.util.HashSet;
3  import java.util.Arrays;
4
5  class List02_1 {
6      public static void main( String argv[] ) {
7          // 配列からリストを作成する
8          String ar[] = {"みかん","ぶどう","バナナ","りんご"};
9          ArrayList<String> lst = new ArrayList<String> ( Arrays.asList(ar) );
10         System.out.print("配列からリストへ：");
11         for ( String s : lst ) {
12             System.out.print(s+",");
13         }
14         System.out.println();
15
16         // 配列からセットを作成する
17         HashSet<String> st = new HashSet<String> ( Arrays.asList(ar) );
18         System.out.print("配列からセットへ：");
19         System.out.println(st);
20
21         // リストから配列に変換
22         String ar2[] = lst.toArray( new String[lst.size()] );
23         System.out.print("リストから配列へ：");
24         for ( String s : ar2 ) {
25             System.out.print( s+"," );
26         }
27         System.out.println();
28
29         // セットから配列に変換
30         String ar3[] = st.toArray( new String[st.size()] );
31         System.out.print("セットから配列へ：");
32         for ( String s : ar3 ) {
33             System.out.print( s+"," );
34         }
35     }
36 }
```

このプログラムを実行した例を次に示す．

```
配列からリストへ：みかん, ぶどう, バナナ, りんご,
配列からセットへ：[りんご, ぶどう, みかん, バナナ]
リストから配列へ：みかん, ぶどう, バナナ, りんご,
セットから配列へ：りんご, ぶどう, みかん, バナナ,
```

3.2 編集可能な文字列のクラス： StringBuilder，StringBuffer

文字列を扱うための基本的なクラスに String があるが，String オブジェクトの内容は事後に編集することができない．挿入や追加などの編集が自由にできる可変長の文字列データを扱うために StringBuilder クラスと StringBuffer がある．

StringBuilder，StringBuffer は類似の機能を提供するが，それぞれに特徴がある．StringBuilder はデータの編集が高速であるが，スレッドセーフではない．従って，単一のスレッド内で文字列データを編集するのに適している．これに対して StringBuffer は StringBuilder と比べて編集処理が少し遅いが [21] スレッドセーフであり，複数のスレッドで排他制御しながらデータを共有するのに適している．

3.2.1 コンストラクタ

StringBuilder，StringBuffer クラスのコンストラクタを表 20 に示す．

表 20: コンストラクタの働き

StringBuilder	StringBuffer	説明
`StringBuilder( )`	`StringBuffer( )`	インスタンスを生成し，初期容量として 16 バイトを割り当てる．

インスタンス生成直後は，空の（長さが 0 の）文字列が格納される．

コンストラクタの引数に String オブジェクトを与えて初期値を設定することもできる．これは String オブジェクトを StringBuilder / StringBuffer に変換する方法でもある．

3.2.2 文字列編集のためのメソッド

tringBuilder，StringBuffer クラスのオブジェクトに対する代表的な編集メソッドを表 21 に示す．

表 21: 編集のための代表的なメソッド

メソッド	説明
`capacity( )`	編集領域の容量を返す．
`length( )`	格納されている文字列の長さを返す．
`append(値)`	データの末尾に「値」を追加する．値としては各種の型やクラスのものを与えることができる．
`insert(int pos, 値)`	pos で示される位置に「値」を挿入する．値としては各種の型やクラスのものを与えることができる．
`substring(int s, int e)`	s 番目から e-1 番目までの部分文字列を返す．戻り値は String クラスのオブジェクトとして得られる．第 2 引数を省くと末尾までの部分文字列が得られる．
`indexOf(String s)`	文字列 s が最初に現れる位置を int 型で返す．（文字列検索機能）
`delete(int s, int e)`	s 番目から e-1 番目までの部分を削除する．
`reverse( )`	文字列の順序を反転する．
`toString( )`	文字列を String クラスの値として返す．

3.2.3 サンプルプログラム

0 から 24 までの数をつなげた後に編集を施すサンプルプログラム StringTest01.java を示す．

[21] 扱うデータサイズなどの条件に左右されるので，この比較が正しくないこともある．

プログラム：StringTest01.java

```
1  class StringTest01 {
2      public static void main(String argv[]) {
3          StringBuilder sb = new StringBuilder();
4          String s;
5          int c;
6          // appendによる追加
7          for ( c = 0; c < 25; c++ ) {
8              s = String.valueOf(c);
9              sb.append(s+",");
10         }
11         sb.append("end");
12         // 確認
13         System.out.println(sb);
14         System.out.println("length: "+sb.length());
15         System.out.println("capacity: "+sb.capacity());
16         int p = sb.indexOf("20");   // "20"を探索
17         System.out.println("position of '20': "+p);
18         // "20,"を削除
19         sb.delete(p,p+3);
20         System.out.println("after dropping '20': ");
21         System.out.println(sb);
22         // "25,26,27,"を挿入
23         sb.insert(62,"25,26,27,");
24         System.out.println("after the insertion of '25,26,27,': ");
25         System.out.println(sb);
26         System.out.println("length: "+sb.length());
27         System.out.println("capacity: "+sb.capacity());
28         // 反転
29         sb.reverse();
30         System.out.println("reverse: ");
31         System.out.println(sb);
32     }
33 }
```

解説：

7〜10 行目は，StringBuilder オブジェクトに数字をコンマ区切りでつなげている部分で，11 行目は．StringBuilder オブジェクトの末尾に ‘end’ をつなげる部分である．

16 行目で StringBuilder オブジェクトの中から ‘20’ を見つけ出し，19 行目でそれを削除している．

23 行目で ”25,26,27,” を挿入している．

このプログラムの実行結果の例を次に示す．

```
0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,end
length:  68
capacity:  70
position of '20':  50
after dropping '20':
0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,21,22,23,24,end
after the insertion of '25,26,27,':
0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,21,22,23,24,25,26,27,end
length:  74
capacity:  142
reverse:
dne,72,62,52,42,32,22,12,91,81,71,61,51,41,31,21,11,01,9,8,7,6,5,4,3,2,1,0
```

3.2.4 byte 配列,String/StringBuilder/StringBuffer オブジェクトとの間の変換

■ byte 配列→ String

byte 配列を String オブジェクトに変換するには，String クラスのコンストラクタの引数に文字コード名と共に与える [22] ．

[22] この方法は**文字コードの変換**に利用できる．

例. byte 配列 bt にあるデータを utf-8 の文字コードと見做して String オブジェクトに与える.

```
String str = new String(bt,"utf-8");
```

■ **String → byte 配列**

String オブジェクトを byte 配列に変換するには，getBytes メソッドを用いる.

例. utf-8 の文字コードの String オブジェクト str を byte 配列 bt に変換する.

```
byte bt[] = str.getBytes("utf-8");
```

文字コード名としては "utf-8" の他にも "sjis","utf-16" などが指定できる．それらは扱うデータの文字コード体系に応じて使い分ける.

注意）文字コード名を指定してデータを変換する際には，UnsupportedEncodingException 例外が発生する可能性があるので，この例外の処理を記述する必要がある．この例外の定義は

java.io.UnsupportedEncodingException

にあるので，実装に際してはこれを import しておくこと.

byte 配列と StringBuilder/StringBuffer オブジェクトとの間のデータ変換を実現するには，一旦 String に変換する形をとる.

これらの処理を実行するサンプルプログラム ByteString01.java を示す.

プログラム：ByteString01.java

```
1  import java.io.UnsupportedEncodingException;
2
3  class ByteString01 {
4      public static void main( String argv[] ) {
5          String s1 = "言語";
6          try {
7              // String -> byte[] の変換
8              byte[] bt = s1.getBytes("utf-8");
9              System.out.print("byte[]：");
10             for ( int c = 0; c < bt.length; c++ ) { // 内容確認
11                 System.out.printf("%02x,",bt[c]);
12             }
13             System.out.println();
14             // byte[] -> String の変換
15             String s2 = new String(bt,"utf-8");
16             System.out.println("byte[]->String："+s2);
17             // String -> StringBuilder, StringBuffer
18             StringBuilder sb = new StringBuilder(s2);
19             StringBuffer sbf = new StringBuffer(s2);
20             System.out.println("StringBuilder："+sb);
21             System.out.println("StringBuffer： "+sbf);
22         } catch( UnsupportedEncodingException e ) {
23             System.out.println("エラー："+e);
24         } finally {
25             System.out.println("処理が終了しました.");
26         }
27     }
28 }
```

このプログラムを実行した結果の例を次に示す.

```
byte[]：e8,a8,80,e8,aa,9e,
byte[]->String：言語
StringBuilder：言語
StringBuffer： 言語
処理が終了しました.
```

3.3 Math クラス

Math クラスは各種の数学的関数（表 22）をクラスメソッドの形で提供する．

表 22: Math クラスのクラスメソッド（一部）

関数（メソッド）	説明
abs(n)	n の絶対値（n の型は int/long/float/double のどれか） 戻り値は引数の型に対応する．
double ceil(double n)	切り上げ（n 以上の最も小さな整数値）
double floor(double n)	切り捨て（n 以下の最も大きな整数値）
double rint(double n)	n に最も近い整数値
round(n)	四捨五入（n の型は float/double のどちらか）引数が float の場合は戻り値は float，double の場合は戻り値は long になる
max(x, y)	x,y のうち大きい方を返す．x,y の型は共に int/long/float/double のどれかで，戻り値の型は引数に対応する．
min(x, y)	x,y のうち小さい方を返す．x,y の型は共に int/long/float/double のどれかで，戻り値の型は引数に対応する．
double exp(double x)	e^x を返す．
double log(double x)	$\log_e x$ を返す．
double pow(double x, double y)	x^y を返す．
double sqrt(double x)	$\sqrt{x}$ を返す．
double sin(double x)	$\sin x$（正弦）を返す．
double cos(double x)	$\cos x$（余弦）を返す．
double tan(double x)	$\tan x$（正接）を返す．
double asin(double x)	$\sin^{-1} x$ を返す．
double acos(double x)	$\cos^{-1} x$ を返す．
double atan(double x)	$\tan^{-1} x$ を返す．
double toDegrees(double x)	弧度法の角度 x を 360 進法の角度にして返す．
double toRadians(double x)	360 進法の角度 x を弧度法の角度にして返す．
double random( )	0 以上 1 未満の乱数を返す．
double E	e（ネイピア数）
double PI	π（円周率）

これら関数をプログラムの中の式として記述する場合は **Math. 関数名 (…)** とする．

3.3.1 BigInteger，BigDecimal

多くの場合，数値演算では int や long，float，double といった型の値を扱うが，それらの値は固定されたビット長で表現されるので，一定以上の精度（有効桁数）で演算することができない．**高精度の数値演算**を実行するには，ここで説明する BigInteger クラスや BigDecimal クラスを利用する．

BigInteger は**桁の大きな整数**を扱うためのクラスであり，BigDecimal は**高精度の浮動小数点数**を扱うためのクラスである．結果として小数点数が現れない演算には BigInteger で十分であるが，開平や除算といった演算を含む計算処理の場合は BigDecimal を使うことになる．BigDecimal による浮動小数点数の扱いにおいては，有効桁数や丸めの方法を指定する必要がある．

BigInteger，BigDecimal クラスを利用するには以下のクラスライブラリを読み込む．

java.math.BigInteger
java.math.BigDecimal

3.3.1.1 オブジェクトの生成と値のセット

BigInteger，BigDecimal のコンストラクタに値を与えてオブジェクトを生成する．

例. BigInteger オブジェクトの生成

1) `BigInteger a = new BigInteger("1234567890987654321");`
2) `BigInteger b = new BigInteger("FFFFFFFFFFFFFFFF",16);`

2 番目の例は基数を 16 として（16 進数として）値を与えている例である．

例. BigDecimal オブジェクトの生成

```
BigDecimal x = new BigDecimal("1.23456789098765432101234567 89");
```

3.3.1.2 演算

BigInteger と BigDecimal に共通のメソッドを表 23 に示す．

表 23: BigInteger と BigDecimal に共通の演算メソッド（一部）

メソッド	機能
`add(n)`	自身と n を加えた値を返す
`subtract(n)`	自身から n を引いた値を返す
`multiply(n)`	自身を n 倍した値を返す
`divide(n)`	自身を n で割った値を返す
`pow(n)`	自身を n 乗した値を返す（n は int 型）
`abs()`	自身の絶対値を返す
`max(n)`	自身と n の内，大きい方を返す
`min(n)`	自身と n の内，小さい方を返す
`negate()`	自身の符号を反転した値を返す
`remainder(n)`	自身を n で割った余りを返す
`equals(n)`	自身と n が同じなら true を，異なれば false を返す
`compareTo(n)`	自身が n より大きければ 1 を，小さければ -1 を返す 自身と n が等しければ 0 を返す　（戻り値は int）

`intValue()`
`longValue()`
`doubleValue()`
`toString()`
	自身を int 型に変換した値を返す
	自身を long 型に変換した値を返す
	自身を double 型に変換した値を返す
	自身の値を表す文字列を String クラスの値として返す

（クラスメソッド） `valueOf(n)`	n を BigInteger または BigDecimal に変換した値を返す BigInteger.valueOf(n) の場合は long から BigInteger， BigDecimal.valueOf(n) の場合は long もしくは double から BigDecimal に変換する．

※ 特に断らないかぎり，自身と引数は同じクラスであるとする．

3.3.1.3 MathContext について

BigDecimal クラスで浮動小数点数を扱う場合，**有効桁数**と**丸めの方法**を指定することができる．有効桁数と丸めの方法に関する情報を保持するクラスに MathContext がある．例えば精度と丸めを指定して除算を行うには次のようにする．

例. 精度と丸めを指定した除算

```
BigDecimal one = new BigDecimal("1");
BigDecimal nine = new BigDecimal("9");
MathContext mcx = new MathContext(30,RoundingMode.HALF_UP);
BigDecimal y = one.divide(nine,mcx);
System.out.println(y);
```

MathContext のコンストラクタの第 1 引数に 10 進の有効桁数を int 型で，第 2 引数に丸めの方法を与える．この例では有効桁数 30 桁，四捨五入による丸め（RoundingMode.HALF_UP）を指定している．
上のプログラムを実行すると，

```
0.111111111111111111111111111111
```

と表示される．丸め方は RoundingMode クラスのフィールド（表 24 参照）を与えて指定する．

MathContext は BigDecimal オブジェクト生成時にコンストラクタに与えることもできる．

例. BigDecimal オブジェクト生成時に MathContext を与える

```
MathContext mcx = new MathContext(10,RoundingMode.HALF_UP);
BigDecimal x = new BigDecimal("1.234567890987654321",mcx);
System.out.println(x);
```

このプログラムを実行すると

```
1.234567891
```

と表示される．

表 24: 丸めの例

与える数	UP 切上げ（1）	DOWN 切落し（1）	CEILING 切上げ（2）	FLOOR 切落し（2）	HALF_UP	HALF_DOWN	HALF_EVEN
5.5	6	5	6	5	6	5	6
2.5	3	2	3	2	3	2	2
1.6	2	1	2	1	2	2	2
1.1	2	1	2	1	1	1	1
1.0	1	1	1	1	1	1	1
-1.0	-1	-1	-1	-1	-1	-1	-1
-1.1	-2	-1	-1	-2	-1	-1	-1
-1.6	-2	-1	-1	-2	-2	-2	-2
-2.5	-3	-2	-2	-3	-3	-2	-2
-5.5	-6	-5	-5	-6	-6	-5	-6

※ https://docs.oracle.com/ から引用

3.3.1.4 数学関数の多倍長演算について

BigDecimal クラスには，三角関数をはじめとする数学の関数が用意されていない（平方根すら実装されていない）ので，必要に応じてプログラマが実装する（サンプルプログラム BigCalc01.java 参照）必要がある．

プログラム：BigCalc01.java（平方根を求めるプログラムの例）

```
1  import java.math.BigDecimal;
2  import java.math.MathContext;
3  import java.math.RoundingMode;
4
5  // 計算用クラス
6  class BigMath {
7      public static BigDecimal sqrt( BigDecimal n, int prec ) {
8          //  準備
9          MathContext mcx = new MathContext(prec,RoundingMode.HALF_UP);
10         BigDecimal c2 = new BigDecimal(2);       // 定数：2
11         BigDecimal t1 = n.divide( c2, mcx );     // 初期設定1
```

```
12          BigDecimal t2 = n.divide( t1, mcx );     // 初期設定2
13          BigDecimal av = new BigDecimal("0",mcx);
14          // 計算
15          int m = (int)(3.5*Math.log10((double)prec)+2.0); // 計算回数の設定
16          int c;
17          for ( c = 0; c < m; c++ ) {
18              av = (t1.add(t2)).divide(c2,mcx);
19              t2 = n.divide(av,mcx);
20              t1 = av;
21          }
22          return( av );
23      }
24  }
25
26  // メイン
27  class BigCalc01 {
28      public static void main( String argv[] ) {
29          int prec = 65;       // 精度（桁数）の設定
30          MathContext mcx = new MathContext(prec,RoundingMode.HALF_UP);
31          BigDecimal n = new BigDecimal(2);
32          BigDecimal a = BigMath.sqrt(n,prec);
33          System.out.println("2の平方根："+a);
34          BigDecimal a2 = a.multiply(a,mcx);
35          System.out.println("上記の2乗："+a2);
36      }
37  }
```

このプログラムは平方根を求めるプログラムの例である．計算用のクラス BigMath を定義しており，クラスメソッドとして sqrt を実装している．これを実行した例を次に示す．

```
2の平方根：1.4142135623730950488016887242096980785696718753769480731766797380
上記の2乗：2.0000000000000000000000000000000000000000000000000000000000000000
```

多倍長精度の演算機能を提供するサードパーティのライブラリ Apfloat も存在している．これに関しては巻末付録「E.1 Apfloat」（P.215）で紹介する．

3.4 日付，時刻の扱い

3.4.1 Date，Calender クラス

日付と時刻の情報を保持するために Date クラスと Calendar クラスがある．前者は連続的な時間を表す単純なクラスであるのに対し，後者は暦に対応した “**年，月，日，時，分，秒**” といった各種のフィールドを取り扱う．

これらのクラスは java.util 配下のライブラリとして提供されている．

java.util.Date
java.util.Calendar

ここでは，サンプルプログラムを示しながら Date クラスと Calendar クラスについて説明する．

【サンプルプログラム】

ここで示すサンプルプログラム DateTest01.java は次のような処理を行うものである．

- 現在の時刻を取得して，書式整形して表示
- 指定した日付と時刻を書式整形して表示
- 時間の加算

プログラム：DateTest01.java

```
1  import java.util.Date;
2  import java.util.Calendar;
3  import java.text.SimpleDateFormat;
4
5  class DateTest01 {
6      public static void main( String argv[] ) {
7          //--- 現在の日付と時刻のデータ ---
8          Date d = new Date();
9
10         //--- 日付の書式データ ---
11         SimpleDateFormat f =
12             new SimpleDateFormat("GG yyyy年MM月dd日 HH:mm:ss.S (E) [z/Z]");
13
14         //--- 現在の日付・時刻の出力 ---
15         System.out.println("現在：    "+f.format(d));
16
17         //--- カレンダーに 2019/05/01 01:02:03.0 を設定 ---
18         Calendar cal = Calendar.getInstance();
19         int y = 2019,   m   = 5,    day = 1,
20             h   = 1,    min = 2,    sec = 3,    ms  = 0;
21         cal.set(Calendar.YEAR,  y);
22         cal.set(Calendar.MONTH, m-1);
23         cal.set(Calendar.DATE,  day);
24         cal.set(Calendar.HOUR_OF_DAY, h);
25         cal.set(Calendar.MINUTE, min);
26         cal.set(Calendar.SECOND, sec);
27         cal.set(Calendar.MILLISECOND, ms);
28         // 日付・時刻の書式で出力
29         Date d2 = cal.getTime();
30         System.out.println("令和開始："+f.format(d2));
31         // 5年後の日付
32         cal.add(Calendar.YEAR,5);
33         d2 = cal.getTime();
34         System.out.println("上記の5年後：  "+f.format(d2));
35     }
36 }
```

解説：

- **8 行目：** 現在の日付と時刻の取得

Date クラスのオブジェクトは新規に作成すると，その時点での日付と時刻の情報を保持したものが得られる．このオブジェクトの内容をわかりやすい形に書式整形するために SimpleDateFormat クラス（java.text.SimpleDateFormat）がある．このクラスのオブジェクトには

“紀元，年，月，日，曜日，時，分，秒，タイムゾーン”

といった内容を表現するための書式情報を設定し，それに基いて Date クラスのオブジェクトを文字列オブジェクトの形に変換するのに使用する．

SimpleDateFormat クラスは java.text 配下のライブラリとして提供されており，使用する場合は java.text.SimpleDateFormat をインポートする必要がある．

【時間情報の書式整形】

実行例：
```
SimpleDateFormat f =
        new SimpleDateFormat("GG yyyy 年 MM 月 dd 日 HH:mm:ss.S (E) [z/Z]");
Date d = new Date();
String dtext = f.format(d);
```

解説： format メソッドを用いて，f に設定された書式に従って日付と時刻のデータである d を整形し，でき上がった表現を String クラスの dtext に格納する．

※ format メソッドの戻り値は StringBuffer クラスの値である

整形のための書式を表 25 に示す．

表 25: 日付けのフォーマット

書式文字	意味	書式文字	意味
G	紀元	H	一日における時 (0 〜 23)
y	年	k	一日における時 (1 〜 24)
M	月	K	午前/午後の時 (0 〜 11)
w	年における週	h	午前/午後の時 (1 〜 12)
W	月における週	m	分
D	年における日	s	秒
d	月における日	S	ミリ秒
F	月における曜日	z	一般的なタイムゾーン（例．PST）
E	曜日	Z	RFC 822 タイムゾーン（例．-0800）

● **15 行目：** 書式整形された日付と時刻の表示

● **18 行目：** Calendar クラスのオブジェクトの生成

Calender クラスを用いると，日数や時間の設定，時間経過に関する計算などができる．このクラスのインスタンスを生成するには getInstance メソッド（クラスメソッド）を用いる．インスタンス生成後は，set メソッドを用いて日付や時刻の情報を設定することができる．(19〜27 行目)

【日付・時刻の設定】

実行例：
```
Calendar cal = Calendar.getInstance();
cal.set(Calendar.YEAR, 2019);
```

解説： Calendar クラスのインスタンス cal を生成し，それに対して “2019 年”を設定している．

set メソッドの第 1 引数には，日付や時刻を意味するフィールドを指定するが，これは表 26 の通りである．

表 26: Calendar の値のフィールド

記述	意　味	記述	意　味
Calendar.YEAR	年	Calendar.MINUTE	分
Calendar.MONTH	月	Calendar.SECOND	秒
Calendar.DATE	日	Calendar.MILLISECOND	ミリ秒
Calendar.HOUR_OF_DAY	時		

※注意：

Calendar クラスの「月」の値（Calendar.MONTH）の範囲は 0〜11 であり,「0」は「1 月」を表す．そのため，サンプルプログラムの 22 行目では ‘m-1’ のようにして調整している．

● **29 行目：** Calendar クラスのオブジェクトの値を Date クラスのオブジェクトに変換

Calendar クラスのオブジェクトに対して getTime メソッドを用いて，Date クラスのオブジェクトに変換することができる．

(参考)

getTime メソッドとは逆に setTime メソッドを用いると，Date クラスのオブジェクトの値を Calendar クラスのオブジェクトに変換することができる．

● **32 行目：** Calendar クラスのオブジェクトへの時間の加算

Calendar クラスのオブジェクトに対して add メソッドを実行することで時間を加算することができる．add の第 1 引数には加算する時間単位を指定する．これには表 26 に示すものを用いる．add の第 2 引数には加算する値を指定する．今回のサンプルプログラムでは，5 年後の日付と時刻を計算している．

サンプルプログラムを実行した例を次に示す．

```
現在：　西暦 2019 年 10 月 22 日 11:56:34.125（火）[JST/+0900]
令和開始：西暦 2019 年 05 月 01 日 01:02:03.0（水）[JST/+0900]
上記の 5 年後：　西暦 2024 年 05 月 01 日 01:02:03.0（水）[JST/+0900]
```

3.4.2 時間の計測

オブジェクト System には，現在時刻と 1970 年 1 月 1 日 (UTC) 深夜零時との時間差（システム時刻）を算出するメソッドが存在する．

【システム時刻の取得】

例 1. ミリ秒単位で取得　：　System.currentTimeMillis()

例 2. ナノ秒単位で取得　：　System.nanoTime()

解説： 現在時刻と 1970 年 1 月 1 日 (UTC) 深夜零時との時間差を long 型で返す．

これらメソッドを利用することで，プログラムの実行時間を計測することができる．実行時間を計測するサンプルプログラム DurationTest01.java を次に示す．

プログラム：DurationTest01.java

```
1  class DurationTest01 {
2      public static void main(String argv[]) {
3          // 時刻保持のための変数
4          long    m1, m2, n1, n2;
5          double x;       // 計算処理のための変数
6          // 開始時刻の測定
7          n1 = System.nanoTime();             // ナノ秒（開始）
8          m1 = System.currentTimeMillis();    // ミリ秒（開始）
9          // 計算の繰り返し（9,000,000回）
10         for ( int i = 0; i < 3000; i++ ) {
11             for ( int j = 0; j < 3000; j++ ) {
12                 x = Math.sqrt(2.0);
13                 x = Math.sin(3.14);
14                 x = Math.tan(3.14);
15                 x = Math.exp(1.0);
16                 x = Math.log(2.718);
17             }
18         }
19         // 終了時刻の測定
20         n2 = System.nanoTime();             // ナノ秒（終了）
21         m2 = System.currentTimeMillis();    // ミリ秒（終了）
22         // 結果表示
23         System.out.println("経過時間（ミリ秒）: "+(m2-m1));
24         System.out.println("経過時間（ナノ秒）: "+(n2-n1));
25     }
26 }
```

このプログラムは Math クラスの数学関数による計算を大量に実行（10〜18 行目）し，それに要した時間を計測するものである．このように開始時点と終了時点でのシステム時刻を取得し，それらの差を算出することでプログラムの実行時間を測定することができる．

このプログラムの実行結果の例を次に示す．

```
経過時間（ミリ秒）：  2051
経過時間（ナノ秒）：  2047467800
```

3.4.3 LocalDateTime，LocalDate，LocalTime クラス

Java SE 8 から，ISO 8601 に準拠した形で日付と時刻を扱う新たな API（LocalDateTime，LocalDate，LocalTime クラス）が導入された．新しい API では，和暦や東南アジア諸国の仏暦，中華民国（台湾）の民国暦，イスラム社会のヒジュラ暦などの扱いをサポートしている．また日付や時刻をそれぞれ単独で扱うクラスが新たに加わった．新しい API はスレッドセーフになり，マルチスレッドプログラミングにおいても安全なものになった．

【基本的なクラス】

- LocalDateTime　日付と時刻を保持するクラス（java.time.LocalDateTime）
- LocalDate　日付のみを保持するクラス（java.time.LocalDate）
- LocalTime　時刻のみを保持するクラス（java.time.LocalTime）

それぞれのクラスのインスタンスを生成するには，クラスメソッドの now を用いる．

例 1. LocalDateTime datetime = LocalDateTime.now();
例 2. LocalDateTime datetime = LocalDateTime.now(ZoneId.of("Europe/Paris"));

このような方法で生成されたインスタンスには now メソッド実行時点の日付（時刻）が設定されている．例 2 にある "Europe/Paris" は**タイムゾーン**（表 27 参照）であり，これを与えることで指定した地域における日付・時刻が取得できる．タイムゾーンは ZoneID クラス（java.time.ZoneId）の of メソッドにより生成する．

表 27: タイムゾーンの一覧

Africa/Addis_Ababa	Asia/Dhaka
Africa/Cairo	Asia/Ho_Chi_Minh
Africa/Harare	Asia/Karachi
America/Anchorage	Asia/Kolkata
America/Argentina/Buenos_Aires	Asia/Shanghai
America/Chicago	Asia/Tokyo
America/Indiana/Indianapolis	Asia/Yerevan
America/Los_Angeles	Australia/Darwin
America/Phoenix	Australia/Sydney
America/Puerto_Rico	Europe/Paris
America/Sao_Paulo	Pacific/Apia
America/St_Johns	Pacific/Auckland
	Pacific/Guadalcanal

タイムゾーン毎の現在時刻を表示するサンプルプログラム DateTest02.java を次に示す．

プログラム：DateTest02.java

```
1  import java.time.LocalDate;
2  import java.time.LocalTime;
3  import java.time.ZoneId;
4
5  class DateTest02 {
6      public static void main( String argv[] ) {
7          //---   タイムゾーン  ---
8          String[] tz = {
9              "Africa/Addis_Ababa",   "Africa/Cairo",
```

```
10              "Africa/Harare",        "America/Anchorage",
11              "America/Argentina/Buenos_Aires",
12              "America/Chicago",
13              "America/Indiana/Indianapolis",
14              "America/Los_Angeles",  "America/Phoenix",
15              "America/Puerto_Rico",  "America/Sao_Paulo",
16              "America/St_Johns",     "Asia/Dhaka",
17              "Asia/Ho_Chi_Minh",     "Asia/Karachi",
18              "Asia/Kolkata",         "Asia/Shanghai",
19              "Asia/Tokyo",           "Asia/Yerevan",
20              "Australia/Darwin",     "Australia/Sydney",
21              "Europe/Paris",         "Pacific/Apia",
22              "Pacific/Auckland",     "Pacific/Guadalcanal"
23          };
24
25          //---  現在の日付と時刻のデータ  ---
26          LocalDate[] d = new LocalDate[25];
27          LocalTime[] t = new LocalTime[25];
28          int c;
29          for ( c = 0; c < 25; c++ ) {
30              d[c] = LocalDate.now(ZoneId.of(tz[c]));
31              t[c] = LocalTime.now(ZoneId.of(tz[c]));
32          }
33
34          //---  出力  ---
35          for ( c = 0; c < 25; c++ ) {
36              System.out.println(d[c]+" : "+t[c]+" : "+tz[c]);
37          }
38      }
39  }
```

これを実行すると次のような結果となる．

```
2019-10-22 :  07:56:43.362 :  Africa/Addis_Ababa
2019-10-22 :  06:56:43.409 :  Africa/Cairo
2019-10-22 :  06:56:43.409 :  Africa/Harare
2019-10-21 :  20:56:43.431 :  America/Anchorage
2019-10-22 :  01:56:43.431 :  America/Argentina/Buenos_Aires
2019-10-21 :  23:56:43.431 :  America/Chicago
2019-10-22 :  00:56:43.431 :  America/Indiana/Indianapolis
2019-10-21 :  21:56:43.431 :  America/Los_Angeles
2019-10-21 :  21:56:43.431 :  America/Phoenix
2019-10-22 :  00:56:43.431 :  America/Puerto_Rico
2019-10-22 :  01:56:43.431 :  America/Sao_Paulo
2019-10-22 :  02:26:43.431 :  America/St_Johns
2019-10-22 :  10:56:43.431 :  Asia/Dhaka
2019-10-22 :  11:56:43.431 :  Asia/Ho_Chi_Minh
2019-10-22 :  09:56:43.431 :  Asia/Karachi
2019-10-22 :  10:26:43.431 :  Asia/Kolkata
2019-10-22 :  12:56:43.431 :  Asia/Shanghai
2019-10-22 :  13:56:43.431 :  Asia/Tokyo
2019-10-22 :  08:56:43.431 :  Asia/Yerevan
2019-10-22 :  14:26:43.431 :  Australia/Darwin
2019-10-22 :  15:56:43.431 :  Australia/Sydney
2019-10-22 :  06:56:43.431 :  Europe/Paris
2019-10-22 :  18:56:43.431 :  Pacific/Apia
2019-10-22 :  17:56:43.431 :  Pacific/Auckland
2019-10-22 :  15:56:43.431 :  Pacific/Guadalcanal
```

3.4.3.1 基本的な機能

基本的な機能について，サンプルプログラムを示しながら説明する．

プログラム：DateTest06.java

```
1  import java.time.LocalDateTime;
```

```
2  import java.time.LocalDate;
3  import java.time.LocalTime;
4  import java.time.DayOfWeek;
5
6  class DateTest06 {
7
8      //--- 曜日テーブル ---
9      static String[] JpWeek = {"日","月","火","水","木","金","土"};
10
11     //--- 日付と時刻の表示&うるう年の判定 ---
12     static void disp(LocalDateTime d) {
13         int year, month, day, dayY, dayWn, hour, min, sec, mill;
14         DayOfWeek dayW;
15
16         LocalDate date  = d.toLocalDate();  // 日付のみ取り出し
17         LocalTime tm    = d.toLocalTime();  // 時刻のみ取り出し
18
19         year    = d.getYear();
20         month   = d.getMonthValue();
21         day     = d.getDayOfMonth();
22         dayY    = d.getDayOfYear();
23         dayW    = d.getDayOfWeek();
24         dayWn   = dayW.getValue();
25         hour    = d.getHour();
26         min     = d.getMinute();
27         sec     = d.getSecond();
28         mill    = d.getNano() / 1000000;
29
30         System.out.print(year+"/"+month+"/"+day+
31             "(年通算"+dayY+"日目)"+
32             dayW+"("+JpWeek[dayWn%7]+")  ");
33         System.out.println("\t"+hour+":"+min+":"+sec+"."+mill+
34             " うるう年か?("+date.isLeapYear()+")");
35     }
36
37     //--- メイン ---
38     public static void main( String argv[] ) {
39
40         //ーー  現在の日付と時刻  ーー
41         LocalDateTime d = LocalDateTime.now();
42         System.out.print("d:  ");
43         disp(d);
44
45         //ーー  1年1ヶ月1日1時間1分1秒後の日付  ーー
46         LocalDateTime d2 = d.plusYears(1).plusMonths(1).plusDays(1).
47                            plusHours(1).plusMinutes(1).plusSeconds(1);
48         System.out.print("d2: ");
49         disp(d2);
50
51         //ーー  1年1ヶ月1日1時間1分1秒前の日付  ーー
52         LocalDateTime d1 = d.minusYears(1).minusMonths(1).minusDays(1).
53                            minusHours(1).minusMinutes(1).minusSeconds(1);
54         System.out.print("d1: ");
55         disp(d1);
56
57         //ーー  前後の検査  ーー
58         int c = d.compareTo(d1);
59         if ( c == 0 ) {
60             System.out.println("d == d1");
61         } else if ( c > 0 ) {
62             System.out.println("d > d1");
63         } else {
64             System.out.println("d < d1");
65         }
66
67         //ーー  日付と時刻の生成：一気に設定  ーー
68         LocalDateTime dtn = LocalDateTime.parse("1989-01-08T01:02:03");
69         System.out.print("dtn(1):");
70         disp(dtn);
71
```

```
72          //ーー　日付と時刻の生成：2段階に設定　ーー
73          LocalDate dn = LocalDate.parse("1989-01-08");
74          LocalTime tn = LocalTime.parse("01:02:03");
75          dtn = tn.atDate(dn);
76          System.out.print("dtn(2):");
77          disp(dtn);
78          // 逆も
79          dtn = dn.atTime(tn);
80          System.out.print("dtn(3):");
81          disp(dtn);
82      }
83  }
```

プログラム中で使用したメソッドについての簡単な説明を表 28 に挙げる．

表 28: 基本的なメソッド

メソッド	説明
toLocalDate	LocalDateTime → LocalDate の変換
toLocalTime	LocalDateTime → LocalTime の変換
getYear	「年」の取り出し
getMonthValue	「月」の取り出し
getDayOfMonth	月の内の「日数」の取り出し
getDayOfYear	年内の「日数」の取り出し
getDayOfWeek	週の内の「曜日」の取り出し．戻り値のクラスは DayOfWeek これに対して getValue メソッドを使用することで整数値を取り出すことができる．
getHour	「時」の取り出し
getMinute	「分」の取り出し
getSecond	「秒」の取り出し
getNano	「ナノ秒」の取り出し
isLeapYear	LocalDate オブジェクトがうるう年かどうかを判定する．（戻り値は boolean）

【比較と演算】

日付けと時刻に関する基本的な演算を表 29 に示す．

表 29: 基本的なメソッド

メソッド	説明	メソッド	説明
plusYears(整数値)	「1 年後」を算出	minusYears(整数値)	「1 年前」を算出
plusMonths(整数値)	「1ヶ月後」を算出	minusMonths(整数値)	「1ヶ月前」を算出
plusDays(整数値)	「1 日後」を算出	minusDays(整数値)	「1 日前」を算出
plusHours(整数値)	「1 時間後」を算出	minusHours(整数値)	「1 時間前」を算出
plusMinutes(整数値)	「1 分後」を算出	minusMinutes(整数値)	「1 分前」を算出
plusSeconds(整数値)	「1 秒後」を算出	minusSeconds(整数値)	「1 秒前」を算出
D1.compareTo(D2)	$D1 > D2$ のとき正の値 $D1 = D2$ のとき 0 $D1 < D2$ のとき負の値		

【日付けの値を設定する方法】

次に示すように parse メソッドで年月日，時分秒の値の設定ができる．

【年月日，時分秒の設定】

実行例： LocalDateTime dtn = LocalDateTime.parse("1989-01-08T01:02:03");

実行例： LocalDate dn = LocalDate.parse("1989-01-08");

実行例： LocalTime tn = LocalTime.parse("01:02:03");

【LocalDate,LocalTime から LocalDateTime を合成する方法】

LocalDate クラスのオブジェクト dn と LocalTime クラスのオブジェクト tn がある場合，これらのオブジェクトの値を LocalDateTime クラスのオブジェクト dtn に設定する下記のような方法がある．

例 1. dtn = tn.atDate(dn);

例 2. dtn = dn.atTime(tn);

このように，atDate, atTime メソッドを使用する．

このサンプルプログラムを実行すると次のような表示となる．

```
d:  2019/10/22(年通算 295 日目)TUESDAY(火) 14:15:32.244 うるう年か?(false)
d2:  2020/11/23(年通算 328 日目)MONDAY(月) 15:16:33.244 うるう年か?(true)
d1:  2018/9/21(年通算 264 日目)FRIDAY(金) 13:14:31.244 うるう年か?(false)
d > d1
dtn(1):1989/1/8(年通算 8 日目)SUNDAY(日) 1:2:3.0 うるう年か?(false)
dtn(2):1989/1/8(年通算 8 日目)SUNDAY(日) 1:2:3.0 うるう年か?(false)
dtn(3):1989/1/8(年通算 8 日目)SUNDAY(日) 1:2:3.0 うるう年か?(false)
```

3.4.3.2　和暦の扱い

和暦を扱うためのクラスとして JapaneseDate がある．JapaneseDate クラスのインスタンスは，クラスメソッド now で生成することができ，生成時の日付と時刻の値が設定される．

次に示すサンプルプログラムに基いて説明する．

ソースプログラム：　DateTest07.java

```
1  import java.time.LocalDateTime;
2  import java.time.LocalDate;
3  import java.time.chrono.JapaneseDate;
4  import java.time.temporal.ChronoField;
5
6  class DateTest07 {
7
8      //--- 年号テーブル ---
9      static String[] JpEra = {"明治","大正","昭和","平成","令和"};
10
11     //--- 和暦出力メソッド ---
12     static void jdisp(JapaneseDate dj) {
13         System.out.print(JpEra[dj.get(ChronoField.ERA)+1]+"(");
14         System.out.print(dj.getEra()+").");
15         System.out.println(dj.get(ChronoField.YEAR_OF_ERA));
16     }
17
18     //--- メイン ---
19     public static void main( String argv[] ) {
20
21         //--- 現在の日付と時刻を和暦で取得 ---
22         JapaneseDate dj = JapaneseDate.now();
23         System.out.println("直接取得：  "+dj);
24
25         //--- LocalDateから和暦に変換 ---
26         LocalDateTime datetime = LocalDateTime.now();
27         LocalDate date = datetime.toLocalDate();
28         dj = JapaneseDate.from(date);
29         System.out.println("和暦に変換："+dj);
30
31         //--- 和暦年号の扱い ---
32         int e;
33         //  （明治6年）
34         dj = JapaneseDate.from(LocalDate.parse("1873-01-25"));
35         jdisp(dj);
36         //  （大正元年）
37         dj = JapaneseDate.from(LocalDate.parse("1912-07-30"));
38         jdisp(dj);
```

```
39          //  （昭和元年）
40          dj = JapaneseDate.from(LocalDate.parse("1926-12-25"));
41          jdisp(dj);
42          //  （平成元年）
43          dj = JapaneseDate.from(LocalDate.parse("1989-01-08"));
44          jdisp(dj);
45          //  （令和元年）
46          dj = JapaneseDate.from(LocalDate.parse("2019-05-01"));
47          jdisp(dj);
48      }
49  }
```

サンプルプログラムにあるように，JapaneseDate クラスのクラス・メソッド from を使用して LocalDate クラスのオブジェクトから JapaneseDate クラスに値を変換することができる．

【LocalData クラスのオブジェクトから JapaneseDate への変換】
実行例： JapaneseDate dj = JapaneseDate.from(d);

注意： 和暦は明治 6 年（1873 年）以降しか扱えない．

JapaneseDate オブジェクトから各種のフィールド値を取り出すには get メソッドを使用する．

【JapaneseData クラスのオブジェクト dj から値を取得する】
実行例： int y = dj.get(ChronoField.YEAR_OF_ERA);
解説： 和暦日付 dj から和暦年を取り出す．

get メソッドで値を取り出す場合，ChronoField クラスの必要なメンバを与える（表 30）ことで，必要なフィールドを選択する．

表 30: フィールド値の指定のためのメンバ（一部）

フィールド名	意味
ERA	年号（戻り値：明治→-1, 大正→ 0, 昭和→ 1, 平成→ 2．令和→ 3）
YEAR_OF_ERA	年号内の紀元年
YEAR	西暦年

このサンプルプログラムを実行すると次のような表示になる．

```
直接取得：  Japanese Reiwa 1-10-23
和暦に変換：Japanese Reiwa 1-10-23
明治 (Meiji).6
大正 (Taisho).1
昭和 (Showa).1
平成 (Heisei).1
令和 (Reiwa).1
```

3.5 ファイル，パス，URI

ファイルシステム中のファイルやディレクトリは**パス**で識別され，その表記は OS 毎に異なる.（次の例参照）

OS	パスの表記の例	解説
Windows	C:¥a¥b¥c¥d¥e¥f	ディレクトリ C:¥a¥b¥c¥d¥e にあるファイル f
macOS, Linux	/a/b/c/d/e/f	ディレクトリ /a/b/c/d/e にあるファイル f

※使用する端末装置によっては「¥」はバックスラッシュ「\」として表示されることがある.

3.5.1 パスの作成，パスの分解

ファイル入出力において対象のファイルを開く際，文字列（String）の形式でパスを指定することもできるが，ここで説明する File クラス（java.io.File）や Path インターフェース（java.nio.file.Path）のオブジェクトとしてパスを取り扱うことができる．また，これらのライブラリを使用すると，パスの連結や要素への分解が簡便な形で実現できる.

サンプルプログラム File01.java に基づいて File，Path の使用方法を解説する.

プログラム：File01.java

```
1  import java.io.File;
2  import java.net.URI;
3  import java.nio.file.Path;
4  import java.util.Iterator;
5
6  class File01 {
7      public static void main( String argv[] ) {
8          // パスの連結(1)
9          File f1 = new File("C:","Users");
10         System.out.println("---[File]---");
11         System.out.println("f1："+f1.toString());
12         // パスの連結(2)
13         File f2 = new File(f1,"katsu");
14         System.out.println("f2："+f2.toString());
15         // URIへの変換
16         URI u = f2.toURI();
17         System.out.println("f2.toURI()："+u.toString());
18         // パスの連結(3)
19         File f3 = new File("C:");
20         String dn[] = {"a","b","c","d","e","f"};
21         for ( String s : dn ) {
22             f3 = new File( f3, s );
23         }
24         System.out.println("f3："+f3.toString());
25         // パスへの変換
26         Path p1 = f3.toPath();
27         System.out.println("---[Path]---");
28         System.out.println("p1.getFileName()："+p1.getFileName().toString());
29         System.out.println("p1.getParent()："+p1.getParent().toString());
30         System.out.println("p1.getRoot()："+p1.getRoot().toString());
31         System.out.println("p1.toUri()："+p1.toUri().toString());
32         // パスの要素の取り出し
33         System.out.print("パスの要素（Iteratorによる反復処理）：");
34         Iterator<Path> it = p1.iterator();
35         while ( it.hasNext() ) {
36             System.out.print(it.next()+",");
37         }
38         System.out.println();
39         System.out.print("パスの要素（拡張for文による反復処理）：");
40         for ( Path p : p1 ) {
41             System.out.print(p+",");
42         }
43     }
44 }
```

解説：

File クラスのコンストラクタの引数には String 形式のパス名を与える．

例 1： File f = new File("C:¥katsu")

例 2： File f = new File("C:","katsu")

コンストラクタにはパスを 2 つ与えることができ，その場合は 2 つを連結したパスを生成する．この方法を応用して上記プログラム中では File オブジェクト f1, f2, f3 を生成している．File オブジェクトは toString メソッドにより String 形式に変換することができる．(9〜24 行目)

File のコンストラクタの第 1 引数に与える文字列の先頭の表記によって**絶対パス**か**相対パス**かが決まる．すなわち，第 1 引数に与える文字列が 'C:' や '/' で始まる場合は絶対パスとして，それ以外の場合は相対パスとして File オブジェクトが生成される．

プログラムの 26 行目のように File オブジェクトに対して toPath メソッドを使用すると Path オブジェクトに変換することができる．Path オブジェクトには要素（親ディレクトリ，ルートディレクトリ，ファイル名など）に分解するための各種のメソッド（表 31）が使用できる．

表 31: Path に対するメソッド（一部）

メソッド	解説
getFileName()	パスの右端の要素を返す.
getParent()	パスの右端の要素を取り除いたものを返す.
getRoot()	パスのルートディレクトリを返す.
toFile()	File オブジェクトに変換する.（戻り値は File クラス）
toUri()	URI オブジェクトに変換する.（戻り値は URI クラス）

上記 3 つの戻り値は Path クラスのオブジェクト

プログラムの 28〜30 行目では表 31 にあるメソッドを用いて Path の要素を取り出している．Path オブジェクトの個々の要素を左端から順番に取り出すには Iterator を生成する方法（34〜37 行目）と，拡張 for 文を用いる方法（40〜42 行目）がある．

3.5.2 URI

プログラム File01.java の 16 行目では File オブジェクトを toURI メソッドによって URI オブジェクトに変換している．**URI** とは RFC3986 で規定された**統一資源識別子**（Uniform Resource Identifier）であり，ネットワークを含めた広い範囲でファイルなどの情報資源を識別するもの [23] である．URI の先頭は http:，ftp:，file: といった**スキーム**で始まる．

プログラム File01.java を実行した例を次に示す．

```
---[File]---
 f1：C:¥Users
 f2：C:¥Users¥katsu
 f2.toURI()：file:/C:/Users/katsu/
 f3：C:¥a¥b¥c¥d¥e¥f
---[Path]---
 p1.getFileName()：f
 p1.getParent()：C:¥a¥b¥c¥d¥e
 p1.getRoot()：C:¥
パスの要素（Iterator による反復処理）：a,b,c,d,e,f,
パスの要素（拡張 for 文による反復処理）：a,b,c,d,e,f,
```

[23] **URL** も URI に含まれる．

Path オブジェクトを URI オブジェクトに変換する toUri メソッド[24] もあり，プログラムの 31 行目でそれを実行している．URI オブジェクトに対して toString メソッドを実行すると String に変換することができる．

3.5.3 File クラスのメソッド

File クラスにも Path インターフェースと類似のメソッドがいくつか用意されている．サンプルプログラムを示しながらそれらについて説明する．

プログラム：File02.java

```
1  import java.io.File;
2  import java.io.IOException;
3
4  class File02 {
5      public static void main( String argv[] ) throws IOException {
6          System.out.println("---[パスに対する処理]---");
7          File f1 = new File("File02.java");      // 相対パスでファイル名を取得
8          System.out.println("相対パス："+f1.toString());
9          File f2 = f1.getAbsoluteFile();         // 絶対パス（Fileオブジェクト）
10         System.out.println("絶対パス1："+f2.toString());
11         File cd = f2.getParentFile();           // 親ディレクトリ
12         System.out.println("親ディレクトリ："+cd.toString());
13         System.out.println("↑上がこの場合のカレントディレクトリ");
14         String fn = f2.getName();   // ファイル名/ディレクトリ名（文字列）
15         System.out.println("ファイル名："+fn);
16         // 絶対パスを文字列として取得する方法
17         String ap = f1.getAbsolutePath();       // 絶対パス（Stringオブジェクト）
18         System.out.println("絶対パス2："+ap);
19         // ディレクトリの作成
20         System.out.print("---[ディレクトリ作成：");
21         File newd = new File(cd,"newdir");
22         if ( newd.mkdir() ) {
23             System.out.println(newd.toString()+"：作成成功]---");
24             // ファイルの作成
25             File newf;
26             String fname;
27             for ( int c = 0; c < 3; c++ ) {
28                 fname = String.format("f%03d",c);
29                 newf = new File(newd,fname);
30                 System.out.print("ファイル："+newf.toString());
31                 if ( newf.createNewFile() ) {
32                     System.out.println("：作成成功");
33                 } else {
34                     System.out.println("：作成失敗");
35                 }
36             }
37             // ファイル／ディレクトリの一覧を取得
38             System.out.println("---[ディレクトリ内のファイル一覧]---");
39             File flst[] = newd.listFiles();
40             for ( File f : flst ) {
41                 System.out.println(f.toString());
42             }
43         } else {
44             System.out.println(newd.toString()+"：作成失敗]---");
45         }
46         // ファイルの存在検査
47         System.out.print("ファイル f001 は");
48         if ( (new File(newd,"f001")).exists() ) {
49             System.out.println("存在します．");
50         } else {
51             System.out.println("存在しません．");
52         }
53         System.out.print("ファイル f005 は");
54         if ( (new File(newd,"f005")).exists() ) {
55             System.out.println("存在します．");
56         } else {
57             System.out.println("存在しません．");
```

[24]File オブジェクトに対する toURI と名前が似ているが少し異なることに注意．得られる URI 表記も若干異なる．

```
58            }
59        }
60    }
```

解説：

このプログラム 7〜17 行目までは与えられたパスを要素に分解する処理を示している．まず 7 行目で File オブジェクトを**相対パス**の形で生成しており，getAbsoluteFile メソッド（9 行目）で**絶対パス**の形に変換している．

パスの右端の要素とそれ以外（親ディレクトリ）を分解するために getName メソッド（14 行目），getParentFile メソッド（11 行目）を用いている．getAbsoluteFile に似たメソッドに getAbsolutePath メソッドがあり，これは絶対パスを文字列（String）の形で返す．

File クラスのメソッドの一部を表 32 にまとめる．

表 32: File オブジェクトに対するメソッド（一部）

メソッド	解説
getAbsoluteFile()	絶対パスの形の File オブジェクトにして返す．
getParentFile()	親ディレクトリを示す File オブジェクトを返す．
getName()	ファイル名／ディレクトリ名（パスの右端）を String として返す．
getAbsolutePath()	File オブジェクトの絶対パスを String として返す．
listFiles()	当該ディレクトリに属するファイル（サブディレクトリ）群を File の配列として返す．（対象の File オブジェクトがディレクトリである場合）
mkdir()	File オブジェクトが示すディレクトリを作成する．処理が成功すれば true を，失敗すれば false を返す．
createNewFile()	File オブジェクトが示すファイルを作成する．当該ファイルが既存の場合は処理が失敗して false を返す．作成が成功すれば true を返す．
exists()	File オブジェクトが示すファイル／ディレクトリが存在すれば true を，存在しなければ false を返す．
renameTo(dst)	File オブジェクトのパスを，別の File オブジェクト dst が示すパスに変更する．変更処理が成功すれば true を，失敗すれば false を返す．
delete()	File オブジェクト示すファイル／ディレクトリを削除する．削除が成功すれば true を，失敗すれば false を返す．
toURI()	URI オブジェクトに変換する．

3.5.3.1 ファイル，ディレクトリの作成

ディレクトリを作成するには，当該ディレクトリのパスを示す File オブジェクトに対して mkdir メソッド（22 行目）を実行する．このメソッドは処理が成功すると true を，失敗すると false を返す．このプログラムでは，カレントディレクトリに ‘`newdir`’ というディレクトリを作成している．

プログラム中 25〜36 行目では，作成したディレクトリの中に新たなファイルを作成している．ファイルを作成するには，そのファイルを示す File オブジェクトに対して createNewFile メソッドを実行（31 行目）する．このメソッドはファイルの作成が成功すれば true を，失敗すれば false を返す．このプログラムでは，ディレクトリ ‘`newdir`’ の中に 3 つのファイル

`f000, f001, f002`

を作成している．

3.5.3.2　ディレクトリ内容の取得，ファイルの存在検査

ディレクトリを示す File オブジェクトに対して listFiles メソッドを実行（39 行目）すると，当該ディレクトリに含まれるファイルやサブディレクトリの一覧が Files オブジェクトの配列として得られる．

ファイルやディレクトリが存在するかどうかを検査するには，当該パスの File オブジェクトに対して exists メソッドを実行（48,54 行目）する．

プログラム File02.java を実行した例を次に示す．

```
---[パスに対する処理]---
相対パス：File02.java
絶対パス 1：C:\Users\katsu\java\File02.java
親ディレクトリ：C:\Users\katsu\java
↑上がこの場合のカレントディレクトリ
ファイル名：File02.java
絶対パス 2：C:\Users\katsu\java\File02.java
---[ディレクトリ作成：C:\Users\katsu\java\newdir：作成成功]---
ファイル：C:\Users\katsu\java\newdir\f000：作成成功
ファイル：C:\Users\katsu\java\newdir\f001：作成成功
ファイル：C:\Users\katsu\java\newdir\f002：作成成功
---[ディレクトリ内のファイル一覧]---
 C:\Users\katsu\java\newdir\f000
 C:\Users\katsu\java\newdir\f001
 C:\Users\katsu\java\newdir\f002
ファイル f001 は存在します.
ファイル f005 は存在しません.
```

3.5.3.3　ファイル名の変更（パスの変更）

ファイルやディレクトリの名前（パス）の変更には renameTo メソッドを用いる．これに関してサンプルプログラム File03.java を示して説明する．

プログラム：File03.java

```
 1  import java.io.File;
 2
 3  class File03 {
 4      public static void main( String argv[] ) {
 5          File f1 = new File("File03.java");      // 相対パスでファイル名を取得
 6          File f2 = f1.getAbsoluteFile();         // 絶対パス（Fileオブジェクト）
 7          File cd = f2.getParentFile();   // 親ディレクトリ（カレントディレクトリ）
 8          // サブディレクトリ
 9          File newd = new File(cd,"newdir");
10          if ( newd.exists() ) {
11              // ファイル名の変更（移動）
12              System.out.println("---[ファイル名変更]---");
13              File src = new File(newd,"f001");
14              File dst = new File(newd,"g999");
15              System.out.println(src.toString()+"から");
16              System.out.print("  "+dst.toString()+"への変更：");
17              if ( src.renameTo(dst) ) {
18                  System.out.println("成功");
19              } else {
20                  System.out.println("失敗");
21              }
22              System.out.println("---[ファイル一覧]---");
23              File flst[] = newd.listFiles();
24              for ( File f : flst ) {
25                  System.out.println(f.toString());
26              }
27          }
28      }
29  }
```

このプログラムでは，変更前のファイルのパスを File クラスの src オブジェクトとして（13 行目），変更後のファイルのパスを File クラスの dst オブジェクトとして（14 行目）作成してる．その後 17 行目にあるように，

```
src.renameTo(dst)
```

としてパスの変更を実行している．この処理は成功の場合に true を，失敗の場合に false を返す．

プログラム File03.java を実行した例を次に示す．

```
---[ファイル名変更]---
C:\Users\katsu\java\newdir\f001 から
  C:\Users\katsu\java\newdir\g999 への変更：成功
---[ファイル一覧]---
C:\Users\katsu\java\newdir\f000
C:\Users\katsu\java\newdir\f002
C:\Users\katsu\java\newdir\g999
```

3.5.3.4　ファイル，ディレクトリの削除

ファイルやディレクトリを削除するには，削除対象のパスを示す File オブジェクトに対して delete メソッドを使用する．これに関してサンプルプログラム File04.java を示して説明する．

プログラム：File04.java

```
1  import java.io.File;
2
3  class File04 {
4      public static void main( String argv[] ) {
5          File f1 = new File("File04.java");      // 相対パスでファイル名を取得
6          File f2 = f1.getAbsoluteFile();         // 絶対パス（Fileオブジェクト）
7          File cd = f2.getParentFile();   // 親ディレクトリ（カレントディレクトリ）
8          // サブディレクトリ
9          File newd = new File(cd,"newdir");
10         if ( newd.exists() ) {
11             // ファイルをすべて削除
12             System.out.println("---["+newd.toString()+"の全ファイル削除]---");
13             File flst[] = newd.listFiles();
14             for ( File f : flst ) {
15                 if ( f.delete() ) {
16                     System.out.println(f.toString()+"：削除成功");
17                 } else {
18                     System.out.println(f.toString()+"：削除失敗");
19                 }
20             }
21             System.out.print("ディレクトリ"+newd.toString()+"の削除：");
22             if ( newd.delete() ) {
23                 System.out.println("成功");
24             } else {
25                 System.out.println("失敗");
26             }
27         }
28     }
29 }
```

delete メソッドは処理が成功すると true を，失敗すると false を返す．ディレクトリを削除する場合は，当該ディレクトリ配下の全てのファイルやサブディレクトリを削除しておく必要がある．13〜20 行目は，削除対象のディレクトリ ‘newdir’ を示す File オブジェクト ‘newd’ の内容を取得して，個々に削除する処理（15 行目）を行っている．‘newdir’ 以下の全てのファイルを削除した後，22 行目の delete メソッドでディレクトリ ‘newdir’ を削除している．

プログラム File04.java を実行した例を次に示す．

```
---[C:\Users\katsu\java\newdir の全ファイル削除]---
C:\Users\katsu\java\newdir\f000：削除成功
C:\Users\katsu\java\newdir\f002：削除成功
C:\Users\katsu\java\newdir\g999：削除成功
ディレクトリ C:\Users\katsu\java\newdir の削除：成功
```

【参考】

ファイル／ディレクトリに対する処理のために，java.io.File に代わる新しいライブラリとして java.nio 以下のライブラリが提供されており，新たにプログラムを作成する際はこちらのライブラリを用いることが推奨されている．先に説明した Path インターフェースも新しいライブラリに含まれるものである．java.io.File は後方互換性のために今後も存続するので，これに関しても理解しておく必要がある．

java.nio ライブラリに関しては巻末付録「C．java.nio ライブラリ」（p.208）で紹介する．

4　GUI と 2D グラフィックス

GUI（Graphical User Interface）はウィンドウ，アイコン，ボタンといった仮想的な操作対象のことを指す．今では GUI は広く普及しており，コンピュータの最も一般的なユーザインターフェースとなっている．この章では，Java 言語において広く利用されている API を用いて GUI アプリケーションを構築する方法について説明する．

4.1　GUI プログラミングの基本的な概念

現在のように GUI を備えたプログラムが一般的になる以前は，プログラムの実行途中にユーザの操作が介入することはあまり無く，予め用意したデータをまとめてプログラムに与えて実行を開始し，ユーザは処理が終わるまで待つのが基本であった．これは**バッチ処理**と呼ばれる情報処理の形態であり，GUI のスタイルが一般的になる [25] までの主流であった．図 9 のように，バッチ処理は必要なプログラム群を順次実行することで情報処理を実現する．バッチ処理を構成する各プログラムは「ステップ」と呼ばれ，1 つのバッチ処理は「ジョブ」と呼ばれる．

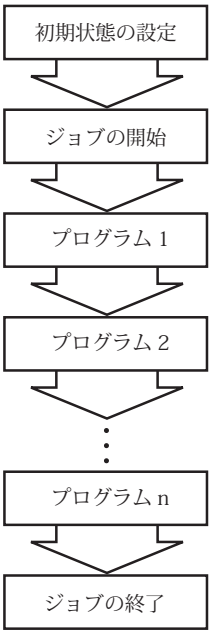


図 9: 「バッチ処理」のイメージ

コンピュータが小型化・高速化するに従い，ユーザからの指示を柔軟に受け付けて情報処理を行うスタイルが求められるようになってきた．実際，図形を高速に表示するグラフィックディスプレイ，ユーザからの入力と指示を即時に受け付ける入力デバイス（マウス，キーボードなど）の普及が，現在の GUI による情報処理を可能にした．

GUI のスタイルによる情報処理では，ユーザが与えた入力や指示を起点として情報処理を行い，得られた結果を即座にグラフィックディスプレイなどに応答する．これは，予めまとめて用意したデータを処理するバッチ処理とは異なるスタイルであり，ユーザの操作などに代表される**イベント**を起点とする情報処理（図 10）である．

このような，イベントを起点とする情報処理の実現を**イベント駆動型プログラミング**という．

4.2　GUI アプリケーション構築の概略

GUI はメニュー，ボタン，テキストフィールド，チェックボックスやラジオボタンといった各種の**コンポーネント** [26] から構成される．それらコンポーネントはウィンドウなどの**コンテナ**に配置されてユーザインターフェースとなる．

[25] 最初の GUI システムは 1970 年代前半に現れ，普及したのは 1980 年代中期以降である．

[26] Swing 以外の GUI ツールキットにおいては**ウィジェット**と呼ばれることが多い．本書でもコンポーネントと同義の言葉としてウィジェットという名称を使用することがある．

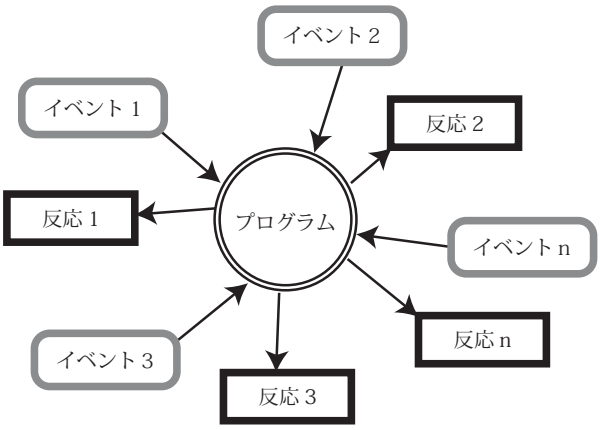


図 10: イベント駆動型システムのイメージ

GUI アプリケーションプログラムの構築は，概ね次の 2 つの作業からなる．

(1) コンテナへのコンポーネントの配置

(2) 配置したコンポーネントからイベントが発生したときの処理の記述

これらの作業を行うために，次に説明する「ツールキット」と呼ばれる API を使用する．

4.2.1 GUI ツールキット

GUI を備えたアプリケーションプログラムを Java で構築するための標準的な方法を 3 つ挙げる．

(1) Swing ツールキットを使用する方法

(2) SWT ツールキットを使用する方法

(3) JavaFX を使用する方法

上記の (1) と (2) は従来の Java（Java SE 7）で GUI アプリケーションプログラムを作成するための標準的な方法である．(3) は Java SE 8 から導入された比較的新しい方法であるが，本書では扱わない．

> ■ **JavaFX について**
>
> Java を配布する Oracle 社が，JDK11 から JavaFX を同梱しないこととしたが，現在では JavaFX は **OpenJFX** として開発が続けられている．

Swing，SWT 共に**ウィジェット**（GUI コンポーネントもしくはその集まり）を利用するためのツールキットである．GUI のためのツールキットとしては Swing，SWT 以外にも AWT と呼ばれるツールキットも存在する．AWT は Java の黎明期から存在しているがその機能は少なく，AWT の後継として開発された Swing が利用されることの方が多い．SWT は米 IBM 社により開発されたツールキットであり，Swing と並んでよく用いられる．

SWT は Swing に比べ，システムリソースの使用量や実行速度において効率が良くなることが多い [27] が，サードパーティ（IBM）が開発したものであり，Java 標準のツールキットではないことから，SWT を利用するには計算機環境に JRE 以外に必要なライブラリを導入しなければならない．また，SWT で提供されるウィジェットは，Java のオブジェクト指向の考え方には完全には沿っていない点に注意する必要がある．

Swing が提供するウィジェットは SWT のものよりも概して高機能であり，高度な GUI アプリケーションを実装する場合に有利なことが多い．また Swing は Java の標準のツールキットであることから，計算機環境に JRE を導入するだけで，Swing を利用したアプリケーションプログラムを実行することができる．

Java で実用的な GUI アプリケーションプログラムを開発するためには Swing，SWT などのライブラリを使用することになる．また，どちらも広く（Microsoft Windows，Apple macOS，Linux などの OS で）用いられているツールキットであり，プログラマは両方に通じておくことが望ましい．

[27] 最近では Java 仮想マシンの性能向上と計算機そのものの性能向上が相まって，Swing を用いたアプリケーションプログラムの実行時の性能は SWT に引けを取らないものになっている．

4.3 Swing

ここでは Swing ツールキットを利用して GUI アプリケーションプログラムを構築する方法について説明する．

4.3.1 GUI の構築

Swing では**コンテナ**に GUI コンポーネントを配置することで GUI を作成する．コンテナには**トップレベルコンテナ**と**中間コンテナ**がある．中間コンテナは，メニューやボタン，テキストフィールドなどの GUI コンポーネントを実際に貼り付けるための基盤と考えることができる．実際に OS のデスクトップに現れるウィンドウがトップレベルコンテナである．すなわち GUI 構築の手順は，

(1) 中間コンテナに GUI コンポーネントを貼り付ける

(2) 中間コンテナをトップレベルコンテナに追加する

という流れになる．図 11 はこの過程を概略的に示したものである．

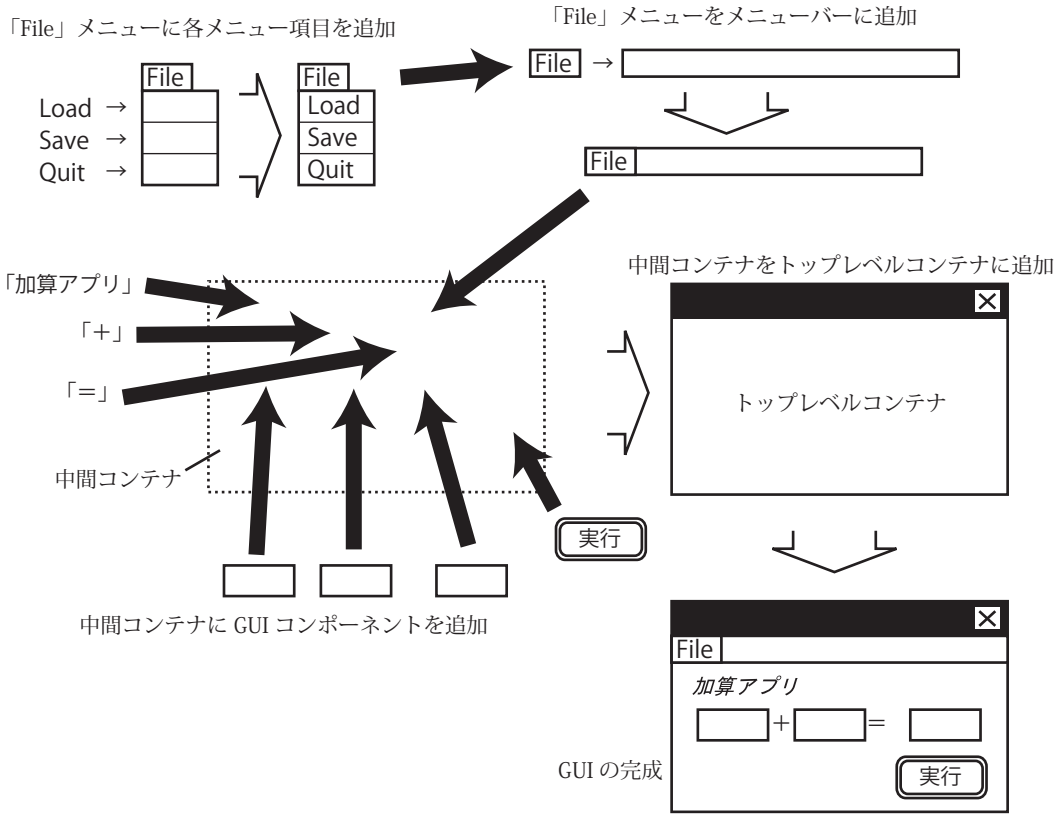


図 11: Swing を用いた GUI の構築（イメージ）

次に，具体的にプログラムの例を見ながら GUI アプリケーションプログラム構築の方法について説明する．

4.3.1.1 GUI 構築の例

図 12 のような，2 つの数の足し算を行う GUI アプリケーションプログラムについて考える．このプログラムは 3 つのテキストフィールドを持ち,「実行」ボタンをクリックすると，はじめの 2 つのフィールドに入力された数を加算して結果を 3 つ目のテキストフィールドに表示するものである．また「File」メニューを備えており，この中に終了の項目「Quit」（図 13）があり，これを選択するとプログラムが終了する．

■ ウィンドウの生成

OS に実際に現れるウィンドウを実現するための **JFrame** クラスを用いる．JFrame オブジェクトはトップレベルコンテナの 1 つである．JFrame オブジェクトに対して行う処理の内で基本的なものを次に挙げる．

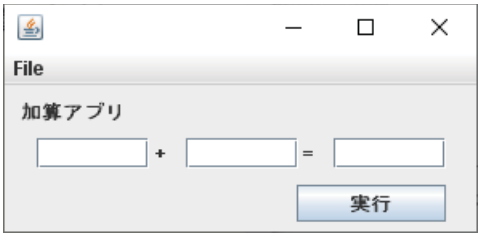


図 12: 足し算をする GUI アプリケーション

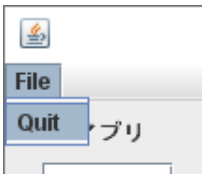


図 13: 「File」メニューの中に「Quit」を備える

(1) 位置と大きさの設定

メソッド： setBounds(横位置, 縦位置, 横幅, 高さ)

ウィンドウを表示するデスクトップ上の位置（横位置，縦位置）とウィンドウの大きさ（横幅，高さ）を指定する．OS のデスクトップ上でのウィンドウの位置とサイズの関係は図 14 のようになる．

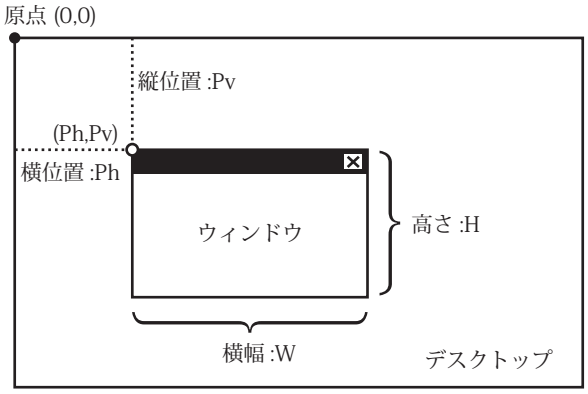


図 14: デスクトップ上でのウィンドウの位置とサイズ

位置とサイズの値は全て int 型の整数で与える．

(2) クローズボタンの動作の設定

メソッド： setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)

ウィンドウのクローズボタン☒がクリックされたら，ウィンドウを閉じてプログラムを終了する設定．

(3) 中間コンテナの取得

中間コンテナは実際に GUI のコンポーネントを配置するための土台となるオブジェクトである．JFrame オブジェクトを生成すると，その中に中間コンテナも生成される．中間コンテナは Container クラスのオブジェクトであり，getContentPane メソッドで取得する [28] ことができる．

JFrame オブジェクト fr を生成し，中間コンテナ con の取得，ウィンドウの位置とサイズの設定，クローズボタンの動作の設定を行う例を次に示す．

[28] 勿論，JPanel などの中間コンテナを新規に生成して JFrame に与えることも可能である．詳しくは後で説明する．

```
例.    JFrame fr = new JFrame();
       Container con = fr.getContentPane();
       fr.setBounds(100, 100, 310, 150);
       fr.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

ここで得られた con に対して GUI コンポーネントを配置してゆく．

■ レイアウトマネージャについて

Swing には**レイアウトマネージャ**という機構があり，GUI コンポーネントをコンテナに配置する際の位置を自動的に決定する．しかし GUI の設計においては，プログラマが自由にコンポーネントの配置を決定できる方が好ましい場合もある．レイアウトマネージャの介入を抑止するには，下記のようにコンテナオブジェクトのレイアウトに ‘null’ を設定すればよい．

例．コンテナ con のレイアウトマネージャを無効にする方法：

```
con.setLayout(null)
```

■ GUI コンポーネントの生成と配置

図 12 のインターフェースを構築するために必要な GUI コンポーネントの生成について説明する．

(1) メニューの構成

必要なオブジェクト：

(a) **メニューバー**　JMenuBar クラス

(b) **メニュー**　JMenu クラス
例．new JMenu(”File”) として生成すると，表示名が ‘File’ のメニューが生成される．

(c) **メニュー項目**　JMenuItem クラス
例．new JMenuItem(”Quit”) として生成すると，表示名が ‘Quit’ のメニュー項目が生成される．

メニュー項目をメニューに登録するには，次の例のように ‘add’ メソッドを呼び出す．

```
例.    JMenuBar mb = new JMenuBar();
       JMenu mFile = new JMenu("File");
       JMenuItem miQuit = new JMenuItem("Quit");
       mb.add(mFile);
       mFile.add(miQuit);
```

メニューの項目としては JMenuItem クラスのオブジェクトだけでなく，JMenu クラスのオブジェクトを追加することができ，階層的なメニューを構成することができる．

(2) ラベル　（コンテナへの文字の配置）

クラス： JLabel

new JLabel(”加算アプリ”) としてラベルを生成すると “加算アプリ”の文字列を持ったラベルが生成される．これを add メソッドを用いてコンテナに追加する．ラベルの配置は setBounds メソッドを用いる．この際，位置はコンテナに対して相対的に取る．すなわち，コンテナの左上を原点として位置を決定する．これら setBounds,add メソッドは他のコンポーネントに対しても同様に作用する．

(3) テキストフィールド　（文字列の入力）

クラス：　JTextField

キーボードから入力された文字列を保持するコンポーネントである．

(4) ボタン

クラス： JButton

マウスのクリックを受け付けるコンポーネントである．

ウィンドウの可視化について

JFrame オブジェクトは生成しただけでは OS のデスクトップに現れない．setVisible メソッドにより，可視属性を ‘true’ にする（下記）ことでウィンドウがデスクトップに表示される．

例．JFrame オブジェクト fr を表示する方法： fr.setVisible(true)

配置した GUI コンポーネントからイベントが発生した場合（メニュー選択やボタンのクリックなど）の動作の記述の方法について次に説明する．

4.3.1.2 イベント処理

イベント駆動型モデルでは，GUI コンポーネントからイベントが発生した際に，対応するメソッドを起動する．Swing では，イベントを発する GUI コンポーネントを**イベントソース**と呼ぶ．また，Swing には**イベントリスナ**と呼ばれる機構が備わっており，イベントソースから発生したイベントはイベントリスナに伝えられる．イベントリスナはイベントを受け取った後，適切なメソッドを呼び出す．

【イベントのクラスについて】

Swing では，イベントを種類ごとに**イベントクラス**として分類している．(表 33)

表 33: イベントクラス

イベントクラス	対応するリスナ
ActionEvent	ActionListener
AdjustmentEvent	AdjustmentListener
ItemEvent	ItemListener
KeyEvent	KeyListener
MouseEvent	MouseListener, MouseMotionListener
TextEvent	TextListener
ComponentEvent	ComponentListener
ContainerEvent	ContainerListener
FocusEvent	FocusListener
WindowEvent	WindowListener
ChangeEvent	ChangeListener

表 33 の中で特に重要なものについて説明する．

- **ActionEvent / ActionListener**
  - ボタン関係のコンポーネントがクリックされた場合
  - メニュー関係のコンポーネントが選択された場合
  - テキストフィールドなどの文字を入力するコンポーネントで Enter キーが押された場合
- **MouseEvent / MouseListener / MouseMotionListener**
  - マウスボタンがクリックされた場合
  - マウスが動かされた場合（MouseMotionListener）
- **WindowEvent / WindowListener**
  - ウィンドウを閉じたり最小化した場合など

● **ChangeEvent / ChangeListener**
・ボタンやスライダ，スピナの状態が変化した場合

● **KeyEvent / KeyListener**
・キーボードが押された場合

【イベント発生時に起動するメソッド】

イベントソースから発生したイベントに対して実際の動作を与えるメソッドを関連付けるには，それらをイベントリスナに登録する必要がある．例えば,「ファイルメニューの Quit が選択されたらプログラムを終了する」という機能を実現するには,

「メニュー項目 miQuit から ActionEvent が発生したらプログラムを終了する」

という設定を次のように行う．

```
miQuit.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.exit(0);
    }
} );
```

イベントソースであるメニュー項目 miQuit に対して addActionListener メソッドを作用させることで，イベントリスナにイベントとそれに対応する動作を登録することができる．イベントに対応する動作は ‘ActionListener(){…}’ を new により「生成」して addActionListener の引数として与える．この ‘…’ の部分に具体的なプログラム（メソッド）を記述する．これは actionPerformed メソッドとして記述する．

イベントソースからの ActionEvent をイベントリスナに登録する方法：

イベントソース.addActionListener(新規アクションリスナ)

新規アクションリスナ：

new ActionListener() { イベントに対応するメソッド }

イベントに対応するメソッド：

public void actionPerformed(ActionEvent e) { 動作 }

この「動作」の部分にプログラムを記述する．
ActionEvent クラスの引数 e は，発生するイベントに付随する情報であり，例えば，マウスの位置情報や，押されたキーに関する情報などがここに保持されている．

下記のプログラム SwingSample01.java は図 12 のアプリケーションプログラムを実現するものである．

プログラム：SwingSample01.java

```
1  import java.awt.*;
2  import java.awt.event.*;
3
4  import javax.swing.*;
5
6  public class SwingSample01 {
7      public static void main(String[] args) {
8          // GUIの構成要素
9          JFrame    fr     = new JFrame();
10         Container con;
11
12         JMenuBar  mb     = new JMenuBar();
13         JMenu     mFile  = new JMenu("File");
14         JMenuItem miQuit = new JMenuItem("Quit");
15
16         JLabel    lb1    = new JLabel("加算アプリ");
```

```
17          JLabel    lb2    = new JLabel("+");
18          JLabel    lb3    = new JLabel("=");
19
20          JButton   btn1   = new JButton("実行");
21
22          JTextField tf1 = new JTextField();
23          JTextField tf2 = new JTextField();
24          JTextField tf3 = new JTextField();
25
26          // 中間コンテナの取得
27          con = fr.getContentPane();
28
29          // ウィンドウのサイズとレイアウトの設定
30          fr.setBounds(100, 100, 310, 150);
31          fr.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
32          con.setLayout(null);
33
34          // メニューをメニューバーに追加
35          mb.add(mFile);
36
37          // メニュー項目をメニューに追加
38          mFile.add(miQuit);
39
40          // メニューバーのサイズの設定と中間コンテナへの追加
41          mb.setBounds(0, 0, 300, 20);
42          con.add(mb);
43
44          // ラベルのサイズの設定と中間コンテナへの追加
45          lb1.setBounds(10, 30, 80, 13);
46          con.add(lb1);
47
48          lb2.setBounds(94, 56, 19, 13);
49          con.add(lb2);
50
51          lb3.setBounds(186, 56, 19, 13);
52          con.add(lb3);
53
54          // テキストフィールドのサイズの設定と中間コンテナへの追加
55          tf1.setBounds(20, 53, 70, 19);
56          con.add(tf1);
57
58          tf2.setBounds(113, 53, 70, 19);
59          con.add(tf2);
60
61          tf3.setBounds(205, 53, 70, 19);
62          con.add(tf3);
63
64          // ボタンのサイズの設定と中間コンテナへの追加
65          btn1.setBounds(182, 82, 93, 23);
66          con.add(btn1);
67
68          // ボタンがクリックされた際の処理（イベントの登録）
69          btn1.addActionListener(new ActionListener() {
70              public void actionPerformed(ActionEvent arg0) {
71                  double x, y, z;
72                  x = Double.parseDouble(tf1.getText());
73                  y = Double.parseDouble(tf2.getText());
74                  z = x + y;
75                  tf3.setText(Double.toString(z));
76              }
77          });
78
79          // メニュー操作「File」→「Quit」の処理（イベントの登録）
80          miQuit.addActionListener(new ActionListener() {
81              public void actionPerformed(ActionEvent e) {
82                  System.exit(0);
83              }
84          });
85
86          // ウィンドウの可視化
```

```
87          fr.setVisible(true);
88      }
89  }
```

注意：

上記のプログラムは GUI アプリケーション構築について理解しやすく記述したものであり，正しく動作はするが，スレッド管理の観点からは安全な形にはなっていない．安全な実装に関しては後で説明する．

4.3.1.3　キーボードイベントからのキーの取り出し

キーボードの操作は KeyEvent であり，イベント e に対して getKeyChar メソッドを実行することにより押されたボタンのコードを取得することができる．例えば，テキストフィールド tf1 のキーボードイベントからキーボードのコードを取得するには次のようにする．

```
// キーボードイベントの登録
tf1.addKeyListener(new KeyAdapter() {
    public void keyTyped(KeyEvent e) {
        char c = e.getKeyChar();
        （c を用いた処理）
    }
} );
```

getKeyChar メソッドは char 型の値を返す．

4.3.1.4　JFrame クラスとしてのアプリケーションの構築

SwingSample01.java の例で説明した方法は，アプリケーションの主たるクラス SwingSample01 の中でウィンドウのクラス JFrame のオブジェクト fr を生成し，それに対して GUI コンポーネントの配置とイベント処理の登録を行うというものである．よく用いられる別の方法として，アプリケーションの主たるクラスそのものを JFrame クラスの拡張クラス（サブクラス）として定義するというものがある．

下記のプログラム SwingSample02.java では，アプリケーションの主たるクラス SwingSample02 のメンバとして GUI コンポーネントを定義し，このクラスのコンストラクタの中でウィンドウの構築とイベントの設定などを行なっている．

プログラム：SwingSample02.java

```
1   import java.awt.event.*;
2   import javax.swing.*;
3
4   public class SwingSample02 extends JFrame {
5       // テキストフィールド
6       JTextField tf1;
7       JTextField tf2;
8       JTextField tf3;
9
10      // メイン
11      public static void main(String[] args) {
12          // GUIの生成
13          SwingSample02 fr = new SwingSample02();
14          // ウィンドウの可視化
15          fr.setVisible(true);
16      }
17
18      // コンストラクタ
19      public SwingSample02() {
20          // 中間コンテナの生成とウィンドウへの追加，レイアウトの設定
21          JPanel con;
22          con = new JPanel();
23          setContentPane(con);
```

```
24          con.setLayout(null);
25
26          // ウィンドウのサイズ設定
27          setBounds(100, 100, 310, 150);
28          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
29
30          // メニューバーの生成とサイズの設定，中間コンテナへの追加
31          JMenuBar mb = new JMenuBar();
32          mb.setBounds(0, 0, 310, 20);
33          con.add(mb);
34
35          // メニューをメニューバーに追加
36          JMenu mFile = new JMenu("File");
37          mb.add(mFile);
38
39          // メニュー項目をメニューに追加
40          JMenuItem miQuit = new JMenuItem("Quit");
41          mFile.add(miQuit);
42
43          // ラベルの生成とサイズの設定，中間コンテナへの追加
44          JLabel lb1 = new JLabel("加算アプリ");
45          lb1.setBounds(10, 30, 80, 13);
46          con.add(lb1);
47
48          JLabel lb2 = new JLabel("+");
49          lb2.setBounds(94, 56, 19, 13);
50          con.add(lb2);
51
52          JLabel lb3 = new JLabel("=");
53          lb3.setBounds(186, 56, 19, 13);
54          con.add(lb3);
55
56          // テキストフィールドのサイズの設定と中間コンテナへの追加
57          tf1 = new JTextField();
58          tf1.setBounds(20, 53, 70, 19);
59          con.add(tf1);
60          tf1.setColumns(10);
61
62          tf2 = new JTextField();
63          tf2.setBounds(113, 53, 70, 19);
64          con.add(tf2);
65          tf2.setColumns(10);
66
67          tf3 = new JTextField();
68          tf3.setBounds(205, 53, 70, 19);
69          con.add(tf3);
70          tf3.setColumns(10);
71
72          // ボタンの生成とサイズの設定，中間コンテナへの追加
73          JButton btn1 = new JButton("実行");
74          btn1.setBounds(182, 82, 93, 23);
75          con.add(btn1);
76
77          // ボタンがクリックされた際の処理（イベントの登録）
78          btn1.addActionListener(new ActionListener() {
79              public void actionPerformed(ActionEvent e) {
80                  double x, y, z;
81                  x = Double.parseDouble(tf1.getText());
82                  y = Double.parseDouble(tf2.getText());
83                  z = x + y;
84                  tf3.setText(Double.toString(z));
85              }
86          });
87
88          // メニュー操作「File」→「Quit」の処理（イベントの登録）
89          miQuit.addActionListener(new ActionListener() {
90              public void actionPerformed(ActionEvent arg0) {
91                  System.exit(0);
92              }
93          });
```

```
    }
}
```

注意：

このプログラムも正しく動作するが，スレッド管理の観点からは安全な形にはなっていない．

スレッドとは Java が管理する処理の単位で，1 つのプログラムから別の（複数の）処理を起動する際に用いられる．GUI アプリケーションでは，複数の処理を同時に実行することがあり，基本的にはスレッドプログラミングのモデルを採用する．（スレッドプログラミングについては後述する）

4.3.1.5 より安全な実装

SwingSample02.java をスレッド管理の観点から安全なプログラムにするためには，
java.awt.EventQueue をインポートし，main メソッドを下記のように書き換える．

プログラムの書き換え部分：安全な main メソッド

```
    public static void main(String[] args) {
        // スレッドの生成
        EventQueue.invokeLater( new Runnable() {
            // スレッドの実行内容の記述
            public void run() {
                try {  // スレッドの試行
                    SwingSample02s fr = new SwingSample02s();
                    fr.setVisible(true);
                } catch (Exception e) {   // スレッドの実行が
                    e.printStackTrace();  // 失敗した場合の処理
                }
            }
        });
    }
```

4.3.1.6 レイアウトマネージャ

ここまでは中間コンテナ（JPanel のオブジェクト）にレイアウトマネージャを設定しない形（setLayout で null を設定）で GUI コンポーネントを配置する例を示してきたが，現実的な GUI アプリケーション開発においては**レイアウトマネージャ**を使用してユーザインターフェースを構築する．例えば図 15 に示すような BorderLayout を使用する場合を考える

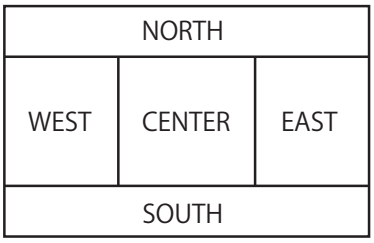


図 15: BorderLayout

BorderLayout は NORTH，SOUTH，WEST ，EAST，CENTER の 4 つの領域から成る．このレイアウトを用いるとコンテナ内の座標位置を余り意識せずに GUI コンポーネントを配置することができる．例えばメニューバーを NORTH に，フッターの記述を SOUTH に，その他の GUI コンポーネントを WEST，CENTER，EAST に add メソッドで配置するだけで GUI のデザインが完成する．

BorderLayout を利用して，コンテナ内の座標指定をせずにボタンを配置するサンプルプログラム SwingSample03_1.java を示す．

プログラム：SwingSample03_1.java

```
import java.awt.EventQueue;
import java.awt.BorderLayout;
```

```
 3 import javax.swing.JFrame;
 4 import javax.swing.JPanel;
 5 import javax.swing.JButton;
 6
 7 public class SwingSample03_1 extends JFrame {
 8     //--- メイン ---
 9     public static void main(String[] args) {
10         // スレッドの生成
11         EventQueue.invokeLater( new Runnable() {
12             // スレッドの実行内容の記述
13             public void run() {
14                 try {  // スレッドの試行
15                     SwingSample03_1 fr = new SwingSample03_1();
16                     fr.setVisible(true);
17                 } catch (Exception e) {   // スレッドの実行が
18                     e.printStackTrace();  // 失敗した場合の処理
19                 }
20             }
21         });
22     }
23     //--- コンストラクタ ---
24     public SwingSample03_1() {
25         setTitle("BorderLayout");
26         // ウィンドウのサイズ設定
27         setBounds(100, 100, 300, 150);
28         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);     // 閉じるボタン
29         // 中間コンテナ
30         JPanel con = new JPanel();
31         setContentPane(con);
32         con.setLayout( new BorderLayout() );
33         // ラベルを配置
34         JButton b1 = new JButton("NORTH");
35         con.add(b1,BorderLayout.NORTH);
36         JButton b2 = new JButton("SOUTH");
37         con.add(b2,BorderLayout.SOUTH);
38         JButton b3 = new JButton("WEST");
39         con.add(b3,BorderLayout.WEST);
40         JButton b4 = new JButton("EAST");
41         con.add(b4,BorderLayout.EAST);
42         JButton b5 = new JButton("CENTER");
43         con.add(b5,BorderLayout.CENTER);
44     }
45 }
```

このプログラムでは，中間コンテナとして JPanel オブジェクト con を生成し，それに対して setLayout メソッドを用いて BorderLayout オブジェクトを登録（32 行目）している．BorderLayout の各領域にそれぞれボタンオブジェクト b1 〜 b5 を add メソッドによって登録している．このとき add メソッドの第 2 引数に BorderLayout の各領域を意味する BorderLayout.NORTH，BorderLayout.SOUTH，BorderLayout.WEST，BorderLayout.CENTER，BorderLayout.EAST を指定している．

今回のプログラム SwingSample03_1.java では，座標の位置を指定せずに図 16 に示すような GUI アプリケーションを構成することができる．

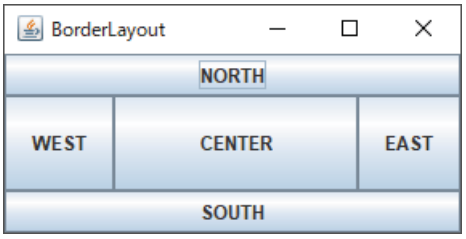


図 16: BorderLayout による GUI の作成

■ 各種のレイアウト

コンテナに使うことができるレイアウトは先の BorderLayout 以外にも多数あり（表 34），作成する GUI アプリ

ケーションのデザインに応じて使い分けることができる．

表 34: コンテナに使用できるレイアウト

レイアウト	特徴
BorderLayout	NORTH,SOUTH,WEST,CENTER,EAST の 5 つの領域にコンポーネントを配置する．
BoxLayout	縦もしくは横一列にコンポーネントを配置する．
GroupLayout	GUI コンポーネントをグループ毎にまとめて自動配置する．
CardLayout	複数の異なる GUI を個別に構築して切り替える場合に有用．
FlowLayout	GUI コンポーネントを横並びに順番に配列する．
GridLayout	GUI コンポーネントを縦横の升目に配列する．
GridBagLayout	GUI コンポーネントを縦横の升目に配列する．（升目の連結可）
OverlayLayout	GUI コンポーネントの重ね合わせ配置が可能
SpringLayout	GUI コンポーネント間隔の自動調整を行う．

レイアウトマネージャの詳細に関しては Oracle の公式インターネットサイトを参照のこと．

4.3.2 Swing でのグラフィックス作成の概要

Java にはグラフィックス作成のために AWT クラスがあり，基本的なグラフィックス作成機能を提供している．AWT クラスには **Graphics** クラスが含まれており，平面図形の描画に関する基本的な機能が利用できる．Swing ツールキットは AWT よりも高度なグラフィックス作成機能を提供しており，Graphics クラスよりも高度な平面図形描画クラス **Graphics2D** が利用できる．Graphics2D は Graphics の拡張であり，これら 2 つのクラスのメソッドを同時に使用することも可能である．

4.3.2.1 グラフィックス作成の流れ

Swing では平面図形は中間コンテナに対して描画する．従ってグラフィックスの描画は JFrame などのトップレベルコンテナが生成されたとき，あるいはそれらがアクティブになって表示されたときに行われる．Swing において GUI アプリケーションを作成する流れは，

(1) トップレベルコンテナの生成
(2) 中間コンテナの取得（あるいは生成）
(3) 中間コンテナへのコンポーネントの登録（GUI の構築）
(4) トップレベルコンテナの可視化

というものであるが，グラフィックスの作成は (3) の段階で行い，(4) の段階で実際に表示される．中間コンテナ JPanel に描画された図形は paintComponent メソッドで実際のウィンドウに描画される．このメソッドはウィンドウの再描画が必要となったタイミング [29] で自動的に実行される．従って，図形の描画を行うには，この paintComponent メソッドの中で行う必要がある．paintComponent の中にプログラマが独自の処理を記述するには，中間コンテナの拡張クラス（サブクラス）を定義して，その中でこのメソッドをオーバライドする．（次の例を参照）

JPanel の拡張クラス GPanel の定義と paintComponent メソッドのオーバーライド

```
class GPanel extends JPanel {
    public void paintComponent(Graphics g) {
            （図形描画の記述）
    }
}
```

この例では，中間コンテナ JPanel の拡張クラスとして GPanel（クラス名はプログラマが自由に決めることができる）を定義し，その中で paintComponent メソッドを記述（オーバーライド）している．

[29] 初めてウィンドウが表示されたときや，ウィンドウのサイズが変更されたとき，あるいはウィンドウを最小化した後に再表示したときなど

4.3.2.2 プログラムの例

ウィンドウに直線を 1 本だけ描画するプログラム G2Dsample01.java を次に挙げる.

プログラム：G2Dsample01.java

```
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

public class G2Dsample01 extends JFrame {
    // 描画用中間コンテナ
    GPanel contentPane;
    // メイン
    public static void main(String[] args) {
        // スレッドの生成
        EventQueue.invokeLater( new Runnable() {
            public void run() {
                try {
                    G2Dsample01 fr = new G2Dsample01();
                    fr.setVisible(true);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        });
    }
    // コンストラクタ
    public G2Dsample01() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setBounds(50, 150, 270, 120);
        setTitle("G2Dsample01");
        // 中間コンテナの生成
        contentPane = new GPanel();
        // 中間コンテナをトップレベルコンテナに登録
        setContentPane(contentPane);
        // レイアウトマネージャの無効化
        contentPane.setLayout(null);
    }
}

// 描画用コンテナのクラス定義
class GPanel extends JPanel {
    public void paintComponent(Graphics g) {
        // Graphics2Dに変換
        Graphics2D g2 = (Graphics2D)g;
        // 線の属性を設定する変数
        Stroke st;
        // 描画色と線の太さの設定
        g2.setPaint(Color.red);
        st = new BasicStroke(10.0f);
        g2.setStroke(st);
        // 太線
        g2.drawLine(10,15,240,70);
    }
}
```

このプログラムを実行すると図 17 のようなグラフィックスが表示される.

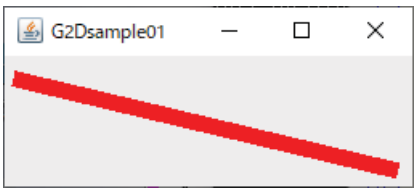


図 17: G2Dsample01.java の実行結果

4.3.2.3　paintComponent の引数について

paintComponent メソッド実行時には，引数として Graphics クラスのオブジェクトが渡される．これはグラフィックスの描画対象であり，各種の図形描画はこのオブジェクトに対して行われる．上のプログラムの例では Graphics2D の使用を前提としているので，描画開始時に与えられた g を，Graphics2D クラスのオブジェクト g2 に変換している．すなわち，各種描画メソッドはこの g2 に対して実行される．

具体的な描画メソッドに関しては後で説明する．

4.3.3　座標系について

描画に際しての座標系は，描画対象の中間コンテナに対する相対的なものであり，左上を原点とする．

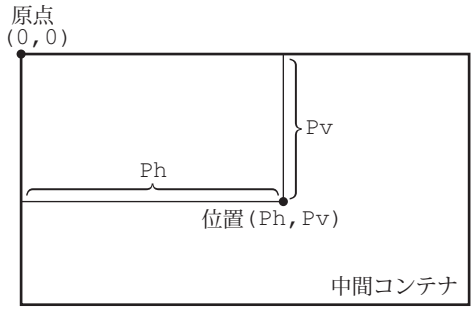


図 18: 描画のための座標系

4.3.4　グラフィックス作成のための基本的なメソッド

Graphics2D クラスのオブジェクト g2 に対する描画を例に挙げて各種図形の描画について説明する．

4.3.4.1　描画色の指定

描画に用いる色は Paint インターフェースで指定する．またこれを実際に描画に反映させるには setPaint メソッドを用いる．Paint インターフェースの実装の 1 つに描画色を設定するための Color クラス [30] がある．

使用例：　**g2.setPaint**(色指定)

‘色指定’ の部分に Color クラスのオブジェクトを与える．例えば青で描画を行う際は

g2.setPaint(Color.BLUE)

と記述する．色の指定に用いる値を表 35 に挙げる．

表 35: Color クラス：色の指定

設定値	色	設定値	色
Color.BLACK	黒	Color.MAGENTA	マゼンタ
Color.BLUE	青	Color.ORANGE	オレンジ
Color.CYAN	シアン	Color.PINK	ピンク
Color.DARK_GRAY	濃灰色	Color.RED	赤
Color.GRAY	灰色	Color.WHITE	白
Color.GREEN	緑	Color.YELLOW	黄色
Color.LIGHT_GRAY	淡灰色		

表 35 にない色を使う場合，光の 3 原色に基づいて色を合成する方法もある．この場合はコンストラクタに 3 原色の成分を与えて Color クラスのオブジェクトを生成し，それを setPaint の引数に与える．

[30] 他にもグラデーションを与える GradientPaint，LinearGradientPaint，RadialGradientPaint，テクスチャで塗りつぶすための TexturePaint といったクラスがある．

3 原色の指定で色を設定する例： g2.setPaint(new Color(r,g,b))
r,g,b にそれぞれ赤，緑，青の成分を 0〜255 の整数値で与える．

参考： setPaint では色の指定だけでなく，グラデーション（階調色）やテクスチャなども設定できる．

4.3.4.2 直線の描画： drawLine メソッド

使用例： g2.drawLine(Ph1, Pv1, Ph2, Pv2)

始点 (Ph1,Pv1) から終点 (Ph2,Pv2) にかけて直線を描く．（位置の指定は int 型整数）

4.3.4.3 折れ線の描画： drawPolyline メソッド

使用例： g2.drawPolyline(Pha, Pva, n)

Pha, Pva は int 型の配列で，折れ線の頂点の座標の列である．Pha には頂点の横位置の列，
Pva には頂点の縦位置の列をそれぞれ格納する．n はデータの個数を現す int 型の値である．

4.3.4.4 線の属性の指定

図形の描画において，線の太さや破線の状態などの属性は Stroke クラスのオブジェクトに格納される．また Stroke クラスのオブジェクトに値を設定するには BasicStroke クラスのオブジェクトを生成し，setStroke メソッドを用いて描画に反映させる．このときののコンストラクタは次の通り．

(1) **線の太さの指定** BasicStroke(Lw)
これは線幅のみを設定する場合のコンストラクタである．Lw は線幅を指定するための float 型の数である．

(2) **線の先端と曲がり角の形状の指定** BasicStroke(Lw, Cap, Join)
線幅に加え，先端形状と曲がり角の形状をそれぞれ Cap と Join で指定する場合のコンストラクタである．Cap と Join は int 型整数であり，図 20 と図 22 のような形状が選択できる．

(3) **破線の長さと間隔の指定** BasicStroke(Lw, Cap, Join, MiterLimit, Dash, DashPhase)

実際に折れ線を表示する例を見ながら線の属性について確認する．先のプログラム G2Dsample01.java の GPanel の部分を次のように書き換えて実行する．

プログラムの変更部分：

```
1  // 描画用コンテナのクラス定義
2  class GPanel extends JPanel {
3      public void paintComponent(Graphics g) {
4          // Graphics2Dに変換
5          Graphics2D g2 = (Graphics2D)g;
6          // 線の属性を設定する変数
7          Stroke st;
8          // 描画色と線の太さの設定
9          g2.setPaint(new Color(255,0,255) );
10         st = new BasicStroke(30.0f);
11         g2.setStroke(st);
12         // 折れ線
13         int x[] = {30,230,180,380,330};
14         int y[] = {125,50,125,50,125};
15         g2.drawPolyline(x,y,5);
16     }
17 }
```

上のように変更して実行した例を図 19 に示す．

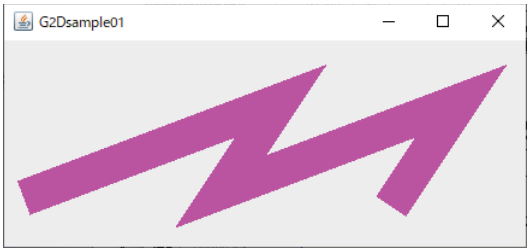


図 19: プログラム変更後の実行結果

この例では，色がマゼンタになり，太さが 30.0 になっている．更に，先端の形状を変更して試してみる．Stroke クラスの st を生成している部分を次のように変える．

1. 半円状の先端：(BasicStroke.CAP_ROUND)
st = new BasicStroke(30.0f, BasicStroke.CAP_ROUND, BasicStroke.JOIN_MITER);

2. 正方形の先端：(BasicStroke.CAP_SQUARE)
st = new BasicStroke(30.0f, BasicStroke.CAP_SQUARE, BasicStroke.JOIN_MITER);

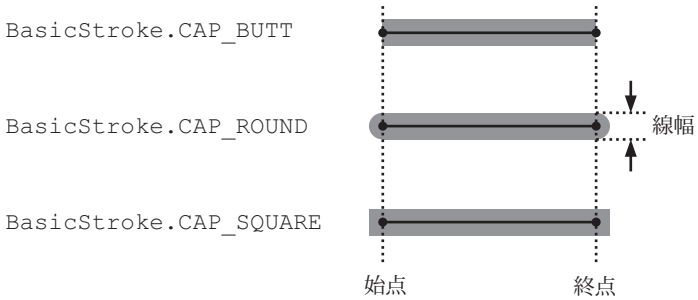


図 20: 線の先端の形状

それぞれの実行結果を図 21 に示す．

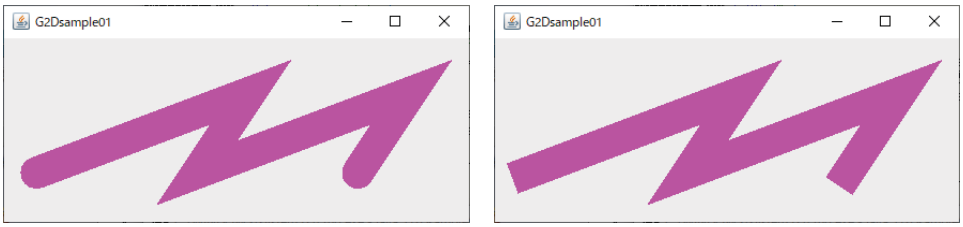


図 21: 先端の形状の指定
左が CAP_ROUND，右が CAP_SQUARE

多角形や折れ線を描画する場合，図 22 に示すような曲がり角の形状の設定ができる．

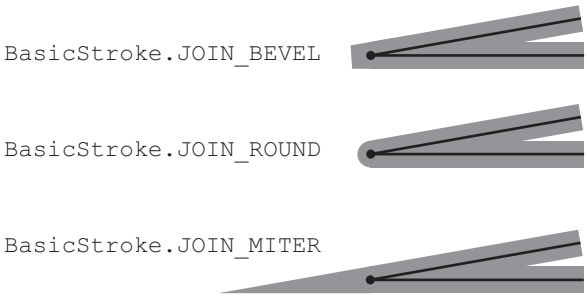


図 22: 折れ線の曲がり角の形状

これを利用して，曲がり角の形状を変更して試してみる．Stroke クラスの st を生成している部分を次のように変える．

1. 曲がり角の刈り取り：(BasicStroke.JOIN_BEVEL)

```
st = new BasicStroke(30.0f, BasicStroke.CAP_BUTT, BasicStroke.JOIN_BEVEL);
```

2. 丸い曲がり角：(BasicStroke.JOIN_ROUND)

```
st = new BasicStroke(30.0f, BasicStroke.CAP_BUTT, BasicStroke.JOIN_ROUND);
```

それぞれの実行結果を図 23 に示す．

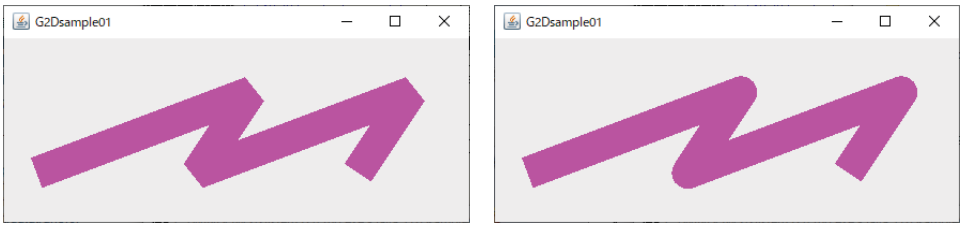


図 23: 曲がり角の形状の指定
左が CAP_BEVEL，右が CAP_ROUND

次に破線の設定について説明する．破線の設定は「線」と「間隔」の大きさを配列変数に格納して指定する．例えば 2 点鎖線で折れ線を描画するには先のプログラムで

```
float ds[] = {15.0f,5.0f,5.0f,5.0f,5.0f,5.0f};
```

のような配列 ds を定義し，更に Stroke クラスの st を生成している部分を次のように変える．

```
st = new BasicStroke(5.0f, BasicStroke.CAP_BUTT, BasicStroke.JOIN_MITER,10.0f,ds,0.0f);
```

このようにプログラムを変更して実行した結果を図 24 に示す．

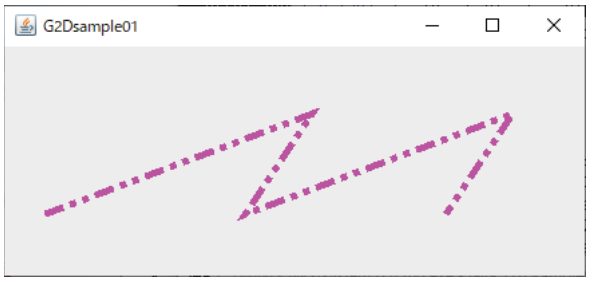


図 24: 破線の描画

■ MiterLimit と DashPhase について

折れ線の描画において曲がり角の角度が小さい場合，図 25 のように曲がり角の突出が大きくなる．

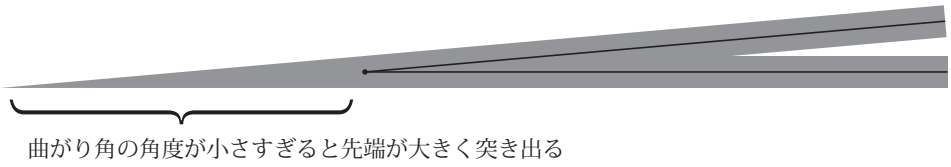


図 25: 曲がり角の突出

```
new BasicStroke( Lw, Cap, Join, MiterLimit, Dash, DashPhase )
```

として BasicStroke クラスのオブジェクトを生成する際，MiterLimit（float 型）の数値を設定することで曲がり角の突出を制限することができる．実際の突出が，線幅と MiterLimit の積より大きくなる場合は曲がり角が BasicStroke.JOIN_BEVEL の指定をしたのと同じ形状となる．

DashPhase（float 型）には破線が始まるオフセットを指定する．

4.3.4.5　長方形の描画 (1) 枠：　drawRect メソッド

使用例：　g2.drawRect(Ph, Pv, W, H)

コンテナ内の位置 (Ph,Pv) に横幅 W，高さ H の長方形枠を描く.（引数は int 型整数）

4.3.4.6　長方形の描画 (2) 塗りつぶし：　fillRect メソッド

使用例：　g2.fillRect(Ph, Pv, W, H)

コンテナ内の位置 (Ph,Pv) に横幅 W，高さ H の長方形（塗りつぶし）を描く.（引数は int 型整数）

4.3.4.7　楕円の描画 (1) 枠：　drawOval メソッド

使用例：　g2.drawOval(Ph, Pv, W, H)

コンテナ内の位置 (Ph,Pv) に横幅 W，高さ H の楕円形枠を描く.（引数は int 型整数）

4.3.4.8　楕円の描画 (2) 塗りつぶし：　fillOval メソッド

使用例：　g2.fillOval(Ph, Pv, W, H)

コンテナ内の位置 (Ph,Pv) に横幅 W，高さ H の楕円形（塗りつぶし）を描く.（引数は int 型整数）

4.3.4.9　多角形の描画 (1) 枠：　drawPolygon メソッド

使用例：　g2.drawPolygon(Pha, Pva, n)

Pha, Pva は int 型の配列で，多角形の頂点の座標の列である．Pha には頂点の横位置の列，
Pva には頂点の縦位置の列をそれぞれ格納する．n はデータの個数を現す int 型の値である．

4.3.4.10　多角形の描画 (2) 塗りつぶし：　fillPolygon メソッド

使用例：　g2.fillPolygon(Pha, Pva, n)

多角形の塗りつぶしを描画する．引数は drawPolygon と同様である．

4.3.4.11　文字の描画

文字の描画は drawString メソッドを用いて行う．

使用例：　g2.drawString(s, Ph, Pv)

コンテナ内の位置 (Ph,Pv) に文字列 s（String 型）を描画する．この場合の位置は
文字列のベースライン（図 26）の左端の位置である．

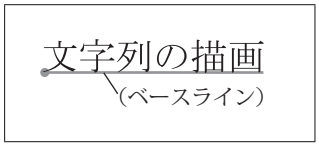


図 26: 文字列のベースライン

■ フォントの指定方法：

フォントの指定には Font クラスのオブジェクトを用いる．Font クラスのオブジェクトを生成する際，下記のようにコンストラクタにフォント名，スタイル，サイズを指定する．

コンストラクタ：　Font(フォント名, スタイル, サイズ)

フォント名は String 型でシステムで利用できるフォントの名前を指定する．スタイルは表 36 の中から選ぶ．サイズは int 型整数でポイント単位の値を与える．

フォント指定の例.

```
Font ft = new Font("Times New Roman",Font.ITALIC,18);
g2.setFont(font);
g2.drawString("Italic String",300,60);
```

表 36: フォントのスタイル（一部）

値	スタイル
Font.PLAIN	通常のスタイル
Font.BOLD	太字（ボールド）
Font.ITALIC	斜体（イタリック）

この例は”Times New Roman”フォント，スタイルは斜体，大きさは 18 ポイントを指定して”Italic String”を表示するものである．

■ **システムで利用できるフォントを調べる方法：**

GraphicsEnvironment クラスには，使用している環境のデバイスやフォントに関する情報が格納されている．クラスメソッド getLocalGraphicsEnvironment を呼び出すことで，それらの情報を取得することができる．

グラフィック環境に関する情報を取得する例.

```
GraphicsEnvironment genv = GraphicsEnvironment.getLocalGraphicsEnvironment();
```

これを実行することで，グラフィック環境の各種情報が genv オブジェクトとして取得できる．更にこのオブジェクトに対して次の例のように getAllFonts メソッドを用いることで使用可能なフォントのリストが得られる．

フォントのリストを取得する例.

```
Font font[] = genv.getAllFonts();
```

Font クラスがフォント情報を扱うためのクラスで，上のように実行することで，全てのフォントの情報が配列の形で font オブジェクトとして得られる．具体的なフォント名を取得するには，この配列の要素に対して getName メソッドを使用する．次のプログラム ShowFonts.java は，これらのメソッドを使ってフォント名の一覧を出力する例である．

プログラム：ShowFonts.java

```
1  import java.awt.Font;
2  import java.awt.GraphicsEnvironment;
3
4  public class ShowFonts {
5      public static void main( String[] argv ) {
6          // グラフィックス環境に関する情報の取得
7          GraphicsEnvironment genv =
8                  GraphicsEnvironment.getLocalGraphicsEnvironment();
9          // フォントリストの取得
10         Font font[] = genv.getAllFonts();
11         int c, l = font.length;
12         for ( c = 0; c < l; c++ ) {
13             System.out.println(c+": "+font[c].getName() );
14         }
15         System.out.println(l+" fonts available");
16     }
17 }
```

このプログラムを実行すると，標準出力にフォント名のリストが出力される．これをテキストファイルにリダイレクトすると良い.（次の例参照）

コマンド操作の例： java ShowFonts > fonts.txt

これによりフォント名のリストが ‘fonts.txt’ として作成される．

4.3.4.12　階調色の指定

Paint インターフェースと setPaint メソッドを用いて階調色（グラデーション）の設定ができる．先に Color クラスによる単純な色指定について説明したが，これと並んで提供される GradientPaint クラスを使用する．

GradientPaint クラスのオブジェクトは階調色を決定するもので，このクラスのオブジェクトを生成する際にコンストラクタで階調の始点と終点を指定する．(図 27)

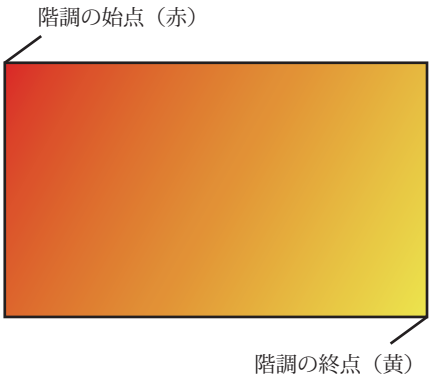


図 27: 階調の作成

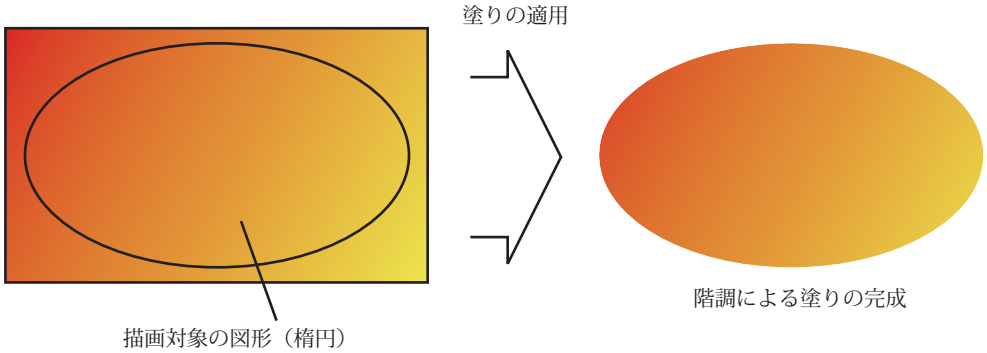


図 28: 階調の適用

例えば，階調の始点の位置が (10,20) で色は赤，終点の位置が (300,200) で色は黄にする場合，次のようにして階調オブジェクトを生成する．

```
GradientPaint gr;
gr = new GradientPaint( 10,20,Color.RED,300,200,Color.YELLOW );
```

この gr を setPaint メソッドにより描画に適用して図形を作成する（図 28）ことで階調による塗りが実現できる．

先のプログラム G2Dsample01.java の GPanel の部分を次のように書き換えて，階調による塗りを試みる．

プログラムの変更部分：　GPanel クラス

```
1  // 描画用コンテナ
2  class GPanel extends JPanel {
3      public void paintComponent(Graphics g) {
4          // Graphics2Dに変換
5          Graphics2D g2 = (Graphics2D)g;
6          // 階調の設定
7          GradientPaint gr = new GradientPaint( 10,20,Color.RED,
8                                                300,200,Color.YELLOW );
9          g2.setPaint(gr);
10         // 楕円
11         g2.fillOval(15,25,285,175);
12     }
13 }
```

プログラムの実行結果を図 29 に示す．

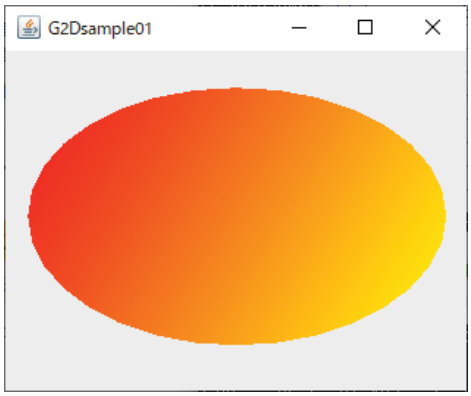


図 29: 階調による塗りの実行結果

4.3.5 BufferedImage の利用

実用的なレベルでグラフィックスを取り扱うには，主要なフォーマット [31] のビットマップ画像に対する各種の処理が求められる．ここでは画像データの入出力や加工，表示のための方法について説明する．

paintComponent メソッドによるコンテナへの描画は，ウィンドウの状態変更に伴って自動的に行われる．これはグラフィックスを常に正しく表示するために必要な処理であるが，生成に時間のかかる複雑なグラフィックスを表示するには適切な方法であるとはいえない．(ウィンドウの状態変更のたびに時間のかかる画像生成を毎回行うのは現実的ではない．)

BufferedImage は主記憶上に確保される仮想的な描画領域であり，生成に時間のかかる画像は一旦 BufferedImage オブジェクトに作成しておくことで，ウィンドウの再描画のたびに画像生成を繰り返す必要がなくなる．この方法を採用すると，paintComponent メソッド内では作成済みの BufferedImage オブジェクトをコンテナに表示するだけでよい．

4.3.5.1 BufferedImage に対する描画

BufferedImage オブジェクトを利用したグラフィクス作成の流れは次の通りである．

(1) **BufferedImage クラスのオブジェクトの生成**

描画領域の横幅と高さ，イメージタイプを指定して生成する．コンストラクタには

BufferedImage(横幅, 高さ, イメージタイプ);

のような引数を与える．例えば，横幅が 600，高さが 400 のカラーイメージを作成するには

BufferedImage(600, 450, BufferedImage.TYPE_INT_RGB);

として生成する．イメージタイプに指定できるものは多いが，特に使用頻度の高い代表的なものを表 37 に挙げる．

表 37: 主なイメージタイプ

値	説明
BufferedImage.TYPE_INT_RGB	RGB（赤・緑・青）各色 256 階調のカラーモデル
BufferedImage.TYPE_USHORT_GRAY	2 バイトグレースケール（65536 階調）
BufferedImage.TYPE_BYTE_GRAY	1 バイトグレースケール（256 階調）

(2) **上のオブジェクトに対する Graphics2D オブジェクトの生成**

描画処理は Graphics2D オブジェクトに対して行われるため，BufferedImage オブジェクトに対する Graphics2D オブジェクトを生成する必要がある．これには createGraphics メソッドを用いる．BufferedImage クラスのオ

[31] JPEG,GIF,BMP,PNG といった，頻繁に利用されるビットマップ画像フォーマット

ブジェクト bi から Graphics2D クラスのオブジェクト img を生成している例を次に示す．

```
bi = new BufferedImage(600, 450, BufferedImage.TYPE_INT_RGB);
Graphics2D img = bi.createGraphics();
```

(3) 上で生成した Graphics2D オブジェクトに対する描画

先に説明した各種の描画メソッドを用いてグラフィックスを描画する．

(4) paintComponent によるコンテナへの表示

paintComponent メソッド内で drawImage メソッドを用いて BufferedImage オブジェクトの内容を中間コンテナに表示する．drawImage メソッドは

drawImage(BufferedImage オブジェクト, 横位置, 縦位置, オブザーバ)

という形で引数を与えて呼び出す．例えば AppClass というクラス内で宣言されている BufferedImage オブジェクト bi を，中間コンテナの位置 (0,0) に表示する例を次に示す．（中間コンテナの Graphics2D オブジェクトは g2 である場合.）

```
g2.drawImage( AppClass.bi, 0,0, this );
```

オブザーバには通常 'this' を指定する．

4.3.5.2 BufferedImage オブジェクトを画像データとして出力する方法

ImageIO クラスの write メソッドを用いることで，BufferedImage オブジェクトを画像ファイルとして出力することができる．このメソッドは

ImageIO.write(BufferedImage オブジェクト, 出力フォーマット, File オブジェクト);

のように引数を与えて呼び出す．BufferedImage オブジェクト bi を BMP フォーマットでファイル 'out.bmp' に出力する例を次に示す．

```
ImageIO.write(bi, "bmp", new File("out.bmp"));
```

注意： このメソッドはファイルに対する入出力処理であるため，適切な例外処理をする必要がある．

現在 Java で扱える代表的なフォーマットを表 38 に示す．

表 38: 代表的な画像フォーマット

フォーマット	説明
"BMP" , "bmp"	未圧縮のビットマップ形式（劣化なしフルカラー）
"gif"	上限 256 色のビットマップ（読込み時のみ指定可能）
"png"	Portable Network Graphics フルカラー（劣化あり，圧縮形式）
"JPEG" , "JPG" , "jpeg" , "jpg"	Joint Photographic Experts Group フルカラー（劣化あり，圧縮形式）

グラフィックスを BufferedImage オブジェクトとして生成してウィンドウに表示し，更に画像ファイルとして出力するプログラムの例 BIsample01.java を次に示す．

プログラム：BIsample01.java

```
1  import java.io.*;
2  import java.awt.*;
3  import java.awt.geom.*;
4  import java.awt.image.*;
5  import javax.swing.*;
6  import javax.imageio.*;
7
8  public class BIsample01 extends JFrame {
9      GPanel contentPane;          // 描画用中間コンテナ
```

```
10      static BufferedImage bi;    // 描画対象
11      // メイン
12      public static void main(String[] args) {
13          // スレッドの生成
14          EventQueue.invokeLater( new Runnable() {
15              public void run() {
16                  try {
17                      BIsample01 fr = new BIsample01();
18                      fr.setVisible(true);
19                  } catch (Exception e) {
20                      e.printStackTrace();
21                  }
22              }
23          });
24      }
25      // コンストラクタ
26      public BIsample01() {
27          setTitle("BIsample01");
28          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
29          setBounds(10, 10, 600, 450);
30          // グラフィックスの作成
31          genGraphics();
32          // 中間コンテナの生成
33          contentPane = new GPanel();
34          setContentPane(contentPane);
35          contentPane.setLayout(null);
36          // グラフィックスをファイルに出力
37          try {
38              ImageIO.write(bi, "bmp", new File("BIsample01.bmp"));
39          } catch( Exception e ) {
40              e.printStackTrace();
41          }
42      }
43      // 描画
44      public void genGraphics() {
45          bi = new BufferedImage(600, 450, BufferedImage.TYPE_INT_RGB);
46          Graphics2D img = bi.createGraphics();
47          // 背景色の設定
48          img.setBackground(Color.WHITE);
49          img.clearRect(0,0,599,449);
50          // ストロークの設定
51          Stroke st = new BasicStroke(25);
52          img.setStroke(st);
53          // 画素の描画
54          int r, g, b, h, v;
55          for ( r = 0; r < 4; r++ ) {
56              for ( g = 0; g < 4; g++ ) {
57                  for ( b = 0; b < 4; b++ ) {
58                      img.setPaint(new Color(85*r,85*g,85*b));
59                      h = r*150;
60                      v = (g*4+b)*25+20;
61                      img.drawLine(h,v,h+125,v);
62                  }
63              }
64          }
65          // 文字の描画
66          img.setPaint(Color.WHITE);
67          Font font = new Font("ＭＳ ゴシック",Font.PLAIN,30);
68          img.setFont(font);
69          img.drawString("BufferedImageオブジェクトを用いた描画",10,50);
70          img.setPaint(Color.BLACK);
71          font = new Font("Times New Roman",Font.PLAIN,32);
72          img.setFont(font);
73          img.drawString("Plain",450,280);
74          font = new Font("Times New Roman",Font.BOLD,32);
75          img.setFont(font);
76          img.drawString("Bold",450,320);
77          font = new Font("Times New Roman",Font.ITALIC,32);
78          img.setFont(font);
79          img.drawString("Italic",450,360);
```

```
80          font = new Font("Times New Roman",Font.ITALIC | Font.BOLD,32);
81          img.setFont(font);
82          img.drawString("BoldItalic",450,400);
83          // グレージェント
84          st = new BasicStroke(20);
85          img.setStroke(st);
86          GradientPaint paint = new GradientPaint(10,100,Color.red,
87                                                  200,200,Color.yellow);
88          img.setPaint(paint);
89          img.drawOval(10,100,200,200);
90          // 破線
91          float ds[] = new float[]{0.0f,39.2f};
92          st = new BasicStroke(30,BasicStroke.CAP_ROUND,
93                               BasicStroke.JOIN_ROUND,0.0f,ds,0.0f);
94          img.setStroke(st);
95          paint = new GradientPaint(230,100,
96                  Color.red,430,300,Color.yellow);
97          img.setPaint(paint);
98          img.drawOval(230,100,200,200);
99      }
100 }
101
102 // 描画用コンテナ
103 class GPanel extends JPanel {
104     static int n = 0; // 描画回数
105     public void paintComponent(Graphics g) {
106         Graphics2D g2 = (Graphics2D)g;
107         int h, w;
108         h = getHeight();  // ウィンドウ横幅
109         w = getWidth();   // ウィンドウ高さ
110         n++;
111         System.out.println(n+"> paintComponent: width="+w+", height="+h);
112         // BI描画
113         g2.drawImage(BIsample01.bi,0,0,this);
114     }
115 }
```

このプログラムを実行すると図 30 のようなウィンドウが表示される．このグラフィックスは genGraphics メソッド内で 1 度だけ描画されており，paintComponent メソッド内で drawImage メソッドにより再描画される．更に，同じグラフィックスが画像ファイル ‘BIsample01.bmp’ に出力される．

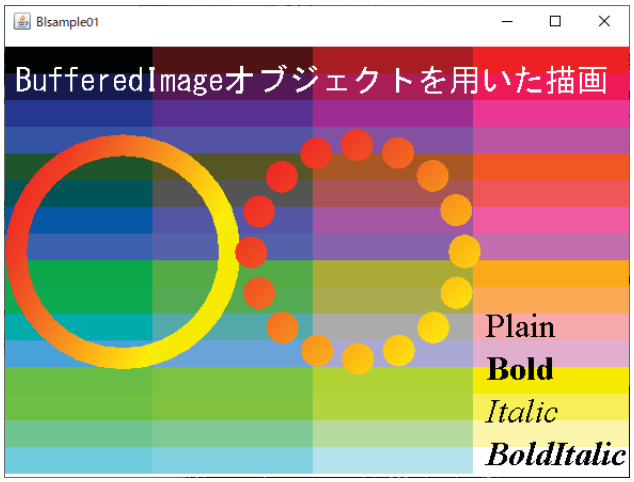


図 30: BufferedImage オブジェクトとして描画した例

4.3.5.3 画像ファイルの読み込みと表示

ImageIO クラスの read メソッドを用いることで，画像ファイルを読み込んで BufferedImage オブジェクトにすることができる．このメソッドは

ImageIO.read(File オブジェクト);

のように引数を与えて呼び出す．処理の結果，BufferedImage オブジェクトが返される．ファイル ‘Lenna.bmp’ [32] を読み込んで BufferedImage クラスのオブジェクト bi にセットする例を次に示す．

bi = ImageIO.read(new File("Lenna.bmp"));

注意： このメソッドはファイルに対する入出力処理であるため，適切な例外処理をする必要がある．

画像ファイルを読み込んでウィンドウに表示するプログラムの例 BIsample02.java を次に示す．

プログラム：BIsample02.java

```
1  import java.io.*;
2  import java.awt.*;
3  import java.awt.geom.*;
4  import java.awt.image.*;
5  import javax.swing.*;
6  import javax.imageio.*;
7
8  public class BIsample02 extends JFrame {
9      GPanel contentPane;          // 描画用中間コンテナ
10     static BufferedImage bi;     // 描画対象
11     // メイン
12     public static void main(String[] args) {
13         // スレッドの生成
14         EventQueue.invokeLater( new Runnable() {
15             public void run() {
16                 try {
17                     BIsample02 fr = new BIsample02();
18                     fr.setVisible(true);
19                 } catch (Exception e) {
20                     e.printStackTrace();
21                 }
22             }
23         });
24     }
25     // コンストラクタ
26     public BIsample02() {
27         // 画像の読み込み
28         try {
29             bi = ImageIO.read(new File("Lenna.bmp"));
30         } catch( Exception e ) {
31             e.printStackTrace();
32         }
33         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
34         setBounds(10, 10, 265, 285);
35         setTitle("BIsample02");
36         // 中間コンテナの生成
37         contentPane = new GPanel();
38         setContentPane(contentPane);
39         contentPane.setLayout(null);
40     }
41 }
42
43 // 描画用コンテナ
44 class GPanel extends JPanel {
45     static int n = 0; // 描画回数
46     public void paintComponent(Graphics g) {
47         Graphics2D g2 = (Graphics2D)g;
48         int h, w;
49         h = getHeight();  // ウィンドウ横幅
50         w = getWidth();   // ウィンドウ高さ
51         n++;
52         System.out.println(n+"> paintComponent: width="+w+", height="+h);
53         // BI描画
54         g2.drawImage(BIsample02.bi,0,0,this);
55     }
56 }
```

[32] 標準画像の 1 つ

このプログラムを実行した結果，図 31 のようなウィンドウが表示される．

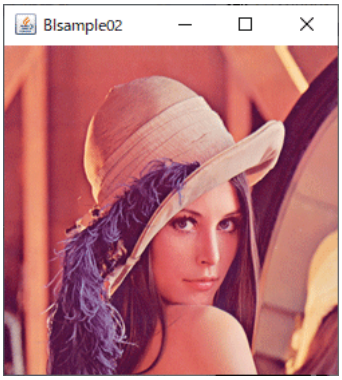


図 31: 画像を読み込んで表示した例

4.3.5.4　画素に対する操作

setRGB メソッドを使用することで，BufferedImage オブジェクトの画素を直接操作することができる．このメソッドは，下記のように指定した位置の画素に直接色を設定するものである．

setRGB(横位置, 縦位置, 色指定（RGB 値）);

引数は全て int 型で指定する．色指定の RGB 値は図 32 のように各色の値をシフトし，論理和による重ね合わせをして得る．

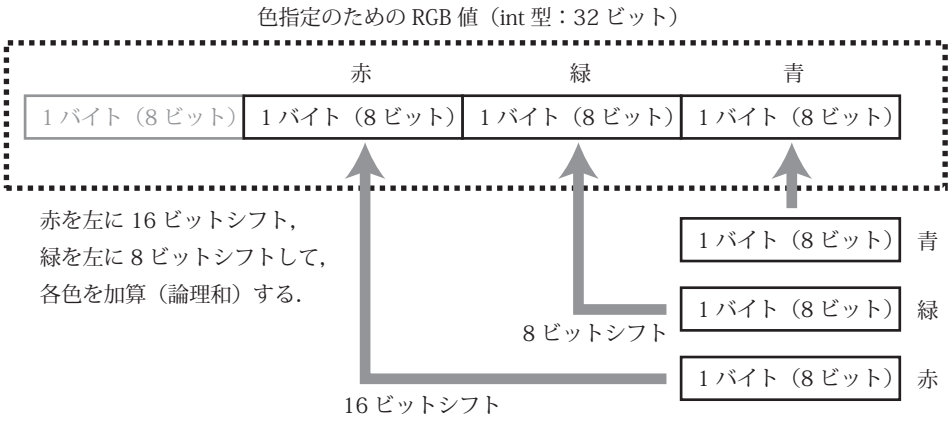


図 32: RGB 値の構造と合成の方法

直接画素を操作して画像を生成するプログラムの例 BIsample03.java を次に示す．

プログラム：BIsample03.java

```
import java.io.*;
import java.lang.Math;
import java.awt.*;
import java.awt.geom.*;
import java.awt.image.*;
import javax.swing.*;
import javax.imageio.*;

public class BIsample03 extends JFrame {
    GPanel contentPane;
    static BufferedImage bi;
    // メイン
    public static void main(String[] args) {
        // スレッドの生成
        EventQueue.invokeLater( new Runnable() {
            public void run() {
```

```
17                 try {
18                     BIsample03 fr = new BIsample03();
19                     fr.setVisible(true);
20                 } catch (Exception e) {
21                     e.printStackTrace();
22                 }
23             }
24         });
25     }
26     // コンストラクタ
27     public BIsample03() {
28         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
29         setBounds(10, 10, 600, 167);
30         setTitle("BIsample03");
31         // グラフィックスの作成
32         genGraphics();
33         // 中間コンテナの生成
34         contentPane = new GPanel();
35         setContentPane(contentPane);
36         contentPane.setLayout(null);
37         // グラフィックスをファイルに出力
38         try {
39             ImageIO.write(bi, "bmp", new File("BIsample03.bmp"));
40         } catch( Exception e ) {
41             e.printStackTrace();
42         }
43     }
44     // RGB値生成
45     public int vRGB( int r, int g, int b ) {
46         int ret = (r<<16) | (g<<8) | (b<<0);
47         return(ret);
48     }
49     // 描画
50     public void genGraphics() {
51         bi = new BufferedImage(600, 150, BufferedImage.TYPE_INT_RGB);
52         Graphics2D img = bi.createGraphics();
53         // 画素の設定
54         int c, r, g, b, h, v;
55         for ( r = 0; r < 8; r++ ) {
56          for ( g = 0; g < 8; g++ ) {
57           for ( b = 0; b < 8; b++ ) {
58            for ( h = 0; h < 600; h++ ) {
59              v = (int)(70.0 - 70.0*Math.sin((double)(h-300-64*r+8*g+b)/70.0));
60              c = vRGB(r*36+3, g*36+3, b*36+3);
61              bi.setRGB(h,v,c);
62            }
63           }
64          }
65         }
66     }
67 }
68
69 // 描画用コンテナ
70 class GPanel extends JPanel {
71     static int n = 0; // 描画回数
72     public void paintComponent(Graphics g) {
73         Graphics2D g2 = (Graphics2D)g;
74         int h, w;
75         h = getHeight();  // ウィンドウ横幅
76         w = getWidth();   // ウィンドウ高さ
77         n++;
78         System.out.println(n+"> paintComponent: width="+w+", height="+h);
79         // BI描画
80         g2.drawImage(BIsample03.bi,0,0,this);
81     }
82 }
```

このプログラムを実行した結果，図33のようなウィンドウが表示され，同じグラフィックスがファイルBIsample03.bmpにも保存される．

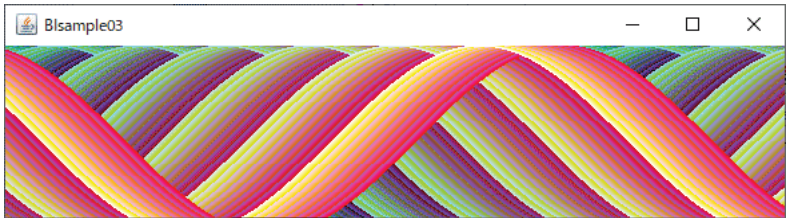


図 33: 画像を直接生成して表示した例

4.3.6 ファイル選択ダイアログ

実用的なアプリケーションプログラムでは，ファイルの読み込みや保存の際に，ファイルを選択するためのダイアログを表示してユーザに操作を促す．Swing にもファイル選択ダイアログを表示するための API が用意されている．ここでは，読込み用と保存用の 2 種類のファイル選択のダイアログについて説明する．

ファイル選択ダイアログは JFileChooser クラス（javax.swing.JFileChooser）のオブジェクトである．ファイルの読込み用にダイアログを表示する場合は，このクラスのオブジェクトに対して showOpenDialog メソッドを，ファイルの保存用にダイアログを表示する場合は showSaveDialog メソッドを実行する．

【読込み用ファイル選択ダイアログを表示する例】

```
JFileChooser fc = new JFileChooser();
int sl = fc.showOpenDialog(jframe);
```

上記 jframe は，読込み用ダイアログを表示するためのアプリケーションのウィンドウ（JFrame クラス）である．

このような方法で，既存のファイルを選択することができる．選択結果の戻り値は int 型の整数で得られる．戻り値の意味を表 39 に挙げる．

表 39: showOpenDialog メソッドの戻り値

値	意　　味
JFileChooser.APPROVE_OPTION	正常にファイル選択が完了した
JFileChooser.CANCEL_OPTION	ファイル選択をキャンセルした
JFileChooser.ERROR_OPTION	ファイル選択時にエラーが発生した

これら戻り値によって，ファイルの選択結果の状態が判断できる．

【保存用ファイル指定ダイアログを表示する例】

```
JFileChooser fc = new JFileChooser();
int sl = fc.showSaveDialog(jframe);
```

上記 jframe は，保存用ダイアログを表示するためのアプリケーションのウィンドウ（JFrame クラス）である．

このような方法で，保存先のファイルを指定することができる．処理結果の戻り値は int 型の整数で得られる．戻り値の意味は先の表 39 に示したものと同じである．

【選択したファイルの取り扱い】

正常にファイル選択が終了した場合，戻り値の JFileChooser オブジェクトに対して getSelectedFile メソッドを実行（次の例参照）すると，当該ファイルが File クラスのオブジェクトとして得られる．

例. JFileChooser クラスのインスタンス fc がある場合

File f = fc.getSelectedFile();

これで，選択したファイルが File クラス（java.io.File）のインスタンス f として得られる．

読込み用，保存用それぞれのダイアログを表示する（ファイルの読み込みと保存の処理は実際には行わない）サンプルプログラム FileDialogSwing1.java を示す．

プログラム：FileDialogSwing1.java

```
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class FileDialogSwing1 {
    public static void main(String[] args) {
        // GUIの構成要素
        JFrame  fr   = new JFrame();
        Container con = fr.getContentPane();    // 中間コンテナ
        // ボタン
        JButton   btn1   = new JButton("選択");
        JButton   btn2   = new JButton("保存");
        // ウィンドウのサイズとレイアウトの設定
        fr.setBounds(100, 100, 310, 150);
        fr.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        con.setLayout(null);
        // ボタンのサイズの設定と中間コンテナへの追加
        btn1.setBounds(50, 30, 90, 25);
        con.add(btn1);
        btn2.setBounds(150, 30, 90, 25);
        con.add(btn2);
        //---- ボタンがクリックされた際の処理（イベントの登録）----
        // 既存ファイルの選択
        btn1.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent arg0) {
                JFileChooser fc = new JFileChooser();
                int sl = fc.showOpenDialog(fr);
                if ( sl == JFileChooser.APPROVE_OPTION ) {
                    File f = fc.getSelectedFile();
                    String fn = f.getPath();
                    System.out.println("ファイル \""+fn+"\" が選択されました．");
                } else if ( sl == JFileChooser.CANCEL_OPTION ) {
                    System.out.println("ファイル選択がキャンセルされました．");
                } else if ( sl == JFileChooser.ERROR_OPTION ) {
                    System.out.println("ファイル選択時にエラーが発生しました．");
                }
            }
        });
        // 保存先ファイルの指定
        btn2.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent arg0) {
                JFileChooser fc = new JFileChooser();
                int sl = fc.showSaveDialog(fr);
                if ( sl == JFileChooser.APPROVE_OPTION ) {
                    File f = fc.getSelectedFile();
                    String fn = f.getPath();
                    System.out.println("ファイル \""+fn+"\" が指定されました．");
                } else if ( sl == JFileChooser.CANCEL_OPTION ) {
                    System.out.println("ファイル指定がキャンセルされました．");
                } else if ( sl == JFileChooser.ERROR_OPTION ) {
                    System.out.println("ファイル指定時にエラーが発生しました．");
                }
            }
        });
        // ウィンドウの可視化
        fr.setVisible(true);
    }
}
```

このプログラムを実行すると，図 34 のようなウィンドウが表示される．

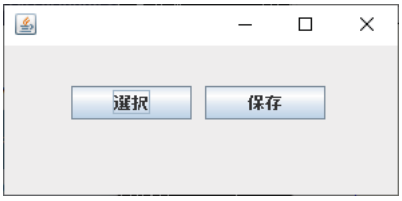


図 34: 実行したところ

ここで「選択」ボタンをクリックすると図 35 のようなダイアログが表示され，既存のファイルを選択することができる．また「保存」ボタンをクリックすると図 36 のようなダイアログが表示され，保存用にディレクトリとファイル名を指定することができる．

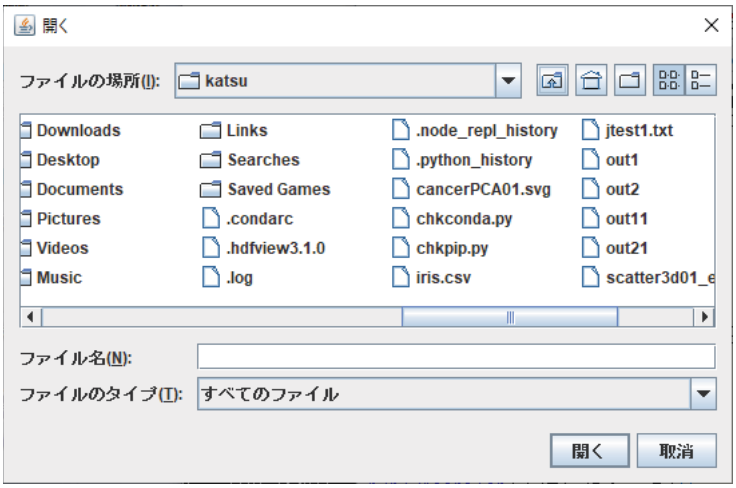


図 35: 読込むファイルを選択するダイアログ

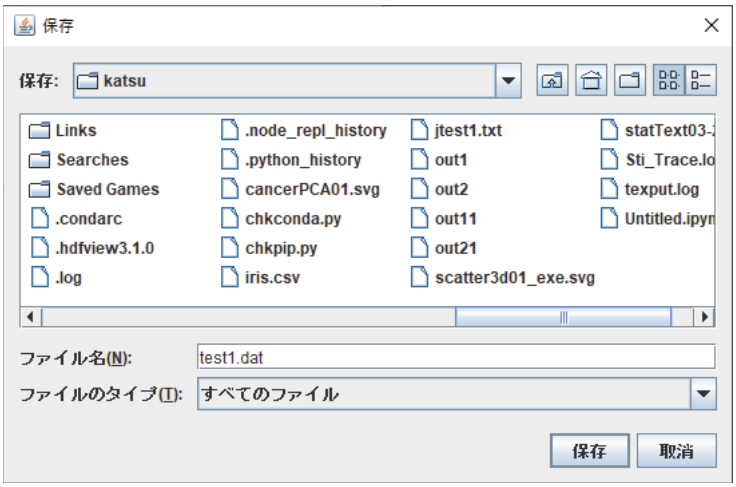


図 36: 保存するファイルを指定するダイアログ

4.4 SWT

SWT では，GUI アプリケーションプログラムの構築は **Display クラス**のオブジェクトの取得から始まる．Display クラスは，OS のウィンドウシステムと Java アプリケーションを仲介するものと見ることができ，SWT による GUI の実装は全てこのクラスのオブジェクトの下で管理される．

GUI アプリケーションの中核となるウィンドウは **Shell クラス**のオブジェクトとして生成する．このオブジェクトに対して GUI コンポーネントを登録し，イベント処理の登録をすることで GUI アプリケーションができあがる．

4.4.1 GUI アプリケーション構築の流れ

【1】Display オブジェクトの取得

getDefault メソッドで Display オブジェクトを取得する．

```
例.　Display disp = Display.getDefault();
```

これで，Display オブジェクト disp が取得でき，この後は disp の配下でアプリケーションを構築することになる．

【2】Shell オブジェクトの生成

GUI アプリケーションのためのウィンドウを生成する．

```
例.　Shell sh = new Shell();
```

これで，ウィンドウ（Shell オブジェクト）sh が生成される．ウィンドウのタイトルバーに表示する文字列を設定するには，setText メソッドを用いる．

```
例. ウィンドウ sh へのタイトルの設定　sh.setText("ウィンドウのタイトル");
```

この設定により，タイトルバーに「ウィンドウのタイトル」という文字列が表示される．

SWT にも Swing 同様にコンポーネントのレイアウトを自動化する機構が備わっている．レイアウトの設定には setLayout メソッドを用いる．

```
例. ウィンドウ sh へのレイアウトの設定　sh.setLayout(null);
```

この例ではレイアウトとして ‘null’ を設定しており，プログラマが自由に GUI コンポーネントを配置できる．

【3】GUI の構築とイベント処理の登録

生成した Shell オブジェクトに対して必要な GUI コンポーネントを生成して登録し，更にそれらコンポーネントに対してイベント処理を登録する．

【4】Shell オブジェクトの表示

open メソッドを用いて生成したウィンドウ（Shell オブジェクト）を表示する．

```
例.　sh.open();
```

これで，ウィンドウ（Shell オブジェクト）sh が表示される．

【5】イベントループの実行

GUI アプリケーションはイベントを待ち続け，イベントが発生した際に適切なメソッドを呼び出す．このサイクルを**イベントループ**（図 37）と呼ぶ．

Swing の場合はイベントループは見えない形で自動的に実行されるが，SWT ではプログラムとして明に記述する必要がある．イベントループを構成するための基本的な考え方は

「ウィンドウが表示されている間，イベント受信とメソッド起動を繰り返す」

というものである．ウィンドウが表示されているかどうかを調べるメソッドとして ‘isDisposed’ がある．

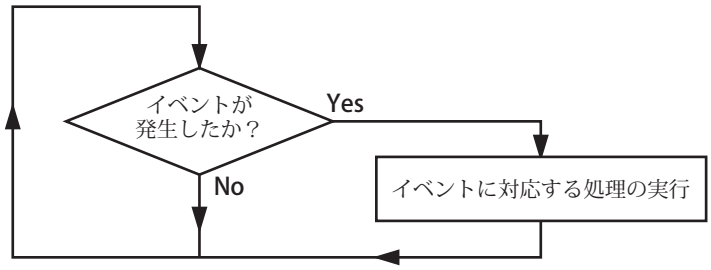


図 37: イベントループ

例. Shell クラスのオブジェクト sh が閉じられたことを調べる方法

```
sh.isDisposed()
```

isDisposed メソッドは boolean 値を返す．sh が閉じられていれば true を，表示されていれば false を返す．

イベント発生を受けて対応するメソッドを起動する処理を実現するには，readAndDispatch を使用する．

例. Display クラスのオブジェクト disp にイベントが発生したことを受けて，対応するメソッドを起動する方法

```
disp.readAndDispatch()
```

readAndDispatch メソッドは boolean 値を返す．イベントが発生していれば true を，発生していなければ false を返す．

Display クラスのオブジェクトが disp で，ウィンドウオブジェクト（Shell クラスのオブジェクト）sh が存在する状況でイベントループを構成する例を次に示す．

```
while ( !sh.isDisposed() ) {
    if ( !disp.readAndDispatch() ) {
        disp.sleep();
    }
}
```

イベントループの実行において，イベントが何も発生していない場合は無意味な繰り返しが発生する．(図 37 の左側のループ） CPU に無意味な負荷をかけることを避けるために，sleep メソッドでプログラムの実行を一時停止することができる．

例. Display クラスのオブジェクト disp でのプログラムの実行を一時停止する方法

```
disp.sleep()
```

これにより，次回のイベントが発生するまでプログラムは一時停止する．

4.4.2 GUI コンポーネントの生成と配置

GUI コンポーネントはウィンドウ（Shell クラスのオブジェクト）に対して配置する．Shell クラスのオブジェクト sh がある場合を例に挙げて，各種 GUI コンポーネントを配置する方法について説明する．

【1】メニューの構築

メニューは Menu クラスのオブジェクトとして生成する．

● メニューバーの設置

例.

```
Menu mb = new Menu(sh,SWT.BAR);
sh.setMenuBar(mb);
```

これにより，メニューバーオブジェクト mb を生成してウィンドウ sh に設置することができる．Menu のコンストラクタの第 1 引数には追加対象のウィンドウを指定する．第 2 引数の SWT.BAR はメニューバーを生成することを意味する．

● **メニューへの項目の追加**

メニューに追加する項目としては，

(1) サブメニュー（更に下部構造を持つメニュー）

(2) 階層を持たない項目

の 2 種類がある．メニュー項目は MenuItem クラスのオブジェクトとして生成する．MenuItem 生成時のコンストラクタの第 1 引数には項目を追加する対象のメニューを指定する．第 2 引数には追加する項目の種別などを指定する．

メニュー項目 miFile を生成し，メニューバー mb に「File」の項目として追加する例を次に示す．

```
例.  MenuItem miFile = new MenuItem(mb, SWT.CASCADE);
     miFile.setText("File");
```

SWT.CASCADE を指定することで，この項目が更に下部構造を持つメニュー（カスケードメニュー）となる．miFile に対して setText メソッドを用いて “File” という表示名を設定している．

この時点では，項目 miFile はまだ下部構造をもたない．この項目の配下に項目を格納するために，下記のように更に Menu クラスのオブジェクト mFile を生成して miFile に接続する．

```
例.  Menu mFile = new Menu(miFile);
     miFile.setMenu(mFile);
```

Menu のコンストラクタの引数に miFile が指定されているが，これは接続先のオブジェクトを意味する．実際の接続は 2 行目のように setMenu メソッドを用いて行う．

下部構造を持たないメニュー項目の追加の場合は MenuItem オブジェクトを生成する際のコンストラクタの第 2 引数に SWT.NONE を指定する．例えば今回の mFile メニューにメニュー項目 “Quit” を追加するには次のように記述する．

```
例.  MenuItem miQuit = new MenuItem(mFile, SWT.NONE);
     miQuit.setText("Quit");
```

【2】テキストフィールドの設置

テキストフィールドは Text クラスのオブジェクトとして生成する．

```
例.  Text tf1 = new Text(sh1, SWT.BORDER);
```

この例では，Text クラスのオブジェクト tf1 を生成している．コンストラクタの第 1 引数には設置先のウィンドウ，第 2 引数には属性を指定する．SWT.BORDER を指定すると，枠のあるテキストフィールドとなる．オブジェクトのサイズと設置場所の設定は，Swing の場合と同様に setBounds メソッドを使用する．

【3】ラベルの設置

ラベルは Label クラスのオブジェクトとして生成する．

```
例.  Label lb1 = new Label(sh1, SWT.NONE);
     lb1.setBounds(10, 10, 53, 12);
     lb1.setText("身長 (cm)");
```

この例では，Label クラスのオブジェクト lb1 を生成している．コンストラクタの第 1 引数には設置先のウィンドウ，第 2 引数には属性を指定する．SWT.NONE を指定すると，枠のないラベルとなる．オブジェクト内の表示を設定するには，setText メソッドを使用する．

【4】ボタンの設置

ボタンは Button クラスのオブジェクトとして生成する．

```
例.    Button btn1 = new Button(sh1, SWT.NONE);
       btn1.setBounds(166, 49, 72, 22);
       btn1.setText("計算");
```

この例では，Button クラスのオブジェクト btn1 を生成している．コンストラクタの第 1 引数には設置先のウィンドウ，第 2 引数には属性を指定する．

4.4.3 イベント処理の登録

SWT では，イベントを発生するオブジェクトを**コントロール**と呼ぶ．これは Swing におけるイベントソースに対応する概念であり，コントロール毎に異なる種類のイベントが起こる．発生したイベントはイベントリスナによって受信され，適切な**イベントアダプタ**に渡され対応するメソッドが起動される．イベントの種類により，それを受け取るイベントリスナの種類も異なる．

【例】メニュー項目の選択

メニュー項目を選択すると，そのメニュー項目のオブジェクトから SelectionEvent が発生する．先の例で挙げた miQuit オブジェクトが選択された場合の処理の登録を例示する．

```
miQuit.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(SelectionEvent arg0) {
        System.exit(0);
    }
} );
```

addSelectionListener メソッドにより，miQuit から発生したイベントの取り扱いをイベントリスナに登録している．このイベントにより実際に起動されるメソッドは widgetSelected であり，これは SelectionAdapter を新規に生成してイベントリスナに登録する．SelectionEvent のハンドリングの際には常に SelectionAdapter メソッドと widgetSelected メソッドを使用する．

ボタンのクリックの場合も発生するイベントは SelectionEvent であり，この例と同様に扱う．

4.4.4 サンプルプログラム

身長と体重から BMI（Body Mass Index）を算出する GUI アプリケーション（図 38）の構築を考える．

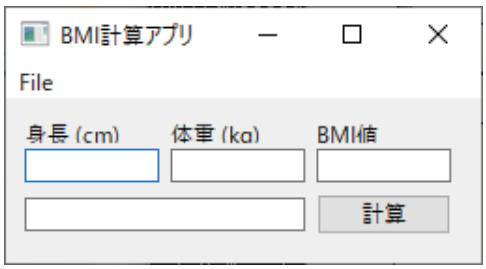


図 38: アプリケーションのウィンドウ

このアプリケーションは 4 つのテキストフィールドを持ち，ユーザが身長と体重の数値を入力して「計算」ボタンをクリックすると，BMI の値とそれに基づいたメッセージが表示される．このアプリケーションは図 39 のようなメニューを備える．

このアプリケーションを実装した例を次のプログラム SWTsample01.java に示す．

プログラム：SWTsample01.java

```
1 import org.eclipse.swt.*;
2 import org.eclipse.swt.widgets.*;
3 import org.eclipse.swt.events.*;
```

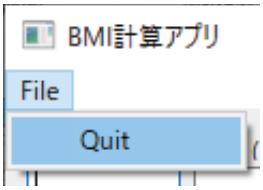


図 39: メニューの構成

```
4
5  public class SWTsample01 {
6      public static void main(String[] args) {
7          // Displayの取得
8          Display disp = Display.getDefault();
9          // ウィンドウの生成と設定
10         Shell sh1 = new Shell();
11         sh1.setSize(270, 145);
12         sh1.setText("BMI計算アプリ");
13         sh1.setLayout(null);  // レイアウトの設定
14         // メニューの構築
15         Menu mb = new Menu(sh1, SWT.BAR);   // メニューバー
16         sh1.setMenuBar(mb);
17         MenuItem miFile = new MenuItem(mb, SWT.CASCADE);    // メニュー
18         miFile.setText("File");
19         Menu mFile = new Menu(miFile);
20         miFile.setMenu(mFile);
21         MenuItem miQuit = new MenuItem(mFile, SWT.NONE);
22         miQuit.setText("Quit");
23         // テキストフィールドの設置と位置・サイズの設定
24         final Text tf1 = new Text(sh1, SWT.BORDER);
25         final Text tf2 = new Text(sh1, SWT.BORDER);
26         final Text tf3 = new Text(sh1, SWT.BORDER);
27         final Text tf4 = new Text(sh1, SWT.BORDER);
28         tf1.setBounds(10, 25, 72, 18);
29         tf2.setBounds(88, 25, 72, 18);
30         tf3.setBounds(166, 25, 72, 18);
31         tf4.setBounds(10, 51, 150, 18);
32         // ラベルの設置
33         Label lb1 = new Label(sh1, SWT.NONE);
34         lb1.setBounds(10, 10, 53, 12);
35         lb1.setText("身長 (cm)");
36         Label lb2 = new Label(sh1, SWT.NONE);
37         lb2.setBounds(88, 10, 53, 12);
38         lb2.setText("体重 (kg)");
39         Label lb3 = new Label(sh1, SWT.NONE);
40         lb3.setBounds(166, 10, 53, 12);
41         lb3.setText("BMI値");
42         // ボタンの設置
43         Button btn1 = new Button(sh1, SWT.NONE);
44         btn1.setBounds(166, 49, 72, 22);
45         btn1.setText("計算");
46         // ボタンからのイベントに対応する処理
47         btn1.addSelectionListener(new SelectionAdapter() {
48             public void widgetSelected(SelectionEvent arg0) {
49                 double h, w, b;
50                 h = Double.parseDouble(tf1.getText()) / 100.0;
51                 w = Double.parseDouble(tf2.getText());
52                 b = w / h / h;
53                 tf3.setText(Double.toString(b));
54                 if ( b < 18.5 ) {
55                     tf4.setText("痩せすぎです");
56                 } else if ( b <= 24.9 ) {
57                     tf4.setText("正常です");
58                 } else {
59                     tf4.setText("太り過ぎです");
60                 }
61             }
62         });
63         // File -> Quit で発生するイベントに対応する処理
```

```
64          miQuit.addSelectionListener(new SelectionAdapter() {
65              public void widgetSelected(SelectionEvent arg0) {
66                  System.exit(0);
67              }
68          });
69          // ウィンドウを表示
70          sh1.open();
71          // アプリケーションのループ（イベントループ）
72          while (!sh1.isDisposed()) {
73              if (!disp.readAndDispatch()) {
74                  disp.sleep();
75              }
76          }
77      }
78  }
```

同じ機能を持ったアプリケーションを，主たるクラスが Shell クラスの派生クラスとして構築することができる．構築例をプログラム SWTsample02.java として次に挙げる．

プログラム：SWTsample02.java

```
 1  import org.eclipse.swt.*;
 2  import org.eclipse.swt.widgets.*;
 3  import org.eclipse.swt.events.*;
 4
 5  public class SWTsample02 extends Shell {
 6      // クラスのメンバ（テキストフィールド）
 7      Text tf1;
 8      Text tf2;
 9      Text tf3;
10      Text tf4;
11      // メインメソッド
12      public static void main(String args[]) {
13          // ディスプレイの取得
14          Display disp = Display.getDefault();
15          // ウィンドウの生成
16          SWTsample02 sh1 = new SWTsample02(disp);
17          // ウィンドウの表示
18          sh1.open();
19          sh1.layout();
20          // アプリケーションのループ
21          while (!sh1.isDisposed()) {
22              if (!disp.readAndDispatch()) {
23                  disp.sleep();
24              }
25          }
26      }
27      // コンストラクタ
28      public SWTsample02(Display disp) {
29          // ウィンドウの設定
30          setText("BMI値計算アプリ");
31          setSize(270, 145);
32          setLayout(null); // レイアウト
33          // メニューの構築
34          Menu mb = new Menu(this, SWT.BAR);
35          setMenuBar(mb);
36          MenuItem miFile = new MenuItem(mb, SWT.CASCADE);
37          miFile.setText("File");
38          Menu mFile = new Menu(miFile);
39          miFile.setMenu(mFile);
40          MenuItem miQuit = new MenuItem(mFile, SWT.NONE);
41          miQuit.setText("Quit");
42          // テキストフィールドの生成と位置・サイズの設定
43          tf1 = new Text(this, SWT.BORDER);
44          tf2 = new Text(this, SWT.BORDER);
45          tf3 = new Text(this, SWT.BORDER);
46          tf4 = new Text(this, SWT.BORDER);
47          tf1.setBounds(10, 25, 72, 18);
48          tf2.setBounds(88, 25, 72, 18);
```

```
49          tf3.setBounds(166, 25, 72, 18);
50          tf4.setBounds(10, 51, 150, 18);
51          // ラベルの設置
52          Label lb1 = new Label(this, SWT.NONE);
53          lb1.setBounds(10, 10, 64, 12);
54          lb1.setText("身長 (cm)");
55          Label lb2 = new Label(this, SWT.NONE);
56          lb2.setBounds(80, 10, 53, 12);
57          lb2.setText("体重 (kg)");
58          Label lb3 = new Label(this, SWT.NONE);
59          lb3.setBounds(139, 10, 53, 12);
60          lb3.setText("BMI値");
61          // ボタンの設置
62          Button btn1 = new Button(this, SWT.NONE);
63          btn1.setBounds(166, 49, 72, 22);
64          btn1.setText("計算");
65          // ボタンからのイベントに対応する処理
66          btn1.addSelectionListener(new SelectionAdapter() {
67              public void widgetSelected(SelectionEvent arg0) {
68                  double h, w, b;
69                  h = Double.parseDouble(tf1.getText()) / 100.0;
70                  w = Double.parseDouble(tf2.getText());
71                  b = w / h / h;
72                  tf3.setText(Double.toString(b));
73                  if ( b < 18.5 ) {
74                      tf4.setText("痩せすぎです");
75                  } else if ( b <= 24.9 ) {
76                      tf4.setText("正常です");
77                  } else {
78                      tf4.setText("太り過ぎです");
79                  }
80              }
81          });
82          // File -> Quit で発生するイベントに対応する処理
83          miQuit.addSelectionListener(new SelectionAdapter() {
84              public void widgetSelected(SelectionEvent arg0) {
85                  System.exit(0);
86              }
87          });
88      }
89      protected void checkSubclass() {} // checkSubclass() の無効化
90  }
```

【より安全な実装】

GUI の実装とイベントのハンドリングで発生するエラーの処理（例外処理）をしておく方がより安全であり，main メソッドを下記のように記述することが望ましい．

プログラムの書き換え部分：安全な main メソッド：

```
1       public static void main(String args[]) {
2           try {
3               Display disp = Display.getDefault();
4               SWTsample02 sh1 = new SWTsample02(disp);
5               sh1.open();
6               sh1.layout();
7               while (!sh1.isDisposed()) {
8                   if (!disp.readAndDispatch()) {
9                       disp.sleep();
10                  }
11              }
12          } catch ( Exception e ) {
13              e.printStackTrace();
14          }
15      }
```

Swing のプログラムと同様に，エラー発生時に printStackTrace メソッドを呼び出している．このメソッドはプログラム中で呼び出された一連のメソッドの記録を出力するもので．問題解決のための情報を提供する．

4.4.5　キーボードイベントからのキーの取り出し

キーボードの操作で発生するイベントは KeyEvent であり，イベント e のメンバ keyCode に押されたキーボードのコードが格納されている．例えば，テキストフィールド tf1 のキーボードイベントからキーボードのコードを取得するには次のようにする．

```
// キーボードイベントの登録
tf1.addKeyListener(new KeyAdapter() {
    public void keyPressed(KeyEvent e) {
        int c = e.keyCode;
        （c を用いた処理）
    }
} );
```

keyCode は int 型の値である．

4.4.6　SWT でのグラフィックスの作成の概要

SWT ライブラリはグラフィックスの描画のための GUI のコンポーネントとして **Canvas クラス**を提供する．また，具体的な描画対象として **Drawable インターフェイス**があり，その実装の 1 つに **Image クラス**がある．Image クラスはビットマップ画像を扱うためのもので，通常はグラフィックスの描画は Image オブジェクトに対して行う．

4.4.6.1　GC（Graphic Context）

GC（Graphic Context）は実際の描画メソッドを適用する対象である．すなわち，具体的な描画処理は GC クラスのオブジェクトに対して行う．GC は Drawable インターフェイスから得られるもので，例えば描画対象の Image クラスのオブジェクトから得られた GC オブジェクトに対して具体的な描画処理を行う．

4.4.6.2　グラフィックス作成の流れ

(1) Canvas オブジェクトの生成

描画領域である Canvas クラスのオブジェクトを取得する．この処理は次のようにして Canvas クラスのコンストラクタを呼び出すことで行う．

```
Canvas( 登録対象のコンテナ, スタイル )
```

例えば，Shell オブジェクトのコンストラクタ内で Canvas オブジェクト cv を生成するには，

```
Canvas cv = new Canvas(this, SWT.NONE);
```

とする．この例では特にスタイルを指定しないので，第 2 引数は SWT.NONE としている．

(2) Image オブジェクトの生成

グラフィックスの画像を保持するための Image オブジェクトを生成する．この処理は次のように Image クラスのコンストラクタを呼び出すことで行う．

```
Image( Display オブジェクト, 横幅, 高さ )
```

例えば，Display オブジェクト disp の下で Image オブジェクト img を生成するには

```
Image img = new Image( disp, 400, 300 );
```

とする．これにより横幅 400，高さ 300 の描画領域が確保される．

(3) GC オブジェクトの生成

具体的な描画処理を行う対象である GC オブジェクトを生成する．この処理は，次のように GC クラスのコンストラクタを呼び出すことで行う．

```
GC( Drawable のオブジェクト )
```

例えば，Image オブジェクト img に対する GC オブジェクトを g として生成するには

GC g = new GC(img);

とする．この後，g に対して描画メソッドを呼び出すことで img にグラフィックスが描かれる．

(4) **Canvas オブジェクトに対する描画イベントの設定**

実際の描画のタイミングは，ウィンドウ生成時やコンテナの状態が変化したときである．具体的には，Canvas オブジェクトに対して PaintEvent イベントが発生したときであり，このイベントが発生したときに Image オブジェクトを Canvas オブジェクトに表示する処理を行うようなイベント処理を登録する．この手続は，Canvas オブジェクトに対して PaintListener を生成・登録するものであり，具体的な処理は paintControl メソッドとして与える．

Image オブジェクトの Canvas オブジェクトへの表示は，その Canvas オブジェクトの GC に対して drawImage メソッドを適用することで行う．drawImage メソッドは次のように Image オブジェクトと表示位置を指定して呼び出す．

drawImage(Image オブジェクト, 横位置, 縦位置)

例えば，Canvas オブジェクト cv に対して PaintEvent e が発生した場合に，Image オブジェクト img を表示するようにイベント処理を登録するには次のように記述する．

```
cv.addPaintListener(new PaintListener() {
    public void paintControl(PaintEvent e) {
        e.gc.drawImage(img, 0, 0 );
    }
} );
```

この例からわかるように，paintControl メソッド内では，イベント e のメンバ gc として描画対象の GC が参照される．

(5) **各種描画メソッドによるグラフィックスの作成**

GC オブジェクトに対して描画メソッドを用いてグラフィックスを描画する．

4.4.7 座標系について

Swing の場合と同様に，図 18 のような座標系を前提とする．

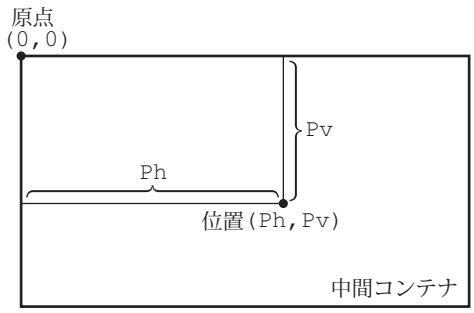


図 18: 描画のための座標系（再録）

4.4.8 色の指定

描画のためのメソッドは大きく 2 つに分けることができる．1 つは ‘draw’ で始まる名前のメソッドで，主に枠を描くメソッドである．2 つ目は ‘fill’ で始まる名前のメソッドで，主に塗りつぶしを行うメソッドである．線画図形の多くは図 40 のように「枠」と「塗り」から構成されており，これら 2 種類のメソッドを使うことで描くことができる．

SWT では枠を描く場合の色属性（Foreground）と，塗りつぶしを行う際の色属性（Background）を分けており，それぞれ色を指定するメソッドが異なる．

Foreground の色属性の設定例： setForeground(blue);

Background の色属性の設定例： setBackground(red);

この例の blue と red は次に説明する Color オブジェクトである．

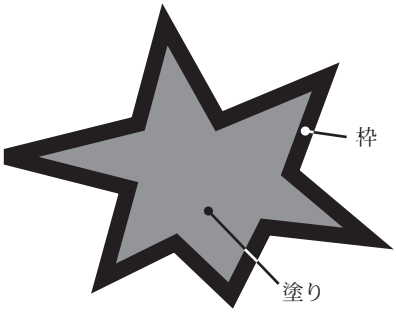


図 40: Foreground（枠）と Background（塗り）

色の指定には Color クラスのオブジェクトを用いる．コンストラクタには次のように引数を指定して Color オブジェクトを生成する．

Color(Display オブジェクト, 赤の階調, 緑の階調, 青の階調)

例えば，Display オブジェクト disp の下で，赤の色属性を持った Color オブジェクト red を生成するには次のように記述する．

```
Color red = new Color(disp,255,0,0);
```

4.4.9 描画のための基本的なメソッド

GC クラスのオブジェクト g に対する描画を例に挙げて各種図形の描画について説明する．

4.4.9.1 直線の描画： drawLine メソッド

使用例： g.drawLine(Ph1, Pv1, Ph2, Pv2)

始点 (Ph1,Pv1) から終点 (Ph2,Pv2) にかけて直線を描く．（位置の指定は int 型整数）

4.4.9.2 折れ線の描画： drawPolyline メソッド

使用例： g.drawPolyline(Pos)

Pos は int 型の配列で，折れ線の頂点の座標の列である．例えば，(h1,v1),(h2,v2),...
という頂点がある場合，Pos に格納するデータは h1,v1,h2,v2,... となる．

	形　　状	値	
(1)		SWT.LINE_SOLID	実線
(2)		SWT.LINE_DASH	破線
(3)		SWT.LINE_DOT	点線
(4)		SWT.LINE_DASHDOT	一点鎖線
(5)		SWT.LINE_DASHDOTDOT	二点鎖線

図 41: 線のスタイル

4.4.9.3 線の属性の設定

SWT における線の属性には太さとスタイルの 2 種類がある．線の太さの設定には下記のように setLineWidth メソッドを用いる．

線の太さを 5 に設定する例： g.setLineWidth(5);

線のスタイルには図 41 に挙げるようなものがあり，下記のように setLineStyle メソッドで設定する．

線のスタイルを設定する例： g.setLineStyle(SWT.LINE_DOT);

4.4.9.4　長方形の描画 (1)　枠：　drawRectangle メソッド

使用例：　g.drawRectangle(Ph, Pv, W, H)

コンテナ内の位置 (Ph,Pv) に横幅 W，高さ H の長方形枠を描く.（引数は int 型整数）

4.4.9.5　長方形の描画 (2)　塗りつぶし：　fillRectangle メソッド

使用例：　g.fillRectangle(Ph, Pv, W, H)

コンテナ内の位置 (Ph,Pv) に横幅 W，高さ H の長方形（塗りつぶし）を描く.（引数は int 型整数）

4.4.9.6　楕円の描画 (1)　枠：　drawOval メソッド

使用例：　g.drawOval(Ph, Pv, W, H)

コンテナ内の位置 (Ph,Pv) に横幅 W，高さ H の楕円形枠を描く.（引数は int 型整数）

4.4.9.7　楕円の描画 (2)　塗りつぶし：　fillOval メソッド

使用例：　g.fillOval(Ph, Pv, W, H)

コンテナ内の位置 (Ph,Pv) に横幅 W，高さ H の楕円形（塗りつぶし）を描く.（引数は int 型整数）

4.4.9.8　多角形の描画 (1)　枠：　drawPolygon メソッド

使用例：　g.drawPolygon(Pos)

Pos は int 型の配列で，多角形の頂点の座標の列である．例えば，(h1,v1),(h2,v2),...
という頂点がある場合，Pos に格納するデータは h1,v1,h2,v2,... となる.

4.4.9.9　多角形の描画 (2)　塗りつぶし：　fillPolygon メソッド

使用例：　g.fillPolygon(Pos)

多角形の塗りつぶしを描画する．引数は drawPolygon と同様である.

4.4.10　文字の描画

SWT では，フォントやスタイル，位置，大きさを指定して文字を描画する．位置の指定については Swing の場合と異なり，図 42 のようになる.

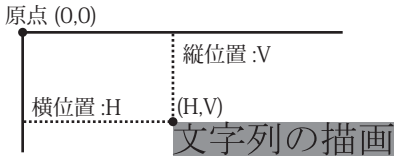


図 42: 文字の描画位置

フォント，スタイル，大きさの指定には **Font クラス**のオブジェクトを用いる．Font オブジェクトのコンストラクタには次のような引数を与える.

Font(Display オブジェクト, フォント名, サイズ, スタイル)

例えば,

```
Font fnt = new Font(disp,"Arial",18,SWT.NORMAL);
```

とすると，フォント名が"Arial"で大きさが 18 ポイントの通常スタイルを指定するオブジェクト fnt が生成される．フォントのスタイルには表 40 のようなものがある.

実際にフォントを設定するには setFont メソッドを用いる.

使用例：　g.setFont(fnt);

これにより，以降の文字描画に fnt の設定が適用される.

表 40: フォントのスタイル（一部）

値	スタイル
SWT.NORMAL	通常のスタイル
SWT.BOLD	太字（ボールド）
SWT.ITALIC	斜体（イタリック）

文字列の描画には drawText メソッドを用いる．

使用例： g.drawText("Hello, Java",10,20)

この例では，(10,20) の座標位置に文字列"Hello, Java"を表示する．

4.4.11 画素に対する操作

個々の画素に色を設定することで，直接的に画像を生成することができる．画素の集まりを扱うために **ImageData クラス**があり，このクラスのオブジェクトに対して各種の処理を行うことで画像の生成ができる．ImageData オブジェクトを生成するには，コンストラクタに下記のように引数を指定する．

ImageData(横幅, 高さ, カラーの深さビット長, パレットデータ)

横幅と高さは int 型の整数で，描画領域の画素のサイズを与える．カラーの深さのビット長には，1 つの画素を表現するのに用いるビット長を int 型整数で与える．ここに指定できるのは 1,2,4,8,16,32 の数（表 41）である．

表 41: カラーの深さのビット長と表現できる色の数

ビット長	表現できる色	
1	($2^1 = 2$)	2 色
2	($2^2 = 4$)	4 色
4	($2^4 = 16$)	16 色
8	($2^8 = 256$)	256 色
16	($2^{16} = 65536$)	65,536 色
32	フルカラー	

パレットデータは色を与えるパレットのためのデータ構造であり，PaletteData クラスのオブジェクトを与える．

パレットの考え方：

画素に与える色は**パレット**から選ぶ．これは PaletteData オブジェクトとして生成する．PaletteData には**インデックス型**と**ダイレクト型**の 2 種類がある．インデックス型のパレットでは，予め使用する色を配列の形で用意（図 43）しておき，色はその配列の添字で指定する．

図 43: インデックス型のパレット

パレットに登録する色はプログラマが任意に設定することができる．PaletteData オブジェクト plt をインデックス型で生成する例を次に示す．

（インデックス型オブジェクトを生成する例）

```
PaletteData plt = new PaletteData(new RGB[] {
    new RGB(0xFF, 0x00, 0x00),
    new RGB(0x00, 0xFF, 0x00),
    new RGB(0x00, 0x00, 0xFF)
} );
```

RGB クラスのオブジェクトは赤・緑・青の各色の成分を保持するもので，この例では，

```
plt[0]  →  赤
plt[1]  →  緑
plt[2]  →  青
```

に設定される．

（ダイレクト型）

ダイレクト型のパレットでは，RGB の成分を直接指定して色を与える．Swing の場合に図 32 のような色成分の格納構造について説明したが，ここでも同様の考え方をする．

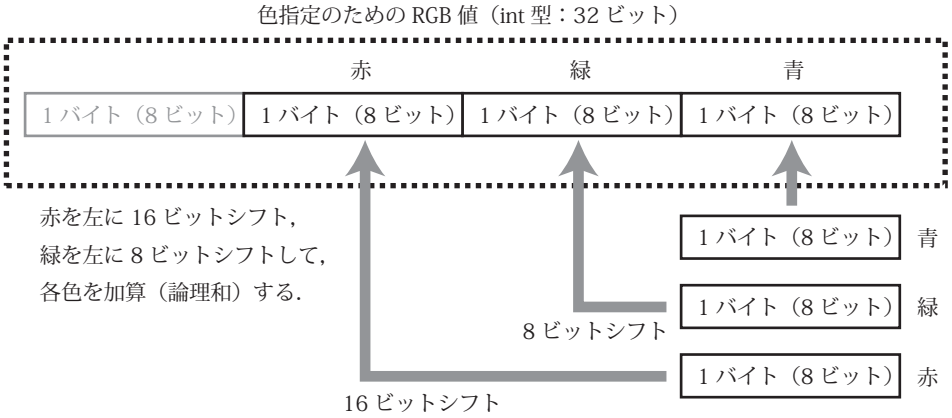


図 32: RGB 値の構造と合成の方法（再録）

ダイレクト型の PaletteData を生成するには，RGB 各色の値を与えるビット列の位置を意味する**ビットマスク**（次の解説参照のこと）をコンストラクタに与える．（下記）

PaletteData(赤のビットマスク, 緑のビットマスク, 青のビットマスク)

例えば，図 32 のような構造で色を指定するパレット plt を生成するには

PaletteData plt = new PaletteData(0xFF<<16,0xFF<<8,0xFF<<0);

とする．‘0xFF’ は 255 の 16 進数表現であり，ビット表現では ‘11111111’ である．すなわちこの値を適切なビット数だけ左にシフトすることで，各色のビット位置を与えることができる．

解説：　ビットマスク

ビットマスクは，ビット列の中の「ある特定の部分」を取り出す場合など（図 44）に用いられる．int 型整数は 32 ビットのビット列で，通常は 1 つの変数に 1 つの値を格納するのに用いるが，更に細かくビット領域を分割して複数の情報を埋め込むという使い方もある．例えば図 32 のように，RGB の 3 つの色の値を 1 つの int 型変数に埋め込むという使い方がそれに当たる．このように，複数の情報が埋め込まれているビット列から，対象のビット列部分だけを取り出す際にビット位置を指定するのにビットマスクを用いる．

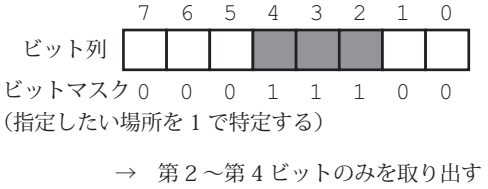


図 44: ビットマスクを用いたビット位置の指定

PaletteData オブジェクト plt が得られたあと ImageData オブジェクトを生成する．次の例は ImageData オブジェクト imd を生成するものである．

```
ImageData imd = new ImageData(400,300,32,plt);
```

この例ではピクセルサイズが 400 × 300 でフルカラーの ImageData を生成している．

4.4.11.1 画素の着色処理

setPixel メソッドを使用することで，ImageData オブジェクトの指定した位置の画素に着色することができる．このメソッドには次のように引数を与える．

```
setPixel(横位置, 縦位置, 色)
```

例えば，インデックス型パレット使用の場合は，

```
imd.setPixel(h,v,plt.getRGB(0));
```

とすることで (h,v) の位置の画素の色をパレットの 0 番目の色にする．またダイレクト型パレット使用の場合は，

```
c = (255<<16) | (0<<8) | (0<<0);
imd.setPixel(h,v,c);
```

とすると (h,v) の位置の画素が赤になる．

参考： 既存の ImageData オブジェクトに対して getRGBs() メソッドを適用するとパレットの色の配列 RGB[] が得られる．これはパレット型の場合で，ダイレクト型の場合は null が返される．この処理は画像を加工する処理の際に必要となる．

4.4.12 画像ファイルの入出力

4.4.12.1 画像ファイルの読み込み

画像ファイルを読み込んで Image オブジェクトにするには，Image のコンストラクタにファイル名を指定する．例えば，Display オブジェクト disp の下で画像ファイル"Balloon.bmp" [33] を読み込んでそれを Image オブジェクト pic とするには

```
final Image pic = new Image(disp,"Balloon.bmp");
```

とする．

4.4.12.2 画像ファイルへの出力

画像ファイルの出力には ImageLoader クラスを用いる．ImageLoader クラスを用いて画像ファイルを出力する手順を次に示す．

(1) ImageLoader オブジェクトを生成する
(2) 上記オブジェクトのメンバ data に出力したい ImageData オブジェクトを指定する
(3) 上記オブジェクトに save メソッドを適用して出力処理を実行する

例．ImageData オブジェクト dat をファイル 'out.bmp' に出力する処理

```
ImageLoader il = new ImageLoader();
il.data = new ImageData[]{ dat };
il.save("out.bmp", SWT.IMAGE_BMP);
```

il.data には ImageData オブジェクトを配列の形で与えている．これは，**レイヤー構造** [34] をもつ画像フォーマットに対応するためである．

例．Image オブジェクト img をファイル 'out.bmp' に出力する処理

```
ImageData dat = img.getImageData();
ImageLoader il = new ImageLoader();
il.data = new ImageData[]{ dat };
il.save("out.bmp", SWT.IMAGE_BMP);
```

[33] 標準画像の 1 つ

[34] 画像処理における重要な概念の 1 つ．複数のビットマップ画像を重ねて 1 つの画像を構成する場合に用いられる．GIF アニメーションもこの考えを応用したものである．

この例では，img から ImageData オブジェクト dat を生成して出力している．

4.4.13　サンプルプログラム

1. 描画メソッドによる描画，画像ファイルの読み込み，画像ファイルへの保存

プログラム：SWT2d01.java

```
import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.graphics.*;
import org.eclipse.swt.events.*;

public class SWT2d01 extends Shell {
    public static void main(String args[]) {
        try {
            Display disp = Display.getDefault();
            SWT2d01 shell = new SWT2d01(disp);
            shell.open();
            shell.layout();
            while (!shell.isDisposed()) {
                if (!disp.readAndDispatch()) {
                    disp.sleep();
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    // コンストラクタ
    public SWT2d01(Display disp) {
        setLayout(null);
        setText("Sample Program : SWT2d01");
        setSize(1090, 320);
        // canvasの生成
        Canvas cv = new Canvas(this, SWT.NONE);
        cv.setBounds(10, 10, 1060, 256);
        // GCの生成
        final Image img = new Image(disp,1060,256);
        GC g = new GC(img);
        // 画像の読み込み
        final Image pic = new Image(disp,"Balloon.bmp");
        // 色の取得
        Color red   = new Color(disp,255,0,0);
        Color blue  = new Color(disp,0,0,255);
        Color white = new Color(disp,255,255,255);
        Color black = new Color(disp,0,0,0);
        g.setBackground(red);
        g.setForeground(blue);
        // 線の太さの設定
        g.setLineWidth(5);
        // 長方形
        g.fillRectangle(10,10,50,50);
        g.drawRectangle(10,10,50,50);
        // 楕円
        g.fillOval(80,10,100,50);
        g.drawOval(80,10,100,50);
        // 直線
        g.drawLine(200,10,220,60);
        // 折れ線
        int pos[] = new int[]{10,70,190,70,155,150,100,100,55,150,10,70};
        g.drawPolyline(pos);
        // 多角形
        pos = new int[]{10,170,190,170,155,250,100,200,55,250,10,170};
        g.fillPolygon(pos);
        g.drawPolygon(pos);
        // 角の丸い長方形
        g.fillRoundRectangle(240,10,100,50,15,15);
        g.drawRoundRectangle(240,10,100,50,15,15);
        // 弧
```

```
63          g.fillArc(240,70,100,50,30,300);
64          g.drawArc(240,70,100,50,30,300);
65          // 破線
66          g.setLineWidth(3);
67          int h, v = 135, d = 1;
68          for ( h = 200; h <= 280; h += 18 ) {
69              g.setLineStyle(d++);
70              g.drawOval(h,v,(295-h)*2,(190-v)*2);
71              v += 5;
72          }
73          // 文字の描画
74          g.setBackground(white);
75          g.setForeground(black);
76          Font fnt = new Font(disp,"Arial",18,SWT.NORMAL);
77          g.setFont(fnt);
78          g.drawText("Arial Normal",410,5);
79          fnt = new Font(disp,"Arial",18,SWT.BOLD);
80          g.setFont(fnt);
81          g.drawText("Arial Bold",410,30);
82          fnt = new Font(disp,"Arial",18,SWT.ITALIC);
83          g.setFont(fnt);
84          g.drawText("Arial Italic",410,55);
85          fnt = new Font(disp,"Arial",18,SWT.ITALIC|SWT.BOLD);
86          g.setFont(fnt);
87          g.drawText("Arial Bold Italic",410,80);
88          fnt = new Font(disp,"ＭＳ 明朝",18,SWT.NORMAL);
89          g.setFont(fnt);
90          g.drawText("ＭＳ 明朝 標準",410,105);
91          fnt = new Font(disp,"ＭＳ 明朝",18,SWT.BOLD);
92          g.setFont(fnt);
93          g.drawText("ＭＳ 明朝 ボールド",410,130);
94          fnt = new Font(disp,"ＭＳ 明朝",18,SWT.ITALIC);
95          g.setFont(fnt);
96          g.drawText("ＭＳ 明朝 イタリック",410,155);
97          fnt = new Font(disp,"ＭＳ 明朝",18,SWT.ITALIC|SWT.BOLD);
98          g.setFont(fnt);
99          g.drawText("ＭＳ 明朝 ボールドイタリック",410,180);
100         fnt = new Font(disp,"HG行書体",20,SWT.ITALIC|SWT.BOLD);
101         g.setFont(fnt);
102         g.drawText("HG行書体 ボールドイタリック",400,210);
103         // 画像として保存
104         ImageData dat  = img.getImageData();
105         ImageLoader il = new ImageLoader();
106         il.data = new ImageData[]{ dat };
107         il.save("SWT2d01.bmp", SWT.IMAGE_BMP);
108         // 描画をcanvasに反映
109         cv.addPaintListener(new PaintListener() {
110             public void paintControl(PaintEvent e) {
111                 e.gc.drawImage(pic,   0, 0 );
112                 e.gc.drawImage(img, 260, 0);
113             }
114         });
115     }
116     // Disable the check that prevents subclassing of SWT components
117     protected void checkSubclass() { }
118 }
```

このプログラムを実行すると図 45 のように表示される.

2. 画素の生成

[プログラム：SWT2d03.java]

```
1  import org.eclipse.swt.SWT;
2  import org.eclipse.swt.widgets.*;
3  import org.eclipse.swt.graphics.*;
4  import org.eclipse.swt.events.*;
5
6  public class SWT2d03 extends Shell {
7      public static void main(String args[]) {
```

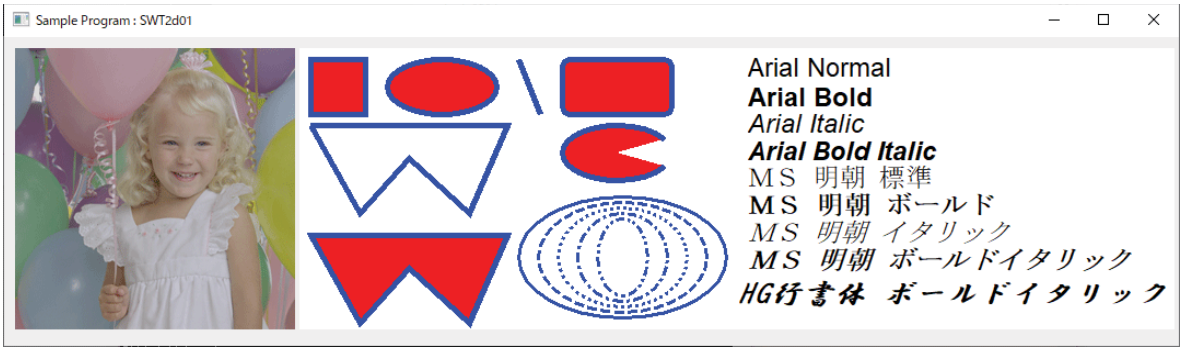


図 45: 各種グラフィックスの表示

```
        try {
            Display disp = Display.getDefault();
            SWT2d03 shell = new SWT2d03(disp);
            shell.open();
            shell.layout();
            while (!shell.isDisposed()) {
                if (!disp.readAndDispatch()) {
                    disp.sleep();
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    // RGB値生成
    public int vRGB( int r, int g, int b ) {
        return( (r<<16) | (g<<8) | (b<<0) );
    }
    // コンストラクタ
    public SWT2d03(Display disp) {
        setLayout(null);
        setText("Sample Program : SWT2d01");
        setSize(440, 210);
        // canvasの生成
        Canvas cv = new Canvas(this, SWT.NONE);
        cv.setBounds(10, 10, 400, 150);
        // PaletteDataの作成
        PaletteData plt = new PaletteData(0xFF<<16,0xFF<<8,0xFF<<0);
        // ImageDataの確保
        ImageData imd = new ImageData(400,150,32,plt);
        int m, h, v;
        // RGBレベル
        for ( h = 0; h < 400; h++ ) {
            m = 255*h/399;
            for ( v = 0; v < 50; v++ ) {
                imd.setPixel(h,v,vRGB(m,0,0));
            }
        }
        for ( h = 0; h < 400; h++ ) {
            m = 255*h/399;
            for ( v = 50; v < 100; v++ ) {
                imd.setPixel(h,v,vRGB(0,m,0));
            }
        }
        for ( h = 0; h < 400; h++ ) {
            m = 255*h/399;
            for ( v = 100; v < 150; v++ ) {
                imd.setPixel(h,v,vRGB(0,0,m));
            }
        }
        // 描画用イメージとGCの生成
        final Image img = new Image(null,imd);
        GC g = new GC(img);
        // 画像として保存
```

```
62          ImageLoader il = new ImageLoader();
63          il.data = new ImageData[]{ imd };
64          il.save("SWT2d03.bmp", SWT.IMAGE_BMP);
65          // 描画をcanvasに反映
66          cv.addPaintListener(new PaintListener() {
67              public void paintControl(PaintEvent e) {
68                  e.gc.drawImage(img, 0, 0);
69              }
70          });
71      }
72      // Disable the check that prevents subclassing of SWT components
73      protected void checkSubclass() { }
74  }
```

このプログラムを実行した結果を図 46 に示す．

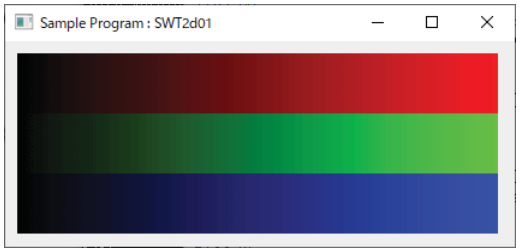


図 46: 画素を直接生成して作画した例

4.4.14 ファイル選択ダイアログ

実用的なアプリケーションプログラムでは，ファイルの読み込みや保存の際に，ファイルを選択するためのダイアログを表示してユーザに操作を促す．SWT にもファイル選択ダイアログを表示するための API が用意されている．ここでは，読込み用と保存用の 2 種類のファイル選択のダイアログについて説明する．

ファイル選択ダイアログは FileDialog クラスのオブジェクトである．
このクラスのオブジェクトを生成する際，コンストラクタの第 1 引数にアプリケーションのウィンドウ（Shell クラスのオブジェクト）を，第 2 引数にダイアログの**スタイル**（表 42 参照）を整数型で与える．

表 42: FileDialog のスタイル

値	意　　味
SWT.OPEN	既存のファイルを選択する際のダイアログ
SWT.SAVE	保存先ファイルを指定する際のダイアログ

【読込み用ファイル選択ダイアログを表示する例】

```
FileDialog fd = new FileDialog(sh,SWT.OPEN);
String fn = fd.open();
```

上記 sh は，読込み用ダイアログを表示するためのアプリケーションのウィンドウ（Shell クラス）で，このウィンドウに属するダイアログオブジェクト fd が生成される．この fd に対して open メソッドを実行すると，ファイルを読み込むためのダイアログが表示される．選択結果の戻り値は String 型のオブジェクトで，選択したファイルのパス名が値として保持されている．ユーザのキャンセル操作などによってファイルが選択されなかった場合の戻り値は null である．

読込み用，保存用それぞれのダイアログを表示するサンプルプログラム FileDialogSWT1.java を示す．

プログラム：FileDialogSWT1.java

```
1  import org.eclipse.swt.*;
2  import org.eclipse.swt.widgets.*;
3  import org.eclipse.swt.events.*;
4
```

```
5  public class FileDialogSWT1 {
6      public static void main(String[] args) {
7          // Displayの取得
8          Display disp = Display.getDefault();
9          // ウィンドウの生成と設定
10         Shell sh1 = new Shell();
11         sh1.setSize(400, 150);
12         sh1.setText("File Dialog");
13         sh1.setLayout(null);  // レイアウトの設定
14         // ボタンの設置
15         Button btn1 = new Button(sh1, SWT.NONE);
16         btn1.setBounds(100, 40, 72, 22);
17         btn1.setText("選択");
18         Button btn2 = new Button(sh1, SWT.NONE);
19         btn2.setBounds(200, 40, 72, 22);
20         btn2.setText("保存");
21         //---- ボタンからのイベントに対応する処理（イベント処理の登録）----
22         //  読込み用
23         btn1.addSelectionListener(new SelectionAdapter() {
24             public void widgetSelected(SelectionEvent arg0) {
25                 FileDialog fd = new FileDialog(sh1,SWT.OPEN);
26                 String fn = fd.open();
27                 if ( fn == null ) {
28                     System.out.println("ファイルを選択しませんでした. ");
29                 } else {
30                     System.out.println("ファイル \""+fn+"\" を選択しました. ");
31                 }
32             }
33         });
34         //  保存用
35         btn2.addSelectionListener(new SelectionAdapter() {
36             public void widgetSelected(SelectionEvent arg0) {
37                 FileDialog fd = new FileDialog(sh1,SWT.SAVE);
38                 String fn = fd.open();
39                 if ( fn == null ) {
40                     System.out.println("保存先ファイルを指定しませんでした. ");
41                 } else {
42                     System.out.println("保存先 \""+fn+"\" を指定しました. ");
43                 }
44             }
45         });
46         // ウィンドウを表示
47         sh1.open();
48         // アプリケーションのループ（イベントループ）
49         while (!sh1.isDisposed()) {
50             if (!disp.readAndDispatch()) {
51                 disp.sleep();
52             }
53         }
54     }
55 }
```

このプログラムを実行すると，図 47 のようなウィンドウが表示される．

図 47: 実行したところ

ここで「選択」ボタンをクリックすると図 48 のようなダイアログが表示され，既存のファイルを選択することができる．また「保存」ボタンをクリックすると図 49 のようなダイアログが表示され，保存用にディレクトリとファイル名を指定することができる．

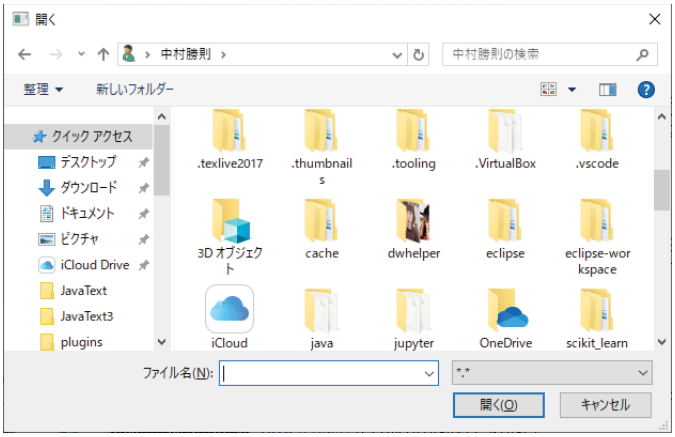


図 48: 読込むファイルを選択するダイアログ

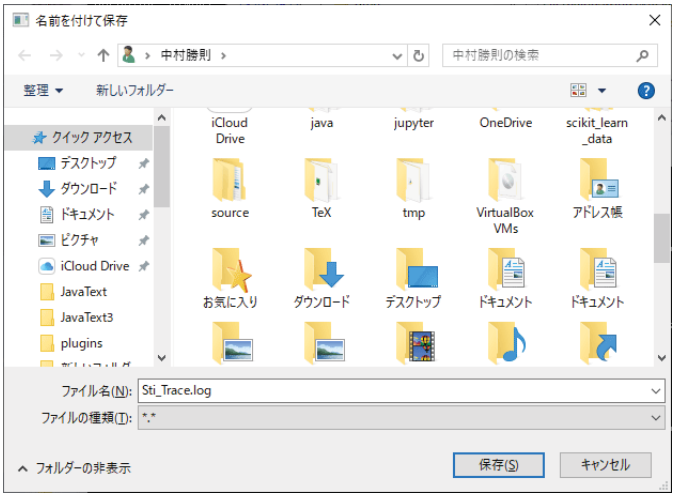


図 49: 保存するファイルを指定するダイアログ

5　スレッドとプロセス

プログラムは文を順番に実行する流れである．分岐や反復を含むプログラムも 1 つの流れに沿って実行され，文が 1 つずつ実行される．このような 1 続きの処理の脈絡を**スレッド**と呼ぶ．Java では複数のスレッドを同時に並行して実行する**マルチスレッドプログラミング**が可能である．

実用的なアプリケーションプログラムでは，異なる複数の処理を同時に実行することが求められることがある．例えば HTML コンテンツを閲覧するための Web ブラウザは，通信やコンテンツのレンダリング，マルチメディアの再生処理などを同時に並行して実行するが，そのような処理を実現するための 1 つの手段[35] が**マルチスレッドプログラミング**である．

5.1　マルチスレッドプログラミング

5.1.1　Thread クラス

スレッドを取り扱うには **Thread クラス**を利用する．スレッドは Thread クラスの run メソッドを呼び出すことで実行される．従ってプログラマは Thread クラスの拡張クラスを定義し，その中で run メソッドをオーバーライドすることで独自のスレッドを記述する．

スレッドの記述：

```
class クラス名 extends Thread {
    public void run() {
        （実行内容の記述）
    }
}
```

このように記述されたスレッドをオブジェクトとして生成して起動する．スレッドオブジェクトを起動するには start メソッドを用いる．

Thread の拡張クラス MyThread のオブジェクト mt1 を実行する例：

```
MyThread mt1 = new MyThread();
mt1.start();
```

以下でサンプルプログラムを見ながらスレッドプログラミングについて説明する．

不定期にメッセージを 5 回表示するプログラムを例に，Thread クラスの利用方法を説明する．
2 秒（2,000 ミリ秒）未満の不定時間毎にメッセージを 5 回表示するスレッドを，2 つ別々に起動するプログラム ThreadTest1.java を次に示す．

プログラム：ThreadTest1.java

```
1  class MyThread extends Thread {
2      // スレッド開始メソッド
3      public void run() {
4          int i;
5          double d;
6          long b, t, w;
7          b = System.currentTimeMillis();
8          String name = Thread.currentThread().getName();
9          for ( i = 0; i < 5; i++ ) {
10             w = (int)(2000.0*Math.random());    // ランダムな待ち時間の生成
11             try {
12                 Thread.currentThread().sleep(w);
13             } catch ( Exception e ) {
```

[35]別の手段として**マルチプロセシング**があるが，これに関しては後で取り上げる．

```
14                  break;
15              }
16              t = System.currentTimeMillis();
17              d = (long)(t - b) / 1000.0;
18              System.out.println(d+": "+name+" waited "+w+" milli seconds.");
19          }
20      }
21  }
22  // メイン
23  class ThreadTest1 {
24      public static void main( String argv[] ) {
25          MyThread mt1 = new MyThread();
26          mt1.start();
27          MyThread mt2 = new MyThread();
28          mt2.start();
29          System.out.println("main ended.");
30      }
31  }
```

このプログラムでは，スレッドプログラムを記述するための Thread の拡張クラス MyThread を定義しており，その中の run メソッドとして実行内容を記述している．

メッセージを表示する時間間隔 w は乱数を用いて設定（10 行目）し，その時間が経過（12 行目）するごとにメッセージを表示（18 行目）して，それが 5 回終われば実行を終了するというスレッドプログラムを記述している．

使用したメソッドの説明：

- System に対するクラスメソッド currentTimeMills はシステム時刻をミリ秒単位で返す．
- スレッドプログラム内で Thread クラスメソッド currentThread を実行すると，そのスレッドオブジェクト自体が返される．
- スレッドオブジェクトに対して getName メソッドを実行すると，その スレッドの名前 が返される．
- スレッドオブジェクトに対して sleep メソッドを実行すると，そのスレッドの実行を指定した時間（ミリ秒単位）停止する．

このプログラムでは main メソッドから 2 つのスレッド mt1,mt2 が生成・起動される．スレッドは生成した直後は実行されず，start メソッドによってその実行が開始される．このプログラムでは，main メソッドは 2 つのメソッドが起動した直後に終了する．main,mt1,mt2 は並行して実行され，それぞれの時間経過の関係は図 50 のようになる．

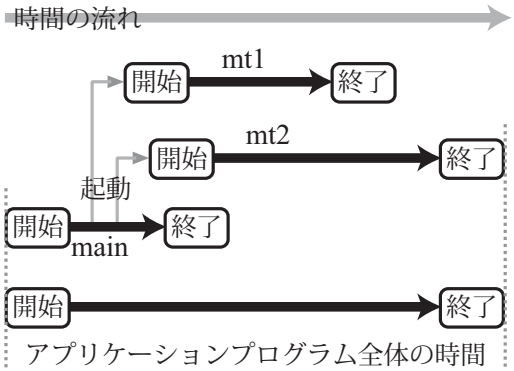


図 50: 各スレッドの時間経過の関係

各スレッドは，時刻，スレッド名，待ち時間を 1 行に表示する．

ThreadTest1.java を実行した例を次に示す.

```
main ended.      0.501:  Thread-0 waited 487 milli seconds.
0.599:  Thread-0 waited 96 milli seconds.
0.877:  Thread-0 waited 277 milli seconds.
0.963:  Thread-1 waited 949 milli seconds.
1.102:  Thread-1 waited 138 milli seconds.
1.83:  Thread-0 waited 953 milli seconds.
2.816:  Thread-1 waited 1713 milli seconds.
3.142:  Thread-0 waited 1311 milli seconds.
4.677:  Thread-1 waited 1861 milli seconds.
4.72:  Thread-1 waited 42 milli seconds.
```

この結果から，mt1，mt2 それぞれのスレッドが別々に並行して実行されている様子がわかる.（mt1,mt2 起動直後に main が終了していることもわかる）

5.1.2 Runnable インターフェース

Thread クラスの拡張クラスを定義してスレッドを作成する方法には 1 つの問題がある．Java ではクラスの多重継承ができないため，Thread クラス（とそれを継承するクラス）以外のクラスのメソッドをそのままスレッドにすることはできない.

Java には **Runnable インターフェース**があり，これを用いることで Thread クラスとは無関係のクラスに run メソッドを記述してスレッドにすることができる.

次のプログラム ThreadTest2.java は先のプログラムと動作はほぼ同じであるが，Runnable インターフェースを用いて実装した例である.

プログラム：ThreadTest2.java

```
1  class MyRunnable implements Runnable {
2      // スレッド開始メソッド
3      public void run() {
4          int i;
5          double d;
6          long b, t, w;
7          b = System.currentTimeMillis();
8          String name = Thread.currentThread().getName();
9          for ( i = 0; i < 5; i++ ) {
10             w = (int)(2000.0*Math.random());    // ランダムな待ち時間の生成
11             try {
12                 Thread.currentThread().sleep(w);
13             } catch ( Exception e ) {
14                 break;
15             }
16             t = System.currentTimeMillis();
17             d = (long)(t - b) / 1000.0;
18             System.out.println(d+": "+name+" waited "+w+" milli seconds.");
19         }
20     }
21 }
22 // メイン
23 class ThreadTest2 {
24     public static void main( String argv[] ) {
25         MyRunnable r1 = new MyRunnable();
26         Thread t1 = new Thread(r1);
27         MyRunnable r2 = new MyRunnable();
28         Thread t2 = new Thread(r2);
29         t1.start();
30         t2.start();
31         System.out.println("main ended.");
32     }
33 }
```

Runnable インターフェースを用いたクラスに run メソッドを記述（下記）することで，そのクラスのオブジェクトをスレッドにすることができる．

Thread クラスと無関係のクラス MyRunnable をスレッドにする例：

```
class MyRunnable implements Runnable {
    public void run() {
        （実行内容の記述）
    }
}
```

上の例のように記述することで MyRunnable クラス（任意のクラスを継承することが可能）のオブジェクトをスレッドにすることができる.（次の例を参照）

先の例の MyRunnable クラスのオブジェクトをスレッドにする例：

```
MyRunnable r1 = new MyRunnable();
Thread t1 = new Thread(r1);    // スレッドにするオブジェクトをコンストラクタに与える
t1.start();
```

スレッドを生成するには最終的には Thread クラスのオブジェクトを生成する必要がある．それを行なっているのが上記の

```
Thread t1 = new Thread(r1);
```

の部分である．

5.1.3 スレッドの同期

実用的なスレッドプログラミングを行うには，スレッド間での実行タイミングの制御（同期）を行う必要がある．上に挙げたプログラムの例では，各スレッドは個々別々の動作をしており，互いに無関係である．ここでは，スレッドの実行状態を変える方法や，スレッド間の同期を取る方法について説明する．

5.1.3.1 他のスレッドの終了を待機する方法： join メソッド

Thread オブジェクトが終了するまで待つには join メソッドを用いる．例えば Thread オブジェクト t1 が終了するのを待つには，

```
t1.join();
```

と記述する．先のプログラム ThreadTest2.java では，main メソッドは Thread オブジェクト t1,t2 を起動した直後に終了していたが，t1,t2 両方のスレッドが終了するまで待つ形にするには main メソッドを次のように変更する．

プログラムの変更：main メソッド

```
1  class ThreadTest2 {
2      public static void main( String argv[] ) {
3          MyRunnable r1 = new MyRunnable();
4          Thread t1 = new Thread(r1);
5          MyRunnable r2 = new MyRunnable();
6          Thread t2 = new Thread(r2);
7          t1.start();
8          t2.start();
9          try {
10             t1.join();
11             t2.join();
12         } catch (Exception e) {
13             e.printStackTrace();
14         }
15         System.out.println("main ended.");
```

```
16        }
17  }
```

このように変更することで，各スレッドの時間経過の関係は図 51 のようになる．

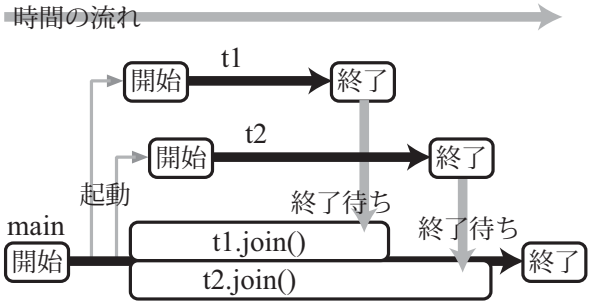


図 51: 各スレッドの時間経過の関係

実行結果の例を次に示す．

```
0.021:  Thread-0 waited 21 milli seconds.
0.175:  Thread-1 waited 175 milli seconds.
1.236:  Thread-0 waited 1212 milli seconds.
1.781:  Thread-0 waited 545 milli seconds.
1.964:  Thread-1 waited 1789 milli seconds.
3.276:  Thread-1 waited 1311 milli seconds.
3.441:  Thread-0 waited 1660 milli seconds.
3.5:  Thread-0 waited 58 milli seconds.
4.979:  Thread-1 waited 1703 milli seconds.
6.221:  Thread-1 waited 1242 milli seconds.
main ended.
```

スレッド t1,t2 が終了した後に “main ended.”のメッセージが表示されていることがわかる．

join メソッドの応用例　複数スレッドによる並列分散処理を行う際，全てのスレッドが終了するのを確認してから次の処理に移るという制御を行うことができる．

5.1.3.2　synchronized 修飾子

マルチスレッドプログラミングでは，

(1) 資源への排他的アクセス

(2) スレッドの同期

の 2 つの点が重要である．変数などの記憶資源に対して複数のスレッドが同時にアクセスすると値の整合性に問題が生じる．これを回避するためには資源への排他的なアクセスが求められる．すなわち，同一の記憶資源に対して 1 つのスレッドが読み書きを行う間は，他のスレッドからのアクセスを抑制することが必要になる．

先に説明した join メソッドによるスレッド終了の待ち受けに加えて，スレッドを待機させたり，待機させたスレッドの実行を再会する操作を行うことで，より高度なスレッド実行のスケジュールが可能になる．以降では，**排他的制御**を含めたより高度な同期処理を実現するための **synchronized 修飾子** とそれに関連するメソッドの使用方法について説明する．

【記憶資源の排他制御】

複数のスレッドが同時並行的に 1 つの記憶資源にアクセスする例を考える．図 52 の例では，1 つの共有記憶の変数 *number* に対して，複数のスレッドが 1 を加える処理を行っている．

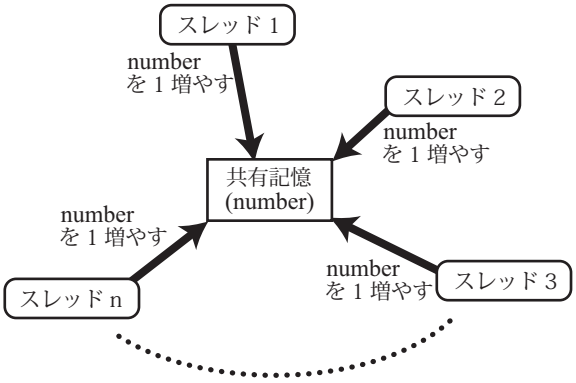


図 52: 複数のスレッドによる共有記憶へのアクセスの例

これを実装したプログラム例 ThreadTest3.java について考える.

プログラム：ThreadTest3.java

```
1  // 共有記憶
2  class ShMem {
3      int number;     // 共有するメンバ
4      // コンストラクタ
5      ShMem() {
6          number = 0;
7      }
8      // インクリメント
9      void inc() {
10         int n;
11         n = number;
12         n++;
13         number = n;
14     }
15 }
16 // スレッドのためのRunnableクラス
17 class MyRunnable implements Runnable {
18     public void run() {
19         int n;
20         try {
21             Thread.currentThread().sleep(100);
22         } catch ( Exception e ) {
23             e.printStackTrace();
24         }
25         ThreadTest3.shmem.inc();
26     }
27 }
28 // メイン
29 class ThreadTest3 {
30     // 共有記憶
31     public static ShMem shmem;
32
33     public static void main( String argv[] ) {
34         // 共有記憶の初期化
35         shmem = new ShMem();
36         System.out.println("初期値  : "+shmem.number);
37         // スレッドの生成と起動
38         MyRunnable r[] = new MyRunnable[1000];
39         Thread t[] = new Thread[1000];
40         int c;
41         for ( c = 0; c < 1000; c++ ) {
42             r[c] = new MyRunnable();
43             t[c] = new Thread(r[c]);
44             t[c].start();
45         }
46         // スレッド終了の待機
47         for ( c = 0; c < 1000; c++ ) {
48             try {
49                 t[c].join();
```

```
50                } catch ( Exception e ) {
51                    e.printStackTrace();
52                }
53            }
54            // 結果表示
55            System.out.println("処理結果: "+shmem.number);
56        }
57    }
```

このプログラムでは，共有記憶のためのクラス ShMem が定義されており，そのメンバとして変数 number がある．またこのクラスには number に 1 を加える操作を行うメソッド inc があり，複数のスレッド（この例では 1000 個のスレッド）がこれを実行する．

処理を行うスレッドの個数が n であるとして，処理の衝突が起こらなければ number の値は n 増えることになる．例のプログラムでは，number の初期値は 0 であるので処理が終わった後は 1000 になるかと思われるが，実際には処理の衝突が起こり，次の実行結果のように，number の値は 1000 を下回る．

```
初期値  :  0
処理結果:  771
```

共有記憶を 1 増やす inc メソッドに対して下記のように synchronized 修飾子を付けることで，そのメソッドの実行を 1 つのスレッドに限定することができる．

変更箇所：inc メソッド

```
1       // インクリメント
2       synchronized void inc() {
3           int n;
4           n = number;
5           n++;
6           number = n;
7       }
```

このように記述することで，1 つのスレッドが inc メソッドを実行している途中で別のスレッドが同じメソッドを実行しようとすると，そのスレッドは一時停止の状態になり，実行中の inc メソッドが終了するまで待たされる．

上記のように inc メソッドを変更したプログラムの実行結果を次に示す．

```
初期値  :  0
処理結果:  1000
```

number の値がメソッドの個数と同じ 1000 になっていることがわかる

synchronized 修飾子はプログラム中の特定のブロックに指定することもできる．（下記）

変更箇所：inc メソッド

```
1       // インクリメント
2       void inc() {
3           int n;
4           synchronized (this) {
5               n = number;
6               n++;
7               number = n;
8           }
9       }
```

排他的処理の対象となるオブジェクトを指定して {…} 内の処理を排他的に行うための synchronized 修飾子の記述法を次に示す．

```
synchronized ( 対象オブジェクト ) {
        (処理の記述)
}
```

このように，排他的処理の対象となるプログラムの部分を**クリティカルリージョン**（クリティカルセクション）と呼ぶ．

考察． run メソッドに synchronized 修飾子を付けてはならない．その理由を考えよ．

注意） ここで示したような多数のスレッドの同時実行においては Tread クラスを用いる方法は安全ではなく，後の「5.2 concurrent ライブラリ」（p.150）で解説する **concurrent ライブラリ**を用いるべきである．

5.1.3.3 スレッドの待機／実行の再開

wait メソッド，notifyAll メソッドを用いることで，スレッド実行の一時停止と再開の制御が可能になる．

■ wait メソッド

スレッドの中で **wait メソッド**を実行することで，そのスレッドの実行を一時停止することができる．このメソッドは synchronized 指定されたメソッド（あるいはブロック）内で実行する．

■ notifyAll メソッド

Thread クラスやそれを継承したクラス，あるいは Runnable インターフェースを実装したクラスのメソッド内で **notifyAll** [36] メソッドを実行することで，一時中断したメソッドの実行を再開する．このメソッドは synchronized 指定されたメソッド（あるいはブロック）内で実行する．

notifyAll メソッドで再開されるスレッドは，同じモニタ（共有オブジェクト）内で待機中の全てのスレッドである．

【サンプルプログラム】

スレッド実行の一時停止と再開を行うプログラムの例を示す．次のプログラム ThreadTest4.java は不定期にメッセージを 3 回表示する 3 つのスレッドを起動するものであり，それら 3 つのスレッドは起動直後に一時停止する．main メソッド内で実行再開をスレッドに対して指示すると，スレッドは一斉にメッセージを出力し，全てのスレッドが終了した後，main メソッドが終了する．

プログラム：ThreadTest4.java

```
1  // スレッドのためのRunnableクラス
2  class MyRunnable implements Runnable {
3      // スレッド実行の入り口
4      public void run() {
5          go();
6      }
7      // スレッドの動作の本体
8      synchronized void go() {
9          int c, w;
10         try {
11             wait();
12         } catch ( Exception e ) {
13             e.printStackTrace();
14         }
15         for ( c = 0; c < 3; c++ ) {
16             w = (int)(1000.0*Math.random());
17             try {
18                 Thread.currentThread().sleep(w);
19             } catch ( Exception e ) {
20                 e.printStackTrace();
21             }
```

[36] 他にも notify メソッドもあるが，これの使用は Java の開発元が推奨していない．

```
22                 System.out.println(Thread.currentThread().getName()+" : "+c);
23             }
24             System.out.println(Thread.currentThread().getName()+" ended.");
25         }
26         // スレッド実行再開
27         synchronized void begin() {
28             notifyAll();
29         }
30     }
31     // メイン
32     class ThreadTest4 {
33         public static void main( String argv[] ) {
34             int c;
35             MyRunnable r[] = new MyRunnable[3];
36             Thread   t[] = new Thread[3];
37             // スレッドの生成と実行
38             for ( c = 0; c < 3; c++ ) {
39                 r[c] = new MyRunnable();
40                 t[c] = new Thread(r[c]);
41                 t[c].start();
42             }
43             System.out.println("マルチスレッド実行待機");
44             for ( c = 3; c > 0; c-- ) {
45                 System.out.println(c+"秒前");
46                 try {
47                     Thread.currentThread().sleep(1000);
48                 } catch ( Exception e ) {
49                     e.printStackTrace();
50                 }
51             }
52             System.out.println("マルチスレッド実行再開");
53             // スレッド実行の再開
54             for ( c = 0; c < 3; c++ ) {
55                 r[c].begin();
56             }
57             // スレッド終了の待ち受け
58             for ( c = 0; c < 3; c++ ) {
59                 try {
60                     t[c].join();
61                 } catch ( Exception e ) {
62                     e.printStackTrace();
63                 }
64             }
65             System.out.println("main ended.");
66         }
67     }
```

次の実行例のように，このプログラムを実行すると秒読みの後，3 つのスレッドが一斉に実行を再開する．

```
マルチスレッド実行待機
3 秒前
2 秒前
1 秒前
マルチスレッド実行再開
Thread-2 :  0
Thread-1 :  0
Thread-0 :  0
Thread-1 :  1
Thread-2 :  1
Thread-1 :  2
Thread-1 ended.
Thread-2 :  2
Thread-2 ended.
Thread-0 :  1
Thread-0 :  2
Thread-0 ended.
main ended.
```

全てのスレッドが終了した後で main が終了しているのがわかる.

5.1.4 スレッドの状態について

スレッドの状態の移り変わりを図 53 に示す.

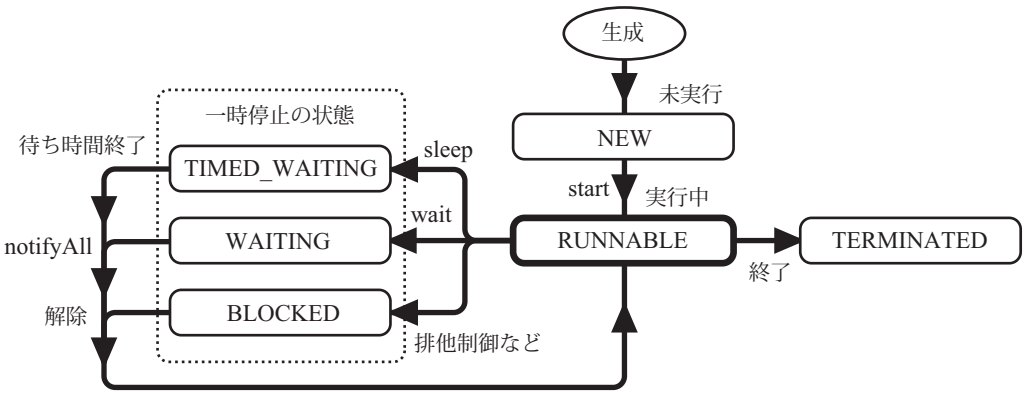


図 53: スレッドの状態遷移

スレッドの状態は，スレッドに対して getState メソッドを実行することで調べることができる. getState メソッドによって得られるスレッドの状態を表 43 に示す.

表 43: スレッドの状態（Thread.State）

状態	説明
NEW	生成された直後でまだ起動されていない
RUNNABLE	実行中
TIMED_WAITING	待機時間付きで待機中（sleep などによる）
WAITING	待機中（wait などによる）
BLOCKED	ブロックされている（排他制御などによる）
TERMINATED	終了した状態

5.2 concurrent ライブラリ

マルチスレッドプログラミングには，Thread クラス以外にも concurrent ライブラリ（java.util.concurrent）を用いる方法がある．concurrent ライブラリでは**スレッドプール**の概念を導入しており，Tread クラスに比べてより安全で効率的なマルチスレッドの扱いができる．

スレッドプールとは，複数のスレッドを管理する機構であり，扱うスレッド数の上限の設定や，終了したスレッドの再利用などを行う．concurrent ライブラリでは，以下で説明する Executor や ExecutorService といったインターフェースを用いてスレッドプールを扱う．

5.2.1 Executor と ExecutorService

Executor インターフェース（java.util.concurrent.Executor）でマルチスレッドプログラミングを実現するための手順は次の通り．

(1) スレッドプール用の Executor オブジェクトの生成

(2) 上で生成した Executor オブジェクトに対して **execute** メソッドでスレッドを登録

スレッドとして実行するのは先に説明した **Runnable オブジェクト** などである．また execute メソッドの戻り値はない（void）．

次に示すサンプルプログラム CncrntTest1.java を例に挙げて具体的な事柄について説明する．

プログラム：CncrntTest1.java

```
1  import java.util.concurrent.Executor;
2  import java.util.concurrent.Executors;
3
4  class MyRunnable implements Runnable {
5      public String name;
6      // スレッド開始メソッド
7      public void run() {
8          int i;
9          double d;
10         long b, t, w;
11         b = System.currentTimeMillis();
12         name = Thread.currentThread().getName();
13         w = (int)(2000.0*Math.random());    // ランダムな待ち時間の生成
14         try {
15             Thread.currentThread().sleep(w);
16         } catch ( Exception e ) {
17             fin();
18         }
19         t = System.currentTimeMillis();
20         d = (long)(t - b) / 1000.0;
21         fin();
22     }
23     // 終了報告
24     public void fin() {
25         System.out.print(name+",");
26     }
27 }
28 // メイン
29 class CncrntTest1 {
30     public static void main( String argv[] ) throws Exception {
31         int N = 8;
32         Executor ex = Executors.newFixedThreadPool(N);
33         for ( int c = 0; c < 40; c++ ) {
34             ex.execute( new MyRunnable() );
35         }
36         System.out.println("main ended.");
37     }
38 }
```

このプログラムで定義している MyRunnable クラスは Runnable インターフェースを実装したものであり，実行するスレッドとなるものである．処理内容は「2 秒以内の待ち時間の後に，メッセージ（スレッド名）を出力して終了する」というものである．

スレッド実行のためのスレッドプールは Executor オブジェクトとして 32 行目で生成している．このとき，**Executors クラス**（java.util.concurrent.Executors）の newFixedThreadPool メソッドを使用している．スレッドプール生成のためのメソッドは newFixedThreadPool 以外にもあり（表 44），生成するスレッドプールの種類を選ぶことができる．

表 44: スレッドプールを生成するメソッド（一部）

メソッド	解説
newFixedThreadPool(N)	固定数（int N）のスレッドを再利用するスレッドプールを生成する．
newCachedThreadPool( )	必要に応じて新規スレッドを生成し，既存のスレッドの再利用も行うスレッドプールを生成する．短時間で終了する多数のスレッドを実行する場合に向いている．

この他にも，特定のプロセッサに向けたスレッドプールを生成するものなど多数存在する．

プログラム CncrntTest1.java では同時に実行するスレッド数を 8 として（31 行目）いる．また 33〜35 行目の反復で合計 40 個のスレッドを execute メソッドで起動している．

このプログラムでは多数のスレッドを実行した直後に main メソッドが終了するが，その後もスレッドプールは稼働し続けるため，プログラムは終了しない．このプログラムを実行した例を次に示す．

```
main ended.
pool-1-thread-2,pool-1-thread-7,pool-1-thread-8,pool-1-thread-3,pool-1-thread-5,
pool-1-thread-4,pool-1-thread-8,pool-1-thread-4,pool-1-thread-1,pool-1-thread-3,
pool-1-thread-6,pool-1-thread-2,pool-1-thread-3,pool-1-thread-1,pool-1-thread-6,
pool-1-thread-6,pool-1-thread-5,pool-1-thread-7,pool-1-thread-4,pool-1-thread-4,
pool-1-thread-8,pool-1-thread-2,pool-1-thread-5,pool-1-thread-3,pool-1-thread-1,
pool-1-thread-6,pool-1-thread-2,pool-1-thread-7,pool-1-thread-4,pool-1-thread-8,
pool-1-thread-4,pool-1-thread-6,pool-1-thread-2,pool-1-thread-5,pool-1-thread-7,
pool-1-thread-1,pool-1-thread-3,pool-1-thread-6,pool-1-thread-4,pool-1-thread-8,
```

末尾に番号を伴うスレッド名が表示されている．40 個全てのスレッドが終了した後もスレッドプールは稼働しており，プログラムは終了しないので，強制終了などの方法（ Ctrl+C の打鍵など）でプログラムを終了させる．

次に，Executor インターフェースとは別の **ExecutorService インターフェース**を用いたスレッドの扱いについて説明する．ExecutorService を用いる方法は Executor の場合とよく似ており，表 44 にあるような Executors クラスのメソッドによってスレッドプールを生成する．また，スレッドプールへのスレッドの登録は submit メソッドを使用する．このメソッドの戻り値は Future オブジェクト（後述）である．

ExecutorService を用いて先と同様の動作を実現するプログラムの例を CncrntTest2.java に示す．

プログラム：CncrntTest2.java

```
1  import java.util.concurrent.ExecutorService;
2  import java.util.concurrent.Executors;
3
4  class MyRunnable implements Runnable {
5      public String name;
6      // スレッド開始メソッド
7      public void run() {
8          int i;
9          double d;
10         long b, t, w;
11         b = System.currentTimeMillis();
12         name = Thread.currentThread().getName();
13         w = (int)(2000.0*Math.random());    // ランダムな待ち時間の生成
14         try {
15             Thread.currentThread().sleep(w);
16         } catch ( Exception e ) {
```

```
17                fin();
18            }
19            t = System.currentTimeMillis();
20            d = (long)(t - b) / 1000.0;
21            fin();
22        }
23        // 終了報告
24        public void fin() {
25            System.out.print(name+",");
26        }
27    }
28    // メイン
29    class CncrntTest2 {
30        public static void main( String argv[] ) throws Exception {
31            int N = 8;
32            ExecutorService ex = Executors.newFixedThreadPool(N);
33            for ( int c = 0; c < 40; c++ ) {
34                ex.submit( new MyRunnable() );
35            }
36            ex.shutdown();
37            System.out.println("main ended.");
38        }
39    }
```

プログラムの 32 行目でスレッドプールを生成して ExecutorService オブジェクト ex としており，34 行目で submit メソッドによってスレッドを登録している．33〜35 行目の反復で 40 個のスレッドを登録した後，ExecutorService オブジェクトに対して shutdown メソッドを実行しているが，これにより新規のスレッドを ExecutorService オブジェクトに対して登録することができなくなり，全てのスレッドが終了した時点でスレッドプールの稼働が終了する．従ってこの CncrntTest2.java では，全スレッドが終了した時点でプログラムの動作が終了する．

5.2.2 Callable インターフェースと Future インターフェース

Runnable で実装されたスレッドを execute メソッドで実行した場合，処理の終了後は値を返さない（void 型）が，ここで説明する **Callable インターフェース** を実装したスレッドを submit メソッドで実行すると，処理の結果を **Future** オブジェクトとして返す．すなわち Callable のスレッドは値を返すことができる．

submit によって実行されるスレッドは，その処理がいつ終わるかは予めわからない．従ってスレッドが返す値もいつ得られるかは予めわからない．このような処理を**非同期処理**（非同期計算）と呼ぶ．Future インターフェースは非同期処理の結果を受け取るための機構である．

Callable を実装したクラスには call メソッドを定義しておく．そしてそのクラスのオブジェクトがスレッドとして登録されるとその call メソッドを実行する形でスレッドの処理が行われる．以上のことに関してサンプルプログラム CncrntTest3_1.java を挙げて説明する．

プログラム：CncrntTest3_1.java

```
1   import java.util.concurrent.ExecutorService;
2   import java.util.concurrent.Executors;
3   import java.util.concurrent.Callable;
4   import java.util.concurrent.Future;
5
6   class MyCallable implements Callable<Integer> {
7       int n;      // 値を受け取るための変数（メンバ）
8       // コンストラクタ
9       MyCallable( int m ) {   // スレッド生成時に引数を受け取って…
10          n = m;              // それをクラスのメンバに設定する．
11      }
12      // スレッド開始メソッド
13      public Integer call() {
14          int r = n*n;        // 2乗の計算
15          try {
16              Thread.currentThread().sleep(2000);     // 終了まで2秒待機
17          } catch(Exception e) {
18              System.out.println("スレッドのエラー");
```

```
19          }
20          return( r );    // 計算結果を返す
21      }
22  }
23  // メイン
24  class CncrntTest3_1 {
25      public static void main( String argv[] ) throws Exception {
26          // スレッドプールの生成
27          ExecutorService ex = Executors.newCachedThreadPool();
28          // スレッドに値を与えて実行し，未だ得られていない戻り値を
29          // Futureオブジェクトで受信準備（スレッド終了時に実際の値が得られる）
30          Future<Integer> fut = ex.submit( new MyCallable(16) );
31          ex.shutdown();  // スレッドが終了するのを待つ
32          System.out.println("ExecutorService ended.");
33          // この時点でFutureオブジェクトに値が得られているので，それを実際に
34          // 取り出す.
35          int r = fut.get();
36          System.out.println("Value of Future:"+r);   // 結果表示
37      }
38  }
```

このプログラム中で定義されている MyCallable クラスのインスタンスはスレッドとして実行される．具体的な処理としては，与えられた整数を 2 乗し，2 秒後に計算結果を返す．これを行うのが call メソッドである．

27 行目でスレッドプール ex を生成し，30 行目で submit メソッドによりスレッドを生成（引数に整数値の 16 を与えている）して登録している．また，いずれ得られる戻り値を Future オブジェクト fut で受け取る準備をしている．

このプログラムではスレッドプールにスレッドを 1 つだけ登録し，31 行目の shutdown メソッドでスレッドプールの終了準備に入っている．スレッドプールには submit メソッドを用いて次々とスレッドを登録できるが，shutdown メソッドを実行するとそれ以後のスレッド登録はできなくなり，実行中のスレッドが全て終了するのを待つ．

shutdown メソッドが終了した後は，仕掛けておいた Future オブジェクトに実際に値が得られている．Future オブジェクトに得られた値を取り出すには get メソッドを使用（35 行目）する．

このプログラムを実行した結果の例を次に示す．

```
ExecutorService ended.
Value of Future:256
```

スレッド生成時に引数として与えた値である 16 を 2 乗した結果（256）が表示されている．

サンプルプログラム CncrntTest3_1.java では，スレッドプールの生成に newCachedThreadPool メソッドを使用している．この形で生成されたスレッドプールは，実行時間の短いスレッドを大量に処理するのに適している．このプログラムを拡張した CncrntTest3_2.java では，100 個のスレッドをスレッドプールに登録してそれらを同時に実行している．これら 100 個のスレッドは全て非同期に値を返す（いつ値が得られるか判らない）が，それらを受け付けるための 100 個の Future オブジェクトを ArrayList の要素として保持している．

プログラム：CncrntTest3_2.java

```
1   import java.util.concurrent.ExecutorService;
2   import java.util.concurrent.Executors;
3   import java.util.concurrent.Callable;
4   import java.util.concurrent.Future;
5   import java.util.ArrayList;
6
7   class MyCallable implements Callable<Integer> {
8       int n;      // 値を受け取るための変数（メンバ）
9       // コンストラクタ
10      MyCallable( int m ) {   // スレッド生成時に引数を受け取って…
11          n = m;              // それをクラスのメンバに設定する.
12      }
13      // スレッド開始メソッド
14      public Integer call() {
15          int r = n*n;        // 2乗の計算
```

```
16          try {
17              Thread.currentThread().sleep(2000);     // 終了まで2秒待機
18          } catch(Exception e) {
19              System.out.println("スレッドのエラー");
20          }
21          return( r );    // 計算結果を返す
22      }
23  }
24  // メイン
25  class CncrntTest3_2 {
26      public static void main( String argv[] ) throws Exception {
27          // 同時に実行するスレッドの数を与えてスレッドプールを生成
28          int N = 100;
29          ExecutorService ex = Executors.newCachedThreadPool();
30          // スレッドの戻り値（複数個）を受け取るためのFutureオブジェクトの
31          // リストを作成しておく
32          ArrayList<Future<Integer>> arr = new ArrayList<Future<Integer>>();
33          // スレッドの起動と，各スレッドの戻り値を受信するための
34          // Futureオブジェクトをリストにしておく
35          for ( int c = 0; c < N; c++ ) {
36              arr.add( ex.submit( new MyCallable(c) ) );
37          }
38          ex.shutdown();      // 全スレッドが終了するのを待つ
39          System.out.println("ExecutorService ended.");
40          // リストの要素の各Futureから値を取り出す
41          for ( int c = 0; c < N; c++ ) {
42              System.out.print(arr.get(c).get()+",");
43          }
44          System.out.println("\nmain ended.");
45      }
46  }
```

全てのスレッドの実行が終了すると，ArrayList の全ての要素に計算結果が得られるので，それを標準出力に出力（41〜43 行目）している．

このプログラムを実行した結果の例を次に示す．

```
ExecutorService ended.
0,1,4,9,16,25,36,49,64,81,100,121,144,169,196,225,256,289,324,361,400,441,484,529,576,625,
676,729,784,841,900,961,1024,1089,1156,1225,1296,1369,1444,1521,1600,1681,1764,1849,1936,
2025,2116,2209,2304,2401,2500,2601,2704,2809,2916,3025,3136,3249,3364,3481,3600,3721,3844,
3969,4096,4225,4356,4489,4624,4761,4900,5041,5184,5329,5476,5625,5776,5929,6084,6241,6400,
6561,6724,6889,7056,7225,7396,7569,7744,7921,8100,8281,8464,8649,8836,9025,9216,9409,9604,
9801,
main ended.
```

5.3 外部プロセスの起動

Java のプログラムからオペレーティングシステム（OS）のプロセス（**外部プロセス**）を起動するには ProcessBuilder クラス（java.lang.ProcessBuilder）と Process クラス（java.lang.Process）を用いる．

5.3.1 Java プログラムと外部プロセスの関係

外部プロセスは OS が扱う**プロセス**のことであり，これは実行形式のプログラムファイルを起動することで開始する．この際に必要となるのは，起動する実行形式のプログラム（コマンドファイル）のパスと起動時に与える引数である．これら必要な情報は**プロセス属性**と呼ばれ，これを保持するのが ProcessBuilder オブジェクトである．

実行に必要なプロセス属性を持った ProcessBuilder オブジェクトに対して start メソッドを実行することで OS 上のプロセスが起動する．この後，その外部プロセスは Java プログラム側では Process オブジェクトとして管理される．start メソッドで外部プロセスを開始するに当たって，必要となる環境変数の情報を与えることができる．

実行を開始した外部プロセスの標準入出力と標準エラー出力は，入出力ストリームの形で Java プログラムと接続（図 54）することができる．

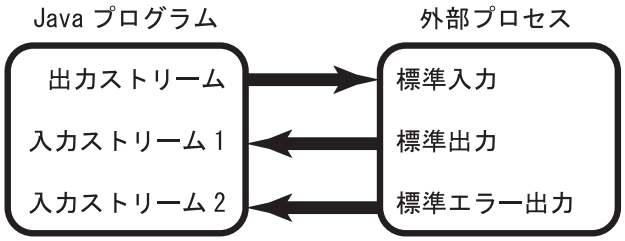


図 54: Java プログラムと外部プロセスの入出力の接続

5.3.2 使用方法

■ ProcessBuilder オブジェクトの生成

プロセス属性（コマンドファイルのパス，実行時の引数）を ProcessBuilder オブジェクトのコンストラクタに与える．

例. `ProcessBuilder pb = new ProcessBuilder("myCommand", "Arg1", "Arg2");`

”myCommand”というファイルとして存在する実行形式プログラムを，”Arg1”, ”Arg2” という引数を与えて起動するための ProcessBuilder オブジェクトを pb として生成する．

■ 環境変数の取得と設定

生成した ProcessBuilder オブジェクトに対して environment メソッドを実行することで，外部プロセス実行のための環境変数を取得することができる．

例. `Map<String, String> env = pb.environment();`

この結果，環境変数の名前と値の組が Map ビュー env として得られる．この Map ビューに対して get メソッドや put メソッドを使用することで環境変数の値の参照や設定ができる．ここで設定した環境変数は外部プロセスからもアクセスできる..

■ 外部プロセス実行のための作業ディレクトリの設定

外部プロセス実行時の作業ディレクトリを指定するには，ProcessBuilder オブジェクトに対して directory メソッドを実行する．

例. `pb.directory(new File("myDir"));`

この結果，外部プロセス実行時の作業用のカレントディレクトリが ”myDir” に設定される．

■ 外部プロセスの実行

用意した ProcessBuilder オブジェクトに対して start メソッドを実行することで外部プロセスが実際に起動する．

例. `Process p = pb.start();`

これにより外部プロセスが起動し，そのプロセスは Process オブジェクト p として管理される．また，start メソッド実行直後に標準入出力，標準エラー出力を Java プログラム側に接続することができる．

外部プロセスの起動に失敗すると IOException 例外が発生する．

■ 標準入出力，標準エラー出力との接続

Process オブジェクトに対して getInputStream メソッド，getOutputStream メソッド，getErrorStream メソッドを実行することで，起動した外部プロセスの標準入出力，標準エラー出力を Java プログラム側の入出力ストリームとして取得することができる．

例 1. `InputStream is = p.getInputStream();`

例 2. `InputStream ise = p.getErrorStream();`

例 3. `OutputStream os = p.getOutputStream();`

この結果，外部プロセスの標準出力と標準エラー出力がそれぞれ `is`，`ise` に入力ストリームとして接続され，それらストリームからデータを受け取ることができる．また，外部プロセスの標準入力が `os` に出力ストリームとして接続され，それに対してデータを送ることができる．

■ 外部プロセスの終了に関すること

・外部プロセスの強制終了

例. `p.destroy();`

外部プロセス p を強制的に終了する．

・外部プロセス終了時の戻り値の取得

例. `p.exitValue();`

外部プロセス p の終了時の戻り値を int 型で取得する．外部プロセスが終了していない状態でこのメソッドを呼び出すと IllegalThreadStateException 例外が発生する．

・外部プロセス終了の同期（待機）

例. `p.waitFor();`

外部プロセス p が終了するまで待つ．

・外部プロセスの例外

外部プロセスの起動時，もしくは実行中に発生する例外（外部プロセスの異常終了など）は InterruptedException である．

5.3.3 サンプルプログラム

ここでは，C 言語で記述した外部のプログラムを Java から起動する例を示しながら，外部プロセスとの連携処理を行う様子を見る．

標準入力から受け取った値を定義域として，それに対する正弦関数の値域のデータを標準出力に出力するプログラムが pbtest01.c（C 言語で記述）である．

外部ププログラム：pbtest01.c

```
1  #include    <stdio.h>
2  #include    <math.h>
3
4  int main(int ac, char **av)
5  {
6      double  x, y;
7      int     c;
8
9      // 引数の受取り
10     fprintf(stderr,"%d個の引数",ac);
11     for ( c = 0; c < ac; c++ ) {
```

```
12          fprintf(stderr,":%s",av[c]);
13      }
14      fprintf(stderr,":処理を開始します\n");
15      fflush(stderr);
16
17      // 入出力処理
18      while ( -1 ) {
19          scanf("%lf",&x);
20          if ( feof(stdin) == 0 ) {
21              y = sin(x);
22              printf("%lf\n",y);
23          } else {
24              break;
25          }
26      }
27      fflush(stdout);
28
29      fprintf(stderr,"処理を終了します");
30      fflush(stderr);
31  }
```

このプログラムをコンパイルして，実行可能なプログラムのファイルが完成しているものと想定する．

このプログラムを外部プロセスとして起動する Java のプログラムが pbtest01.java である．

プログラム：pbtest01.java

```
1   import java.util.*;
2   import java.io.*;
3
4   class pbtest01 {
5       public static void main(String argv[]) {
6           // ProcessBuilderと環境変数の用意
7           ProcessBuilder pb =
8                   new ProcessBuilder("pbtest01","arg1","arg2");
9           Map<String,String> env = pb.environment();
10
11          // 環境変数の一覧表示
12          String key, val;
13          Iterator it = env.keySet().iterator();
14          System.out.println("=== 環境変数（ここから） ===");
15          while ( it.hasNext() ) {
16              key = it.next().toString();
17              val = env.get(key);
18              System.out.println(key+" : "+val);
19          }
20          System.out.println("=== 環境変数（ここまで） ===：以上"+env.size());
21
22          // 外部プロセスの実行
23          try {
24              pb.directory(new File("."));
25              Process p = pb.start();
26              // 標準入出力，標準エラー出力の接続
27              BufferedReader br =
28                      new BufferedReader(
29                              new InputStreamReader(p.getInputStream()));
30              BufferedReader bre =
31                      new BufferedReader(
32                              new InputStreamReader(p.getErrorStream()));
33              PrintWriter pw = new PrintWriter(p.getOutputStream());
34              // 外部プロセスへの送信
35              double x;
36              for ( x = 0.0; x < 6.4; x += 0.4 ) {
37                  pw.println(x);
38              }
39              pw.flush();
40              System.out.println("=== 外部プロセスへの送信完了（標準入力へ）===");
41              pw.close();
42              // 外部プロセスからの受信
43              String recv;
```

```
44                 System.out.println("=== 外部プロセスからの受信（標準出力から）===");
45                 while ( true ) {
46                     recv = br.readLine();
47                     if ( recv == null ) {
48                         break;
49                     } else {
50                         System.out.println(recv);
51                     }
52                 }
53                 System.out.println("=== 外部プロセスからの受信（標準エラーから）===");
54                 while ( true ) {
55                     recv = bre.readLine();
56                     if ( recv == null ) {
57                         break;
58                     } else {
59                         System.out.println(recv);
60                     }
61                 }
62                 System.out.println("=== 受信はここまで ===");
63                 // 終了後の処理
64                 p.waitFor();
65                 int ev = p.exitValue();
66                 System.out.println("外部プロセスの戻り値: "+ev);
67             } catch (IOException e) {
68                 System.out.println("IOException!");
69             } catch (InterruptedException e) {
70                 System.out.println("InterruptedException!");
71             }
72         }
73 }
```

解説：

7〜9 行目で ProcessBuilder と環境変数のための Map ビューを生成している．

12〜19 行目で環境変数の内容を表示している．

24〜25 行目で外部プログラムの作業ディレクトリを設定して，外部プロセスとして起動している．

27〜33 行目で外部プロセスと Java プログラムとの入出力を接続して，相互に入出力を可能にしている．

36〜41 行目で外部プロセスの標準入力に対してデータを与え，終了後に close している．

45〜52 行目で外部プロセスの標準出力からデータを受け取っている．

54〜61 行目で外部プロセスの標準エラー出力からデータを受け取っている．

外部プログラムの起動や入出力の接続に問題が発生すると例外 IOException が発生するので，try〜catch の構造になっている．

このプログラムの実行結果の例を次に示す．

```
=== 環境変数（ここから） ===
 USERDOMAIN_ROAMINGPROFILE : DESKTOP-NS027TL
 PROCESSOR_LEVEL : 6
 （表示が多いので省略）
=== 環境変数（ここまで） ===：以上 52
=== 外部プロセスへの送信完了（標準入力へ）===
=== 外部プロセスからの受信（標準出力から）===
 0.000000
 0.389418
 （表示が多いので省略）
-0.631267
-0.279415
=== 外部プロセスからの受信（標準エラーから）===
 3 個の引数:pbtest01:arg1:arg2:処理を開始します
処理を終了します
=== 受信はここまで ===
外部プロセスの戻り値:  0
```

6　ラムダ式，Stream API，メソッド参照

ラムダ式[37] は，オブジェクトやオブジェクトの列から別のオブジェクトを算出するための簡潔な表現である．

【ラムダ式の書き方】

■ **1 つのオブジェクトから別のオブジェクトを算出する記述**

仮引数 -> 計算式

■ **複数のオブジェクトから別のオブジェクトを算出する記述**

(仮引数 1, 仮引数 2, … , 仮引数 n) -> 計算式

■ **オブジェクトを算出する手続きの記述**

仮引数（あるいは仮引数の列） -> {
（手続きの記述）
return(戻り値);
}

このように「->」という記号を用いて元のオブジェクト（あるいは列）と算出結果のオブジェクトを対応させる．ラムダ式は**関数型インターフェース**のメソッドを定義する際に使うことができる．

6.1　関数型インターフェース

関数型インターフェースとはメソッドを 1 つだけ持つインターフェースである．関数型インターフェースのメソッドの実装にラムダ式を用いる例をサンプルプログラム FuncItf01.java で示す．

プログラム：FuncItf01.java

```
1  @FunctionalInterface
2  interface Func1arg {
3      public int calc( int n );
4  }
5
6  @FunctionalInterface
7  interface Func2args {
8      public int plus( int n1, int n2 );
9  }
10
11 class FuncItf01 {
12     public static void main( String argv[] ) {
13         // 関数型インターフェースの実装(1)
14         Func1arg f = new Func1arg() {
15             public int calc( int n ) {
16                 return( 2*n );
17             }
18         };
19         System.out.println( "(1) 2*123："+f.calc(123) );
20
21         // 関数型インターフェースの実装(2)
22         Func1arg g = n -> 2*n;
23         System.out.println( "(2) 2*321："+g.calc(321) );
24
25         // 関数型インターフェースの実装(3)
26         Func2args h = (n1,n2) -> n1+n2;
27         System.out.println( "(3) 11+22："+h.plus(11,22) );
28
29         // 関数型インターフェースの実装(4)
30         Func1arg j = n -> {
31             int c, r = 0;
32             for ( c = 1; c <= n; c++ ) {
33                 r += c;
34             }
```

[37] Java SE 8 から導入された構文．

```
35                 return( r );
36             };
37             System.out.println( "(4) sum of 1 to 100："+j.calc(100) );
38         }
39 }
```

このプログラムでは関数型インターフェース Func1arg，Func2args を定義している．これらインターフェースの宣言の直前で @FunctionalInterface というアノテーションを記述しているが，これは直後のインターフェースの定義が関数型インターフェースであることを示すものである．

14〜18 行目では Func1arg インターフェースの calc メソッドの定義を通常の形式で記述（オーバーライド）しているが，これと同じ機能を持つメソッドの定義を 22 行目ではラムダ式を用いて 1 行だけで記述している．同様に 26 行目では Func2args インターフェースの plus メソッドの定義をラムダ式を用いて 1 行で記述している．

30〜36 行目では複数行に渡る手続きをラムダ式で記述した例である．

このプログラムを実行した例を次に示す．

```
(1) 2*123：246
(2) 2*321：642
(3) 11+22：33
(4) sum of 1 to 100：5050
```

関数型インターフェースは各種のライブラリやフレームワークで応用されていることが多く，それらを利用する場合はラムダ式の応用でプログラムの可読性が向上する場合がある．

6.2 Stream インターフェース

複数の要素に対する順次的処理や並列的処理を効率的に実現するための Stream インターフェース（java.util.stream.Stream）がある．Stream オブジェクトは要素に対して一斉処理を施すためのものでり，例えば

```
1, 2, 3, 4
```

という Stream があった場合，各要素に対して「2 倍する」という処理を施すと，

```
2, 4, 6, 8
```

という Stream になる．更にこの Stream の各要素に ‘System.out.print’ という処理を施すと全ての要素が標準出力に出力される．この処理をサンプルプログラム で示す．

プログラム：Stream0.java

```
1  import java.util.stream.Stream;
2
3  class Stream0 {
4      public static void main( String argv[] ) {
5          // Streamの作成
6          Stream<Integer> st = Stream.of(1,2,3,4);
7
8          // 各要素に対する処理
9          st.map( e -> 2*e ).forEach( e -> System.out.print(e+",") );
10     }
11 }
```

このプログラムの 6 行目で Stream オブジェクト st を作成している．9 行目に map というメソッドがあるが，これが各要素に対する一斉処理を施した結果を得るものであり「各要素を 2 倍する」処理を行った結果の Stream オブジェクトを返す．また forEach メソッドも各要素に対して一斉処理を施すものであり「各要素を標準出力に出力する」処理を行う．

このプログラムを実行すると

```
2,4,6,8,
```

と表示される．メソッドの詳細について以下で説明する．

6.2.1 Stream オブジェクトの生成

先のプログラム Stream0.java では Stream オブジェクトを宣言する際に

```
Stream<Integer> st = Stream.of(1,2,3,4)
```

としているが，要素の型を総称型（Generics）の表記で指定する．

Stream オブジェクトを生成には下記のようないくつかの方法がある．

■ Stream.of メソッドによる方法

書き方：　Stream.of(要素 1, 要素 2, … , 要素 n)

of メソッドの引数にストリームの要素を書き並べる．

■ 配列から Stream オブジェクトに変換する方法

書き方：　Arrays.stream(配列)

stream は Arrays クラス（java.util.Arrays）のクラスメソッドである．

■ Collection を Stream オブジェクトに変換する方法

書き方：　Collection オブジェクト.stream( )

リストやセットといった Collection のオブジェクトに対して stream メソッドを実行することで Stream オブジェクトが得られる．

6.2.2 forEach メソッド

forEach メソッドは，引数に与えられた処理を Stream の全ての要素に対して実行する．このメソッドは値を返さない．(void)

forEach メソッドの引数には Consumer インターフェース（java.util.function.Consumer）のオブジェクトを与える．Consumer オブジェクトとしては，関数型インターフェースのオブジェクトやラムダ式で与える．先のプログラム Stream0.java では，forEach メソッドの引数にラムダ式を与えている．

6.2.3 filter メソッド

filter メソッドは，指定した条件を満たす要素を抽出し，それを Stream オブジェクトとして返す．filter メソッドの引数には Predicate インターフェース（java.util.function.Predicate）のオブジェクトを与える．Predicate オブジェクトとしては，関数型インターフェースのオブジェクトやラムダ式で与える．

filter メソッドを使用したサンプルプログラム Stream1.java を示す．

プログラム：Stream1.java

```
1  import java.util.stream.Stream;
2
3  class Stream1 {
4      public static void main( String argv[] ) {
5          // Streamの作成
6          Stream<Integer> st = Stream.of(1,2,3,4,5,6,7,8,9);
7
8          // 3の倍数を抽出する処理
9          st.filter( e -> e%3==0 ).forEach( e -> System.out.print(e+",") );
10     }
11 }
```

このプログラムの 9 行目で，要素に対する判定処理を表現するラムダ式を filter メソッドの引数に与えている．このプログラムを実行すると次のように表示される．

```
3,6,9,
```

6.2.4 map メソッド

map メソッドは，引数に与えられた処理を Stream の全ての要素に対して実行し，その結果を Stream オブジェクトとして返す．map メソッドの引数には Function インターフェース（java.util.function.Function）のオブジェクトを与える．Function オブジェクトとしては，関数型インターフェースのオブジェクトやラムダ式で与える．先のプログラム Stream0.java では，map メソッドの引数にラムダ式を与えている．

6.2.5 sorted メソッド

sorted メソッドは，要素の順序を整列し，それを Stream オブジェクトとして返す．sorted メソッドの引数には Comparator インターフェース（java.util.Comparator）のオブジェクトを与える．Comparator オブジェクトとしては，関数型インターフェースのオブジェクトやラムダ式で与える．

sorted メソッドの引数は省略できる場合がある．

sorted メソッドを使用したサンプルプログラム Stream2.java を示す．

プログラム：Stream2.java

```
1  import java.util.stream.Stream;
2
3  class Stream2 {
4      public static void main( String argv[] ) {
5          // Streamの作成
6          Stream<Integer> st = Stream.of(6,8,2,1,5,3,7,9,4);  // 乱雑な順序
7
8          // 整列処理
9          System.out.print("整列：");
10         st.sorted().forEach( e -> System.out.print(e+",") );
11
12         // 逆順の整列処理
13         System.out.print("\n逆順：");
14         st = Stream.of(6,8,2,1,5,3,7,9,4);  // Streamを再度生成
15         st.sorted( (k1,k2) -> k2-k1 ).forEach( e -> System.out.print(e+",") );
16     }
17 }
```

このプログラムの 15 行目では，Comparator としてラムダ式（逆順になる式）を引数に与えている．このプログラムを実行すると次のように表示される．

```
整列：1,2,3,4,5,6,7,8,9,
逆順：9,8,7,6,5,4,3,2,1,
```

6.2.5.1 Stream のクローズ（終了）

Stream オブジェクトは一連の処理が終わるとクローズされ，以後は使用できない．先のプログラム Stream2.java では 10 行目の処理で Stream は一旦終了してクローズされるので，14 行目で再度 Stream を生成している．

6.2.6 collect メソッド

Stream を Collection のオブジェクト（リストや連想辞書（Map）など）に変換するメソッドとして collect がある．

書き方：　Stream オブジェクト.collect(コレクタ)

コレクタは Collector インターフェース（java.util.stream.Collector）のオブジェクトであり，Stream の要素を蓄積する処理を意味するものである．コレクタを生成するには Collectors クラス（java.util.stream.Collectors）の各種メソッドを実行する．例えば，Stream を List インターフェース（java.util.List）のオブジェクトに変換するためのコレクタは

Collectors.toList()

として生成[38] する．

[38]他にも Set オブジェクトに変換するコレクタを生成する toSet や，Map オブジェクトに変換するコレクタを生成する toMap がある．

これらメソッドを使用して Stream オブジェクトを List オブジェクトに変換するサンプルプログラム Stream3.java を示す.

プログラム：Stream3.java

```
1  import java.util.stream.Stream;
2  import java.util.stream.Collectors;
3  import java.util.List;
4
5  class Stream3 {
6      public static void main( String argv[] ) {
7          // Streamの作成
8          Stream<Integer> st = Stream.of(1,2,3,4,5,6,7,8,9);
9          // Streamからリストへの変換
10         List<Integer> ar = st.collect( Collectors.toList() );
11         for ( int n : ar ) {
12             System.out.print(n+",");
13         }
14         System.out.println("\n得られたリストのクラス："+ar.getClass().getName());
15     }
16 }
```

10 行目で Stream オブジェクトを List オブジェクトに変換している.

このプログラムを実行した例を次に示す.

```
1,2,3,4,5,6,7,8,9,
得られたリストのクラス：java.util.ArrayList
```

6.3 メソッド参照

関数型インターフェースのオブジェクトやラムダ式と同様の扱いができる表現に**メソッド参照**（method reference）がある．これは

クラス::メソッド

あるいは,

インスタンス::メソッド

というコロン 2 つ「::」を用いた表記で メソッドそのもの を表す.

メソッド参照に関するサンプルプログラム を示す.

プログラム：MethRef01.java

```
1  @FunctionalInterface
2  interface MyInterface {
3      public void writeString( String s );
4  }
5
6  class MethRef01 {
7      public static void main( String argv[] ) {
8          MyInterface f = System.out::println;
9          f.writeString("メソッド参照でprintlnの機能を利用しています. ");
10     }
11 }
```

このプログラムの 8 行目にメソッド参照の記述（System.out クラスの println メソッドを意味する）がある．これはラムダ式を用いて

```
MyInterface f = e -> System.out.println(e);
```

と記述した場合と同じ働きを実現する.

メソッド表記に関するプログラムをもう 1 例示す.

プログラム：MethRef02.java

```
1  import java.util.stream.Stream;
2
3  class MyClass {
4      public static void writeString( String s ) {     // 静的メソッド
5          System.out.print( s+"," );
6      }
7      public void writeMessage( String s ) {           // インスタンスメソッド
8          System.out.print( s+"," );
9      }
10 }
11
12 class MethRef02 {
13     public static void main( String argv[] ) {
14         // Streamの作成
15         Stream<String> st = Stream.of("a","b","c","d","e","f","g");
16         System.out.print("処理1：");
17         // 各要素に対する処理（静的メソッド）
18         st.forEach( MyClass::writeString );
19
20         // Streamの作成（再度）
21         st = Stream.of("a","b","c","d","e","f","g");
22         System.out.print("\n処理2：");
23         // MyClassのインスタンス生成
24         MyClass mc = new MyClass();
25         // 各要素に対する処理（インスタンスメソッド）
26         st.forEach( mc::writeMessage );
27     }
28 }
```

このプログラムの 18 行目では，独自に定義したクラス `MyClass` の静的メソッド `writeString` を Stream の処理に用いている．これはラムダ式を用いて

```
st.forEach( e -> System.out.print(e+",") );
```

と記述した場合と同じ働きを実現する．同様に 26 行目では `MyClass` のインスタンスメソッド `writeMessage` を Stream の処理に用いている．

このプログラムを実行した例を次に示す．

```
処理 1：a,b,c,d,e,f,g,
処理 2：a,b,c,d,e,f,g,
```

7　正規表現

文字列のパターンを表現するものに**正規表現**がある．正規表現は ある規則 に則って記述されるものである．例えば ‘[0-9]’ は正規表現によるパターンの 1 つであり「‘0’〜‘9’ までの数字の記号」を表す．また ‘+’ は「その直前のパターンで表現される文字が 1 回以上連続して並んでいる」ことを意味する．この 2 つの表現を次のように組み合わせ

‘[0-9]+’

と記述すると「数字の記号が並んだ文字列」を意味する表現となる．

正規表現を応用すると，与えられた文字列がある形式になっているかを判定（**パターンマッチ**）したり,「ある規則に沿った文字列」を文書データの中から**探索**（**検索**）する，あるいはパターンにマッチする部分を別の文字列に**置換**するといった処理が実現できる．

正規表現を扱うためのクラスには Pattern クラス（java.util.regex.Pattern）と Matcher クラス（java.util.regex.Matcher）がある．

文字列探索やパターンマッチの処理は，**テキスト**（対象文字列，文書データ）の中から**探索キー**を見つけ出す処理を基本とする．具体的には，**テキスト**は文字列（String），正規表現による探索キーは Pattern オブジェクトとして扱う．上記のクラスライブラリを用いた処理は次のような手順を踏む．

1) 正規表現の文字列を Pattern オブジェクトに変換する．

2) 上で作成した Pattern オブジェクトとテキスト（文書データ）から Matcher オブジェクトを作成する．

3) 上で作成した Matcher オブジェクトを用いて探索や置換を実行する．

以下ではサンプルプログラムを提示する形で具体的な方法について説明する．

7.1　文字列探索：　find メソッド

先に示した正規表現 ‘[0-9]+’ を用いて，対象文字列の中に数字列があるかどうかを判定するプログラム RegEx0.java を示す．

プログラム：RegEx0.java

```
1  import java.util.regex.Pattern;
2  import java.util.regex.Matcher;
3
4  class RegEx0 {
5      public static void main( String argv[] ) {
6          String  str1 = "abcdefghijklmn",     // 数字を含まない文字列
7                  str2 = "abcd836912klmn",     // 数字を含む文字列
8                  ptn0 = "[0-9]+";             // 数字列を表現する正規表現の文字列
9          // 正規表現のパターンオブジェクトを生成
10         Pattern ptn = Pattern.compile( ptn0 );
11         // str1の検査
12         Matcher m = ptn.matcher(str1);       // ptnとstr1のマッチ処理
13         if ( m.find() ) {
14             System.out.println(str1+"は数字列を含みます．");
15         } else {
16             System.out.println(str1+"は数字列を含みません．");
17         }
18         // str2の検査
19         m = ptn.matcher(str2);       // ptnとstr2のマッチ処理
20         if ( m.find() ) {
21             System.out.println(str2+"は数字列を含みます．");
22         } else {
23             System.out.println(str2+"は数字列を含みません．");
24         }
25     }
26 }
```

数字列が含まれない文字列 str1 と数字列を含む文字列 str2 がある．また数字列を表す正規表現が ptn0 であり，これから Pattern オブジェクト ptn を作成している．更に ptn を用いて探索処理をするために，Matcher オブジェクト m を作成している．この段階で文字列探索のための準備ができたことになる．

作成した m に対して find メソッドを実行すると探索結果が真理値（boolean）として得られる．
RegEx0.java を実行した例を次に示す．

```
abcdefghijklmn は数字列を含みません.
abcd836912klmn は数字列を含みます.
```

7.1.1 パターンの検出位置

find メソッドの実行結果が true の場合は，探索対象の文字列中にパターンが含まれている．この場合，Matcher オブジェクトに対して更に start，end ，group メソッド（表 45）によってパターンの検出位置などに関する情報が得られる．

表 45: パターン検出後に使えるメソッド（一部）

メソッド	解説
start()	パターン検出の開始位置（対象文字列内のインデックス）
start(pos)	対象文字列の中の検索を始める位置のインデックス（int 型）を引数に与えることができる．
end()	パターン検出の終了位置（対象文字列内のインデックス+1）
group()	パターンにマッチする部分の文字列（String）を取得する．
group(n)	パターンマッチで複数の部分を同時に取り出す際は，引数に取り出す部分の番号（整数値） n を与える

これらメソッドを使用するサンプルプログラム RegEx1.java を示す．

プログラム：RegEx1.java

```
1  import java.util.regex.Pattern;
2  import java.util.regex.Matcher;
3
4  class RegEx1 {
5      public static void main( String argv[] ) {
6          String  str2 = "abcd836912klmn",    // 数字を含む文字列
7                  ptn0 = "[0-9]+";            // 数字列を表現する正規表現の文字列
8          // 正規表現のパターンオブジェクトを生成
9          Pattern ptn = Pattern.compile( ptn0 );
10         // str2の検査
11         Matcher m = ptn.matcher(str2);      // ptnとstr2のマッチ処理
12         if ( m.find() ) {
13             System.out.println(str2+"は数字列を含みます. ");
14             System.out.print("start："+m.start()+", "); //
15             System.out.print("end："+m.end()+", ");
16             System.out.println("group："+m.group());
17         } else {
18             System.out.println(str2+"は数字列を含みません. ");
19         }
20     }
21 }
```

このプログラムの実行結果の例を次に示す．

```
abcd836912klmn は数字列を含みます.
start：4, end：10, group：836912
```

7.1.2 パターン検出の続行

find メソッドでは対象文字列中で，パターンに最初にマッチするものを検出しようとする．対象文字列中にマッチするパターンが複数ある場合は find メソッドの引数に，探索開始位置のインデックス（整数）を与える形で探索を続行する．（サンプルプログラム RegEx2.java 参照）

7.2 置換処理： replaceFirst，replaceAll

対象文字列の中で正規表現のパターンがマッチする部分の文字列を別の文字列に置換するには replaceFirst，replaceAll メソッド（表 46）を Matcher オブジェクトに対して実行する．

表 46: 置換メソッド

メソッド	解説
replaceFirst(置換文字列)	対象文字列の中で最初にパターンにマッチする部分を「置換文字列」に置換えたもの（String）を返す．
replaceAll(置換文字列)	対象文字列の中でパターンにマッチする全ての部分を「置換文字列」に置換えたもの（String）を返す．

次に示すプログラム RegEx2.java は，数字列を複数含む文字列 str2 の中の数字列の部分を順番に取り出す処理と，replaceFirst，replaceAll による置換処理を例示するものである．

プログラム：RegEx2.java

```
1  import java.util.regex.Pattern;
2  import java.util.regex.Matcher;
3
4  class RegEx2 {
5      public static void main( String argv[] ) {
6          String  str2 = "abc123def456ghi789jkl",      // 数字列を複数含む文字列
7                  ptn0 = "[0-9]+";             // 数字列を表現する正規表現の文字列
8          // 正規表現のパターンオブジェクトを生成
9          Pattern ptn = Pattern.compile( ptn0 );
10         // str2の検査
11         Matcher m = ptn.matcher(str2);       // ptnとstr2のマッチ処理
12         int c = 1, pos = 0;
13         while ( true ) {
14             if ( m.find(pos) ) {    // パターンが見つかった場合
15                 System.out.print(c+") start："+m.start()+", ");
16                 System.out.print("end："+m.end()+", ");
17                 System.out.println("group："+m.group());
18                 pos = m.end();       // 次の探索開始位置
19                 c++;
20             } else {    // パターンが見つからなかった場合
21                 break;       // 終了
22             }
23         }
24         // 数字列を「数字」に置換する
25         String r1 = m.replaceFirst("数字");
26         String r2 = m.replaceAll("数字");
27         System.out.println("最初にマッチしたものを置換："+r1);
28         System.out.println("マッチするものを全て置換："+r2);
29     }
30 }
```

このプログラムの実行結果の例を次に示す．

```
1) start：3, end：6, group：123
2) start：9, end：12, group：456
3) start：15, end：18, group：789
最初にマッチしたものを置換：abc 数字 def456ghi789jkl
マッチするものを全て置換：abc 数字 def 数字 ghi 数字 jkl
```

str2 の中の数字列の部分を「数字」という文字列に置き換える処理を 2 種類実行している．

7.3 パターンの記述方法

パターンを構成するための基本的な要素について説明する．まず 1 文字分のパターンの表記法を表 47 に示す．

表 47: 正規表現のパターン（一部）

パターン	解説
[$c_1c_2c_3\cdots$]	特定の文字の集合 $c_1c_2c_3\cdots$　（これらの内のどれかに該当）
[c_1-c_2]	c_1〜c_2 の間に含まれる文字．文字の範囲をハイフン '-' でつなげる．　例: [a-d] → 'a','b','c','d' のどれか
[^文字の集合]	「文字の集合」にマッチしないもの
.（ドット）	任意の 1 文字
¥d	数字（ [0-9] と同じ）
¥D	数字以外
¥s	空白文字（タブ，改行文字なども含む）
¥S	空白文字以外（¥s でないもの）
¥w	アンダースコアを含む半角英数字（ [a-zA-Z0-9_] と同じ）
¥W	半角英数字以外（¥w でないもの）

注意）円記号「¥」は端末装置によってはバックスラッシュ「\」として表示される．

正規表現のパターンではない文字（列）は，それ自身がパターンとなる．例えば 'Apple' と記述したパターンは 'Apple' という文字列にマッチする．

「繰り返し」の基本的なものを表 48 に挙げる．

表 48: 繰り返しの表記（一部）

表記	解説	表記	解説
+	1 回以上	*	0 回以上
?	0 回かもしくは 1 回	{m,n}	m 回以上 n 回以下（回数が大きい方を優先）
{n}	n 回	{m,}	m 回以上

例. '¥d+' は数字列にマッチする．

対象文字列のパターン照合の位置を表す表現を表 49 に示す．

表 49: パターンマッチの位置の指定

表記	解説	表記	解説
^	行頭でのパターンマッチ	$	行末でのパターンマッチ

正規表現の応用例として，日付表記「yyyy/mm/dd」やメールアドレスの表記「xxxx@yyy.zz」を判定する処理をサンプルプログラム RegEx3.java で示す．

プログラム：RegEx3.java

```
1  import java.util.regex.Pattern;
2  import java.util.regex.Matcher;
3
4  class RegEx3 {
5      // パターンマッチ用メソッド
6      public static void mtch( String str, String ptn0 ) {
7          Pattern ptn = Pattern.compile( ptn0 );
8          Matcher m = ptn.matcher(str);
9          if ( m.find() ) {
10             System.out.println("'"+str+"'は'"+ptn0+"'にマッチします. ");
11         } else {
12             System.out.println("'"+str+"'は'"+ptn0+"'にマッチしません. ");
13         }
14     }
15     // メイン
```

```
16      public static void main( String argv[] ) {
17          String ptn0;
18          // 年月日 yyyy/mm/dd へのマッチ
19          ptn0 = "^\\d+/\\d+/\\d+$";
20          mtch( "2019/12/01", ptn0 );
21          mtch( "日付2019/12/01", ptn0 );
22          mtch( "2019/12/01,時刻21:12:00", ptn0 );
23          System.out.println();
24          // メールアドレス xxxx@yyyy.zz へのマッチ
25          ptn0 = "^\\S+@\\S+$";
26          mtch( "nkatsu2012@gmail.com", ptn0 );
27          mtch( "http://www.google.com/", ptn0 );
28      }
29  }
```

プログラム中では，定義した mtch メソッドでパターンマッチを実行している．正規表現のパターンは文字列 ptn0 に格納しているが,「¥」を表現するにはエスケープシーケンスにする必要があるため「¥¥」と 2 回連続して記述している．

このプログラムの実行結果の例を次に示す．

```
'2019/12/01' は'^¥d+/¥d+/¥d+$' にマッチします.
' 日付 2019/12/01' は'^¥d+/¥d+/¥d+$' にマッチしません.
'2019/12/01, 時刻 21:12:00' は'^¥d+/¥d+/¥d+$' にマッチしません.

'nkatsu2012@gmail.com' は'^¥S+@¥S+$' にマッチします.
'http://www.google.com/' は'^¥S+@¥S+$' にマッチしません.
```

7.4 パターンマッチのグループ

対象の文字列の中から指定したパターンにマッチするものを取り出す場合には group メソッドを使用するが，ここでは，複数のパターンマッチによって複数の部分を取り出す方法について説明する．

例えば Oracle のインターネットサイトの URL を表す文字列 'https://www.oracle.com' からスキームの部分（'https'）と TLD（トップレベルドメイン）の部分（'com'）の 2 つを同時に取得するには次のような正規表現を用いる．

'^(¥S+)://¥S+¥.¥S+¥.(¥S+)$'

注意） ピリオド「.」にマッチさせるには「¥.」と記述する．

この正規表現の中に括弧 '（ ）' で括った部分が 2 箇所あるが，それらはスキームと TLD にマッチする部分である．このように括弧で括った部分は group メソッドによる抽出の対象となる．

このような正規表現でスキームと TLD の部分を取り出すプログラムの例を RegEx4.java に示す．

プログラム： RegEx4.java

```
1   import java.util.regex.Pattern;
2   import java.util.regex.Matcher;
3
4   class RegEx4 {
5       public static void main( String argv[] ) {
6           // URLからスキームとTLDを取り出すパターン
7           Pattern ptn = Pattern.compile( "^(\\S+)://\\S+\\.\\S+\\.(\\S+)$" );
8           Matcher m = ptn.matcher("https://www.oracle.com");  // OracleサイトのURL
9           m.find();   // パターンマッチの実行
10          int n = m.groupCount();     // グループ数の取得
11          System.out.println("グループ数："+n);
12          System.out.println("m.group(0)："+m.group(0));
13          for ( int c = 1; c <= n; c++ ) {
14              System.out.print(m.group(c)+", ");
15          }
16      }
17  }
```

括弧‘（ ）’つきの正規表現でパターンマッチした場合，Matcher オブジェクトに groupCount メソッドを実行することで抽出対象の部分の個数（整数）が得られる．今回のプログラムでは，groupCount メソッドで抽出対象部分の個数を n に得ており，1 から n までの値を group メソッドの引数に与えることでマッチした部分を抽出する．また group メソッドの引数に 0 を与えると，対象文字列全体が得られる．

このプログラムを実行した例を次に示す．

```
グループ数：2
m.group(0)：https://www.oracle.com
https, com,
```

参考）文字列探索やパターンマッチを実行するメソッドには find 以外にも matches や lookingAt といったものがあり，前者は正規表現を対象文字列全体にマッチさせ，後者は対象文字列の先頭にマッチさせる．

7.5 文字列の分割： split メソッド

正規表現を応用して文字列（String）を分割することができる．

書き方： 文字列.split(正規表現の文字列)

「文字列」を「正規表現の文字列」を区切りにして分割し，文字列の配列（ String[ ] ）として返す．split メソッドを応用したサンプルプログラム RegEx5.java を示す．

プログラム：RegEx5.java

```
1  class RegEx5 {
2      public static void main( String argv[] ) {
3          // 対象文字列
4          String str = "aaaaa01bbbbb234ccccc5ddddd67eeeee89fffff";
5          // 数字列を意味する正規表現を区切りとして分割する
6          String sp[] = str.split("\\d+");
7          int n = sp.length;
8          for ( int c = 0; c < n; c++ ) {
9              System.out.print(sp[c]+", ");
10         }
11     }
12 }
```

このプログラムでは文字列 str を数字列を区切りにして分割している．このプログラムを実行した例を次に示す．

```
aaaaa, bbbbb, ccccc, ddddd, eeeee, fffff,
```

split メソッドの引数には正規表現以外の文字列を与えても良い．次に示す RegEx5_2.java は **CSV** [39] を分割するものである．

プログラム：RegEx5_2.java

```
1  class RegEx5_2 {
2      public static void main( String argv[] ) {
3          // 対象文字列
4          String str = "aaaaa,bbbbb,ccccc,ddddd,eeeee,fffff";
5          // コンマを区切りとして分割する（CSVの分割）
6          String sp[] = str.split(",");
7          int n = sp.length;
8          for ( int c = 0; c < n; c++ ) {
9              System.out.print(sp[c]+", ");
10         }
11     }
12 }
```

先の RegEx5.java と同様の結果となる．

[39] RFC 4180：Comma-Separated Values

8　TCP/IP 通信

Java で実装したプログラムは，**クライアント-サーバモデル**に基づく通信（図 55）が可能である．このモデルでは，**サーバ**として機能するプログラムに対して**クライアント**として機能するプログラムが通信の開始を要求し，両者の通信が確立すると双方向にデータの送受信が可能となる．

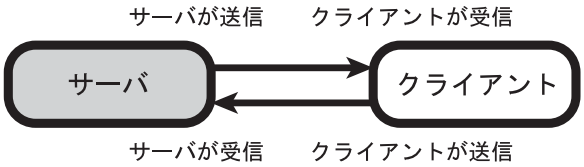


図 55: クライアント-サーバモデル

通信する両者のプログラムが対等ではないのが特徴である．これは，通信を始める前はサーバがサービスの提供者として先に存在し，後からクライアントがサーバに対してサービスを要求するという手順を踏むことによる．

8.1　ソケットの考え方

クライアント-サーバモデルに従って通信するプログラムを考える際に，**ソケット**の概念を理解する必要がある．「ソケット」というのはプラグなどを差し込むための「受け口」という意味の言葉であるが，通信を行うプログラムが，通信ケーブルのプラグを相手プログラムのソケットに差し込んで通信路を確立するという喩えになっている．

サーバ，クライアント共にソケットを実装するが，特にサーバ側プログラムでは**サーバソケット**というものを先に実装しておき，そこからソケットを取得する．クライアント側ではソケットのみを用意する．（図 56 参照）

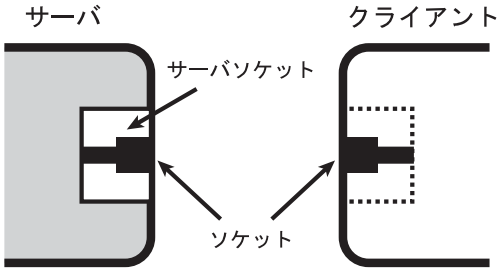


図 56: ソケットのイメージ

サーバ，クライアント共にソケットの用意ができたら，クライアントからサーバに対して接続を要求する．これに対してサーバが応えて通信路が確立すると，以後はソケットを介して双方向の通信が可能になる．（図 57 参照）

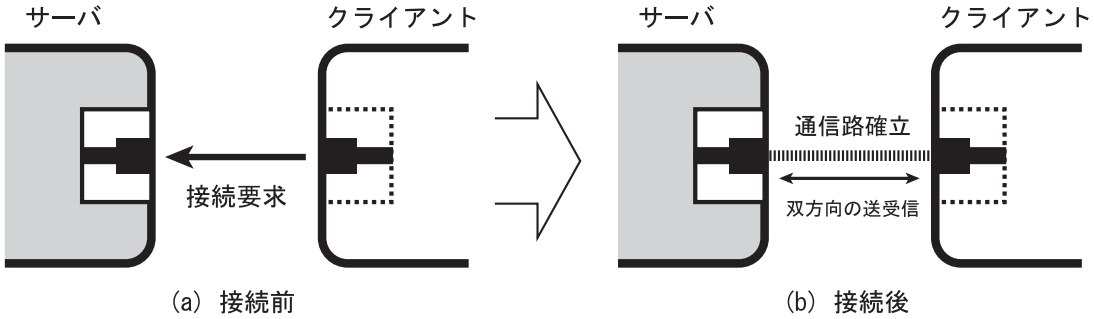


図 57: ソケットの接続

8.2 ソケットに関するクラス

8.2.1 サーバソケット

サーバソケットは ServerSocket クラス（java.net.ServerSocket）のオブジェクトである．

ServerSocket クラス
コンストラクタ： ServerSocket(int PORT)
解説： PORT で指定された番号の TCP ポートで接続を受け付けるサーバソケットを生成する．
実行例： `ServerSocket ssoc = new ServerSocket(8001);`
8001 番の TCP ポートで接続を受け付けるサーバソケット `ssoc` を生成する．

サーバソケットのタイムアウトの設定：
サーバソケットには接続タイムアウトを設定する．サーバソケットが接続要求を待っている状態で，ある指定した時間を超えて接続要求が来ない場合に SocketTimeoutException 例外が発生する．この例外を受けて，サーバプログラムを中断するといった処理を設定することが可能になる．サーバソケットにタイムアウトを設定するには setSoTimeout メソッドを実行する．

例. サーバソケット `ssoc` に 60 秒の接続タイムアウトを設定する．

```
ssoc.setSoTimeout(60000);
```

setSoTimeout メソッドの引数には，タイムアウト時間をミリ秒の単位で与える．
クライアントとサーバの通信路が確立した後で，サーバがクライアントからのデータ受信を待ち受ける際にもここで設定した接続タイムアウトに従って SocketTimeoutException 例外が発生する．

タイムアウトの値として 0 を設定すると無制限となる．

ポート番号について：
先の例に挙げた 8001 番は多くの場合，インターネットラジオのストリームなどに用いられている．実際にアプリケーションを実装する場合は，使用する環境に応じて使用できるポートを慎重に選ぶ必要がある．

8.2.2 ソケット

データの送受信に直接関係するソケットは `Socket` クラス（java.net.Socket）のオブジェクトである．サーバ側プログラムでソケットを実装するには，先に実装した ServerSocket オブジェクトから取得するという形を取る．

サーバ側でソケットを取得する例
実行例： `Socket soc = ssoc.accept();`
解説： ServerSocket クラスのオブジェクト `ssoc` に対して `accept` メソッドを実行することで
サーバ側のソケット `soc` を得ている．

クライアント側プログラムでソケットを実装するには，単に Socket オブジェクトを生成する．その後，接続相手のサーバプログラムの情報（ホスト，ポート）を指定して接続処理を実行する（connect メソッドを実行する）ことで，クライアント - サーバ双方のソケットの間に通信路が確立される．

接続相手の **IP ソケット**[40]（ネットワーク上のプロセスを指定するホストアドレスやポートの情報）は InetSocketAddress クラスのオブジェクトに保持する．

[40] **IP ソケット：** IP 通信をするネットワーク上のプログラムを識別するには，ホストアドレス（IP アドレスやドメイン名）とポート番号からなる「IP ソケット」を用いる．

InetSocketAddress クラス

コンストラクタ： InetSocketAddress(String HOST, int PORT)

解説： ホストアドレス HOST とポート番号 PORT からなる IP ソケットを保持する．

実行例：
```
InetSocketAddress iadr = new InetSocketAddress("localhost",8001);
```
”localhost”の 8001 番のポートを意味する IP ソケットを保持するオブジェクト `iadr` を生成する．

このクラスを使用するには，ライブラリ java.net.InetSocketAddress を読み込む必要がある．

クライアントプログラムのソケットを相手サーバプログラムのソケットに接続して通信路を確立するには `connect` メソッドを使用する．

通信路を確立する例

実行例：
```
soc.connect(iadr,60000);
```
解説： クライアント側のソケット `soc` を `iadr` で指定する相手サーバのソケットに接続する．connect メソッドの第 2 引数には接続タイムアウトの長さ（単位：ミリ秒）を与える．

この例における接続タイムアウトは，クライアントがサーバに接続要求したときから通信路確立までの最大待ち時間（サーバからの接続許可が得られるまでの最大待ち時間）を意味する．この時間を超えてサーバから接続許可が出ない場合はが SocketTimeoutException 例外が発生する．

クライアントとサーバの通信路が確立した後で，クライアントがサーバからのデータ受信を待つ際にも接続タイムアウトの例外（SocketTimeoutException）が発生する．この場合のタイムアウト時間の設定には，クライアントのソケットオブジェクトに対して setSoTimeout メソッドを用いる．

例. クライアントのソケット `soc` に対してタイムアウトを設定する
```
soc.setSoTimeout(60000);
```
この例では 60 秒のタイムアウトが設定されている．

8.3 データの送受信

ソケットを介した通信路は送信，受信の双方向に使用できるが，実装に際してソケットを入出力のストリームとして扱うことができる．すなわち，実装したソケットから入力，出力用のストリームを生成することができ，通常の I/O としての扱いが可能となる．

ソケットからストリームを生成する方法

入力ストリームの生成： `soc.getInputStream()`

出力ストリームの生成： `soc.getOutputStream()`

解説： ソケット `soc` からストリームを生成するには，getInputStream メソッド，getOutputStream メソッドを用いる．

ストリームが得られた後は，更にそれらを元にしてリーダやライタを生成することで，readLine メソッドや println メソッドを実行することが可能となる．

8.4 サンプルプログラム

ここでは，2 つのプログラムが IP 通信を行う例を示す．サーバ側プログラム ComTest01Sv とクライアント側プログラム ComTest01Cl がソケットを介した通信路を確立して互いにメッセージの送受信を行う例である．(図 58 参照)

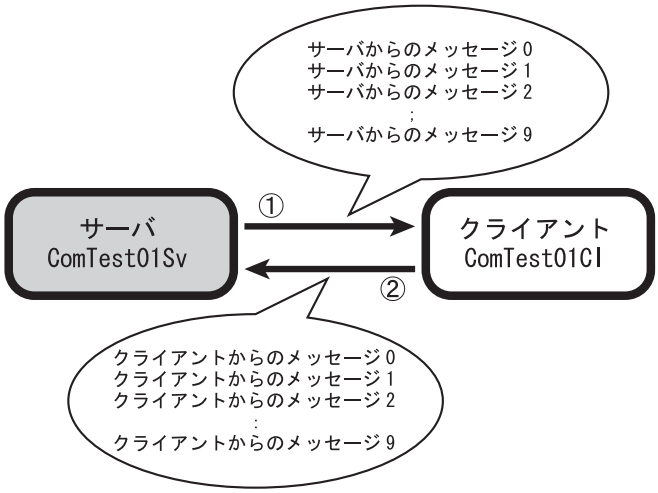


図 58: サンプルプログラムの動き

サーバ側プログラム：ComTest01Sv.java

```
import java.io.*;
import java.net.*;

class ComTest01Sv {
    // サーバの設定
    public static int   PORT        = 8001,     // 使用するポート番号
                        SV_TIMEOUT  = 10000;    // 受信待ちタイムアウト
    // メイン
    public static void main( String argv[] ) {
        // 必要なオブジェクト
        String s;
        // ソケットを開く
        try {
            // 接続処理
            ServerSocket ssoc = new ServerSocket(PORT);
            ssoc.setSoTimeout(SV_TIMEOUT);
            Socket soc = ssoc.accept();
            System.out.println("ポート"+PORT+"接続完了");
            // 通信のための準備(1) ソケットからストリーム生成
            InputStreamReader isr = new InputStreamReader(soc.getInputStream());
            OutputStreamWriter osw = new OutputStreamWriter(soc.getOutputStream());
            // 通信のための準備(2) ストリームからリーダ／ライタ生成
            BufferedReader br = new BufferedReader(isr);
            PrintWriter pw = new PrintWriter(osw);
            // 出力ループ（クライアントへの送信）
            int c;
            for ( c = 0; c < 10; c++ ) {
                pw.println("サーバからのメッセージ"+c);
            }
            pw.println("送信終了");
            pw.flush();
            // 入力ループ（クライアントからの受信）
            while ( true ) {
                s = br.readLine();
                if ( s == null ) {  // 終わりに達していたら終了
                    break;
                } else {
                    System.out.println("クライアントから受信> "+s);
                    if ( s.compareTo("送信終了") == 0 ) {
                        break;
                    }
```

```
42                  }
43              }
44              System.out.println("クライアントから受信が終了しました. ");
45              // 終了
46              ssoc.close();
47          } catch ( SocketTimeoutException e ) {  // タイムアウトの場合
48              System.out.println("ソケット生成がタイムアウトしました. ");
49          } catch ( SocketException e ) {         // ソケット例外
50              System.out.println("ソケット例外が発生しました. ");
51          } catch ( IOException e ) {             // 入出力例外
52              System.out.println("入出力例外が発生しました. ");
53          }
54      }
55  }
```

クライアント側プログラム：ComTest01Cl.java

```
 1  import java.io.*;
 2  import java.net.*;
 3
 4  class ComTest01Cl {
 5      // 通信の設定
 6      public static String    SV_HOST = "localhost";  // サーバ
 7      public static int       PORT    = 8001,         // 使用するポート番号
 8                              TIMEOUT = 10000;        // 接続タイムアウト
 9      // メイン
10      public static void main( String argv[] ) {
11          // 必要なオブジェクト
12          String s;
13          // ソケットを開く
14          try {
15              // 接続処理
16              Socket soc = new Socket();
17              InetSocketAddress iadr = new InetSocketAddress(SV_HOST,PORT);
18              soc.connect(iadr,TIMEOUT);
19              soc.setSoTimeout(TIMEOUT);
20              System.out.println("ポート"+PORT+"接続完了");
21              // 通信のための準備(1) ソケットからストリーム生成
22              InputStreamReader isr = new InputStreamReader(soc.getInputStream());
23              OutputStreamWriter osw = new OutputStreamWriter(soc.getOutputStream());
24              // 通信のための準備(2) ストリームからリーダ／ライタ生成
25              BufferedReader br = new BufferedReader(isr);
26              PrintWriter pw = new PrintWriter(osw);
27              // 入力ループ（サーバからの受信）
28              while ( true ) {
29                  s = br.readLine();
30                  if ( s == null ) {  // 終わりに達していたら終了
31                      break;
32                  } else {
33                      System.out.println("サーバから受信> "+s);
34                      if ( s.compareTo("送信終了") == 0 ) {
35                          break;
36                      }
37                  }
38              }
39              System.out.println("サーバから受信が終了しました. ");
40              // 出力ループ（サーバへ送信）
41              int c;
42              for ( c = 0; c < 10; c++ ) {
43                  pw.println("クライアントからのメッセージ"+c);
44              }
45              pw.println("送信終了");
46              pw.flush();
47              // 終了
48              soc.close();
49          } catch ( SocketTimeoutException e ) {  // タイムアウトの場合
50              System.out.println("ソケット接続がタイムアウトしました. ");
51          } catch ( SocketException e ) {         // ソケット例外
52              System.out.println("ソケット例外が発生しました. ");
53          } catch ( IOException e ) {             // 入出力例外
```

```
54                System.out.println("入出力例外が発生しました.");
55            }
56        }
57    }
```

サーバ側のプログラム ComTest01Sv を先に起動し，続いてクライアント側のプログラム ComTest01Cl を起動すると，それぞれのプログラムの出力は次の例のようになる．

```
ポート 8001 接続完了
クライアントから受信> クライアントからのメッセージ 0
クライアントから受信> クライアントからのメッセージ 1
クライアントから受信> クライアントからのメッセージ 2
クライアントから受信> クライアントからのメッセージ 3
クライアントから受信> クライアントからのメッセージ 4
クライアントから受信> クライアントからのメッセージ 5
クライアントから受信> クライアントからのメッセージ 6
クライアントから受信> クライアントからのメッセージ 7
クライアントから受信> クライアントからのメッセージ 8
クライアントから受信> クライアントからのメッセージ 9
クライアントから受信> 送信終了
クライアントから受信が終了しました.
```

(a) サーバ側のプログラム ComTest01Sv

```
ポート 8001 接続完了
サーバから受信> サーバからのメッセージ 0
サーバから受信> サーバからのメッセージ 1
サーバから受信> サーバからのメッセージ 2
サーバから受信> サーバからのメッセージ 3
サーバから受信> サーバからのメッセージ 4
サーバから受信> サーバからのメッセージ 5
サーバから受信> サーバからのメッセージ 6
サーバから受信> サーバからのメッセージ 7
サーバから受信> サーバからのメッセージ 8
サーバから受信> サーバからのメッセージ 9
サーバから受信> 送信終了
サーバから受信が終了しました.
```

(b) クライアント側のプログラム ComTest01Cl

9　サウンドデータの再生

Java でサウンドを取り扱うための基本的なクラス，インターフェースとして次のものがある．

(1) AudioInputStream　（javax.sound.sampled.AudioInputStream）
サウンドデータの入力元となるクラス

(2) SourceDataLine　（javax.sound.sampled.SourceDataLine）
サウンドデータの出力先となるインターフェース

(1) のオブジェクトから読み込んだサウンドデータを，(2) のオブジェクトに出力することでサウンドが再生される．読み込みには read メソッドを，出力には write メソッドを使用する．（詳しくは後述する．）

サウンド再生に必要なこれらのクラスを生成するに当たって，いくつかの属性情報（下記参照）を取得する必要がある．

(3) AudioFormat　（javax.sound.sampled.AudioFormat）
取り扱うサウンドの形式に関する各種の情報を取り扱うためのクラス

(4) DataLine　（javax.sound.sampled.DataLine）
システムのサウンド機能に関する各種の情報を取り扱うためのインターフェース

(3),(4) のオブジェクトは，出力のためのデータラインとなるオブジェクト (2) を生成するために必要となる．サウンドデータの流れや属性情報などの関連を図 59 に概略的に示す．

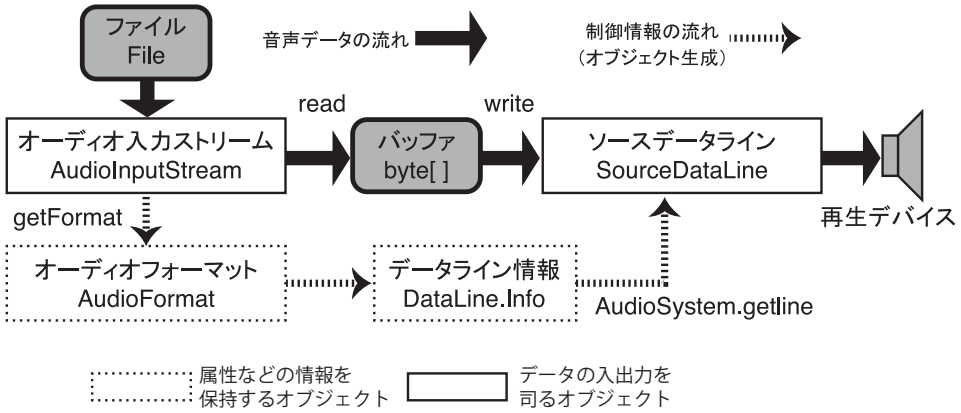


図 59: サウンド再生のシステム構成

図にある通り，サウンドデータの再生は，**オーディオ入力ストリーム**から読み込んで**バッファ**に保持し，それを**ソースデータライン**に書き出すことで実現する．

【サウンドデータの基本的な形式】

Java では基本的に **WAV 形式**のサウンドデータを扱う．WAV 形式は Microsoft 社と IBM 社が開発したサウンドデータ形式（サウンドフォーマット）であり，Microsoft 社の OS（Windows）において基本的な形式として採用されている．このフォーマットはデータ圧縮に関する仕様を持っているが，圧縮処理されていない PCM サウンドのデータを保存する場合に用いられることが多く，Windows に限らず多くの OS，アプリケーションにおいて，PCM サウンドデータの保存のために用いられている．

9.1　最も素朴な方法

サウンドデータファイルの内容を再生するための方法について，サンプルプログラムを示しながら具体的に説明する．次に示すプログラム SoundTest01.java は，WAV 形式のサウンドデータ music001.wav の内容を読み込んで再生するプログラムである．

プログラム：SoundTest01.java

```
1  import java.io.*;
2  import javax.sound.sampled.*;
3
4  public class SoundTest01 {
5      public static void main(String[] args) throws Exception {
6
7          //  オブジェクトの宣言
8          AudioFormat         fmt;
9          DataLine.Info       inf;
10         SourceDataLine      dl;
11         File                af;
12         AudioInputStream    as;
13         byte[ ]             buf;
14         long                sizeDat;
15         int                 sizeBuf;
16
17         //  入力の準備
18         af = new File("music001.wav");  //  音声ファイルを開く
19         as = AudioSystem.getAudioInputStream(af);   //  入力用音声ストリーム作成
20         sizeBuf = as.available( );      //  入出力用バッファサイズ取得
21         buf = new byte[sizeBuf];        //  入出力用バッファ確保
22         fmt = as.getFormat();           //  入力用音声ストリームの情報取得
23
24         //  事前調査
25         sizeDat = fmt.getFrameSize()*as.getFrameLength();
26         System.out.println("総データ長: "+sizeDat+" bytes");
27         System.out.println("バッファ長: "+sizeBuf+" bytes");
28
29         //  出力の準備
30         inf = new DataLine.Info(SourceDataLine.class,fmt);
31         dl = (SourceDataLine)AudioSystem.getLine(inf);  //  出力用データライン作成
32         dl.open(fmt);                                   //  出力用データラインを開く
33         dl.start();     //  出力用データラインの使用開始
34
35         //  データ読込み＆出力
36         as.read(buf,0,sizeBuf);         //  音声データ読込み
37         System.out.println("再生開始");
38         dl.write(buf,0,(int)sizeDat);   //  出力用データラインに出力
39
40         //  終了処理
41         dl.drain();     //  出力用データラインにデータが残留しておれば排出
42         dl.close();     //  出力用データラインを閉じる
43         as.close();     //  入力用音声ストリームを閉じる
44         System.out.println("再生終了");
45     }
46 }
```

解説：

これはサウンドデータファイルを再生するための最も基本的な機能を実装したプログラムである．

8〜15 行目： オーディオファイルの再生に必要な各種のオブジェクトや，制御に必要な変数などを宣言している．

18 行目： サウンドデータファイル “music001.wav” を File クラスのオブジェクト af に割り当てている．

19 行目： AudioSystem クラスのメソッド getAudioInputStream を用いて File クラスのオブジェクト af から AudioInputStream クラスのオブジェクト as を生成している．
以後はこの as がサウンドデータの入力元となる．

20 - 21 行目： オブジェクト as に対して available メソッド を実行することで，サウンド入出力に必要なバッファのサイズ（単位：バイト）を取得している．この値を元に 21 行目でバッファ用の記憶域を確保している．

22 行目： オブジェクト as に対して getFormat メソッドを実行することでサウンドの形式に関する情報を取得し，それを AudioFormat クラスのオブジェクト fmt に設定している．

25 行目： オブジェクト fmt に対して getFrameSize メソッド を実行することでサウンドの**フレームサイズ**（単位：バイト）を取得している．また，オブジェクト as に対して getFrameLength メソッドを実行することで，サウンドデータ全体を再生するのに必要な**フレームの個数**を取得している．すなわち，これら 2 つの数値の積が，再生に必要な有効バイト長となる．

30 行目： オブジェクト fmt からシステムのサウンド機能に関する情報を取得して DataLine.Ino のインスタンス inf に設定している．

31 行目： オブジェクト inf を元にして AudioSystem クラス のメソッド getLine を実行し，サウンド出力用のデータラインを取得してオブジェクト dl に設定している．
以後はこの dl がサウンドの出力先となる．

ここまでがサウンドデータ再生のための準備である．

32 - 33 行目： 出力用データラインをオープン（open メソッド）して，出力可能にして（start メソッド）いる．

36 行目： オーディオストリーム as から実際に音声データを読み込み，バッファ buf に格納している．

read メソッドの説明： as.read(buf,0,sizeBuf);
as からデータを読込み，buf の先頭（0 バイト目）から sizeBuf の長さだけ取得する．

38 行目： 出力データライン dl に buf の内容を出力する．

write メソッドの説明： dl.write(buf,0,sizeDat);
出力データライン dl に対して buf の内容を先頭（0 バイト目）から sizeDat の長さだけ出力する．

41 - 42 行目： 出力用データラインの終了処理．drain メソッドでデータラインに残留しているサウンドを排出し，close メソッドで閉じている．

43 行目： オーディオストリーム as を close メソッドを用いて閉じている．

このサンプルプログラム SoundTest01.java では，ファイルに保存されているサウンドデータを一度に読み取ってバッファに格納し，それを一度に出力データラインに送っている．

このプログラムを実行すると次のように表示して music001.wav のサウンドを再生する．

```
総データ長:  14701760 bytes
バッファ長:  14701760 bytes
再生開始
```

再生するサウンドデータがあまり大きくない場合（数 MB 程度）は今回の方法でも特に問題はないと思われるが，データサイズが非常に大きい場合（数十 MB あるいはそれ以上）はバッファが確保できないケースも発生しうる．また，再生の開始や停止の処理において大きな遅延時間が生じる場合もあり，入出力用バッファのサイズはあまり大きくしない方がよいこともある．

次に，小さめのバッファで「入出力を小分けにして繰り返す」形のサウンド再生について考える．

9.2 サイズの大きなサウンドデータの再生

ここでは，小さめのサイズの入出力用バッファを用いて「入出力を小分けにして繰り返す」形のサウンド再生について，サンプルプログラム SoundTest02.java を示しながら説明する．

プログラム：SoundTest02.java

```
1  import java.io.*;
2  import javax.sound.sampled.*;
3
4  public class SoundTest02 {
5      public static void main(String[] args) throws Exception {
```

```
 6
 7          //  オブジェクトの宣言
 8          AudioFormat         fmt;
 9          DataLine.Info       inf;
10          SourceDataLine      dl;
11          File                af;
12          AudioInputStream    as;
13          byte[]              buf;
14          long                sizeDat;
15          int                 sizeBuf;
16
17          //  入力の準備
18          af = new File("music001.wav");  //  音声ファイルを開く
19          as = AudioSystem.getAudioInputStream(af);   //  入力用音声ストリーム作成
20          sizeBuf = 262144;           //  入出力用バッファサイズ決定： 256KB
21          buf = new byte[sizeBuf];        //  入出力用バッファ確保
22          fmt = as.getFormat();           //  入力用音声ストリームの情報取得
23
24          //  事前調査
25          sizeDat = fmt.getFrameSize()*as.getFrameLength();
26          System.out.println("総データ長: "+sizeDat+" bytes");
27          System.out.println("バッファ長: "+sizeBuf+" bytes");
28
29          //  出力の準備
30          inf = new DataLine.Info(SourceDataLine.class,fmt);
31          dl = (SourceDataLine)AudioSystem.getLine(inf);  //  出力用データライン作成
32          dl.open(fmt);   //  出力用データラインを開く
33          dl.start();     //  出力用データラインの使用開始
34
35          //  データ読込み＆出力
36          dl.start();     //  出力用データラインの使用開始
37          System.out.println("再生開始");
38          int c = 0;      //  読み込んだ回数
39          int lenRead;    //  読み込んだデータ長（1回分）
40          int sizeRead = 0;   //  読み込んだデータ長（トータル）
41          while ( (lenRead = as.read(buf,0,buf.length)) >= 0 ) {  // 音声データ読込み
42              c++;
43              sizeRead += lenRead;
44              System.out.printf("(%02d) %d byte  (total: %d byte)\n",
45                      c,lenRead,sizeRead);
46              dl.write(buf,0,(int)lenRead);   //  出力用データラインに出力
47          }
48
49          //  終了処理
50          dl.drain(); //  出力用データラインにデータが残留しておれば排出
51          dl.close(); //  出力用データラインを閉じる
52          as.close(); //  入力用音声ストリームを閉じる
53          System.out.println("再生終了");
54      }
55  }
```

解説：

処理の大まかな流れは SoundTest01.java とほぼ同じであるが，入出力用のバッファのサイズが異なる．今回のサンプルでは，20 行目で指定しているように 256KB のバッファサイズにしている．このバッファの大きさを超えるサイズのサウンドデータを再生するには，そのデータ全てを読込み終えるまで，入力と出力のサイクルを繰り返すことになる．この繰り返しを行っているのが 41〜47 行目の部分である．

このプログラムを実行したときの表示の例を次に示す．

```
総データ長:  14701760 bytes
バッファ長:  262144 bytes
再生開始
 (01) 262144 byte (total:  262144 byte)
 (02) 262144 byte (total:  524288 byte)
 (03) 262144 byte (total:  786432 byte)
                :
```

この実行例は，サウンドを再生しながら読込みと出力を行う度にメッセージを表示していることを示す．

9.3 実用的なサウンド再生

先に紹介したサンプルプログラム SoundTest01.java，SoundTest02.java は専らサウンドデータの読込みと再生を行うものであり，当然ながらこれらのプログラムは，サウンドの再生をしている間は他の処理はできない．

多くの実用的なアプリケーションプログラムの場合，サウンドの再生を伴う処理を行う際は，GUI の操作といった別の処理を平行して行うことができる．ここでは，サウンドデータの読込みと再生を独立したスレッド（Thread）で実行し，サウンド再生中も他の処理を可能にする例について説明する．（スレッドについては「5.1.1 Thread クラス」（p.140）を参照のこと）

プログラム：SoundTest03.java

```
1  import java.io.*;
2  import javax.sound.sampled.*;
3
4  class Player extends Thread {
5
6      //  オブジェクトの宣言
7      public AudioFormat      fmt;
8      public DataLine.Info    inf;
9      public SourceDataLine   dl;
10     public File             af;
11     public AudioInputStream as;
12     public byte[]           buf;
13     public long             sizeDat;
14     public int              sizeBuf;
15     public boolean          playable = false;
16     public int c = 0;       //  読み込んだ回数
17     public int lenRead;     //  読み込んだデータ長（1回分）
18     public int sizeRead = 0;    //  読み込んだデータ長（トータル）
19
20     //  準備処理
21     public void ready( ) throws IOException,
22                 UnsupportedAudioFileException, LineUnavailableException {
23
24         //  入力の準備
25         af = new File("music001.wav");  //  音声ファイルを開く
26         as = AudioSystem.getAudioInputStream(af);   //  入力用音声ストリーム作成
27         sizeBuf = 262144;           //  入出力用バッファサイズ決定： 256KB
28         buf = new byte[sizeBuf];    //  入出力用バッファ確保
29         fmt = as.getFormat();       //  入力用音声ストリームの情報取得
30
31         //  事前調査
32         sizeDat = fmt.getFrameSize()*as.getFrameLength();
33         System.out.println("総データ長: "+sizeDat+" bytes");
34         System.out.println("バッファ長: "+sizeBuf+" bytes");
35
36         //  出力の準備
37         inf = new DataLine.Info(SourceDataLine.class,fmt);
38         dl = (SourceDataLine)AudioSystem.getLine(inf);  //  出力用データライン作成
39         dl.open(fmt);           //  出力用データラインを開く
40
41         playable = true;
42     }
43
44     //  サウンド再生
45     public void play( ) throws IOException {
46         //  データ読込み＆出力
47         dl.start();     //  出力用データラインの使用開始
48         System.out.println("再生開始");
49         while ( (lenRead = as.read(buf,0,buf.length)) >= 0 && playable ) {
50             c++;
51             sizeRead += lenRead;
52             dl.write(buf,0,(int)lenRead);   //  出力用データラインに出力
53         }
54     }
55
56     //  状態表示
```

```
57      public void report( ) {
58          System.out.printf("%02d回目I/O, %d byte 読込み  (total: %d byte)\n",
59                  c, lenRead, sizeRead);
60      }
61
62      //  終了処理
63      public void terminate( ) throws IOException {
64          //  終了処理
65          dl.drain();     //  出力用データラインにデータが残留しておれば排出
66          dl.close();     //  出力用データラインを閉じる
67          as.close();     //  入力用音声ストリームを閉じる
68          System.out.println("再生終了");
69          playable = false;
70      }
71
72      //  再生実行
73      @Override
74      public void run( ) {
75          if ( playable ) {
76              try {
77                  play( );
78                  terminate( );
79                  playable = false;
80              } catch (IOException e) {
81              }
82          } else {
83              System.out.println("再生の準備ができていません. ");
84          }
85      }
86  }
87
88  class SoundTest03 {
89      public static void main(String[] args) throws Exception {
90          //  サウンド再生
91          Player pl = new Player( );
92          pl.ready( );
93          pl.start( );
94
95          //  コマンドループ
96          int c;
97          while ( true ) {
98              System.out.print("再生中：'9'で終了 > ");
99              c = System.in.read( );
100             if ( c == '9' ) {
101                 pl.playable = false;
102                 break;
103             } else {
104                 pl.report( );
105             }
106         }
107         pl.report( );
108     }
109 }
```

解説：

このプログラムでは Player という名前のクラスを定義して，その中にサウンドデータの読込み，再生，停止といった処理を行うための次のようなメソッドを記述している．

- **ready メソッド：** （21〜42 行目）オーディオ入力ストリーム，サウンド出力用のデータラインの準備．
- **play メソッド：** （45〜54 行目）サウンドデータの読込みと再生を繰り返すループを主とした処理．
- **terminate メソッド：** （63〜70 行目）サウンド再生終了後にオーディオ入力ストリームや出力用データラインを閉じる処理．
- **report メソッド：** （57〜60 行目）サウンド再生中に，処理経過（読み取った回数やデータサイズなど）を表示する処理．　（これは必須の処理ではない）

Player クラスは，Java のスレッドプログラミングのための Thread クラスを拡張したものであり，このクラスの

インスタンスに対して start() メソッドを実行すると，Thread クラスの run() メソッドが起動する．run() メソッドは，拡張クラス側でオーバーライド（@Override）して具体的な処理内容を記述する．この run() メソッドは，main メソッドを実行するスレッドとは別の独自したスレッドとして実行される．

このプログラムの main メソッドの中では，Player クラスのインスタンス pl を生成している．これがオーディオ再生を単独で行う「サウンドプレーヤー」として振る舞う．この pl に対して ready() メソッドを実行すると，オーディオ入力ストリームとサウンド出力用のデータラインを準備する．その後で pl に対して start() メソッド を実行すると，オーバーライドした run() メソッドが起動して独立スレッドとなり，play() メソッドを使用してサウンドを再生する．

97〜106 行目の部分はコマンドループとなっており，サウンドの再生とは別のループである．
Enter キーを押すと入出力の状況報告を表示する．また ‘9’ を入力するとサウンド再生を中断してプログラム全体を終了する．これら操作を行っている例を次に示す．

```
総データ長:  14701760 bytes
バッファ長:  262144 bytes
再生開始

再生中：'9' で終了 > Enter
 04 回目 I/O, 262144 byte 読込み (total:  1048576 byte)
再生中：'9' で終了 > Enter
 04 回目 I/O, 262144 byte 読込み (total:  1048576 byte)
再生中：'9' で終了 >Enter
 08 回目 I/O, 262144 byte 読込み (total:  2097152 byte)
再生中：'9' で終了 > 9   Enter
 14 回目 I/O, 262144 byte 読込み (total:  3670016 byte)
再生終了
```

10　ライブラリ，パッケージに関すること

Java のプログラミングにおいては各種のライブラリ（標準ライブラリ，サードパーティのライブラリ）を利用することが多い．それらのライブラリは開発元の団体（あるいは個人）が Java 言語でクラスの定義を記述したプログラムライブラリである．

Java のクラスライブラリは**パッケージ名**を持ち，パッケージ名の元で各種クラスを利用する形を取る．例えば，ファイル入出力の際に用いる File クラスは java.io.File というライブラリをインポートして使用するが，ライブラリ名の前部の java.io がパッケージ名であり，File はこのパッケージに属するクラスである．

標準ライブラリのパッケージ名は概ね java··· で始まる．これに対してサードパーティが提供するパッケージの名前は，その前部に開発元の団体のドメイン名を 逆順 にしたものが用いられることが一般的である．

例. 多倍長精度演算パッケージ Apfloat [41]
公式サイト：http://www.apfloat.org/　→　パッケージ：org.apfloat.*

開発したライブラリを公開しない場合は，パッケージ名を任意に命名しても特に問題は無い．

10.1　ライブラリの作成

作成したクラスをライブラリとして使用する方法について説明する．作成するパッケージの名前を org.MyLib1 として例を示す．次のような独自のクラスライブラリ org.MyLib1.Calc を作成することを考える．

プログラム：Calc.java

```
1 package org.MyLib1;
2
3 public class Calc {
4     public void methodTest() {
5         System.out.println("MyLib1のmethodTestです. ");
6     }
7 }
```

プログラムの 1 行目にあるように，

package　パッケージ名;

として作成するパッケージの名前を宣言する．以後は通常の形でクラス定義を記述するが，他のプログラムから（パッケージ外から）クラスを利用するために **public 修飾子**を付ける．

10.1.1　ライブラリファイル（クラスファイル）の保存場所

作成したクラスファイル（コンパイル済みの *.class ファイル）はパッケージ名を反映したディレクトリ位置に配置する．すなわち org.MyLib1 というパッケージの場合は ‘org/MyLib1/Calc.class’ としてクラスファイルを配置する．

このクラスライブラリを利用する次のようなアプリケーションプログラム MyApp01 を実行する．

プログラム：MyApp01.java

```
1 import org.MyLib1.*;    // ライブラリの読込み
2
3 class MyApp01 {
4     public static void main( String argv[] ) {
5         // Calcクラスのインスタンス生成
6         Calc c = new Calc();
7         c.methodTest();     // メソッド実行
8     }
9 }
```

[41] 巻末付録「E.1 Apfloat」（P.215）で紹介する．

このプログラムは，先のパッケージのディレクトリ org があるディレクトリと同じディレクトリ に配置して実行する．実行結果の例を次に示す．

```
MyLib1 の methodTest です.
```

10.1.2　異なるパッケージが同名のクラスを提供する場合の扱い

先に例示したクラスライブラリ org.MyLib1.Calc と異なる次のようなクラスライブラリ org.MyLib2.Calc が存在する場合について考える．

プログラム：Calc.java（先とは別のプログラム）

```
1  package org.MyLib2;
2
3  public class Calc {
4      public void methodTest() {
5          System.out.println("MyLib2のmethodTestです. ");
6      }
7  }
```

この場合は異なるパッケージが同名のクラスを提供しており，それらクラスを使い分けるには注意が必要である．org.MyLib1.Calc，org.MyLib2.Calc の両方（両方とも適切なディレクトリに配置されているとする）を読み込んで Calc クラスを使い分けるアプリケーションプログラム MyApp02.java を示す．

プログラム：MyApp02.java

```
1  import org.MyLib1.*;    // ライブラリの読込み(1)
2  import org.MyLib2.*;    // ライブラリの読込み(2)
3
4  class MyApp02 {
5      public static void main( String argv[] ) {
6          // org.MyLib1.Calcクラスのインスタンス生成
7          org.MyLib1.Calc c1 = new org.MyLib1.Calc();
8          c1.methodTest();        // メソッド実行
9
10         // org.MyLib2.Calcクラスのインスタンス生成
11         org.MyLib2.Calc c2 = new org.MyLib2.Calc();
12         c2.methodTest();        // メソッド実行
13     }
14 }
```

このプログラムではどのパッケージの Calc クラスであるかを明に示すためにパッケージ名を含めた形でクラス名を記述（7，11 行目）している．このような形のクラス名を**完全修飾名**（fully qualified name ）と呼ぶ．

MyApp02.java を実行した例を次に示す．

```
MyLib1 の methodTest です.
MyLib2 の methodTest です.
```

異なるパッケージ間で同じ名前のクラスがある場合は，完全修飾名による記述で区別する必要がある．

10.1.3　jar ファイルの作成

JDK では，クラスファイルを **jar 形式**のファイルにするためのプログラムが利用できる．例として，先の説明で作成した org ディレクトリを MyLib.jar という jar 形式ファイルにするには次のような操作を行う．

```
jar cvf MyLib.jar org
```

11　データベースへの接続

大量のデータを扱うには**データベース**の取り扱いが必須となる．データベースとしてデータを蓄積管理するためのシステムは DBMS [42]（データベース管理システム）と呼ばれる独立したシステムである．データベースの機能を使用するアプリケーションは，この DBMS を介してデータを抽出したり，データの登録や更新などを行う．この際の DBMS に対する処理の依頼を**クエリ**（query）と呼び，クエリのための言語として SQL がある．本書では DBMS として主にパブリックドメインのフリーソフトウェアである SQLite の扱いを例に挙げて SQL によるデータベースの使用方法について説明する．SQLite は充分な性能がある上に導入方法も簡単であるため，データベースの機能についての学習や，アプリケーションへの組み込み，あるいは個人的な運用に適している．

11.1　データベースについての基本的な考え方

本書では**関係モデル** [43] に基づく**リレーショナル・データベース**（関係データベース，RDB）を取り扱う．RDB では，データを**対象**，**属性**，**値**の組として扱う．具体的には，これら組の 1 つを 1 行として扱い，複数の列（**項目**）の値を横方向に書き並べて 1 つの**レコード**とする．これは表計算ソフトウェアによる表の扱いに似ている．すなわち，左端の列に対象物の名称や記号を記述し，それより右側には各種の項目の値を書き並べる形式である．関係データベースは表を参照することによって対象の属性の値を返す，すなわち「…の〜は何？」という問い合わせ（クエリ）に対して,「△△△です」と値を返すシステムであると見做すことができる．

RDB では，複数の**表**（table）をまとめて 1 つの**データベース**（database）とする．表計算ソフトウェアの場合では，1 つのデータファイル（ブック）は複数の表（スプレッドシート）を束ねた形 [44] になっており，この意味でも，RDB と表計算ソフトウェアの類似性（表 50）が見られる．

表 50: RDB と表計算ソフトウェアの類似性

RDB	表計算ソフトウェア
table	シート
database	シートを束ねたデータファイル

RDB では特定の項目（列）を**主キー**（primary key）として定め，主キーによる高速なデータ検索を可能にしている．また，table の検索の結果として得られた値を**外部キー**（foreign key）として，別の table の主キーを検索することができ，複雑な情報検索を可能にする．

RDB では表の構成を設計する際に，どのような項目の集まりとするかに注意を払う必要があり，可能な限り，同じような項目が繰り返し現れることのないように配慮する必要がある．このような，無駄や矛盾が起こらないような配慮のために**正規化**と呼ばれる設計上の工夫が重要であるが，これに関しては本書では触れない．

11.1.1　データベースに対する基本的な操作

RDB に対する基本的な操作は**選択**，**射影**，**結合**の 3 種類である．データベースは多量のデータを保持するものであり，必要な行（レコード）や列（項目）を絞り込んでデータを抽出することが重要であり，選択は「必要な行の抽出」，射影は「必要な列の抽出」を意味する．また，RDB では 1 つのデータベース（database）に複数の表（table）があり，それら複数の表を連結して 1 つの表のように扱う機能が「結合」である．これらに関しては後に事例を挙げて具体的に説明する．

[42] 本書では**関係データベース**の DBMS を前提とする．具体的なソフトウェアとしては，Microsoft 社の Access や SQL Server，オープンソースの PostgreSQL や MySQL，パブリックドメインの SQLite などがある．

[43] IBM のエドガー・F・コッドによって考案された現在もっとも広く用いられているデータモデルである．

[44] Microsoft 社の Excel における標準的なデータファイル形式である「ブック形式」などがその例である．

11.2　本書で取り扱うソフトウェア

11.2.1　SQLite

SQLite はパブリックドメインの DBMS であり，ソフトウェア本体と関連情報が公式インターネットサイト

https://www.sqlite.org/

から入手できる．一般的な DBMS は**ミドルウェア**[45] として常時稼働し，複数の利用者から同時に**トランザクション**[46] を受け付けるという形態で運用される．これに対して SQLite はソフトウェアライブラリの形で提供され，アプリケーションに組み込んで[47] 使用する形態のソフトウェアである．SQLite が扱うデータベースは単一のファイルであり，運用と管理が極めて単純である．このため実際に多くのアプリケーションシステムに組み込まれて[48] 利用されている．

本書ではデータベースの DBMS としてこの SQLite を前提とする．

11.2.2　JDBC

関係データベースを扱うための API である ODBC を元にして作られた API に JDBC[49] がある．これは Java で関係データベースを扱うためのものであり，本書では JDBC を前提としてデータベースの取り扱いについて説明する．

11.3　トランザクション

データベース利用の一連の手続きを**トランザクション**という．JDBC によるトランザクションのための基本的なインターフェースが

Connection（java.sql.Connection）
Statement（java.sql.Statement）

の 2 つである．JDBC によるトランザクションは，

(1) DBMS に接続し，接続状態を表す Connection オブジェクトを取得する
(2) Connection オブジェクトから Statement オブジェクト（SQL 文を実行する対象）を取得する
(3) Statement オブジェクトに対して SQL 文を実行する（必要となる一連の SQL を実行する）
(4) DBMS の使用を終了する

というステップから成る．各ステップで使用するメソッドなどを以下に説明する．

11.3.1　データベースへの接続

DriverManager クラス（java.sql.DriverManager）の getConnection メソッドを使用することで実際にデータベースに接続する．

データベースへの接続の例：

```
Connection con = DriverManager.getConnection(
                    データベースの URL, ユーザ名, パスワード )
```

getConnection メソッドの引数は文字列（String）で与える．SQLite の場合はユーザ名，パスワードは省略できる．また，SQLite においては，接続先のデータベースが存在しない場合はそのデータベースが自動的に作成される．この例では接続状態を表す Connection オブジェクト con が得られている．

[45] OS とアプリケーションの中間に位置するもので，各種アプリケーションからのサービス要求に応えるといった役割を持つソフトウェア．
[46] データベースに対する一連の処理の手続きで，アクセスの基本的な流れとなる．
[47] SQLite は動的リンクライブラリ（Windows では DLL）の形で提供される．
[48] SQLite 本体（Windows の場合は sqlite3.dll）を対象アプリケーションのディレクトリに配置するなど，導入が単純である．
[49] SQLite 用の JDBC ライブラリ（jar ファイル）が Bitbucket のインターネットサイト（https://bitbucket.org/）から入手できる．

11.3.2 Statement オブジェクトの取得

データベースへの接続状態を表す Connection オブジェクトに対して createStatement メソッドを実行することで Statement オブジェクトが得られる．SQL 文はこのオブジェクトに対して発行する．

Statement オブジェクト取得の例：

Statement st = con.createStatement();

この例では Statement オブジェクト st が得られている．

11.3.3 SQL 文の実行

Statement オブジェクトに対して SQL 文を実行するメソッドはいくつか存在する．

SQL 文を実行する例 (1)：

int ret = st.executeUpdate("SQL 文")

検索結果などの データを返さない SQL 文，例えば INSERT 文，UPDATE 文，DELETE 文 といったものを実行する．

この例では，実行結果として整数型の ret が得られている．戻り値は，更新したレコードの件数などを意味する．

SQL 文を実行する例 (2)：

ResultSet res = st.executeQuery("SQL 文")

検索結果などの データを返す SQL 文，例えば SELECT 文といったものを実行する．実行結果の戻り値は ResultSet インターフェース（java.sql.ResultSet）のオブジェクトである．

11.3.3.1 検索結果の取り出し

SQL の SELECT 文でデータを選択すると複数のレコードが得られるのが一般的である．executeQuery メソッドで SELECT 文を実行すると ResultSet オブジェクトが得られるが，それに対して getMetaData メソッドを実行すると，それについての管理情報が ResultSetMetaData インターフェース（java.sql.ResultSetMetaData）のオブジェクトとして得られる．ResultSetMetaData オブジェクトに対して getColumnCount メソッドを実行すると，ResultSet オブジェクトが持つレコードの件数が得られる．これらの利用方法については後でサンプルプログラムを挙げて例示する．

11.3.4 トランザクションの終了

DBMS の使用を終了し，接続を終えるには Statement オブジェクトと Connection オブジェクトに対して close メソッドを実行する．

実際のトランザクションに関して，後でサンプルプログラムを挙げて例示する．

11.4 SQL

SQL はデータベースに対するクエリを記述するための言語として最も多く用いられており，多くの DBMS で標準的に採用されている．本書では，Java でデータ処理を行う際に必要となる最低限の範囲で SQL について例を示す形で説明する．

サンプルとして作成するテーブルを表 51 に示す.

表 51: 作成するテーブルのサンプル

コード（整数）	都道府県（文字列）
主キー	-
13	東京都
23	愛知県
26	京都府
27	大阪府

テーブル名：tblPref

都道府県（文字列）	面積（整数）
主キー	-
東京都	2191
神奈川県	2416
愛知県	5172
大阪府	1905

テーブル名：tblGeom

表 51 の tblPref は都道府県コードに対する都道府県名を保持するもので，tblGeom は都道府県の面積を保持するものである．表 51 にある**主キー**とは，当該テーブルから目的のレコードを見つけ出すための検索キーであり，関係データベースの重要な要素である．

関係データベースでは，情報のカテゴリや利用目的に沿った形のテーブルを複数用意して，それらを関連付けて高度な情報検索を実現する．ここでは，上記 2 つのテーブルを**結合**して，両方のテーブルの全ての情報を 1 つのテーブルのように扱うことができることを例示する．

11.4.1 データベースの作成

複数のテーブルを保持するデータベースを新規に作成するには，SQL の文 ‘CREATE DATABASE’ をクエリとして発行する．具体的には作成するデータベースの名前を指定して

CREATE DATABASE IF NOT EXISTS **データベース名**

と記述する．ただし SQLite の場合は，存在しないデータベースに接続を試みると，そのデータベースが自動的に作成されるので，この処理は必要ない．

11.4.2 テーブルの作成

テーブルを新規に作成するには，‘CREATE TABLE’ 文をクエリとして発行する．

《テーブルの作成》

書き方：　CREATE TABLE テーブル名 (カラム名 1　型, カラム名 2　型,　…,
PRIMARY KEY(対象のカラム))

「テーブル名」で指定したテーブルを作成する．この際，保持するカラムとそのデータ型を指定する．
PRIMARY KEY に主キーとするカラムをを指定する.（複数可）

11.4.2.1 カラムのデータ型

カラムのデータ型を表 52 に挙げる．

表 52: カラムのデータ型（SQLite の場合）

型	解説	型	解説
INTEGER	整数	REAL, NUMERIC	浮動小数点数
TEXT	文字列	NONE	型なし

サンプルとして示したテーブル tblPref を作成してレコードを追加するには次のような SQL 文を実行する．

テーブルの作成：

```
CREATE TABLE tblPref(コード INTEGER, 都道府県 TEXT, PRIMARY KEY(コード))
```

レコードの追加：

```
INSERT INTO tblPref (コード, 都道府県) VALUES
                    (13,"東京都"), (22,"愛知県"), (27,"大阪府"), (26,"京都府")
```

注意） 既存のテーブルと同じ名前のテーブルを作成するクエリを発行するとエラーとなる[50] ので注意すること.

テーブルにレコードを挿入する 'INSERT INTO' 文の書き方は次の通り.

書き方：　INSERT INTO テーブル名 (カラム 1,　カラム 2,　…)
VALUES (カラム 1 の値, カラム 2 の値),
(カラム 1 の値, カラム 2 の値), …)

「テーブル名」で指定したテーブルにレコードを挿入（追加）する．その際の各カラムの値を記述する．VALUES 句の後ろには複数の値の列を記述することができ，複数のレコードを挿入することができる．

同様の方法でテーブル tblGeom を作成してレコードを追加する．SQL 文の例を次に示す.

テーブルの作成：

```
CREATE TABLE tblGeom(都道府県 TEXT, 面積 INTEGER, PRIMARY KEY(都道府県))
```

レコードの追加：

```
INSERT INTO tblGeom (都道府県, 面積) VALUES
                    ("東京都",2191), ("愛知県",5172), ("大阪府",1905), ("神奈川県",2416)
```

以上の処理で，2 つのテーブル tblPref, tblGeom が作成されて値が設定された．この後は SELECT 文によるクエリを発行してテーブルのレコードを抽出（選択処理）することができる．SELECT 文の書き方を次に示す.

書き方：　SELECT カラム 1, カラム 2, … FROM テーブル名 WHERE 条件式

「テーブル名」で指定したテーブルから,「カラム 1, カラム 2,…」のカラムを指定してレコードを選択する．このとき，WHERE 句にレコードを選択するための条件を記述する．WHERE 句を省略すると，全てのレコードが選択される．カラムの記述をアスタリスク '*' とすると，全てのカラムが対象となる.

11.4.2.2　サンプルプログラム (1)

ここまでに説明した SQL 文を使用して表 TblDBsmpl に挙げたテーブルを作成して内容を閲覧するサンプルプログラム SQLite01.java を示す.

プログラム：SQLite01.java

```
1  import java.sql.Connection;
2  import java.sql.DriverManager;
3  import java.sql.Statement;
4  import java.sql.ResultSet;
5  import java.sql.ResultSetMetaData;
6
7  class SQLite01 {
8      public static void main( String argv[] ) {
9          try {
10             Class.forName("org.sqlite.JDBC");   // イニシャライザ
```

[50] SQLite ではデータベースが単一のファイルとして作成されるので，トランザクションが開いていない状態であれば，データベースのファイルを削除することで全てのテーブルを破棄することができる.

```
11              // データベースへの接続
12              Connection con = DriverManager.getConnection(
13                                          "jdbc:sqlite:c:/SQLite/JPN01.db");
14              System.out.println("con："+con);
15              Statement st = con.createStatement();   // Statementオブジェクトの取得
16              System.out.println("st："+st);
17              System.out.println();
18
19              // SQL実行に必要なもの
20              String sql;
21              ResultSet res;
22              ResultSetMetaData rsmd;
23              int ret;
24
25              // テーブルの作成とレコードの追加(1)
26              System.out.println("tblPrefの作成");
27              sql = "CREATE TABLE tblPref(コード INTEGER, 都道府県 TEXT, "+
28                                                  "PRIMARY KEY(コード))";
29              ret = st.executeUpdate(sql);
30              System.out.println("ret（更新件数）："+ret);
31              System.out.println("tblPrefへのレコードの追加");
32              sql = "INSERT INTO tblPref (コード, 都道府県) VALUES "+
33                                          "(13,\"東京都\"), (22,\"愛知県\"), "+
34                                          "(27,\"大阪府\"), (26,\"京都府\")";
35              ret = st.executeUpdate(sql);
36              System.out.println("ret（更新件数）："+ret);
37
38              // テーブルの作成とレコードの追加(2)
39              System.out.println("tblGeomの作成");
40              sql = "CREATE TABLE tblGeom(都道府県 TEXT, 面積 INTEGER, "+
41                                                  "PRIMARY KEY(都道府県))";
42              ret = st.executeUpdate(sql);
43              System.out.println("ret（更新件数）："+ret);
44              System.out.println("tblGeomへのレコードの追加");
45              sql = "INSERT INTO tblGeom (都道府県, 面積) VALUES "+
46                                      "(\"東京都\",2191), (\"愛知県\",5172), "+
47                                      "(\"大阪府\",1905), (\"神奈川県\",2416)";
48              ret = st.executeUpdate(sql);
49              System.out.println("ret（更新件数）："+ret+"\n");
50
51              // テーブルの閲覧(1)
52              sql = "SELECT * FROM tblPref";
53              res = st.executeQuery(sql);     // 選択結果
54              rsmd = res.getMetaData();       // 上記のメタデータ
55              System.out.println("tblPrefの閲覧");
56              while ( res.next() ) {
57                  for ( int i = 1; i <= rsmd.getColumnCount(); i++ ) {
58                      System.out.print( res.getString(i)+", " );
59                  }
60                  System.out.println();
61              }
62              System.out.println();
63
64              // テーブルの閲覧(2)
65              sql = "SELECT * FROM tblGeom";
66              res = st.executeQuery(sql);     // 選択結果
67              rsmd = res.getMetaData();       // 上記のメタデータ
68              System.out.println("tblGeomの閲覧");
69              while ( res.next() ) {
70                  for ( int i = 1; i <= rsmd.getColumnCount(); i++ ) {
71                      System.out.print( res.getString(i)+", " );
72                  }
73                  System.out.println();
74              }
75
76              // 終了
77              st.close();
78              con.close();
79          } catch(Exception e) {
80              System.out.println("エラー："+e);
```

```
81            }
82        }
83    }
```

SELECT 文の実行によって選択されたレコード群を扱う方法について，先の「11.3.3.1 検索結果の取り出し」(p.188) の所で簡単に説明したが，このサンプルに沿って更に具体的に説明する．

■ テーブルのカーソルについて

関係データベースのテーブルには，テーブル内のレコード位置を指し示す**カーソル**と呼ばれるものがある．SELECT 文の実行によって選択された複数のレコードの内で特定のレコードをカーソルで指し示し，カーソル位置の示すレコードから必要となる**カラム**（列）のデータを取り出すことができる．

SELECT 文の実行によって得られた ResultSet オブジェクトは複数のレコードを保持しており，それに対して next メソッドを実行するとカーソルが 1 つずつ進む [51]．(図 60 参照)

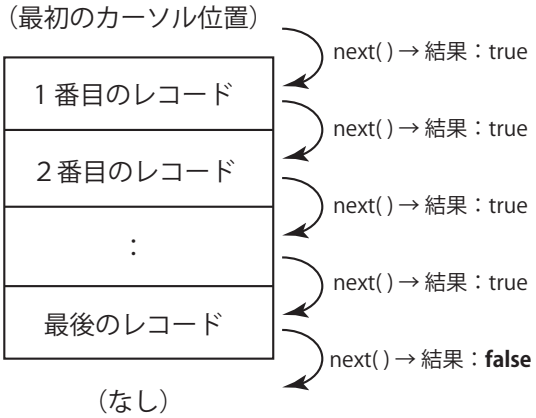


図 60: カーソルのイメージ

next メソッドは実行の結果として真理値（Boolean）を返す．カーソルが ResultSet オブジェクトの末尾のレコードを指し示している状態で next メソッドを実行すると，指し示すべきレコードが存在しないため false を返し，それ以外の場合（存在するレコードを指し示している場合）は true を返す．

■ レコードのカラムの取り出し

カーソルが指し示すレコードからカラムの値を取り出す方法に getString メソッドがある．これは ResultSet オブジェクトに対するメソッドであり，このメソッドの引数にカラムの位置を意味する整数値を与えて実行すると，当該カラムの値が文字列として得られる．先頭カラムの位置は整数値の 1 である．また，レコードが持つカラムの個数は ResultSetMetaData オブジェクトに対して getColumnCount メソッドを実行すると得られる．

プログラム SQLite01.java を実行した結果の例を「実行例 11.4.2.2」に示す．

11.4.3 テーブルの結合

データベースの複数のテーブルを**結合**して 1 つのテーブルのように扱うことができる．実際のデータベースシステムでは，データのカテゴリや使用目的毎に複数のテーブルが作られており，それらを結合して高度な情報抽出を行う．具体的には SELECT 文の INNER JOIN 句などを記述して複数のテーブルの結合を指定する．

書き方：　SELECT カラムの記述 FROM 主たるテーブル名 INNER JOIN 結合するテーブル
ON 結合するカラムの対応

[51] カーソルを 1 つ戻すには previous メソッドを使用する．

実行例 11.4.2.2：SQLite01.java の実行結果の例

```
con：org.sqlite.jdbc4.JDBC4Connection@5e9f23b4
st：org.sqlite.jdbc4.JDBC4Statement@4783da3f

tblPref の作成
ret（更新件数）：0
tblPref へのレコードの追加
ret（更新件数）：4
tblGeom の作成
ret（更新件数）：0
tblGeom へのレコードの追加
ret（更新件数）：4

tblPref の閲覧
13，東京都，
22，愛知県，
26，京都府，
27，大阪府，

tblGeom の閲覧
東京都，2191，
愛知県，5172，
大阪府，1905，
神奈川県，2416，
```

ここで扱っているサンプルでは，tblPref のテーブルに「都道府県」の名前を持つカラムがあり，これを tblGeom の主キーに結合することができる．それによって，2 つのテーブルを結合して 1 つのテーブルのように扱うことができる．以下にその例を示す．

テーブルの作成：

```
SELECT tblPref.コード，tblPref.都道府県，tblGeom.面積 FROM tblPref
                        INNER JOIN tblGeom ON tblPref.都道府県=tblGeom.都道府県
```

更に別の結合方法である**左外部結合** LEFT OUTER JOIN についても紹介する．

書き方：　SELECT カラムの記述 FROM 主たるテーブル名 LEFT OUTER JOIN 結合するテーブル ON 結合するカラムの対応

11.4.3.1　サンプルプログラム (2)

内部結合，左外部結合ともにテーブルを結合するが，結合結果がそれぞれどのような形になるかをサンプルプログラム SQLite02.java で確認する．

プログラム：SQLite02.java

```
1  import java.sql.Connection;
2  import java.sql.DriverManager;
3  import java.sql.Statement;
4  import java.sql.ResultSet;
5  import java.sql.ResultSetMetaData;
6
7  class SQLite02 {
8      public static void main( String argv[] ) {
9          try {
10             Class.forName("org.sqlite.JDBC");   // イニシャライザ
11             // データベースへの接続
12             Connection con = DriverManager.getConnection(
13                                 "jdbc:sqlite:c:/SQLite/JPN01.db");
14             Statement st = con.createStatement();   // Statementオブジェクトの取得
15
```

```
16              // SQL実行に必要なもの
17              String sql;
18              ResultSet res;
19              ResultSetMetaData rsmd;
20
21              // テーブルの結合：内部結合
22              System.out.println("--- 内部結合（INNER JOIN）---");
23              sql = "SELECT tblPref.コード, tblPref.都道府県, "+
24                      "tblGeom.面積 FROM tblPref "+
25                      "INNER JOIN tblGeom ON tblPref.都道府県=tblGeom.都道府県";
26              res = st.executeQuery(sql);
27              // テーブルの閲覧(1)
28              rsmd = res.getMetaData();
29              while ( res.next() ) {
30                  for ( int i = 1; i <= rsmd.getColumnCount(); i++ ) {
31                      System.out.print( res.getString(i)+", " );
32                  }
33                  System.out.println();
34              }
35              System.out.println();
36
37              // テーブルの結合：左外部結合
38              System.out.println("--- 左外部結合（LEFT OUTER JOIN）---");
39              sql = "SELECT tblPref.コード, tblPref.都道府県, "+
40                  "tblGeom.面積 FROM tblPref "+
41                  "LEFT OUTER JOIN tblGeom ON tblPref.都道府県=tblGeom.都道府県";
42              res = st.executeQuery(sql);
43              // テーブルの閲覧(2)
44              rsmd = res.getMetaData();
45              while ( res.next() ) {
46                  for ( int i = 1; i <= rsmd.getColumnCount(); i++ ) {
47                      System.out.print( res.getString(i)+", " );
48                  }
49                  System.out.println();
50              }
51
52              // 終了
53              st.close();
54              con.close();
55          } catch(Exception e) {
56              System.out.println("エラー："+e);
57          }
58      }
59  }
```

このプログラムの実行結果は次のようになる．

```
--- 内部結合（INNER JOIN）---
 13, 東京都, 2191,
 22, 愛知県, 5172,
 27, 大阪府, 1905,
--- 左外部結合（LEFT OUTER JOIN）---
 13, 東京都, 2191,
 22, 愛知県, 5172,
 26, 京都府, null,
 27, 大阪府, 1905,
```

内部結合では，テーブルの間で外部キーと主キーの対応が取れているレコードのみが得られている．それに対して左外部結合では，外部キーに対応する主キーが結合先のテーブルに存在しないレコードに関しては「null」となることがわかる．

付録

A　Swing 簡易リファレンス

A.1　多くのコンポーネントで共通のメソッド

- void **setBounds(int Ph, int Pv, int W, int H)**　位置とサイズの設定
  コンテナ内の位置 (Ph,Pv) に，横幅 W，高さ H でコンポーネントを配置する．
- void **setFont(Font f)**　フォントの設定
  文字の表示フォントを f にする．
- void **setForeground(Color c) / setBackground(Color c)**　色の設定
  文字の色を setForeground で，背景色を setBackground で c に設定する．

A.2　各種のコンポーネント

■ **JTextField**　文字列の入出力のためのコンポーネント

主なメソッド：

- String **getText()**　文字の取得
  取得した文字列が String 型で返される．
- void **setText(String s)**　文字の設定
  s の内容が格納・表示される．

■ **JPasswordField**　パスワード入力のためのコンポーネント

主なメソッド：

- char[] **getPassword()**　パスワードの取得
  フィールドからパスワード文字列を取得して char の配列として返す．
- void **setEchoChar(char c)**　エコー文字の設定
  エコー文字（パスワード入力時に表示される隠蔽文字）を c に設定する．

■ **JLabel**　文字列の表示のためのコンポーネント

表示する文字列は，生成時にコンストラクタで指定する．

例．new JLabel(”文字列”)

■ **JButton**　ボタン

ボタンに表示する文字列は，生成時にコンストラクタで指定する．

例．new JButton(”文字列”)

ボタンに画像をアイコンとして表示することも可能である．画像ファイルを指定して ImageIcon オブジェクトを生成し，それを JButton のコンストラクタに指定すればよい．

例．ImageIcon icon = new ImageIcon(”画像ファイルのパス”);
　　JButton button = new JButton(icon);

ボタンがクリックされたら actionPerformed イベントが発生する．

■ **JCheckBox**　チェックボックス

チェックボックスに表示する文字列は，生成時にコンストラクタで指定する．

例．new JCheckBox(”文字列”)

主なメソッド：

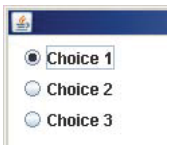
- boolean **isSelected()**　チェックの状態の取得
  ボックスがチェックされていれば true，チェックされていなければ false を返す．

- void **setSelected(boolean b)**　状態の設定

  ボックスの状態を設定する．コンストラクタで初期のチェック状態を設定することもできる．下記のようにコンストラクタの第 2 引数に true を与えると，チェックされた状態が初期状態となる．

  例．new JCheckBox(”文字列”,true)

■ **JRadioButton**　ラジオボタン



ラジオボタンに表示する文字列は，生成時にコンストラクタで指定する．

例．new JRadioButton(”文字列”)

ボタングループについて

ラジオボタンは，複数のものから択一式でチェックする形式をとる場合が多い．複数のラジオボタンをある**ボタングループ**のオブジェクトに登録することで，択一式の選択ができる．例えば，ButtonGroup オブジェクト bg の下にラジオボタン rb1,rb2,rb3 を登録して択一式選択をするには次のように記述する．

```
ButtonGroup bg = new ButtonGroup();    // ボタングループ bg の生成
JRadioButton rb1 = new JRadioButton("Choice 1");    // ラジオボタン 1
JRadioButton rb2 = new JRadioButton("Choice 2");    // ラジオボタン 2
JRadioButton rb3 = new JRadioButton("Choice 3");    // ラジオボタン 3
bg.add(rb1);          // ラジオボタン 1 を bg に登録
bg.add(rb2);          // ラジオボタン 2 を bg に登録
bg.add(rb3);          // ラジオボタン 3 を bg に登録
```

この例のように，add メソッドを用いてラジオボタンをボタングループに登録する．

主なメソッド：

- boolean **isSelected()**　状態の取得

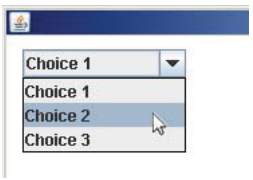
  ラジオボタンが on の状態であれば true，off の状態であれば false を返す．

- void **setSelected(boolean b)**　状態の設定

  ラジオボタンの状態を設定する．コンストラクタで初期の状態を設定することもできる．下記のようにコンストラクタの第 2 引数に true を与えると，on の状態が初期状態となる．

  例．new JRadioButton(”文字列”,true)

■ **JComboBox**　コンボボックス（選択ボックス）



選択項目はオブジェクト生成時にコンストラクタに指定する．例えば，

String itm[] = {”Choice 1”,”Choice 2”, ”Choice 3”};

JComboBox cbox = new JComboBox(itm);

とすると，itm に格納されている項目を選択項目として持つコンボボックス cobx が生成される．

主なメソッド：

- **getSelectedItem()**　選択されている項目の取得

  コンボボックスの現在選択されている項目を取得して返す．戻り値の型は選択項目のデータ型となる

- int **getSelectedIndex()**　選択されている項目のインデックスの取得

  コンボボックスの現在選択されている項目のインデックス値（n 番目）を取得して返す．

【スクロール領域の設置】

スクロール領域は JScrollPane クラスのオブジェクトであり，テキスト編集領域といった，不定の（大きな）表示サイズを必要とするコンポーネントを配置するのに用いられる．JScrollPane オブジェクト自体は他のコンポーネントを保持する「容器」と見ることができ，他のコンポーネントを保持するための**ビューポート**を内部に持つ．従って，スクロール機能を持つコンポーネントを実装するには，

(1) JScrollPane オブジェクト A の生成と配置
(2) GUI コンポーネント B の生成
(3) A のビューポートへの B の追加

という手順を踏む．以下にテキスト編集領域のための JTextArea クラスとともに説明する．

■ **JTextArea**　テキスト編集領域

主なメソッド：

- String **getText()**　テキスト（文字列）の取得
  取得したテキストが String 型で返される．JTextField と違い，複数の行にまたがるテキストを取得できる．
- void **setText(String s)**　テキストの設定
  s の内容が格納・表示される．

JTextArea オブジェクトをスクロール可能な編集領域にするには，setViewportView メソッドを用いて次の例のように JScrollPane クラスのオブジェクトのビューポートに追加する．

例.
```
JScrollPane scp = new JScrollPane();
JTextArea txt = new JTextArea();
scp.setViewportView(txt);
```

JTextArea オブジェクト txt から取得したテキストを行単位に分解するには次の例のように split メソッドを用いる．

```
String s[] = txt.split("¥n");
```

この例では，改行文字"¥n"を区切りにして txt を分解したものを文字列の配列 s[ ] に与えている．

参考：　テキスト編集領域としては JTextArea 以外にも，高度な表示機能や編集機能を備えた JEditPane クラスや JTextPane クラスがある．

■ **JSpinner**　値を選択するためのコンポーネント (1)

JSpinner は定められた選択肢を与える **SpinnerModel インターフェース**に基づいて値の範囲を提示し，その範囲の中から値を選択して取得するためのコンポーネントである．

SpinnerModel インターフェース

SpinnerModel は，どのような型の値をどの範囲で提示するかを定める．例えば「数」の範囲を与えるモデルに SpinnerNumberModel クラス（下記）がある．

```
SpinnerNumberModel( 初期値, 最小値, 最大値, 増減の幅 )
```

次のような例を考える．

```
SpinnerNumberModel mdl = new SpinnerNumberModel( 1, -4, 6, 2 );
JSpinner spn = new JSpinner(mdl);
```

この例では，整数-4〜6 の範囲で 2 刻みで値を選択する JSpinner オブジェクト spn が生成され，初期値として 1 が表示される．最大値や最小値に null を指定すると上限や下限を無制限にできる．

SpinnerListModel

SpinnerListModel は任意の選択肢データを配列の型で与えるものである．

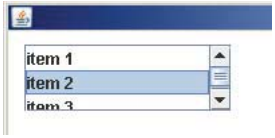
例.
```
String items[ ] = {"Choice - A","Choice - B","Choice - C"};
SpinnerListModel mdl = new SpinnerListModel(items);
JSpinner spinner = new JSpinner(mdl);
```

この例では選択肢として"Choice - A","Choice - B","Choice - C"が提示される．

参考：　SpinnerModel インターフェースには，これらの他にも SpinnerDateModel などがある．

JSpinner オブジェクトから値を取得するには，そのオブジェクトに設定されている SpinnerModel のオブジェクトに対して getValue メソッドを実行する．

■ **JList**　値を選択するためのコンポーネント (2)



JList は定められた選択肢の範囲の中から値を選択して取得するためのコンポーネントである．選択肢の数によっては，JScrollPane オブジェクトのビューポートに与えて使用するのが良い．

項目の設定方法

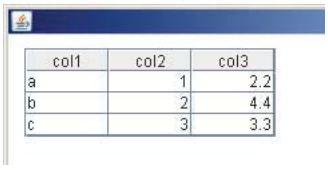
JList オブジェクト生成時のコンストラクタに，選択項目を格納した配列を与える．

```
String items[] = {"item 1","item 2","item 3","item 4"};
JList lst = new JList(items);
JScrollPane scp = new JScrollPane();
scrollPane.setViewportView(lst);
```

JList オブジェクトの選択されている項目を取得するには，getSelectedValue メソッド を用いる方法と，getSelectedIndex メソッドを用いる方法がある．前者は選択されている項目を，後者は項目のインデックスを返す．

参考：　項目を動的に追加・削除するには **ListModel インターフェース**を使用する．

■ **JTable**　表



col1	col2	col3
a	1	2.2
b	2	4.4
c	3	3.3

行列形式の表を実装する．実際のデータは TableModel インターフェースのオブジェクトに保持される．TableModel インターフェースを実装するクラスに DefaultTableModel クラスがある．

DefaultTableModel クラス：

表形式でデータを保持するデータモデルである．このクラスのオブジェクトを生成する際に，コンストラクタに次のように指定する．

```
DefaultTableModel( 2 次元の配列, 列見出しの文字列の配列 )
```

生成したテーブルモデルは setModel メソッドにより JTable オブジェクトに与える．

上記の図のような表を作成するには次の例のように記述する．

```
DefaultTableModel tbm = new DefaultTableModel(
    new Object[][] {
        {"a", 1, 2.2},
        {"b", 2, 4.4},
        {"c", 3, 3.3},
    },
    new String[] {"col1", "col2", "col3"}
);
tbl = new JTable();
tbl.setModel(tbm);
```

（先に javax.swing.table.*を import しておく必要がある.）

表のサイズによっては，JTable オブジェクトを JScrollPane オブジェクトのビューポートに与えて使用するのが良い．

表のセルの値を取得するには getValueAt メソッド，設定するには setValueAt メソッドを JTable オブジェクトに対して実行する．

セルの値の取得：　getValueAt(行, 列)

セルの値の設定： setValueAt(値, 行, 列)

■ **JTree** 階層構造図

階層構造図を実装する．階層構造のデータは TreeNode インターフェースのオブジェクトに保持される．TreeNode インターフェースを実装するクラスに DefaultMutableTreeNode クラスがある．

DefaultTableModel クラス：

階層構造のデータを表現するためのクラスである．このクラスのオブジェクトは同クラスの別のオブジェクトを子に持つことができ，それらの連結関係で階層構造を構成する．生成する際に，コンストラクタに次のように指定する．

DefaultMutableTreeNode(ノードの名前)

子ノードの接続には add メソッドを用いる．図 61 のような階層図を構築する例について考える．

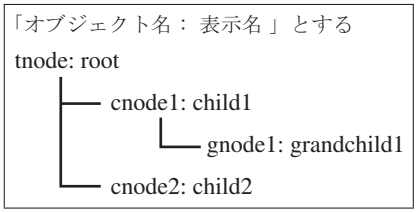


図 61: 階層図の例

下記のように記述することでこの階層図を実装できる．

```
// 階層構造の構築
DefaultMutableTreeNode tnode = new DefaultMutableTreeNode("root");
DefaultMutableTreeNode cnode1 = new DefaultMutableTreeNode("child1");
DefaultMutableTreeNode cnode2 = new DefaultMutableTreeNode("child2");
tnode.add(cnode1);
tnode.add(cnode2);
DefaultMutableTreeNode gnode1 = new DefaultMutableTreeNode("grandchild1");
cnode1.add(gnode1);
// JTree オブジェクトに与える
JTree tree = new JTree(tnode);
```

（先に javax.swing.tree.*を import しておく必要がある.）

この例により tree は図 62 のように表示される．

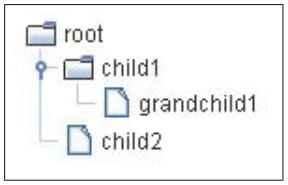
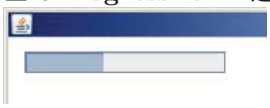


図 62: 実際の階層図

図のサイズによっては，JTree オブジェクトを JScrollPane オブジェクトのビューポートに与えて使用するのが良い．

■ **JProgressBar** 進捗バー



進捗バーは JProgressBar クラスのオブジェクトとして実装される．このオブジェクトに対して 0〜100 の整数値を設定することで，値に応じた進捗状態が表示される．値の設定と取得は setValue メソッドと getValue メソッドで行う．

値 v の設定：　void **setValue(int v)**
値の取得　：　int **getValue()**

値の上限と下限を変更することも可能である．

下限の設定：　void **setMinimum(int min)**
上限の設定：　void **setMaximum(int max)**

■ **JSlider**　スライダ

スライダは JSlider クラスのオブジェクトとして実装される．値の設定と取得，上限・下限の設定は JProgressBar の場合と同じメソッドが使える．

A.3　主な GUI コンポーネントのクラス階層

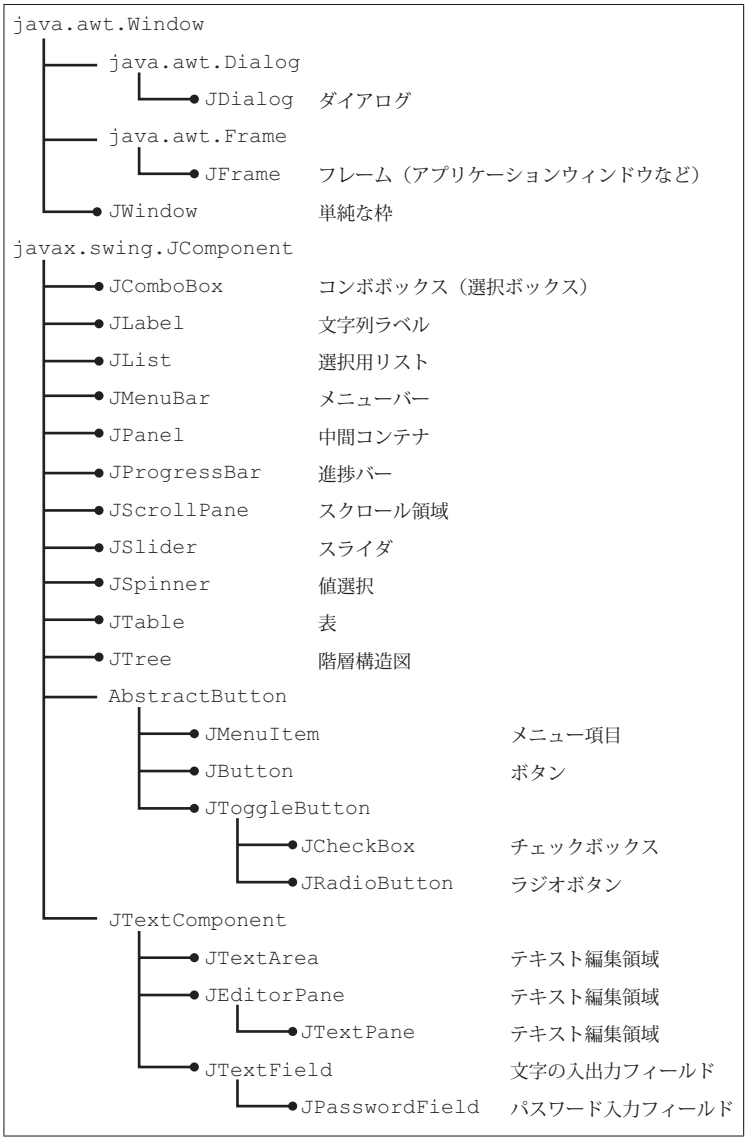


図 63: GUI コンポーネントのクラス階層

A.4　代表的なイベント

代表的なイベント（java.awt.event, javax.swing.event）を図 64 に示す.

```
 イベント                                                          イベントリスナ

(java.awt.event)
ActionEvent ------------------------------------------------------- ActionListener
  └──● actionPerformed    ボタンが押されたときなど

(java.awt.event)
KeyEvent ---------------------------------------------------------- KeyListener
  ├──● keyPressed         キーボードが押下された
  ├──● keyReleased        押されていたキーボードが放された
  └──● keyTyped           キーボードがタイプされた

(java.awt.event)
MouseEvent
  ├──● mouseClicked       マウスボタンがクリックされた
  ├──● mouseEntered       コンポーネントにマウスポインタがさしかかった
  ├──● mouseExited        コンポーネントからマウスポインタが離れた      --- MouseListener
  ├──● mousePressed       マウスボタンが押下された
  ├──● mouseReleased      押されていたマウスボタンが放された
  ├──● mouseDragged       マウスドラッグ                                --- MouseMotionListener
  └──● mouseMoved         マウスが動かされた

(java.awt.event)
ItemEvent --------------------------------------------------------- ItemListener
  └──● itemStateChanged

(javax.swing.event)
ChangeEvent ------------------------------------------------------- ChangeListener
  └──● stateChanged       状態が変わった（JSlider など）

(java.awt.event)
AdjustmentEvent --------------------------------------------------- AdjustmentListener
  └──● adjustmentValueChanged     調整された（スクロールバーなど）

(java.awt.event)
WindowEvent ------------------------------------------------------- WindowListener
  ├──● windowActivated    ウィンドウがアクティブになった
  ├──● windowDeactivated  ウィンドウがアクティブでなくなった
  ├──● windowOpened       ウィンドウが開かれた
  ├──● windowClosed       ウィンドウが閉じられた
  ├──● windowClosing      ウィンドウを閉じようとした
  ├──● windowIconified    ウィンドウが最小化された
  └──● windowDeiconified  最小化されたウィンドウが元に戻った
```

図 64: 代表的なイベント

B　SWT 簡易リファレンス

B.1　多くのコントロールで共通のメソッド

- **setBounds(int Ph, int Pv, int W, int H)**　位置とサイズの設定
  ウィンドウ内の位置 (Ph,Pv) に，横幅 W，高さ H でコントロールを配置する．
- **setFont(Font f)**　フォントの設定
  文字の表示フォントを f にする．
- **setForeground(Color c) / setBackground(Color c)**　色の設定
  オブジェクト前面の色を setForeground で，背景色を setBackground で c に設定する．

B.2　各種のコントロール

■ **Text**　文字列の入出力のためのコントロール

コンストラクタ：　Text(Shell sh, スタイル)

スタイルに設定できるものを表 53 に示す．

表 53: Text のスタイル一覧

値	スタイル
SWT.BORDER	枠の表示
SWT.NONE	枠なし
SWT.PASSWORD	パスワード入力フィールド
SWT.SINGLE	1 行のテキストフィールド（デフォルト）
SWT.MULTI	複数行のテキスト領域
SWT.H_SCROLL	水平スクロールバーの表示
SWT.V_SCROLL	垂直スクロールバーの表示
SWT.WRAP	行の折り返し表示

スタイルはビット和（論理和）で合成することができる．

主なメソッド：

- String **getText()**　文字の取得
  取得した文字列が String 型で返される．
- void **setText(String s)**　文字の設定
  s の内容が格納・表示される．

■ **Label**　文字列の表示のためのコントロール

コンストラクタ：　Label(Shell sh, スタイル)

表示する文字列は，setText メソッドで与える．

スタイルに設定できるものを表 54 に示す．

表 54: Label のスタイル一覧

値	スタイル
SWT.BORDER	枠の表示
SWT.NONE	枠なし
SWT.SEPARATOR	線の表示（水平／垂直の選択には下の 2 つを追加指定する）
SWT.HORIZONTAL	水平（SEPARATOR と併用）
SWT.VERTICAL	垂直（SEPARATOR と併用）

スタイルはビット和（論理和）で合成することができる．

■ **Button**　ボタン

コンストラクタ：　Button(Shell sh, スタイル)

ボタンに表示する文字列は，setText メソッドで与える．
スタイルに設定できるものを表 55 に示す．

表 55: Button のスタイル一覧

値	スタイル
SWT.FLAT	平坦なボタン
SWT.ARROW	矢印ボタン（方向に関しては下の 4 つを追加指定する）
SWT.UP	上向き矢印ボタン（ARROW と併用：デフォルト）
SWT.DOWN	下向き矢印ボタン（ARROW と併用）
SWT.RIGHT	右向き矢印ボタン（ARROW と併用）
SWT.LEFT	左向き矢印ボタン（ARROW と併用）
SWT.RADIO	ラジオボタン
SWT.CHECK	チェックボックス
SWT.TOGGLE	トグルボタン

スタイルはビット和（論理和）で合成することができる．
チェックボックスやラジオボタンの場合，setSelection メソッドで選択状態を設定することができる．

例． chkBtn.setSelection(true);
これは，ボタン chkBtn をチェック状態にする例である．

ラジオボタンについて

ラジオボタンは択一式の選択をするコントロールであり，基本的には 1 つのウィンドウ（Shell オブジェクト）内に設置されたラジオボタンの中から 1 つのボタンを選択をする．また，**グループ**（Group）と呼ばれる**コンポジット**（Composite）にラジオボタンを配置すると，同一コンポジット内で 1 つの選択ができるようになる．すなわち，1 つの Shell 内に複数のグループコンポジットオブジェクトを配置することでラジオボタンのグループ化ができる．
コンポジットは Swing の中間コンテナに例えることができ，そこに GUI のコントロールを配置することができる．

例．ウィンドウ sh 内の Group オブジェクト gr にラジオボタン r1,r2 を配置する例

```
// グループオブジェクトの生成
Group gr = new Group(sh, SWT.NONE);
gr.setText("Group1");
gr.setBounds(10, 12, 110, 80);
// ラジオボタン 1
Button r1 = new Button(gr, SWT.RADIO);
r1.setBounds(0, 20, 72, 22);
r1.setText("radio-1");
// ラジオボタン 2
Button r2 = new Button(gr, SWT.RADIO);
r2.setBounds(0, 48, 72, 22);
r2.setText("radio-2");
```

これを実行すると図 65 のような表示となる．

図 65: ラジオボタンのグループ化

■ **Combo**　コンボボックス（選択ボックス）

コンストラクタ：　Combo(Shell sh, スタイル)

スタイルに設定できるものを表 56 に示す．

表 56: Combo のスタイル一覧

値	スタイル
SWT.SIMPLE	単純な選択ボックス（カーソルキーで選択）
SWT.DROP_DOWN	選択ボタン付き（デフォルト）

選択項目は setItems メソッドを用いて登録する．

例．選択項目の登録

```
String items[] = {"item-0","item-1","item-2","item-3"};
Combo cmb = new Combo(sh, SWT.NONE);
cmb.setItems(items);
```

この例は，items に格納されている項目を選択項目として持つコンボボックス cmb を生成するものである．

選択の初期値を設定するには select メソッドを用いる．

初期状態の設定：　select(int i)

項目の配列の i 番目を初期状態とする．

Combo オブジェクトで選択されている値（文字列）を取得するには getText メソッドを用いる．

■ **Spinner**　値を選択するためのコントロール (1)

Spinner は設定した範囲から値を選択する．

コンストラクタ：　Spinner(Shell sh, スタイル)

スタイルに設定できるものを表 57 に示す．

表 57: Spinner のスタイル

値	スタイル
SWT.BORDER	枠の表示

重要なメソッド

最小値の設定：　setMinimum(int min)

最大値の設定：　setMaximum(int max)

増減幅の設定：　setIncrement(int inc)

初期値の設定：　setSelection(int ini)

値の取得：　getSelection()

■ **List**　値を選択するためのコントロール (2)

コンストラクタ：　List(Shell sh, スタイル)

スタイルに設定できるものを表 58 に示す．

表 58: List のスタイル一覧

値	スタイル
SWT.BORDER	枠の表示
SWT.MULTI	複数項目同時選択の許可
SWT.V_SCROLL	垂直スクロールバーの表示

項目の設定は Combo コントロールと同様に setItems メソッドを用いる．選択されている項目を取得するには **getSelection** メソッドを用いる．List コントロールでは複数の項目を同時選択する可能性があるので，getSelection は配列を返す．

■ **Table**　表

コンストラクタ：　Table(Shell sh, スタイル)

スタイルに設定できるものを表 59 に示す.

表 59: Table のスタイル一覧

値	スタイル
SWT.BORDER	枠の表示
SWT.H_SCROLL	水平スクロールバーの表示
SWT.V_SCROLL	垂直スクロールバーの表示
SWT.FULL_SELECTION	表内選択可能

表の構造と設置方法

Table は **TableColumn** クラスの「列」と，**TableItem** クラスの「行」を加えて使用する．表を構成するには，最初に Table オブジェクトを生成し，それに対して TableColumn オブジェクトを加えることで列（項目あるいはフィールド）の構造を決定する．更に TableItem オブジェクトを順次加えてゆくことで行を増やしてゆく．

TableColumn オブジェクトに setText メソッドを実行することで列のタイトルを設定し，表のセルへの値の設定は，TableItem オブジェクトに対して setText メソッドを実行することで行う．最後に表の各列に対して pack メソッドを実行して調整を行い，表示できる状態にする．

次に 7 行 10 列の表を作成する例を示す.

例. ウィンドウ sh 上に 7 行 10 列の表を作成する

```
// テーブルの生成
tbl = new Table(sh, SWT.BORDER);
// 列構造の決定
int i, j;
String nm;
// 列構造を決めるループ
for ( i = 0; i < 10; i++ ) {
    // 列オブジェクトを順次 tbl に登録
    TableColumn tc = new TableColumn(tbl,SWT.NULL);
    nm = "col"+i;
    // 列のタイトルの設定
    tc.setText(nm);
}
// 行を追加するループ
for ( j = 0; j < 7; j++ ) {
    // 行オブジェクトの生成と tbl への追加
    TableItem ti = new TableItem(tbl,SWT.NULL);
    nm = "item"+i;
    ti.setText(nm);
    // 行の各フィールドへの値の設定
    for ( i = 0; i < 10; i++ ) {
        nm = "item("+j+","+i+")";
        ti.setText(i,nm);
    }
}
// 各列の調整
for ( i = 0; i < 10; i++ ) {
    tbl.getColumn(i).pack();
}
```

表の各セルへのアクセスは，

(1) Table オブジェクトに対して getItem メソッドを用いて「行」を取り出す

(2) 上で得られた行に対して getText,setText メソッドを用いて値の取得／設定を行う

という手順で行う．行オブジェクト ti の n 番目のセルに対してアクセスするには次のようにする．

文字列 s をセット　　：　ti.setText(n,s);
取り出した値を文字列 s に取得：　s = ti.getText(n);

■ **Tree**　階層構造図

コンストラクタ：　Tree(Shell sh, スタイル)

スタイルに設定できるものを表 60 に示す．

表 60: Tree のスタイル一覧

値	スタイル
SWT.BORDER	枠の表示
SWT.H_SCROLL	水平スクロールバーの表示
SWT.V_SCROLL	垂直スクロールバーの表示

Tree の構造と設置方法

Tree には **TreeItem** クラスのオブジェクトを項目として加えて階層構造を構築する．生成した Tree オブジェクトに項目を追加するには TreeItem オブジェクトを順次生成してゆく．生成した TableItem オブジェクトに対して setText メソッドを用いて項目のデータを設定する．TreeItem オブジェクト生成時のコンストラクタの引数には次のように指定する．

TreeItem(親オブジェクト, スタイル)

次に階層構造を構築する例を示す．

例. ウィンドウ sh 上に Tree オブジェクト tr を作成して階層を構築する（深さ=3）

```
Tree tr = new Tree(sh, SWT.BORDER);
for ( i = 0; i < 4; i++ ) {
    TreeItem tr1 = new TreeItem(tr,SWT.NONE);
    tr1.setText("level("+i+")");
    for ( j = 0; j < 4; j++ ) {
        TreeItem tr2 = new TreeItem(tr1,SWT.NONE);
        tr2.setText("level("+i+","+j+")");
        for ( k = 0; k < 4; k++ ) {
            TreeItem tr3 = new TreeItem(tr2,SWT.NONE);
            tr3.setText("level("+i+","+j+","+k+")");
        }
    }
}
```

Tree の各項目へのアクセスは getItem メソッドを用いる．

getItem(int n)　：　n 番目の子のオブジェクトを返す．

これに対して getText ／ setText メソッドで値の取得と設定を行う．

■ **ProgressBar**　進捗バー

コンストラクタ：　ProgressBar(Shell sh, スタイル)

スタイルに SWT.SMOOTH を指定すると間隙なしのバーになる．

主なメソッド：

バーの最小値の設定：　setMinimum(int min);
バーの最大値の設定：　setMaximum(int max);
バーへの値の設定：　setSelection(int v);
バーの値の取得：　getSelection();

■ **Slider / Scale**　スライダ／スケール

コンストラクタ：　Slider(Shell sh, スタイル)

コンストラクタ：　Scale(Shell sh, スタイル)

スタイルに SWT.HORIZONTAL を指定すると水平，SWT.VERTICAL を指定すると垂直のコントロールになる．メソッドに関しては進捗バーとほぼ同じものが使用できる．

B.3　代表的なイベント

SWT で扱う代表的なイベントを表 61 に示す．

表 61: 代表的なイベント

イベントリスナ／アダプタ	登録メソッド	イベント	
SelectionAdapter	addSelectionListener	widgetSelected	ボタンが押されたときや メニューが選択されたときなど
MouseMoveListener	addMouseMoveListener	mouseMove	マウスが動かされたとき
MouseTrackAdapter	addMouseTrackListener	mouseEnter mouseExit mouseHover	マウスがさしかかった マウスが出て行った マウスポインタが上にある
MouseWheelListener	addMouseWheelListener	mouseScrolled	マウスホイールが操作された
MouseAdapter	addMouseListener	mouseDoubleClick mouseDown mouseUp	マウスボタンがダブルクリックされた マウスボタンが押された マウスボタンが放された
KeyAdapter	addKeyListener	keyPressed keyReleased	キーボードが押された キーボードが放された
ShellAdapter	addShellListener	shellActivated shellDeactivated shellClosed shellIconified shellDeiconified	ウィンドウがアクティブになった ウィンドウがアクティブでなくなった ウィンドウが閉じられた ウィンドウが最小化された 最小化されたウィンドウが元に戻った

C　java.nio ライブラリ

ファイル入出力では File クラスが用いられてきたが，新しい版の Java ではパスとファイルの扱いにおいて java.nio ライブラリの使用が推奨されている.（File クラスも後方互換性のために今後も存続する予定である）

C.1　Path インターフェースと Paths クラス

■ パスの作成

Paths クラス（java.nio.file.Paths）を用いると Path オブジェクトを生成することができる.

書き方：　Paths.get(要素 1, 要素 2, ··· , 要素 n)

例. `Paths.get("/","home","katsu","test1.dat")`
これは '`/home/katsu/test1.dat`' の Path オブジェクトとなる.

■ パスを文字列に変換

Paths オブジェクトに対して toString() メソッドを実行すると，そのパスを表す文字列を返す.

書き方：　Path オブジェクト.toString( )

C.2　Files クラス

Files クラス（java.nio.file.Files）には表 62 に示すようなクラスメソッドがあり，Path オブジェクトを用いたファイル入出力ができる.

表 62: java.nio.file.Files のクラスメソッド（一部）

メソッド	解説
Files.newInputStream(Path p)	Path p が示す InputStream オブジェクトを返す.
Files.newOutputStream(Path p)	Path p が示す OutputStream オブジェクトを返す.
Files.newBufferedReader(Path p)	Path p が示す BufferedReader オブジェクトを返す.
Files.newBufferedWriter(Path p)	Path p が示す BufferedWriter オブジェクトを返す.
Files.copy(Path p1, Path p2)	Path p1 が示すファイルを Path p2 として複製する.
Files.createFile(Path f)	Path f が示す空のファイルを作成する.
Files.createDirectories(Path d)	Path d が示すディレクトリを作成する.
Files.delete(Path p)	Path p が示すファイルを削除する.
Files.exists(Path p)	Path p が示すファイルが存在すれば true を，存在しなければ false を返す.
Files.isDirectory(Path p)	Path p がディレクトリならば true を，そうでなければ false を返す.
Files.list(Path d)	Path d が示すディレクトリ配下の要素（ファイル，サブディレクトリなどディレクトリ内容の一覧）を Stream オブジェクトとして返す.
Files.size(Path p)	Path p が示すファイルのサイズ（バイト数）を long 型で返す.

※ この他にも多数のメソッドがある．また，メソッドの引数には様々なオプションを与えることができる．詳しくは Oracle の公式インターネットサイトを参照のこと.

Path オブジェクトと Files クラスを用いてファイル入出力を行うサンプルプログラム NioTest01.java を示す.

プログラム：NioTest01.java

```
1 import java.nio.file.Path;
2 import java.nio.file.Paths;
3 import java.nio.file.Files;
4 import java.io.BufferedWriter;
5 import java.io.BufferedReader;
6 import java.io.PrintWriter;
```

```
 7 import java.io.IOException;
 8
 9 class NioTest01 {
10     public static void main( String argv[] ) throws IOException {
11         // パスの作成
12         Path pth = Paths.get("C:","Users","katsu","java","jtext01.txt");
13         System.out.println("パス："+pth);
14
15         // ファイル出力
16         BufferedWriter bw = Files.newBufferedWriter( pth );
17         PrintWriter pw = new PrintWriter(bw);
18         pw.println("テキストの1行目です. ");
19         pw.println("2行目です. ");
20         pw.println("さんぎょうめ");
21         pw.close();
22
23         // ファイルの読み込み
24         String s;
25         BufferedReader f = Files.newBufferedReader( pth );
26         while ( ( s = f.readLine()) != null ) {
27             System.out.println(s);
28         }
29         f.close();
30
31         // ファイルのサイズ
32         System.out.println("サイズ："+Files.size(pth));
33     }
34 }
```

このプログラムでは，出力ファイルを示す Path オブジェクト `pth` から一旦 BufferedWriter オブジェクト `bw` を作成し，それを元に出力用の PrintWriter オブエジェクト `pw` を生成している．このステップは toString メソッドを用いると更に短縮することができる．例えば 16 行目と 17 行目を

```
PrintWriter pw = new PrintWriter( pth.toString() );
```

と 1 つの行にする（BufferedWriter を中継しない形にする）ことができる．

このプログラムを実行した例を次に示す．

```
パス：C:¥Users¥katsu¥java¥jtext01.txt
テキストの 1 行目です.
 2 行目です.
さんぎょうめ
サイズ：71
```

出力用のファイル `C:¥Users¥katsu¥java¥jtext01.txt` が作成され，保存内容が表示されている．

次に，ディレクトリとファイルの作成，一覧作成，削除を行うサンプルプログラム NioTest02_1.java，NioTest02_2.java，NioTest02_3.java を示す．

プログラム：NioTest02_1.java

```
 1 import java.nio.file.Path;
 2 import java.nio.file.Paths;
 3 import java.nio.file.Files;
 4 import java.io.IOException;
 5
 6 class NioTest02_1 {
 7     public static void main( String argv[] ) throws IOException {
 8         // ディレクトリの作成
 9         Path d = Paths.get("C:","Users","katsu","TestDir");
10         Files.createDirectories( d );
11         System.out.println("ディレクトリ"+d.toString()+"を作成しました. ");
12
13         // ファイルの作成
14         String fname;
15         Path f;
16         for ( int c = 1; c <= 3; c++ ) {
```

```
17                fname = String.format("file%02d",c) + ".txt";
18                f = Paths.get("C:","Users","katsu","TestDir",fname);
19                Files.createFile( f );
20                System.out.println("ファイル"+f.toString()+"を作成しました. ");
21            }
22        }
23    }
```

このプログラムはディレクトリを作成してその中にファイルを作成するものである．実行すると次のように表示される．

ディレクトリ C:¥Users¥katsu¥TestDir を作成しました.
ファイル C:¥Users¥katsu¥TestDir¥file01.txt を作成しました.
ファイル C:¥Users¥katsu¥TestDir¥file02.txt を作成しました.
ファイル C:¥Users¥katsu¥TestDir¥file03.txt を作成しました.

注意）作成しようとするディレクトリやファイルが既に存在する場合はエラー（FileAlreadyExistsException）が発生する．

プログラム：NioTest02_2.java

```
1  import java.nio.file.Path;
2  import java.nio.file.Paths;
3  import java.nio.file.Files;
4  import java.io.IOException;
5  import java.util.stream.Stream;
6
7  class NioTest02_2 {
8      public static void main( String argv[] ) throws IOException {
9          // ディレクトリのパス
10         Path d = Paths.get("C:","Users","katsu","TestDir");
11         System.out.println("ディレクトリ"+d.toString()+"以下のファイル：");
12         // ディレクトリ内の要素
13         Stream<Path> e = Files.list( d );
14         e.forEach( f -> System.out.println(f.toString()) );
15     }
16 }
```

このプログラムはディレクトリ配下の要素（ファイル，サブディレクトリなど）を列挙するものである．実行すると次のように表示される．

ディレクトリ C:¥Users¥katsu¥TestDir 以下のファイル：
C:¥Users¥katsu¥TestDir¥file01.txt
C:¥Users¥katsu¥TestDir¥file02.txt
C:¥Users¥katsu¥TestDir¥file03.txt

注意）対象となるディレクトリが存在しない場合はエラー（NoSuchFileException ）が発生する．

プログラム：NioTest02_3.java

```
1  import java.nio.file.Path;
2  import java.nio.file.Paths;
3  import java.nio.file.Files;
4  import java.io.IOException;
5
6  class NioTest02_3 {
7      public static void main( String argv[] ) throws IOException {
8          // ファイルの削除
9          String fname;
10         Path f;
11         for ( int c = 1; c <= 3; c++ ) {
12             fname = String.format("file%02d",c) + ".txt";
13             f = Paths.get("C:","Users","katsu","TestDir",fname);
14             Files.delete( f );
15             System.out.println("ファイル"+f.toString()+"を削除しました. ");
16         }
17
```

```
18          // ディレクトリの削除
19          Path d = Paths.get("C:","Users","katsu","TestDir");
20          Files.delete( d );
21          System.out.println("ディレクトリ"+d.toString()+"を削除しました. ");
22      }
23  }
```

このプログラムはディレクトリ内のファイルを全て削除した後に，そのディレクトリを削除するものである．実行すると次のように表示される．

```
ファイル C:¥Users¥katsu¥TestDir¥file01.txt を削除しました.
ファイル C:¥Users¥katsu¥TestDir¥file02.txt を削除しました.
ファイル C:¥Users¥katsu¥TestDir¥file03.txt を削除しました.
ディレクトリ C:¥Users¥katsu¥TestDir を削除しました.
```

注意）削除しようとするディレクトリやファイルが存在しない場合はエラー（NoSuchFileException）が発生する．

D　Javadoc

Java を配布する Oracle 社の公式インターネットサイトでは，各種の API に関する多くのドキュメントが公開されている．

https://docs.oracle.com/javase/jp/7/api/java/util/ArrayList.html

図 66: Oracle 社が公開する ArrayList クラスに関するドキュメント

このような HTML ドキュメントは /* ～ */ で括るコメントを用いて作成することができる．例えばつぎのようなプログラム tasu.java について考える．

プログラム：tasu.java

```
1  /**
2   *  <pre>
3   *  2つのコマンド引数の足し算を実行するプログラム.
4   *  計算結果を標準出力に出力します.
5   *  </pre>
6   *  @author      中村勝則
7   */
8  public class tasu {
9      /**
10      *  <pre>コンストラクタは特に必要ありません</pre>
11      */
12     tasu() {
13     }
14     /**
15      *  <pre>
16      *  プログラム本体となるメソッド. 引数列の最初の2つの要素を加算します.
17      *  </pre>
18      *  @param argv      コマンド引数の列
19      *  @param argv[0]   加算に用いる数(1)
20      *  @param argv[1]   加算に用いる数(2)
21      */
22     public static void main( String argv[] ) {
23         int a = Integer.parseInt(argv[0]);       // 最初の引数を整数に変換
24         int b = Integer.parseInt(argv[1]);       // 次の引数を整数に変換
25         int c = a + b;
26         // 加算結果を表示
27         System.out.println( argv[0]+"+"+argv[1]+"="+c );
28     }
29 }
```

これは，起動時のコマンド引数を加算する単純なプログラムであるが，クラスやメソッドの定義の直前に /* ～ */ で括られたコメントが記述されている．それらコメントの中には ‘@’ で始まるドキュメント用の**タグ**が記述されてい

る．これらタグを用いて開発者の名前や，クラス，メソッド，引数の意味など（その他多数記述可）を記述する．

javadoc コマンドを用いると，このようなコメント記述を施したプログラムから HTML ドキュメントを生成することができる．

例． HTML ドキュメントの作成

```
javadoc tasu.java -d html  [Enter]     ←コマンドの発行
ソース・ファイル tasu.java を読み込んでいます...
Javadoc 情報を構築しています...
標準 Doclet バージョン 1.8.0_231
全パッケージとクラスの階層ツリーを作成しています...
.¥tasu.html の生成中...
.¥package-frame.html の生成中...
                :
          （途中省略）
                :
.¥index.html の生成中...
.¥help-doc.html の生成中...
```

コマンド起動時のオプション ‘-d html’ は，生成する文書群の保存先を指定するものであり，この例では保存先として ‘html’ というディレクトリを指定している．この処理の結果，次に示すような HTML などの文書群が ‘html’ の中に生成される．

例． 生成された文書群

allclasses-frame.html	allclasses-noframe.html	constant-values.html
deprecated-list.html	help-doc.html	index-all.html
index.html	overview-tree.html	package-frame.html
package-summary.html	package-tree.html	tasu.html

これら HTML 文書の内，index.html を Web ブラウザで表示した例が図 67 である．

図 67: 作成された HTML ドキュメント（index.html）

D.1　ソースプログラムの文字コード（エンコーディング）の指定

ソースプログラムの文字コード（エンコーディング）は javadoc 起動時にオプション -encoding で指定する．

例． ソースプログラムが utf-8 で記述されている場合

```
javadoc -encoding utf-8 tasu.java -d html
```

D.2　コメントの記述形式とタグ

Javadoc のコメントは，解説の対象となるクラスやメソッドの直前に記述する．またその場合

```
/**
        （コメントの記述）
*/
```

のように，コメントの書き始めは '*' を 2 つ以上続ける．

Javadoc のタグの重要なものを表に示す．

表 63: Javadoc のタグ（一部）

タグ	解説
`<pre>` 解説文 `</pre>`	解説文
`@author` 氏名	開発者名
`@param` 引数名 解説	引数に関する解説
`@return` 解説	戻り値に関する解説
`@version` バージョン番号	バージョン番号

E　サードパーティのライブラリ

E.1　Apfloat

Apfloat は，多倍長精度の浮動小数点数の演算を行うためのライブラリである．このソフトウェアはフィンランドの Mikko Tommila 氏が開発し，LGPL でライセンスされている．このライブラリと関連の情報は Apfloat の公式インターネットサイト `http://www.apfloat.org/` から入手することができる．このパッケージを利用するには，必要な jar ファイルを入手した後，それを利用する Java プログラムでインポート（`org.apfloat.*`）する．

【使用例】

■ コンストラクタ

多倍長精度の数値は Apfloat クラスのオブジェクトに保持する．例えば

```
Apfloat af1 = new Apfloat(1,80);
```

とすると，有効桁数が 80 桁で数値 1 を持つ Apfloat クラスのインスタンス `af1` が生成される．

例. 1/9 を 40 桁の有効桁数で近似値を求める

```
Apfloat af1 = new Apfloat(1,40);
Apfloat af9 = new Apfloat(1,40);
Apfloat afx = af1.divide(af9);
System.out.println(afx);
```

これを実行すると，

```
1.1111111111111111111111111111111111111111e-1
```

と表示される．より自然な形の浮動小数点表現で表示するには，出力時に

```
System.out.println(afx.toString(true));
```

とすると良い．この結果，

```
0.1111111111111111111111111111111111111111
```

と表示される．

■ 各種の関数

Apfloat パッケージで提供されているクラス ApfloatMath には各種の数学関数がクラス関数の形で定義されている．

例. 平方根 sqrt

```
Apfloat af2 = new Apfloat(2,40);
Apfloat afx = ApfloatMath.sqrt(af2);
System.out.println(afx.toString(true));
```

これを実行すると

```
1.4142135623730950488016887242096980785 69
```

と表示される．

例. 指数関数 exp

```
Apfloat af1 = new Apfloat(1,40);
Apfloat afx = ApfloatMath.exp(af1);
System.out.println(afx.toString(true));
```

これを実行すると

```
2.7182818284590452353602874713526624977 57
```

と表示される．

例. 円周率 pi(有効桁数)

```
Apfloat afx = ApfloatMath.pi(72);
System.out.println(afx.toString(true));
```

これを実行すると

```
3.141592653589793238462643383279502884197169399375105820974944592307816 4
```

と表示される.

表 64 に ApfloatMath クラスの数学関数の例を示す.

表 64: ApfloatMath クラスの数学関数の例

メソッド	機能	メソッド	機能
`abs(Apfloat x)`	x の絶対値	`pow(Apfloat x, Apfloat y)`	x^y（y は long も可）
`sqrt(Apfloat x)`	x の平方根	`cbrt(Apfloat x)`	x の立方根
`exp(Apfloat x)`	x の指数関数	`log(Apfloat x)`	x の対数関数
`sin(Apfloat x)`	x の正弦関数	`asin(Apfloat x)`	x の逆正弦関数
`cos(Apfloat x)`	x の余弦関数	`acos(Apfloat x)`	x の逆余弦関数
`tan(Apfloat x)`	x の正接関数	`atan(Apfloat x)`	x の逆正接関数
`toDegrees(Apfloat x)`	弧度法から度への変換	`toRadians(Apfloat x)`	度から弧度法への変換

F　サンプルプログラム

F.1　ArrayList，LinkedList の性能比較

リストは使用頻度の高いデータ構造である．ここでは ArrayList と LinkedList の性能を比較する．次に示すサンプルプログラム List02.java は，リストへの要素の追加，書き換え，参照，挿入，削除を行うのに要する時間を計測するものである．

プログラム：List02.java

```
import java.util.ArrayList;
import java.util.LinkedList;

class List02 {
    public static void main( String argv[] ) {
        int c, max = 100000;
        long t1;
        double t, v;
        // 種類の異なるリストの作成
        ArrayList<Double>   lst1 = new ArrayList<Double>();
        LinkedList<Double>  lst2 = new LinkedList<Double>();

        // 値の追加（ArrayList）
        t1 = System.nanoTime();
        for ( c = 0; c < max; c++ ) {
            lst1.add( Math.random() );
        }
        t = (double)(System.nanoTime()-t1)/1000000000.0;
        System.out.println(max+"個の要素をArrayListに追加する時間："+t);
        // 値の追加（LinkedList）
        t1 = System.nanoTime();
        for ( c = 0; c < max; c++ ) {
            lst2.add( Math.random() );
        }
        t = (double)(System.nanoTime()-t1)/1000000000.0;
        System.out.println(max+"個の要素をLinkedListに追加する時間："+t);
        System.out.println();

        // 値のセット（ArrayList）
        t1 = System.nanoTime();
        for ( c = 0; c < max; c++ ) {
            lst1.set( c, Math.random() );
        }
        t = (double)(System.nanoTime()-t1)/1000000000.0;
        System.out.println(max+"個のArrayListの要素を書き換える時間："+t);
        // 値のセット（LinkedList）
        t1 = System.nanoTime();
        for ( c = 0; c < max; c++ ) {
            lst2.set( c, Math.random() );
        }
        t = (double)(System.nanoTime()-t1)/1000000000.0;
        System.out.println(max+"個のLinkedListの要素を書き換える時間："+t);
        System.out.println();

        // 値の参照（ArrayList）
        t1 = System.nanoTime();
        for ( c = 0; c < max; c++ ) {
            v = lst1.get( c );
        }
        t = (double)(System.nanoTime()-t1)/1000000000.0;
        System.out.println(max+"個のArrayListの要素を参照する時間："+t);
        // 値の参照（LinkedList）
        t1 = System.nanoTime();
        for ( c = 0; c < max; c++ ) {
            v = lst2.get( c );
        }
        t = (double)(System.nanoTime()-t1)/1000000000.0;
        System.out.println(max+"個のLinkedList要素を参照する時間："+t);
        System.out.println();
```

```
60
61          // 値の挿入（ArrayList）
62          t1 = System.nanoTime();
63          for ( c = 0; c < max; c++ ) {
64              lst1.add( 0, Math.random() );
65          }
66          t = (double)(System.nanoTime()-t1)/1000000000.0;
67          System.out.println(max+"個の要素をArrayListに挿入する時間："+t);
68          // 値の挿入（LinkedList）
69          t1 = System.nanoTime();
70          for ( c = 0; c < max; c++ ) {
71              lst2.add( 0, Math.random() );
72          }
73          t = (double)(System.nanoTime()-t1)/1000000000.0;
74          System.out.println(max+"個の要素をLinkedListに挿入する時間："+t);
75          System.out.println();
76
77          // 要素の削除（ArrayList）
78          int max2 = 2*max;
79          t1 = System.nanoTime();
80          for ( c = 0; c < max2; c++ ) {
81              lst1.remove( 0 );
82          }
83          t = (double)(System.nanoTime()-t1)/1000000000.0;
84          System.out.println(max2+"個のArrayLisの要素を個々に削除する時間："+t);
85          // 要素の削除（LinkedList）
86          t1 = System.nanoTime();
87          for ( c = 0; c < max2; c++ ) {
88              lst2.remove( 0 );
89          }
90          t = (double)(System.nanoTime()-t1)/1000000000.0;
91          System.out.println(max2+"個のLinkedLisの要素を個々に削除する時間："+t);
92      }
93  }
```

このプログラムを実行した例を次に示す．

```
100000 個の要素を ArrayList に追加する時間：0.0086872
100000 個の要素を LinkedList に追加する時間：0.010256

100000 個の ArrayList の要素を書き換える時間：0.0044078
100000 個の LinkedList の要素を書き換える時間：4.594506

100000 個の ArrayList の要素を参照する時間：0.0069724
100000 個の LinkedList 要素を参照する時間：4.5320498

100000 個の要素を ArrayList に挿入する時間：1.6733076
100000 個の要素を LinkedList に挿入する時間：0.0105839

200000 個の ArrayLis の要素を個々に削除する時間：2.7644442
200000 個の LinkedLis の要素を個々に削除する時間：0.007409
```

使用した計算機環境）CPU: Intel Corei7-6770HQ 2.6GHz，OS: Windows 10 Pro

この結果から，

- 要素の挿入や削除が多発する処理には LinkedList が適している
- それ以外は ArrayList が適している

と結論できる．

索 引

’, 5
(int), 22
*/, 6
*=, 18
++, 18
+=, 18
–, 18
-=, 18
-cp, 3
. , 28
/*, 6
//, 6
/=, 18
:, 16
;, 6
〈pre〉, 214
=, 4, 7
==, 15
?, 16
”, 5
$, 168
%, 39
&=, 13
&&, 15
|, 12
～, 12
!=, 15
|=, 13
||, 15
!, 15
&, 12
<, 15
<<, 12
<<=, 12
<=, 15
>, 15
>>, 12
>>=, 12
>=, 15
@, 212
¥n, 8
¥¥, 9
¥’, 9
¥”, 9
¥t, 9
¥u, 9
ˆ, 168
ˆ, 12
ˆ=, 13
@FunctionalInterface, 160
@Override, 33, 183
@ author, 214
@ param, 214
@ return, 214
@ version, 214
1 の補数, 12
3 項演算子, 16

abs, 69, 70, 216
AbstractButton, 200
abstract 修飾子, 32
accept, 172
acos, 69, 216
ActionEvent, 94, 201
ActionListener, 94, 201
actionPerformed, 95, 195, 201
add, 53, 59, 70, 75, 93, 196, 199
addActionListener, 95
addAll, 59
addKeyListener, 207
addMouseListener, 207
addMouseMoveListener, 207
addMouseTrackListener, 207
addMouseWheelListener, 207
addSelectionListener, 123, 207
addShellListener, 207
AdjustmentEvent, 201
AdjustmentListener, 201
adjustmentValueChanged, 201
anonymous class, 35
Apfloat, 215
ApfloatMath, 215
append, 66
APPROVE_OPTION, 117
ArrayList, 53
Arrays, 23, 55, 161
asin, 69, 216
asList, 55
atan, 69, 216

atDate, 80
atTime, 80
AudioFormat, 177
AudioInputStream, 177, 178
AudioSystem, 179
available, 178

BasicStroke.CAP_BUTT, 105
BasicStroke.CAP_ROUND, 105
BasicStroke.CAP_SQUARE, 105
BasicStroke.JOIN_BEVEL, 106
BasicStroke.JOIN_MITER, 105
BasicStroke.JOIN_ROUND, 106
BasicStroke クラス, 104
BigDecimal, 69
BigInteger, 69
BLOCKED, 149
Boolean, 37
boolean, 5, 15
BorderLayout, 99, 101
BoxLayout, 101
BoxLayoutGroupLayout, 101
break, 17
BufferedImage, 110
BufferedImage.TYPE_BYTE_GRAY, 110
BufferedImage.TYPE_INT_RGB, 110
BufferedImage.TYPE_USHORT_GRAY, 110
BufferedInputStream, 50
BufferedOutputStream, 50
BufferedReader, 47
Button, 122, 202
ButtonGroup, 196
Byte, 37
byte, 5, 67, 68
byte 配列, 67

Calendar, 73, 74
call, 152, 153
Callable, 152
CANCEL_OPTION, 117
Canvas クラス, 127
capacity, 66
CardLayout, 101
catch, 44
cbrt, 216
ceil, 69
CEILING, 71
CENTER, 99
ChangeEvent, 95, 201
ChangeListener, 95, 201
char, 5
Character, 37
charAt, 7
ChronoField, 81
clar, 62
Class, 37
ClassNotFoundException, 38
CLASSPATH, 3
clear, 53, 59
CLI, 1, 39
close, 44, 179, 188
collect, 162
Collection, 161, 162
Collections, 55
Collector, 162
Collectors, 162
Color クラス, 103
Combo, 204
Comparator, 23, 56, 162
compare, 56
compareTo, 16, 58, 70, 79
Composite, 203
concurrent, 150
connect, 173
Connection, 187
Consumer, 161
Container, 92
contains, 59
containsKey, 62
containsValue, 62
copy, 208
cos, 69, 216
CREATE DATABASE, 189
CREATE TABLE, 189
createDirectories, 208
createFile, 208
createGraphics, 110
createNewFile, 85
createStatement, 188
CSV, 170
CUI, 1
currentThread, 141
currentTimeMillis, 75
currentTimeMills, 141

DashPhase, 106
DataLine, 177
DataLine.Info, 179
DATE, 74
Date, 73
DayOfWeek, 79
DBMS, 186
default, 17
DefaultMutableTreeNode クラス, 199
DefaultTableModel クラス, 198
default 実装, 35
DELETE, 188
delete, 66, 85, 87, 208
destroy, 156
directory, 155
Display クラス, 120
divide, 70
do, 19
Double, 37
double, 5
doubleValue, 70
DOWN, 71
drain, 179
Drawable インターフェイス, 127
drawImage, 111, 128
drawLine, 104, 129
drawOval, 107, 130
drawPolygon, 107, 130
drawPolyline, 104, 129
drawRect, 107
drawRectangle, 130
drawText, 131
DriverManager, 187

E, 69
EAST, 99
else, 14
end, 166
Entry, 64
entrySet, 62, 64
environment, 155
equals, 16, 59, 70
ERA, 81
ERROR_OPTION, 117
execute, 150, 151
executeQuery, 188
executeUpdate, 188
Executor, 150
Executors, 151
ExecutorService, 151
exists, 85, 86, 208
exit, 52
exitValue, 156
exp, 69, 215, 216
extends, 31

false, 15
File, 82, 118, 208
FileAlreadyExistsException, 210
FileDialog, 137
FileInputStream, 49
FileIntputStream, 47
FileOutputStream, 43
FileReader, 47
Files, 208
fillOval, 107, 130
fillPolygon, 107, 130
fillRect, 107
fillRectangle, 130
filter, 161
finally, 44
final 修飾子, 51
find, 166
Float, 37
float, 5
FLOOR, 71
floor, 69
FlowLayout, 101
Font.BOLD, 108
Font.ITALIC, 108
Font.PLAIN, 108
Font クラス, 107, 130
for, 20, 24
forEach, 161
format, 40, 74
forName, 37
from, 81
FSM, 25
fully qualified name, 185
Function, 162
Future, 151, 152

GC, 127
Generics, 36
get, 53, 62, 81, 153, 155

getAbsoluteFile, 85
getAbsolutePath, 85
getAllFonts, 108
getAudioInputStream, 178
getBytes, 68
getClass, 37
getColumnCount, 188, 192
getConnection, 187
getContentPane, 92
getDayOfMonth, 79
getDayOfWeek, 79
getDayOfYear, 79
getDefault, 120
getErrorStream, 156
getFileName, 83
getFormat, 178
getFrameLength, 179
getFrameSize, 179
getHour, 79
getInputStream, 156, 173
getInstance, 74
getItem, 206
getKey, 64
getKeyChar, 97
getLine, 179
getLocalGraphicsEnvironment, 108
getMetaData, 188
getMinute, 79
getMonthValue, 79
getName, 37, 85, 141
getNano, 79
getOutputStream, 156, 173
getParent, 83
getParentFile, 85
getPassword, 195
getRGBs, 133
getRoot, 83
getSecond, 79
getSelectedFile, 117
getSelectedIndex, 196, 198
getSelectedItem, 196
getSelectedValue, 198
getSelection, 204
getSelection(), 204
getState, 149
getString, 192
getText, 195, 197, 202, 204, 206
getTime, 75
getValue, 64, 79, 198, 200
getValueAt, 198
getYear, 79
GradientPaint クラス, 109
Graphic Context, 127
Graphics, 101
Graphics2D, 101
GraphicsEnvironment クラス, 108
GridBagLayout, 101
GridLayout, 101
group, 166, 169
groupCount, 170
GUI, 89

HALF_DOWN, 71
HALF_EVEN, 71
HALF_UP, 71
HashMap, 62
HashSet, 59
hasNext, 61
hasNextInt, 48
HOUR_OF_DAY, 74
HTML, 213

if, 14
IllegalThreadStateException, 156
Image, 133
ImageData クラス, 131
ImageIcon, 195
ImageIO, 111
ImageLoader, 133
Image クラス, 127
implements, 34
in, 40
indexOf, 66
InetSocketAddress, 172
INNER JOIN, 192
InputStream, 40
InputStreamReader, 47
INSERT, 188
insert, 66
INSERT INTO, 190
int, 5
INTEGER, 189
Integer, 37
interface, 33, 34
InterruptedException, 156

intValue, 70
IOException, 44, 156, 158
IP アドレス, 172
IP ソケット, 172
isDirectory, 208
isDisposed, 120, 121
isEmpty, 59, 62
isLeapYear, 79
isSelected, 195, 196
ItemEvent, 201
ItemListener, 201
itemStateChanged, 201
Iterator, 61, 83
iterator, 63
itf-8, 68

JapaneseDate, 80
jar 形式, 3, 185
java, 2
java.awt.Dialog, 200
java.awt.Frame, 200
java.awt.Window, 200
java.nio ライブラリ, 208
javac, 1
javadoc, 213
javax.swing.JComponent, 200
Java 仮想マシン, 1
Java 実行環境, 1
Java 開発キット, 1
JButton, 94, 195, 200
JCheckBox, 196, 200
JComboBox, 196, 200
JDBC, 187
JDialog, 200
JDK, 1
JEditorPane, 200
JEditPane, 197
JFileChooser, 117
JFrame, 91, 200
JFrame.EXIT_ON_CLOSE, 92, 93
JLabel, 93, 195, 200
JList, 198, 200
JMenu, 93
JMenuBar, 93, 200
JMenuItem, 93, 200
join, 8, 143
JPanel, 101, 200
JPasswordField, 195, 200
JProgressBar, 199, 200
JRadioButton, 196, 200
JRE, 1
JScrollPane, 200
JSlider, 200
JSpinner, 197, 200
JTable, 198, 200
JTextArea, 197, 200
JTextComponent, 200
JTextField, 93, 195, 200
JTextPane, 197, 200
JToggleButton, 200
JTree, 199, 200
JVM, 1
JWindow, 200

KeyAdapter, 207
keyCode, 127
KeyEvent, 95, 127, 201
KeyListener, 95, 201
keyPressed, 201, 207
keyReleased, 201, 207
keySet, 62, 63
keyTyped, 201

L, 5
Label, 122, 202
LEFT OUTER JOIN, 193
length, 7, 22, 42, 66
LinkedList, 53
List, 162, 204
list, 208
listFiles, 85, 86
ListModel インターフェース, 198
LocalDate, 76
LocalDateTime, 76
LocalTime, 76
log, 69, 216
Long, 37
long, 5
longValue, 70
lookingAt, 170

main, 4, 42
Map, 61, 155
map, 162
Matcher, 165

matches, 170
Math, 69
MathContext, 70
max, 69, 70
Menu, 121
MenuItem, 122
min, 69, 70
minusDays, 79
minusHours, 79
minusMinutes, 79
minusMonths, 79
minusSeconds, 79
minusYears, 79
MiterLimit, 106
mkdir, 85
MONTH, 74
MouseAdapter, 207
mouseClicked, 201
mouseDoubleClick, 207
mouseDown, 207
mouseDragged, 201
mouseEnter, 207
mouseEntered, 201
MouseEvent, 94, 201
mouseExit, 207
mouseExited, 201
mouseHover, 207
MouseListener, 94, 201
MouseMotionListener, 94, 201
mouseMove, 207
mouseMoved, 201
MouseMoveListener, 207
mousePressed, 201
mouseReleased, 201
mouseScrolled, 207
MouseTrackAdapter, 207
mouseUp, 207
MouseWheelListener, 207
MS932, 2, 3
multiply, 70

nanoTime, 75
negate, 70
NEW, 149
new, 21
newBufferedReader, 208
newBufferedWriter, 208
newCachedThreadPool, 151, 153
newFixedThreadPool, 151
newInputStream, 208
newOutputStream, 208
next, 40, 61, 192
nextBoolean, 49
nextByte, 49
nextDouble, 49
nextInt, 48, 49
nextLine, 49
nextLong, 49
NONE, 189
NORTH, 99
NoSuchFileException, 210, 211
notifyAll, 147
now, 76
null, 47
NUMERIC, 189

ODBC, 187
of, 76, 161
OOP, 25
open, 120, 137, 179
OpenJDK, 1
out, 39
OverlayLayout, 101

pack, 205
paintComponent, 101
Paint インターフェース, 103, 109
PaletteData, 131
PaletteData クラス, 131
parse, 79, 81
parseDouble, 10
parseInt, 10, 11
Path, 82, 208
Paths, 208
Pattern, 165
PI, 69
pi, 216
plusDays, 79
plusHours, 79
plusMinutes, 79
plusMonths, 79
plusSeconds, 79
plusYears, 79
pow, 69, 70, 216
PowerShell, 1

Predicate, 161
previous, 192
PRIMARY KEY, 189
printf, 39
println, 4, 39, 173
PrintWriter, 45
private, 29
Process, 155
ProcessBuilder, 155
ProgressBar, 206
protected, 29
public, 28, 184
put, 62, 155

random, 22, 32, 69
RDB, 186
read, 50, 113
readAndDispatch, 121
readLine, 47, 173
REAL, 189
remainder, 70
remove, 53, 59, 62
removeAll, 59
renameTo, 85, 86
replaceAll, 167
replaceFirst, 167
ResultSet, 188
ResultSetMetaData, 188, 192
retainAll, 59
reverse, 55, 66
RGB クラス, 132
rint, 69
round, 69
RoundingMode, 71
run, 183
RUNNABLE, 149
Runnable, 150
Runnable インターフェース, 142
run メソッド, 140

Scale, 207
Scanner, 40, 48
SELECT, 188
select, 204
SelectionAdapter, 123, 207
SelectionEvent, 123
ServerSocket, 172
Set, 59
set, 53, 74
setBackground, 195, 202
setBounds, 92, 122, 195, 202
setDefaultCloseOperation, 92, 93
setEchoChar, 195
setFont, 130, 195, 202
setForeground, 195, 202
setIncrement(int inc), 204
setItems, 204
setLayout, 93
setMaximum, 200
setMaximum(int max), 204
setMenu, 122
setMinimum, 200
setMinimum(int min), 204
setModel, 198
setPaint, 103, 104, 109
setPixel, 133
setRGB, 115
setSelected, 196
setSelection(int ini), 204
setSoTimeout, 172, 173
setStroke, 104
setText, 122, 195, 197, 202, 206
setTime, 75
setValue, 200
setValueAt, 199
setViewportView, 197
setVisible, 94
shellActivated, 207
ShellAdapter, 207
shellClosed, 207
shellDeactivated, 207
shellDeiconified, 207
shellIconified, 207
Shell クラス, 120
Shift_JIS, 2
Short, 37
short, 5
showOpenDialog, 117
showSaveDialog, 117
shutdown, 152, 153
SimpleDateFormat, 73
sin, 69, 216
size, 53, 59, 62, 208
SJIS, 2
sjis, 68

sleep, 121, 141
Slider, 207
Socket, 172
SocketTimeoutException, 172
sort, 23, 55
sorted, 162
SourceDataLine, 177
SOUTH, 99
Spinner, 204
SpinnerListModel クラス, 197
SpinnerModel インターフェース, 197
SpinnerNumberModel クラス, 197
split, 8, 170, 197
SpringLayout, 101
SQL, 188
SQLite, 187
sqrt, 7, 69, 215, 216
start, 140, 141, 155, 166, 179, 183
stateChanged, 201
Statement, 187, 188
static, 29
static イニシャライザ, 30
Stream, 160
stream, 161
Stream のクローズ（終了）, 162
String, 5
StringBuffer, 66
StringBuilder, 66
Stroke クラス, 104
submit, 151, 153
substring, 8, 66
subtract, 70
Swing, 91
switch, 17
SWT, 120
SWT.ARROW, 203
SWT.BAR, 121
SWT.BORDER, 122, 202, 204–206
SWT.CASCADE, 122
SWT.CHECK, 203
SWT.DOWN, 203
SWT.DROP, 204
SWT.FLAT, 203
SWT.FULL_SELECTION, 205
SWT.H_SCROLL, 202, 205, 206
SWT.HORIZONTAL, 207
SWT.LEFT, 203
SWT.MULTI, 202, 204
SWT.OPEN, 137
SWT.PASSWORD, 202
SWT.RADIO, 203
SWT.RIGHT, 203
SWT.SAVE, 137
SWT.SIMPLE, 204
SWT.SINGLE, 202
SWT.TOGGLE, 203
SWT.UP, 203
SWT.V_SCROLL, 202, 204–206
SWT.VERTICAL, 207
SWT.WRAP, 202
synchronized, 144, 146
System, 39, 75

Table, 205
table, 186
TableColumn, 205
TableItem, 205
TableModel インターフェース, 198
tan, 69, 216
TCP/IP 通信, 171
TCP ポート, 172
TERMINATED, 149
TEXT, 189
Text, 122
Thread, 181
Thread.State, 149
Thread クラス, 140
throws, 38, 45
TIMED_WAITING, 149
toArray, 65
toDegrees, 69, 216
toFile, 83
toList, 162
toLocalDate, 79
toLocalTime, 79
toMap, 162
toPath, 83
toRadians, 69, 216
toSet, 162
toString, 10, 66, 70, 83, 84, 208
toURI, 83, 85
toUri, 83, 84
Tree, 206
TreeItem, 206

TreeMap, 62
TreeNode インターフェース, 199
TreeSet, 59
true, 15
try, 44

UNICODE, 9
UnsupportedEncodingException, 68
UP, 71
UPDATE, 188
URI, 83
utf-16, 68
UTF-8, 68
utf-8, 2, 49

valueOf, 10, 70
VALUES, 190
values, 62, 63
void, 29

wait, 147
waitFor, 156
WAITING, 149
WAV 形式, 177
WEST, 99
WHERE, 190
while, 18
widgetSelected, 123, 207
windowActivated, 201
windowClosed, 201
windowClosing, 201
windowDeactivated, 201
windowDeiconified, 201
WindowEvent, 94, 201
windowIconified, 201
WindowListener, 94, 201
windowOpened, 201
write, 44, 51, 111

YEAR, 74, 81
YEAR_OF_ERA, 81

ZoneID, 76

値を返す, 7
アノテーション, 33, 36
イコール, 7
イベント, 89
イベントアダプタ, 123
イベント駆動型プログラミング, 89
イベントクラス, 94
イベント処理, 94, 123
イベントソース, 94
イベントリスナ, 94
イベントループ, 120
インクリメント, 18
インスタンス, 26, 28
インターフェース, 33
インデックス型, 131
インデックス, 22
インデックス型, 131
インヘリタンス, 28
ウィジェット, 90
ウィンドウの可視化, 94
ウィンドウの生成, 91
エスケープシーケンス, 8
オブジェクト指向, 4, 25
オーディオ入力ストリーム, 177
オーバーライド, 183
階調色, 109
拡張 for 文, 24, 56, 61, 63
拡張クラス, 28
カスケードメニュー, 122
型, 4
型の変換, 9
空のファイルを作成, 208
カラム, 192
仮引数, 26
環境変数, 155
関係データベース, 186
関係モデル, 186
緩衝記憶領域, 50
関数型インターフェース, 159
完全修飾名, 185
カーソル, 192
外部キー, 186
外部プロセス, 155
画素, 115, 131, 133
画像ファイルとして出力, 111
画像ファイルの読み込み, 113, 133
画像ファイルへの出力, 133
基数, 10
キャスト, 9
クエリ, 186
クライアント, 171
クライアント-サーバモデル, 171

クラス, 4, 26
クラスの拡張, 31
繰り返しの表記, 168
クリティカルセクション, 147
クリティカルリージョン, 147
継承, 28, 31
桁の大きな整数, 69
結合, 186, 189, 192
検索, 165
高精度の数値演算, 69
高精度の浮動小数点数, 69
コマンド, 1, 39
コマンドサーチパス, 2
コマンド引数, 42
コマンドプロンプトウィンドウ, 1
コメント, 6
コレクタ, 162
コンストラクタ, 27, 28
コンテナ, 89, 91
コントロール, 123
コンポーネント, 89
再帰代入, 12, 13, 18
作業ディレクトリ, 155
サブクラス, 28
算術演算, 5
サーバ, 171
サーバソケット, 171, 172
座標系, 103, 128
指数表現, 5
シフト JIS, 2
シフト演算, 11
シフト演算の再帰代入, 12
射影, 186
集合, 59
修飾子, 29
終了ステータス, 52
主キー, 186
出力時の文字コードの指定, 46
昇順, 55
書式, 39
書式編集, 40
シングルクオーテーション, 5
時間の計測, 75
時刻, 73
辞書, 61
実行の再開, 147
順序の反転, 55
条件式, 15
条件分岐, 14, 17
スキーム, 83
スコープ, 28
スタイル, 137
スレッド, 140, 181
スレッドの終了を待機する方法, 143
スレッドの状態, 149
スレッドの同期, 143
スレッドの名前, 141
スーパークラス, 28
正規化, 186
正規表現, 165
静的な型付け, 4
静的メソッド, 29
静的メンバ, 30
整列, 55, 162
セット, 59
セミコロン, 6
選択, 186
線の先端と曲がり角の形状, 104
線の属性, 104, 129
線の太さ, 104
絶対パス, 83, 85
総称型, 36
相対パス, 83, 85
添字, 22
ソケット, 171, 172
ソースデータライン, 177
ソースプログラム, 1
タイムアウト, 172
タイムゾーン, 76
多重継承, 33
多倍長精度の浮動小数点数, 215
探索, 165
探索キー, 165
ターミナルウィンドウ, 1
ダイアログ, 117, 137
代入の連鎖, 7
ダイレクト型, 131
タブルクオーテーション, 5
置換, 165
置換処理, 167
中間コンテナ, 91
抽象クラス, 32
抽象メソッド, 33
中間言語, 1

テキスト, 165
テキストデータ, 43, 49
テキストファイル, 1
テキストフィールド, 93, 122
ディレクトリ内容の一覧, 208
ディレクトリ内容の取得, 86
ディレクトリの削除, 87
ディレクトリの作成, 85
ディレクトリを作成, 208
デクリメント, 18
データ集合, 59
データベース, 186
データベース管理システム, 186
データベースへの接続, 187
統一資源識別子, 83
匿名クラス, 35
トップレベルコンテナ, 91
トランザクション, 187
同期入力, 41
独自の基準でリストを整列, 56
ドメイン名, 172
入力時の文字コードの指定, 47
排他的制御, 144
排他的論理和, 12
配列, 21, 53
配列宣言時の初期化, 22
配列のサイズの取得, 22
派生クラス, 28
破線の長さと間隔, 104
反転, 55
バイトコード, 1
バイトストリーム, 40, 43, 49
バイナリデータ, 49
バイナリファイル, 49
バッチ処理, 89
バッファ, 50, 177
バッファド I/O, 50
パス, 82
パスの作成, 82, 208
パスの分解, 82
パスを文字列に変換, 208
パターンの記述方法, 167
パターンマッチ, 165
パターンマッチのグループ, 169
パッケージ, 184
パッケージ名, 184
パレット, 131
パレットデータ, 131
引数, 7
左外部結合, 193
日付, 73
非同期処理, 152
表, 186
標準エラー出力, 41
標準出力, 39
標準入力, 40
ビッグエンディアン, 11
ビット演算, 11
ビット毎の論理演算の再帰代入, 12
ビット反転, 12
ビットマスク, 132
ビュー, 62
ファイルのサイズ, 208
ファイルの削除, 87
ファイルの存在検査, 86
ファイル名の変更, 86
ファイルを削除, 208
フォントの指定方法, 107
フォントのスタイル, 131
フォールスルー, 17
フレームサイズ, 179
フレームの個数, 179
部分文字列, 8
プリミティブ型, 37, 53
プログラムの終了, 52
プロセス, 155
プロセス属性, 155
平衡二分探索木, 59
編集可能な文字列, 66
変数, 4
ボタン, 94, 122
ボタングループ, 196
ポート, 172
マッピング, 61
マルチスレッドプログラミング, 140
マルチプロセシング, 140
丸めの方法, 70
ミドルウェア, 187
無名クラス, 35
メインメソッド, 4, 42
メソッド, 4, 7, 26
メソッド参照, 163
メニュー, 93, 121
メニュー項目, 93, 122

メニューバー, 93, 121
メンバ, 26
文字コード, 2, 49
文字コードの変換, 67
文字列生成, 40
文字列探索, 165
文字列の長さ, 7
文字列の比較, 16
文字列の部分を取り出す, 7
文字列の分割, 8
文字列の要素の位置, 7
文字列の連結, 8
文字列編集, 66
有限状態機械, 25
有効桁数, 70
要素の整列, 23
ライブラリ, 40, 184
ラベル, 93, 122
ラムダ式, 159
乱数, 22
リスト, 53
リダイレクト, 41
リトルエンディアン, 11
利用できるフォント, 108
リレーショナル・データベース, 186
レイアウトマネージャ, 99
例外, 44
例外処理, 38, 44
レイヤー構造, 133
レイアウトマネージャ, 93
レコード, 186
連想配列, 61
論理積, 12
論理的結合, 15
論理和, 12
和暦, 80
スレッドプール, 150

「Java 入門」 − 文法，各種ライブラリ，GUI アプリケーション構築

著者：中村勝則

発行：2019 年 12 月 15 日

テキストの最新版と更新情報

本書の最新版と更新情報を，プログラミングに関する情報コミュニティ Qiita で配信しています.

→ https://qiita.com/KatsunoriNakamura/items/5f119bcbf82ed932a5b0

上記 URL の QR コード

本書はフリーソフトウェアです，著作権は保持していますが，印刷と再配布は自由にしていただいて結構です.（内容を改変せずにお願いします）　内容に関して不備な点がありましたら，是非ご連絡ください．ご意見，ご要望も受け付けています.

● 連絡先

nkatsu2012@gmail.com

中村勝則