オブジェクト指向言語 － Java プログラミング（2010 年度）

桑原 寛明
情報理工学部 情報システム学科

1. Java の基礎

Java 言語は，クラスに基づくオブジェクト指向言語(OOPL: Object-Oriented Programming Language)の一つである．基本的な式，制御構造，演算子などの構文は C 言語や C++言語に類似している．

1.1 ソースファイルとクラスファイル

Java プログラムは以下に示す手順で実行する．

(1) ソースファイルをエディタで作成する．
(2) ソースファイルをコンパイルする．
(3) Java 実行環境で実行する．

ソースファイルとは，Java ソースプログラムであり，拡張子".java"を持つテキストファイルである．C 言語のプログラム作成と同様に，Emacs や Windows の「メモ帳」などのテキストエディタを用いて作成することができる．また，Eclipse や JBuilder のような統合開発環境(IDE)を用いてもよい．

ソースファイルを Java コンパイラ(javac)でコンパイルすると，クラスやインタフェースごとにクラスファイルと呼ばれるバイナリファイルが作成される．クラスファイルは，拡張子".class"を持つ．クラスファイルはバイトコード(bytecode)とも呼ばれ，プラットフォームに依存しないという特徴を持つ．ソースファイル `Sample.java` のコンパイルは，以下のコマンドにより行う．

```
% javac Sample.java
```

バイトコードは，Java 仮想機械(JVM: Java Virtual Machine)と呼ばれるインタプリタ上で解釈実行する．インタプリタは，バイトコードを各計算機に向けの機械語に逐次変換して，その処理を実行する．クラスファイル `Sample.class` の実行は，以下のコマンドにより行う．

```
% java Sample
```

Java コンパイラ(javac)，インタプリタ(java)は Java SE(Standard Edition)として，Sun Microsystems Inc.（を買収した Oracle）から提供されている．Java SE は，javac や java 以外にも，文書生成器(javadoc)やデバッガ(jdb)などのコマンド，さらに，クラスライブラリ(フレームワーク)を含む．Java SE のクラスライブラリに含まれる主なパッケージを以下に示す．

`java.awt`: グラフィカルユーザインタフェース(GUI)
`java.io`: 入出力機能
`java.lang`: Java 言語の機能
`java.math`: 算術演算
`java.net`: ネットワーク処理
`java.util`: ユーティリティ機能
`javax.swing`: グラフィカルユーザインタフェース(Swing)

1.2 Java プログラムの例

簡単な Java プログラムを例題 1 に示す.

例題 1: Sample1.java

```
public class Sample1 {  // クラスの宣言
    public static void main(String[] args) {
        // 文字列を画面に出力する
        System.out.println("Enjoy Java programming.");
    }
}
```

予約語 class によりクラス Sample1 を定義する．public は，このクラスを他のパッケージに対して公開することを表す予約語である. main は Java プログラムにおいて特殊なメソッド(手続き)であり，java コマンドの引数として指定したクラスの実行開始点である． java Sample1 と実行した場合は，クラス Sample1 の main から実行が始まる． System.out.println()の System.out は標準出力を意味し, println()は文字列を画面に表示するメソッドである. また, //からその行の終りまでの文字，/*と*/にはさまれる文字はコメントとなる．例題 1 のプログラムをコンパイルし，実行すると，以下のようになる.

```
% javac Sample1.java
% java Sample1
Enjoy Java programming.
```

Java ソースファイルを作成する際の注意として，ファイル内の公開(public)クラスの名前(Sample1)とファイル名(Sample1.java)は一致させる必要がある．1 つのファイルには public なクラスを高々1 つと public でないクラスを複数記述できる.

1.3 識別子と予約語

Java 言語では，クラス，メソッド，変数などの名前を識別子(identifier)と呼ぶ．識別子には，半角英数字，アンダーバー，ドル記号，Unicode 文字を使用できる．ただし，数字を先頭文字にすることはできない．また，ドル記号は，主に処理系が内部で識別子を自動生成する場合に用いられる．大文字と小文字は区別されて扱われる.

さらに，以下に示す予約語は，識別子には使えない.

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

1.4 演算子と文

Java の主な演算子には, 算術演算子(+, -, *, /, %, ++, --, ?:), 代入演算子(=, +=, -=, *=, /=, %=), 比較演算子(<, <=, >, >=, ==, !=)，論理演算子(&&，||，!)がある．また，Java は，条件文(if),

繰り返し文(while，for)，break 文， continue 文，return 文を備える．これらの記法と意味は，C 言語や C++ 言語のものと同様である．

これら以外の重要な演算子として，インスタンスを生成する new，型を変換するキャスト(型を括弧で括る)，インスタンスの型を判定する instanceof，配列の生成やアクセスに用いる[]，インスタンスのメンバにアクセスしたり限定名を形成するドット(.)がある．

1.5 変数と定数

Java プログラムにおいて，データを確保する場合に変数(variable)を用いる．C 言語と同様に，変数を使用する前に変数を宣言しておく必要がある．変数の名前は，識別子で与えられる．

ここで，Java において変数はすべて型を持つ．Java のデータ型には，次に示す基本型(primitive type)と参照型(reference type)がある．基本型には，論理型の boolean (true あるいは false をとる)，整数型の byte (8 ビット符号付)，short (16 ビット符号付)，int (32 ビット符号付)，long (64 ビット符号付)，浮動小数点型の float (32 ビット単精度)，double (64 ビット倍精度)，Unicode 文字型の char (16 ビット)がある．配列，クラス，インタフェースは参照型であり，その変数の値はすべてポインタとなる．

変数の宣言，代入，初期化は，C 言語の場合と同様である．ただし，Java は厳密な静的型言語であるため，あらかじめ規定されている型変換が適用される場合や多態変数に対する代入を除いて，変数の型と異なる型の値を代入することはできない．

さらに，Java の変数はスコープ(scope)を持つ．スコープとは，その変数を単純名(名前のみ)で参照できる範囲である．インスタンス変数のスコープは宣言されたクラスであり，メソッド引数のスコープは宣言されたメソッドである．ローカル変数のスコープは，それが宣言された位置から始まり，それが宣言されたブロックの終りまでとなる．

Java では，定数をリテラル(literal)と呼ぶ．リテラルには以下のものがある．

- 整数リテラル (1, 123)
- 浮動小数点数リテラル (1.0, 3.1415，1.23E5)
- 真偽リテラル (true, false)
- 文字リテラル ('A','#','¥n','¥t')
- 文字列リテラル ("Java")
- Null リテラル (null)

Java では，変数の宣言に final 修飾子をつけることにより，変数を定数のように扱うことができる．たとえば，

```
final int MAX_VALUE = 1024;
```

のように変数 MAX_VALUE を宣言すると，この変数は値を修正できないことになり，定数のように扱うことできる．

整数リテラル，浮動小数点数リテラル，真偽リテラル，文字リテラルに関しては，それぞれ対応する基本型の変数を用いて扱う．文字列リテラルに関しては，それを扱う基本型は用意されていないので，文字列を扱うクラス String あるいは StringBuffer，StringBuilder を用いる．String から生成したインスタンスでは，格納されている文字列を変更することはできない．これに対して，StringBuffer や StringBuilder から生成したインスタンスでは，文字の追加や挿入ができるようになっている．ここで，String のインスタンスは頻繁に利用されるため，new 演算子を使わなくとも，次のようにインスタンスの生成が可能である．

```
String str = "Java";  // String str = new String("Java");と同じ
```

String から生成したインスタンスは参照オブジェクトであるため，文字列の比較には==演算子ではなく，メソッド equals() を用いる．(==演算子はオブジェクト自体を比較する．)
Null リテラルは，null 型に属する唯一の値であり，参照型変数がまだ有効な値(オブジェクト)を指していないことを表す．

2. クラスとインスタンス

クラス(class)とはデータ(属性)とそれに対する処理(操作)をひとまとめにして定義した抽象データ型である．Java プログラムは 1 つ以上のクラスで構成され，属性をフィールド(field)，操作をメソッド(method)と呼ぶ．実行時には，クラスからオブジェクトを生成し，生成したオブジェクトが動作することで処理を行う．生成したオブジェクトのことを，クラスに対してインスタンス(instance)と呼ぶ．つまり，クラスとはインスタンスを生成する雛形のようなものである．インスタンスを生成し，処理を実行する Java プログラムを例題 2 に示す．

例題 2: Sample2.java

```
public class Sample2 {
    public static void main(String[] args) {

        /* Book クラスからインスタンスを生成 */
        Book myBook = new Book("The Java Tutorial", 55);
        System.out.println("My book title = " + myBook.getTitle());
        System.out.println("My book title = " + myBook.title);
        System.out.println("My book price = " + myBook.getPrice());

        /* Book クラスからインスタンスを生成 */
        Book yourBook = new Book("The Java Programming Language");
        yourBook.setPrice(40);
        System.out.println("Your book title = " + yourBook.getTitle());
        System.out.println("Your book price = " + yourBook.getPrice());
    }
}

class Book {                  // クラス Book の宣言
    String title;             // 題名
    private int price;        // 価格(ドル)

    Book(String t, int p) {   // コンストラクタ
        title = t;            // 題名の設定
        price = p;            // 価格の設定
    }
```

```
    Book(String t) {          // コンストラクタ
      this(t, 0);
    }

    public String getTitle() {      // 題名の取得
        return title;
    }

    public int getPrice() {         // 価格の取得
        return price;
    }

    public void setPrice(int price) {     // 価格の設定
        this.price = price;
    }
}
```

Book は本を表すクラスである．属性として本の題名と価格を持ち，それぞれ title という文字列型(String)および price という整数型(int)のインスタンス変数で参照する．さらに，Book は本の題名を取得するメソッド getTitle() と本の価格を取得するメソッド getPrice() を持つ．クラスと同じ名前を持つメソッドは，コンストラクタ(constructor)である．コンストラクタはインスタンス生成時に自動的に実行されるため，インスタンスの初期化に用いる．

クラス Sample2 では，myBook と yourBook という Book 型のインスタンス変数をそれぞれ宣言し，new 演算子によりクラス Book からインスタンスを 2 つ生成する．生成したインスタンスは，それぞれ変数 myBook と yourBook を用いて扱う．たとえば， ドット演算子を用いて，myBook.getTitle() により myBook のメソッド getTitle() を呼び出したり，myBook.title により myBook のインスタンス変数 title の値を参照したりすることができる．

ここで，メソッド setPrice() 内に存在する予約語 this は，自分自身のオブジェクトへの参照を表す．たとえば，クラス Book のインスタンス myBook の setPrice() が呼び出された場合，this は myBook を指し，this.price は myBook のインスタンス変数 price を指す．同様に，クラス Book のインスタンス yourBook のメソッド setPrice() が呼び出された場合，this は yourBook を指し，this.price は yourBook のインスタンス変数 price を指す．メソッド setPrice() 内の代入文の右辺の price はローカル変数（メソッドの引数で宣言）である．よって，この代入文で，引数の値がインスタンス変数に代入される．また，this() はコンストラクタ内のみで使用可能で，自分自身のオブジェクトの別のコンストラクタを呼び出す際に用いる．
例題２のプログラムを実行すると，以下のようになる．

```
% java Sample2
My book title = The Java Tutorial
My book title = The Java Tutorial
My book price = 55
Your book title = The Java Programming Language
Your book price = 40
```

Java では，インスタンスではなくクラスに属するフィールドやメソッドを定義することができる．これらは，宣言や定義において修飾子 `static` を付け，それぞれクラス変数(class variable)およびクラスメソッド(class method)と呼ぶ．たとえば， クラス `System` のフィールド `out` (`System.out`)はクラス変数，クラス `Math` のメソッド `sqrt()` (`Math.sqrt()`)はクラスメソッドである．クラス変数やクラスメソッドは，インスタンスを生成しなくともクラス名を指定するだけで参照や呼び出しが可能である．クラス変数は同一のクラスから生成したインスタンス群で値を共有したい場合や定数を保持するために使う．クラスメソッドは，複数のクラスから利用されるユーティリティ関数の実現に用いる．

3. 合成と継承

オブジェクト指向で構築されたクラスやインスタンスを拡張する方法には，合成(composition)と継承(inheritance)がある．

合成とは，別のオブジェクトを参照あるいは包含することで機能を拡張することである．新規クラスへのメソッド呼び出しを既存クラスのメソッドに委譲(delegation)あるいは転送(forwarding)することで，既存クラスの機能の一部を再利用し，さらに独自の機能を追加することができる．合成により機能を拡張するプログラムを例題３に示す．

例題 3: `Sample3.java`

```
public class Sample3 {
    public static void main(String[] args) {
        BookInLibrary book = new BookInLibrary("The Java Tutorial", 45);
        book.setBorrower("Taro Ritsumei");
        System.out.println("Title = " + book.getTitle());
        System.out.println("Borrower = " + book.getBorrower());
    }
}

class Book {
    ... // 例題２と同じ
}

class BookInLibrary {
    private Book book;              // Book クラスのインスタンスへの参照
    private String borrower;        // 借用者

    BookInLibrary(String t, int p) {
        book = new Book(t, p);      // Book クラスのインスタンスの生成
        borrower = null;
    }

    public void setBorrower(String name) {      // 借用者の設定
```

```
        borrower = name;
    }

    public String getBorrower() {   // 借用者の取得
        return borrower;
    }

    public String getTitle() {      // 題名の取得
        return book.getTitle();     // インスタンス book へ転送
    }
}
```

クラス BookInLibrary のコンストラクタは，クラス Book のインスタンスを生成し，その参照先をインスタンス変数 book に格納する．BookInLibrary は，独自にインスタンス変数 borrower およびそれにアクセスするメソッド setBorrower() と getBorrower() を持つ．また, getTitle() への呼び出しをインスタンス book のメソッド getTitle() に転送することで，その機能を再利用している．例題 3 を見てわかるように，Java では，合成における参照や包含(静的な関連)をフィールドにより実現している．

例題 3 のプログラムを実行すると，次のようになる．

```
% java Sample3
Title = The Java Tutorial
Borrower = Taro Ritsumei
```

継承とは, 既存クラスのインスタンス変数やメソッドを引き継いで新しいクラスを定義することである．継承を用いると，すでに存在するクラスに機能を追加したり，一部の機能を書き換えたりするだけで新しい機能を持つクラスを作成することができる. 継承を用いてクラスを拡張するプログラムを例題 4 に示す．

例題 4: Sample4.java

```
public class Sample4 {
    public static void main(String[] args) {
        Book book = new Book("The Java Tutorial", 55);
        System.out.println("Title = " + book.getTitle());
        System.out.println("Price = " + book.getPrice());

        OnlineBook obook = new OnlineBook(
                "The Java Virtual Machine Specification", 0,
                "http://java.sun.com/docs/books/vmspec/index.html");
        System.out.println("Title = " + obook.getTitle());
        System.out.println("Price = " + obook.getPrice());
        System.out.println("Website = " + obook.getWebsite());
    }
```

```
}

class Book {
    ... // 例題２と同じ
}

class OnlineBook extends Book {    // クラス Book を継承
    public String website;         // URL

    OnlineBook(String t, int p, String website) {
        super(t, p);               // スーパークラスのコンストラクタの呼び出し
        this.website = website;    // URL の設定
    }

    public String getWebsite() {   // URL の取得
        return website;
    }

    public String getTitle() {
        return "Online: " + title;
    }
}
```

予約語 extends は継承の宣言を表す．つまり，クラス OnlineBook はクラス Book を拡張している．Book を親クラスあるいはスーパークラス(super-class)，OnlineBook を子クラスあるいはサブクラス(sub-class)と呼ぶ．Java では，すべてのクラスは Object(java.lang.Object)のサブクラスであり，extends を省略した場合は，Object の直接の子クラスになる．

例題４において，OnlineBook から生成したインスタンスは，OnlineBook のインスタンス変数 website，メソッド getWebsite() に加えて，Book の title，price，getTitle()，getPrice()が利用できる．ただし，コンストラクタは継承されない．継承により引き継がれる機能は，直接の親クラスだけでなく，間接的に継承している親クラスの祖先からも引き継がれる(よって，OnlineBook は Object の機能も引き継ぐ)．さらに，OnlineBook ではメソッド getTitle()を再定義することで，Book の機能の一部を修正している．このように，親クラスに存在するメソッドと同じシグニチャを持つメソッドを子クラスで再定義することをメソッドのオーバーライド(override)という．シグニチャとは，メソッドの名前，引数の型と並びを指す．

ここで， super は親クラスを指す．また，super()はコンストラクタ内部でのみ利用可能で親クラスのコンストラクタを呼び出す際に用いる．

例題４のプログラムを実行すると，次のようになる．

```
% java Sample4
Title = The Java Tutorial
Price = 55
Title = Online: The Java Virtual Machine Specification
Price = 0
```

```
Website = http://java.sun.com/docs/books/vmspec/index.html
```

4. 抽象クラスと多態性

継承を用いると，類似した機能を持つクラスの共通部分をまとめて親クラスで定義しておき，それぞれ異なる機能だけをその子クラスに定義することができる．その際，親クラスではメソッドの実装を定義せずに，そのシグニチャだけを規定しておきたいことがある．このような要求に対して，Java では抽象クラス(abstract class)が用意されている．実装が定義されていないシグニチャだけのメソッドを抽象メソッド(abstract method)という．抽象クラスは実装を持たないメソッドを含むため，直接インスタンスを生成することはできない．抽象クラスの子クラスからインスタンスを生成するためには，その子クラスで抽象メソッドをすべて実装する必要がある．抽象クラスを含むプログラムを例題 5 に示す．

例題 5: `Sample5.java`

```
public class Sample5 {
    public static void main(String[] args) {
        Publication pub;
        pub = new Article("Java Compiler", "IEEE Software");
        pub.showInfo();

        pub = new TechReport("Java Programming Explained",
                             "Ritsumeikan Univ.");
        pub.showInfo();
    }
}

abstract class Publication {       // 抽象クラスの宣言
    protected String title;        // 題名

    Publication(String t) {
        title = t;
    }

    abstract public void showInfo();     // 抽象メソッドの宣言
}

class Article extends Publication {      // クラス Publication を継承
    private String journal;              // 雑誌名

    Article(String t, String j) {
        super(t);
        journal = j;
    }
```

```
    public void showInfo() { // showInfoの()実装
        System.out.println(title + " in " + journal);
    }
}

class TechReport extends Publication {   // クラスPublicationを継承
    private String institution;           // 発行機関

    TechReport(String t, String i) {
        super(t);
        institution = i;
    }

    public void showInfo() {  // showInfoの()実装
        System.out.println(title + " by " + institution);
    }
}
```

修飾子abstractは抽象クラスおよび抽象メソッドに付加する.Publicationは抽象クラス,showInfo()は抽象メソッドである．Publication を継承しているクラス Article および TechReportでは,メソッドshowInfo()を実装しているため,インスタンスが生成可能である．また，クラスSample5では，クラスArticleおよびTechReportから生成したインスタンスをPublication型のインスタンス変数pubで保持している．Javaでは，このように子クラスのインスタンスを親クラスのインスタンスとして保持することが可能である．

例題５のプログラムを実行すると，次のようになる．

```
% java Sample5
Java Compiler in IEEE Software
Java Programming Explained by Ritsumeikan Univ.
```

この実行例をみると，メソッド pub.showInfo()の呼び出しに対して，Article かTechReportのインスタンスのどちらかが選択され，そのインスタンスのメソッドshowInfo()が実行されていることがわかる．このように，インスタンス変数pubに格納されている値の型に応じて実行時に呼び出しメソッドが決定されることを動的束縛(dynamic binding)と呼ぶ．また，showInfo()のように 1 つのメソッド呼び出しで，異なる振る舞いを実現することを多態性(polymorphism，多相性)と呼ぶ．

ここで，メソッドのオーバーロード(overload)も多態性の一種である．オーバーロードとは，シグニチャの異なるメソッドを定義することを指す．Java では，シグニチャが異なれば，同一クラス内でも同じ名前を持つメソッドを多重に定義することができる．たとえば，次の 2 つのメソッドは，オーバーロードの関係にあり，別々に扱われる．

```
public int getPrice() { ... }
public int getPrice(int rate) { ... }
```

Java では，抽象メソッド(と定数)だけで構成されているクラスをインタフェースと呼ぶ．インタフェースの使用例を以下に示す．

```
interface Printable {
    void print();
}

class Article extends Publication implements Printable {
    public void print() { ... }
}
```

インタフェースは抽象メソッドしか持たないため，修飾子 abstract は不要である．インタフェースを実装(継承)する際には，extends ではなく，implements を用いる．インタフェースはメソッドの実装を持たずにそのシグニチャを規定でき，さらにインタフェースを継承するクラスにメソッドの実装を強要するため，フレームワークの構築によく使われる．

5. パッケージとアクセス制御

大規模なプログラムを開発する際には，複数の人間で効率的にクラスを作成したい．また，Java ではクラスごとにクラスファイル(拡張子が".class"のファイル)を作成するため，複数のクラスを同じファイルに記述するよりもクラスごとに Java ソースファイルを作成する方が，不必要なコンパイルを削減できる．たとえば，例題２において，クラス Book をクラス Sample2 以外のクラスから利用する場合は，Book のコードをソースファイル Book.java に分けて記述するとよい．これにより，Sample3.java では Book のコードをあらためて記述する必要はなく，Sample2 と Book，Sample3 と Book は独立に開発することができる．

また，大規模なプログラムを作成する場合は，ほかの人が作成したクラス(あるいはインタフェース)を再利用する場合が多い．さまざまな人が作成したクラスを混在させて利用する場合に名前の衝突を避けるため，Java にはパッケージという仕組みが組み込まれている．ソースファイルの先頭に package 文が存在すると，そのファイルで定義したクラスやインタフェースはそのパッケージ(名前空間，name-space)に属することになる．たとえば，次のように記述すると，

```
package ritsumei;
class File {
    ...
}
```

クラス File はパッケージ ritsumei に属し，完全限定名(fully-qualified name)は，ritsumei.File となる．よって，Java SE のクラスライブラリに含まれる java.io.File (java.io パッケージ)と区別される．

ソースファイルに package 文が存在しない場合は，そのファイルで定義されているクラスは無名パッケージに属することになる．同じパッケージに属するクラスどうしは，特にパッケージ名を指定しなくともお互いに利用可能である．異なるパッケージに属するクラスを利用する場合は，パッケージ名を含む完全限定名で指定する．たとえば，パッケージ ritsumei に含まれるクラス

File を指定する場合は，ritsumei.File とすればよい．また，import 文を用いると，完全限定名を指定しなくともクラス名だけでクラスを指定可能となる．よって，次のコードにおいて，File は ritsumei.File と同じものを指す．

```
import ritsumei.File;

class Directory {
    ritsumei.File file1;      // 完全限定名で指定
    java.io.File file2;       // ritsumei.File とは異なるクラス
    File file3;               // ritsumei.File と同じクラス
}
```

特定のパッケージに属するクラス群をまとめて指定する場合は, 次のように"*"を用いて import 文を記述することができる．

```
import java.io.*;
class Directory { ...
}
```

このように記述することで, パッケージ java.io に属するクラスをクラス名だけで指定できる．ここで，java.lang パッケージについては，特に import 文を記述しなくとも自動的に取り込まれている．

パッケージに関連して，Java では情報隠蔽(information hiding)を実現するために，アクセスを制御する(access control)修飾子がいくつか用意されている．修飾子とアクセス制御の関係を以下にまとめる．

public: 異なるパッケージのクラスから，そのクラス，変数，メソッドにアクセス可能
protected: 同じパッケージとサブクラスから，その変数，メソッドにアクセス可能
なし: 同じパッケージから，そのクラス，変数，メソッドにアクセス可能
private :クラス内からのみ，その変数，メソッドにアクセス可能

6. 例外処理

C 言語によるプログラミングでは，コンパイル時には判定できないエラー(実行時エラー)を起こす可能性がある文の実行に対して，その直後に条件文を用いてエラー検査を行う．この方法では，エラー検査のコードがプログラムに散らばり再利用を妨げる．これに対して，Java は正常な処理とエラー処理を分離する例外(exception)という仕組みを持つ．ここで，例外とは，単純な実行時エラーだけでなく，特別な状況に陥ることも指す．特別な状況とは，ファイルの読み込みが終了したなどである．

Java の例外処理は，例外を送出する部分と送出された例外を捕捉して決められた処理を実行する部分からなる．例外の送出と捕捉を含むプログラムを例題 6 に示す．

例題 6: Sample6.java

```
public class Sample6 {
```

```
    public static void main(String[] args) {
        int[] numbers = new int[10];   // 配列の宣言と確保

        try {                                          // ここから
            numbers[20] = 0;  //  エラー発生
        } catch (ArrayIndexOutOfBoundsException e) {
                                                       // ここまでの例外を捕捉
            System.out.println("Exception occured: " + e);  // 例外処理
        }

        System.out.println("Finish!");  // 終了メッセージの表示
    }
}
```

ArrayIndexOutOfBoundsException は，配列の添字が範囲を越えた際に送出される例外である．Java では，エラーが発生する可能性のあるコードを try と catch にはさむことで例外を捕捉し，catch 文のブロックに例外処理を記述する．捕捉された例外は，catch ブロックの実行後に消滅する．例題 6 のプログラムを実行すると，次のようになる．

```
% java Sample6
Exception occurred: java.lang.ArrayIndexOutOfBoundsException
Finish!
```

エラーが発生しているにもかかわらず，例外処理が実行され，プログラムは正常に終了していることがわかる．また，1 つの try に対して，複数の catch ブロックを書くことも可能である．

ここで，例題 6 の catch ブロックを見てわかるように，Java では例外の種類に応じて用意された例外クラスを用い，そのインスタンスを受け渡すことで例外情報をやり取りする．つまり，例題 6 において，ArrayIndexOutOfBoundsException は例外クラスを表し，そのインスタンスは変数 e で保持されている．

例外クラスは，大きく Error 系の例外と Exception 系の例外に分けることができる．

- Error 系例外: 回復不可能な重大エラーを伝達するために使用する．この種類の例外が発生した場合，通常，最低限の情報を収集・提示し，プログラムを停止させる．クラス Error を祖先に持つクラス群にまとめられており，一般的にはクラス名に”Error”を含む．代表的な例外として，OutOfMemoryError (メモリ不足), StackOverflowError (スタックオーバフロー)などがある．
- Exception 系例外: 回復可能なエラーや特別な状況を伝達するために使用する．この種類の例外が発生した場合，通常，適切な対処をして，プログラムの実行を続行させる．クラス Exception を祖先に持つクラス群にまとめられており，一般的にはクラス名に”Exception”を含む．代表的な例外として，InterruptedException (スレッドの中断), IOException (入出力処理エラーの発生), EOFException (ファイルの終わりに達した), FileNotFoundException (ファイルが見つからない), ArithmeticException (ゼロ除算などの算術エラーの発生), ClassCastException (不正な型への変換), ArrayIndexOutOfBoundsException (配列における範囲外の添字指定),

StringIndexOutOfBoundsException (文字列型 String における範囲外の添字指定), NullPointerException (null インスタンスへのアクセス)などがある.

Error 系例外クラスおよび Exception 系例外クラスは，どちらもクラス Throwable の子孫である．よって，クラス Throwable を使用することで，すべての例外を捕捉することが可能である．また，例外クラスの継承階層により，複数の子クラスの例外をそれらの親クラスでまとめて捕捉することも可能である．

その他の分類として，Java の例外は，チェック例外(checked exception)と非チェック例外(unchecked exception)に分けられる．チェック例外は，コンパイル時に，適切な例外処理が行われているかどうかを検査する．よって，開発者は try-catch を必ず書く必要がある．一方，非チェック例外は，適切な例外処理が行われているかどうかをコンパイラは検査しない．よって，開発者は try-catch を書いても書かなくてもよい．クラス RuntimeException あるいはクラス Error の子孫となる例外クラスは，すべて非チェック例外である．その他の例外クラスはチェック例外である．

例外が発生すると，その時点でプログラム制御が catch ブロックに移り，特定の文が実行されないことが起こる．このことが問題となることがある．たとえば，ファイル読み込みの途中で例外が発生した場合，そのファイルをクローズする処理が実行されないといった状況が起こる．これに対して，Java では，finally 文を用いることで，try ブロック内のすべての文の実行が終了する前に例外が発生しようとしなかろうと，try ブロックあるいは catch ブロックを抜ける際に必ず実行してほしい文を指定することができる．

これまで述べたように, 例外が発生した文を含むメソッド内において捕捉した例外を処理する場合は try-catch を用いる．これに対して，捕捉した例外をそのメソッド内で処理せずに別のメソッド内で処理したい場合，メソッド宣言において予約語 throws を用いる．throws は，それが付加されたメソッド内で例外が発生する可能性があることを指しており, 発生した例外はそのままメソッドの呼び出し側に渡される．このような例外の引き渡しの仕組みを提供することで，メソッドの戻り値を使用しなくとも，メソッド呼び出し側に例外の情報を伝達することができる．また，コンストラクタにおいても, 例外の情報をインスタンス生成側に伝達することが可能となる. さらに, Java では, 独自の例外を作成することや throw 文を用いて明示的に例外を送出することができる.

例題 7 は，例外 NoTitleException を独自に定義するプログラムである(クラス Exception のサブクラスとして NoTitleException を定義している)．クラス Manual のメソッド setTitle()の処理中に例外 NoTitleException が送出される可能性があるため，throws 文を用いてその例外を送出する可能性があることを宣言している．

例題 7: Sample7.java

```
public class Sample7 {
    public static void main(String[] args) {
        try {
            Manual manual = new Manual();
            manual.setTitle("");            // 文字数 0 の題名を設定

        } catch (NoTitleException e) {    // 例外の捕捉
            System.out.println("No title");
        }
```

```
    }
}

class Manual {
    String title;

    public void setTitle(String t) throws NoTitleException {
    // 例外の転送
        if (t.length() == 0) {
            throw new NoTitleException();  // 例外の送出
        }
        title = t;
    }
}

class NoTitleException extends Exception {  //  例外の定義
}
```

7. 配列とコレクションクラス

Java では，複数のオブジェクトをまとめて扱うために配列(array)とコレクションクラス(collection classes)が用意されている．

配列とは，同じ型の複数の変数(要素)を並べたものである．配列は[]を用いて宣言し，i 番目の要素にアクセスする場合は，添字 i を[]に入れて指定する．Java では，配列もクラス(オブジェクト)であるため，new 演算子を用いて，要素数に応じた領域を確保しなければならない．配列の使用例を以下に示す．

```
int[] numbers;                // int numbers[]とも書けるが望ましくない
numbers = new int[10];        // 10 個分の領域を確保(添字 0〜9)
numbers[5] = 1;               // 5 番目の要素に 1 を代入
System.out.println("5th = " + numbers[5]);     // 5 番目の要素を表示
System.out.println("length = " + numbers.length);    // 配列の長さを表示

int max = 10;
String[] strings = new String[max];  // max 個分の領域を確保(添字 0〜max-1)
```

配列の長さを取得したい場合は，配列インスタンスのフィールド length を用いる．また，配列の要素数は生成時に変数により指定可能である．

コレクションクラスとは，任意のオブジェクトをまとめて格納するためのクラスである．Vector, ArrayList, HashSet, TreeSet などのクラスと，List や Set などのインタフェースが，java.util パッケージに含まれている．配列を用いた場合，一度生成された配列の要素数を変えることはできない．これに対して，コレクションクラスを用いた場合は，その大きさを必要に応じて変更することができる．コレクションクラスの利用例(ArrayList)を示す．

```
List list = new ArrayList();       // ArrayList のインスタンスの生成
list.add("Java");                  // 0 番目の要素を追加
list.add("Programming");           // 1 番目の要素を追加
list.add("Language");              // 2 番目の要素を追加
System.out.println(" size = " + list.size()); // 要素数の表示
String first = (String)list.get(1);      // 1 番目の要素を取得(0 から始まる)
list.remove(1);                           // 1 番目の要素を削除
```

ArrayList のメソッド get()の戻り値の型は Object のため，要素の型を元に戻すためには明示的に型変換を行う必要がある．また，Java にはコレクションクラス内部の要素に順番にアクセスするために反復子(インタフェース Iterator)が用意されている．Iterator を用いた各要素への反復アクセスの例を以下に示す．

```
List list = new ArrayList();       // ArrayList のインスタンスの生成
list.add("Java");
list.add("Programming");
list.add("Language");

Iterator it = list.iterator();     // list に対する反復子の取得
while (it.hasNext()) {             // 要素が存在する場合はループ内部へ
  String str = (String)it.next();  // 要素を取り出す
  System.out.println(str);
}
```

反復子を用いることで，コレクションクラスの型にとらわれずに，各要素にアクセス可能となる．たとえば，クラス ArrayList をクラス TreeSet(要素の格納に木構造を用いる)に変更した場合でも，インスタンスの生成以外はコードを変更する必要はない．

ここで，J2SE5.0(Tiger)で導入された総称(Generics)に関して，コレクションクラスとの関係を少し説明しておく．総称を用いることで，クラス，インタフェース，メソッドなどの型をパラメータとして定義することができる．たとえば，J2SE5.0 では，ArrayList を用いた例を，次のように記述することができる．

```
List<String> list = new ArrayList<String>();
```

ArrayList の後ろに<String>と記述することで，list インスタンスに追加できるインスタンスを String 型に制限している．このように，総称を用いることで，list に追加されるオブジェクトの型を保証することが可能である．この型チェックは，実行時ではなくコンパイル時に行われるため，以下のコードは，実行時エラーでなく，コンパイルエラーとなる．

```
list.add(new Integer(123));
Integer i =  list.get(0);
```

また，総称を用いた宣言により，list インスタンスは String 型のインスタンスを保持していることが保証されるため，要素を取り出す際に従来のようなキャストは不要である．

```
String str = list.get(0);  // (String)list.get(0)と同じ
```

8. 入出力処理

Java ではデバイス（キーボード，ファイル，ディスプレイ）への入出力をストリーム(stream)で扱う．これらは，パッケージ java.io に含まれている．ストリームは 4 つのクラス InputStream, OutputStream, Reader, Writer を基礎とする．InputStream と OutputStream はバイト指向ストリーム(バイトデータの入出力に用いる)であり，Reader と Writer は文字指向ストリーム(テキストデータの入出力に用いる)である．バイト指向とはデータをバイナリ形式でそのまま扱うことを意味し, 文字指向とはデータを文字に変換して扱うことを意味する．

通常，ファイルから文字列を入力する場合は，ファイル名を指定してクラス FileReader のインスタンスを生成する．また，ファイルに文字列を出力する場合は，クラス FileWriter のインスタンスを生成する．ファイル入出力のプログラムを例題８に示す．

例題 8: Sample8.java

```
import java.io.*;

public class Sample8 {
    public static void main(String[] args) {
        String filename = "test8.txt";

        try {
            PrintWriter pw = new PrintWriter(
              new BufferedWriter(          // バッファを挿入
              new FileWriter(filename)));  // 出力ストリームの作成
            pw.println("Java");
            pw.println("Programming");     // 文字列を書き出す
            pw.println("Language");
            pw.close();                    // 出力ストリームのクローズ

        } catch (IOException e) {
            System.out.println("Cannot write: " + filename);
            System.exit(1);                // プログラムの終了
        }

        try {
            BufferedReader br = new BufferedReader(    // バッファを挿入
              new FileReader(filename));   // 入力ストリームの作成

            String line;
```

```
            while ((line = br.readLine()) != null) {   // 文字列を読み込む
               System.out.println(line);
            }
            br.close();        // 入力ストリームのクローズ

        } catch (FileNotFoundException e) {
            System.out.println("File Not Found: " + filename);
        } catch (IOException e) {
            System.out.println("Cannot read: " + filename);
            System.exit(1);    // プログラムの終了
        }
    }
}
```

例題 8 では，ファイルの入出力を効率的に行うためにクラス BufferedReader によりバッファリングを行っている．また，ストリームの読み出し，あるいは，書き出しが終了した場合は，メソッド close() をよびだすことで，ストリームをクローズする．例題 8 のプログラムを実行すると，次のようになる．

```
% java Sample8
Java
Programming
Language
% cat test8.txt
Java
Programming
Language
```

Java では，標準入出力処理のために次に示す 3 つのストリームが用意されており，プログラム起動時にオープンされている．

System.in: 標準入力ストリーム（通常はキーボード）
System.out: 標準出力ストリーム（通常はディスプレイ）
System.err: 標準エラー出力ストリーム（通常はディスプレイ）

これらは，クラス System のフィールドであり，フィールド in はクラス InputStream のインスタンス，フィールド out と err はクラス PrintStream のインスタンスである．よって，キーボードから 1 文字を読み込見たい場合は，以下のように記述する．

```
int data = System.in.read();
```

また，最大 80 文字を読み込みたい場合は，以下のように記述する．

```
byte[] date = new byte[80];
```

```
int n = System.in.read(data, 0, 80);
```

この例では，改行の入力があるまで文字列を読む込むことになる．変数 n には，実際に読み込んだ文字の数が入る．80 文字以上の文字が入力された場合は，先頭から 80 文字までが変数 data に格納され(添字 0〜79)，残りの文字に関しては次にメソッド read() が呼び出されたときに読み込まれる．入力が 80 文字より少ない場合は，最後に改行文字が挿入される．

出力に関しては，クラス PrintStream のメソッド print() や println() を使用することになる．ここで，通常，Java のコンソール出力は，PrintStream の提供するメソッドで十分である．しかしながら，PrintStream は実行効率があまり良くないため，別のストリームクラスを使用することもある．文字入力に関してバッファリングを行いたい場合は，クラス InputStreamReader と BufferedReader を用いて，以下のように記述する．

```
BufferedReader cin =
  new BufferdReader(new InputStreamReader(System.in))
```

また，クラス PrintStream ではなく，クラス PrintWriter を使用したい場合は，以下のように記述する．

```
PrintWriter cout = new PrintWriter(System.out, true)
```

第 2 引数は行単位での出力フラッシュ(書き出し)を行うかどうかを決めるもので，true を指定した場合は，メソッド println() の呼び出しごとにデータが書き込まれる．false の場合は，行単位でフラッシュされない．

9. スレッド

通常のプログラムにおいては，たとえ制御が分岐や繰り返しを含んだとしても，処理の流れは 1 本の糸のように実行される．このようなプログラムを，シングルスレッド・プログラム(single thread program)という．これに対して，複数のスレッドを含むプログラムのことを，マルチスレッド・プログラム(multi-thread program)という．Java は，マルチスレッドを扱うことができる．たとえば，画面に表示するスレッドと，マウス操作を処理するスレッドを分離することで，表示中でもマウス操作が可能となる．あるいは，サーバにおいて，複数のスレッドが別々にセッションを管理することで，それぞれのセッションが独立して処理を実行することができる．ここで注意しなければならないこととして，マルチスレッドとは，複数の処理の流れを同時に実行するように見せかけるだけである．複数の CPU がない限り，複数のスレッドが同時に実行できるわけではない．

通常，Java では，プログラムを起動した際に，1 つのスレッド(メインスレッド)が生成され，それが実行されている．メインスレッド以外に，新たにスレッドを作成するには，次の 2 つの方法がある．

(1) クラス Thread を拡張する．
(2) インタフェース Runnable を実装する．

Java のクラス Thread は，スレッドとしての基本的な機能を備えている．そこで，クラス Thread を継承して，メソッド run() を再定義することで，独自のスレッドを定義できる．例題 9 に，独

自スレッドを定義し，そのスレッドを起動するプログラムを示す．

例題 9: Sample9.java

```
public class Sample9 {
    public static void main(String[] args) {
        MyThread th = new MyThread();     // スレッドの生成
        th.start();                       // スレッドの開始

        for (int i = 0; i < 10; i++) {
            System.out.println("main: " + i);
        }
    }
}

class MyThread extends Thread {    // 独自スレッドの定義
    public void run() {            // スレッド処理の再定義
        for (int i = 0; i < 10; i++) {
            System.out.println("my thread: "+ i);
        }
    }
}
```

クラス Thread を拡張したクラスのインスタンス th を生成し，クラス Thread から引き継いだメソッド start() を呼び出すことで，新たなスレッドが起動され，その後自動的に，クラス Thread を拡張したクラスのメソッド run() が呼び出される．メソッド run() の実行が終了すると，スレッドも終了する．スレッドの生成と同時に，そのスレッドを起動したければ，クラス Thread を拡張したクラスのコンストラクタ内でメソッド start() を呼び出せばよい．例題 9 のプログラムを実行すると，次のようになる．

```
% java Sample9
main: 0
my thread：0
main: 1
my thread: 1
main: 2
my thread: 2
...
```

クラス Thread はスレッドに必要な基本的な機能を備えているため，それを拡張することで簡単に独自のスレッドを定義することができる反面，Java では 1 つのクラスしか継承できないため，この方法ではスレッドの定義ができないことがある．このような場合には，インタフェース Runnable に対してメソッド run() を実装したクラスを作成し，そのクラスをクラス Thread に関連付けることで，独自のスレッドを起動することができる．インタフェース Runnable を用い

たプログラムを例題 10 に示す.

例題 10: Sample10.java

```
public class Sample10 {
    public static void main(String[] args) {
        MyThread th = new MyThread();  // スレッド処理を行うインスタンスの生成
        Thread th0 = new Thread(th);   // スレッドの生成と割り当て
        th0.start();                   // スレッドの開始

        for (int i = 0; i < 10; i++) {
            System.out.println("main: " + i);
        }
    }
}

class MyThread implements Runnable {    // 独自スレッド処理の定義
    public void run() {                  // スレッド処理の実装
        for (int i = 0; i < 10; i++) {
            System.out.println("my thread: "+ i);
        }
    }
}
```

まず，インタフェース Runnable を実装したクラスのインスタンスを生成し，そのインスタンスを引数として, クラス Thread のインスタンスを生成する. クラス Thread のメソッド start() を呼び出すことで，新たにスレッドが起動され，その後自動的に，インタフェース Runnable を実装したクラスのメソッド run()が呼び出される．スレッドの生成と同時に，そのスレッドを起動したければ, インタフェース Runnable を実装したクラスのコンストラクタ内でクラス Thread の生成とメソッド start()の呼び出しを行えばよい．例題 10 のプログラムを実行すると，次のようになる.

```
% java Sample10
main: 0
my thread：0
main: 1
my thread: 1
main: 2
my thread: 2
...
```

ここで，メインスレッドは暗黙の内に実行されるため，スレッドを保持する変数を持たない．また，現在実行中のスレッドに関する情報を知りたいことがある．このために，クラス Thread は，以下のメソッドを用意している.

```
Thread th = Thread.currentThread();
```

クラス Thread が提供するメソッド sleep() を使用することで，スレッドを一時停止させることができる．メソッド sleep() の引数には，ミリ秒単位で数値を指定する．

```
Thread.sleep(1000);  // 現在実行中のスレッドを 1000 ミリ秒停止する．
```

一時停止しているスレッドに割り込みたい場合は，メソッド interrupt() を使用する．

```
MyThread th = new MyThread();
th.start();
...
if (th.isInterrupted() == false) {  // 割り込み処理中かどうかを検査する
    aa.interrupt();
}
```

あるメソッドがメソッド sleep() を呼び出すことで一時停止中に，別のスレッドがメソッド interrupt() を呼び出すことで割り込んだ場合，メソッド sleep() は InterruptedException 例外を発生させる．つまり，一時停止中のスレッドは，メソッド sleep() の呼び出しに対して，InterruptedException 例外を捕捉することで，別のスレッドからの割り込みを検出することができる．

プログラムによっては，複数のスレッドが協調して動作することがある．このような場合，あるスレッドが別のスレッドの実行終了を待つ必要がある．このような要求に対して，クラス Thread はメソッド join() を提供する．メソッド join() は次のように使用する．

```
MyThread th = new MyThread();
th.start();
...
try {
  th.join();
} catch (InterruptedException e) {
}
```

スレッドが生存しているかどうかを確認するためにはメソッド isAlive()，スレッドの優先順位を設定するにはメソッド setPriority(),優先順位を取得するためには getPriproty() を使用する．ただし，優先順位はあくまでも Java での処理において，優先順位の高いスレッドの実行を優先させるように努力することを示しているだけで, 実際の実行順序は OS などに依存するため, 優先順位は保証されない．

複数のスレッドが独立して動作している場合は, 特にスレッド間の関係を考える必要はない．しかし，複数のスレッドが同じデータを読み書きしたり，同じリソースを使用したりする場合，問題が発生する．たとえば，預金処理を考えてみる．この処理を行うメソッド deposit() の実装は以下のようになる．

```
class Account {
  private int balance;

  public void deposit(int money) {
      int currentBalance = balance;
      System.out.println("Previous balance = " + currentBalance);
      currentBalance = currentBalance + money;
      System.out.println("Current balance = " + currentBalance);
      balance = currentBalance;
  }
}
```

まず，預金残高を格納する変数 balance に対してその値を読み込んで変数 currentBalance に代入する．次に，currentBalance の値に入金額 money を加算し，最後に変数 balance に値を書き戻す．

いま，スレッド A がこのメソッドを呼び出し，現在の預金残高を読み込んだ瞬間を考える．スレッド A が預金額を加算し，その値を書き戻すまで，別のスレッド B がこの処理を実行しなければ問題は発生しない．しかし，スレッド A が預金額を加算中に，スレッド B が預金残高を読み込むと，その値はスレッド A により預金額が加算される前の預金残高を指している．たとえば，預金残高 1000 円に対してこのような状況が発生すると，スレッド A とスレッド B が 100 円ずつ預金残高を加算しても，最終的な預金残高は 1100 円(1000 円+100 円)にしかならない．このような問題を回避するためには，スレッド A の実行が終了してからスレッド B の実行が行われる，つまり，スレッド A にスレッド B が割り込めないようにすればよい．Java では，これを同期(synchronization)で実現している．以下のようにメソッドの宣言部を記述することで，このメソッドの処理は同時に 1 つのスレッドしか行えないことになる．

```
synchronized void deposit(int money) { ... }
```

このような実行を排他制御(mutual exclusion control)と呼ぶ．Java では，この制御をロック(lock)とアンロック(unlock)という仕組みで実現している．同期メソッド(synchronized 宣言されたメソッド)が呼び出された場合, そのメソッドを含むインスタンスに自動的にロック(鍵)がかけられる．ロックのかかっているインスタンスに対しては，別のスレッドが同期メソッドを実行できない．つまり，ロックが解除(アンロック)されるまで，別のスレッドは待たされることになる．ロックをかけたスレッドの処理が終了すると，自動的にロックは解除される．ここで注意しなければならないことは，ロックの対象はメソッドではなく，インスタンスであることである．あるスレッドが同期メソッドを実行しているときは, そのインスタンスのすべての同期メソッドを実行できない．逆に，同期メソッド以外は実行可能である．また，あるインスタンスにロックがかかっている状態でも，別のインスタンスの同期メソッドは実行可能である．もし，同期メソッドがインスタンスメソッドでなくクラスメソッドであった場合，ロックはインスタンスではなくクラスにかけられる．

同期対象をメソッド全体でなく，メソッド内の一部分としたい場合は，次のように synchronized 文を使用する．

```
synchronized (式) {
  ...
```

```
}
```

式には，参照型のインスタンスを指定する．また，予約語 this を指定することで，実行中のインスタンス自身にロックをかけることもできる．ロック・アンロックによる排他制御は，デッドロックを引き起こす危険性があるため，注意してプログラムを作成する必要がある．

複数のスレッドが処理を実行している場合, それらのスレッドを任意の時点で協調させたいことがある．Java では，これをスレッド間通信で実現している．スレッド間通信に必要な処理は，クラス Object のメソッド wait()，notify()，notifyAll()で提供されている．あるスレッドがメソッド wait()を呼び出すと，そのスレッドは待機状態になる．この状態で，別のスレッドがメソッド notify()を呼び出すと，待機状態にある先頭の(最初に待機状態になった)スレッドの実行が再開される．メソッド notify()の代わりに，メソッド notifyAll()が呼び出されると，待機状態にあるすべてのスレッドの実行が再開される．

10. GUI プログラミング

グラフィカルユーザインタフェース(GUI: Graphical User Interface)とは，文字ベースのコマンドではなく，グラフィクスを仲介してユーザとプログラムが対話する仕組みである．GUI を活用することで，プログラムに直感的で親しみやすい外観と感触(Look & Feel)を与えることができる．グラフィカルユーザインタフェースは GUI コンポーネントを使って構築する．GUI コンポーネントはウィジェット(widget = window gadget の略)と呼ばれることもある．

GUI プログラミングとは，ウインドウ上に GUI コンポーネントを配置し，それらのコンポーネントで発生するイベントやイベントに対応する動作を記述することで, ユーザとの対話を実現するプログラムを作成することである．

10.1 Swing フレームワーク

フレームワーク(framework)とは，特定の領域(ドメイン)や目的のために設計された再利用可能なモジュール群のことである．クラスとそれらの協調関係を内包しており，半完成のアプリケーションとなっている．Java では，GUI プログラミングのために，AWT(Abstract Window Toolkit)パッケージと Swingパッケージが提供されている. Swing は, AWT の機能を強化したものであり，高度な GUI を提供するフレームワークである．GUI に関係する主なパッケージを以下に示す．

java.awt: AWT の基本パッケージ
java.awt.color: 色空間を扱うパッケージ
java.awt.event: イベントを扱うパッケージ
java.awt.font: フォントを扱うパッケージ
javax.swing: Swing の基本パッケージ
javax.swing.event: Swing で追加されたイベントを扱うパッケージ
javax.swing.filechooser: ファイル選択ダイアログを扱うパッケージ
javax.swing.colorchooser: 色選択ダイアログを扱うパッケージ

10.2 コンテナとコンポーネント

Swing の GUI コンポーネントは，機能により次の 3 つに分類できる．

(1) トップレベルコンテナ

トップレベルコンテナとは，画面上に GUI コンポーネントを表示するための土台となるコンテナで，表示するウインドウに対して必ず 1 つ必要となる．コンテナとは，GUI コンポーネントを格納する入れ物である．Swing には，トップレベルコンテナとして，クラス JFrame (タイトルバーや境界枠を持つ通常のウインドウ)，クラス JDialog (ダイアログ用)，クラス JApplet (アプレット用)，クラス JWindow (タイトルバーや境界枠を持たない空のウインドウ)というクラスが用意されている．

(2) 中間コンテナ

中間コンテナとは，GUI コンポーネントをレイアウトしたり，装飾したりするためのコンテナである．Swing の中間コンテナには，クラス JPanel，JScrollPane，JSplitPane，JTabbedPane がある．クラス JPanel は，レイアウトマネージャを備えた，透明で伸縮自在なシートのようなものである．クラス JScrollPane はスクロールバーを内部コンポーネントに付加するコンテナ，クラス JSplitPane は内部コンポーネントを格納するための分割区画を提供するコンテナ，クラス JTabbedPane は内部コンポーネントを格納するための階層区画を提供するコンテナである．コンテナ内部に別のコンテナに格納することも可能である．

(3) アトミックコンポーネント

アトミックコンポーネントは，ラベルやボタンのような実際に表示される個々の GUI コンポーネントを指す．Swing には，クラス JLabel，JButton をはじめとする多くのコンポーネントが用意されている．アトミックコンポーネントは，コンテナに格納されて使用される．

Swing を用いた簡単なプログラムの例を以下に示す．

例題 11: Sample11.java

```
import javax.swing.*;  // Swing パッケージの指定
import java.awt.*;     // AWT パッケージの指定

public class Sample11 {
    public static void main(String[] args) {
        /* トップレベルコンテナの作成 */
        JFrame frame = new JFrame("Swing Sample11");
        Container contentPane = frame.getContentPane();

        /* ラベルコンポーネントの生成と frame への追加 */
        JLabel label = new JLabel("Enjoy Swing.", JLabel.CENTER);
        contentPane.add(label);

        /* frame の表示 */
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(200, 100);
        frame.setVisible(true);
    }
}
```

例題11ではトップレベルコンテナとしてクラスJFrameを用いて,コンテナframe(正確には,frameはクラスJFrameのインスタンスを指す変数)を生成する．このウインドウは，ボーダ，タイトル,「アイコン化」ボタン,「閉じる」ボタンなどの装飾を備える．さらに，このコンテナにコンポーネントを格納するために，コンテントペイン(ContentPane型の変数contentPane)を取得する．ペインとはトップレベルコンテナの表面のガラスのようなものと考えればよい．

クラスJLabelはラベルコンポーネントである．ここでは，引数に表示する文字列と位置を設定してインスタンスlabelを生成し，クラスContentPaneのメソッドadd()を用いてlabelをframeに格納(配置)する．

クラスJFrameのメソッドsetDefaultCloseOperationはユーザがウインドウの「閉じる」ボタンをクリックした際のデフォルトのアクションを設定する．ここでは，プログラムの実行を終了するように指定している．次に，クラスJFrameのメソッドsetSize()によりウインドウの大きさを設定し，メソッドsetVisible()を引数trueで呼び出すことで，実際にウインドウを画面に表示する．

メソッドmain()をみると，クラスJFrameのメソッドsetVisible()の呼び出しでプログラムの実行が終了しているように見える．しかしながら，メソッドsetVisible()を呼び出すことによりSwingのコンポーネントが実現化されると，イベントディスパッチスレッド(event dispatch thread)の実行が開始される．このスレッドがユーザの実行したイベントを受け付けるようになり，実際にはプログラムの実行は終了しない．例題11では,「閉じる」ボタンを押すことによりイベントが発生し，プログラムの実行が終了する．

10.3 GUIコンポーネントの配置

Javaでは，GUIコンポーネントを配置するために具体的な座標を指定するのではなく，レイアウトマネージャを用いる. AWTとSwingで用意されている主なレイアウトマネージャを次に示す.

FlowLayout: 左から右の横一列，あるいは，上から下の縦一列にコンポーネントを配置する.
BorderLayout: 上，下，右，左，中央の５つの領域の相対位置でコンポーネントを配置する.
CardLayout: 複数のコンポーネントをグループ化し，積み重ねたように配置する.
GridLayout: コンテナを格子に分割して，各格子にコンポーネントを配置する.
GridBagLayout: コンテナをきめの細かい格子に分割して，各格子にコンポーネントを配置する.

クラスBorderLayoutを用いたプログラムの例を示す.

例題12: Sample12.java

```
import javax.swing.*;
import java.awt.*;

public class Sample12 {
   public static void main(String[] args) {

      /* トップレベルコンテナの作成 */
      JFrame frame = new JFrame("Swing Sample12");
```

```
        Container contentPane = frame.getContentPane();

        /* 中間レベルコンテナの作成 */
        JPanel panel = new JPanel();
        panel.setLayout(new BorderLayout());
        contentPane.add(panel);

        /* ラベルコンポーネントの生成と panel への追加 */
        JLabel label = new JLabel("Enjoy Java programming.");
        panel.add(label, BorderLayout.NORTH);

        /* ボタンコンポーネントの生成と panel への追加 */
        JButton button = new JButton("Push");
        panel.add(button, BorderLayout.CENTER);

        /* frame の表示 */
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(200, 80);
        frame.setVisible(true);
    }
}
```

例題 12 では，レイアウトマネージャにクラス BorderLayout を用いるため，クラス JPanel のメソッド setLayout()の引数に BorderLayout のインスタンスを指定する．次に，クラス JLabel のインスタンス label を panel の上部(NORTH)に，クラス JButton のインスタンス button を panel の中央(CENTER)に配置し，その後 frame を表示する．

10.4 イベント

GUI プログラムでは，ユーザがボタンを押したり，マウスを移動させたりすることで，処理を実行する．Java では，ユーザが GUI コンポーネントを操作するとイベント(event)と呼ばれる通知メッセージ(イベントオブジェクト)が発行される．このメッセージを捕捉する(イベントを取得する)ことで，ユーザが行った操作を GUI プログラムは知ることができ，イベントに対応した処理を行う．つまり，GUI プログラムでは，ユーザが何か操作を行うとイベントが発生し，イベントに反応して処理が実行されるという形態になる．これをイベント駆動(event driven)と呼ぶ．

Java では，イベントを代理人モデル(delegation model)で扱う．代理人モデルでは，イベント処理に次の 3 つのオブジェクトが関与する．

イベント: 発生したイベントを表すオブジェクト
イベントソース(event source): イベントの発生源オブジェクト
イベントリスナ(event listener): イベントを処理するオブジェクト

さまざまな GUI コンポーネントがイベントソースになることができる．代理人モデルでは，イベントソースとは別にイベントリスナを実装し，実装したリスナをイベントソースに登録することでイベントの伝搬を実現する．Java プログラムでイベント処理を実装するには，以下のようなプログラミングを行う．

1. イベントソースで発生したイベントを処理するイベントリスナを定義する.
2. イベントリスナにおいて処理を記述する.
3. イベントリスナをイベントソースに登録する.

イベントリスナの登録は，メソッド add<Event>Listener() を用いて行う<Event>はイベントクラスを指す．GUI コンポーネントであるイベントソースをユーザが操作する(たとえば，クラス JButton のインスタンスであるボタンを押す) と，その操作に合わせてイベントオブジェクト(クラス ActionEvent のインスタンス)が生成され，あらかじめ登録されたイベントリスナで宣言されているメソッド(インタフェース ActionListener のメソッド actionPerformed())が呼び出される．このメソッドをイベントハンドラ(event handler)と呼ぶ.

パッケージ java.awt.event とパッケージ javax.swing.event に含まれる主なイベントソース，イベントクラス，イベントリスナ，イベントハンドラを以下に示す.

イベントソース	イベントクラス	イベントリスナ	イベントハンドラ
JButton JCheckBox JComboBox JMenu JTextField	ActionEvent	ActionListener	actionPerformed()
JScrollBar	AdjustmentEvent	AdjustmentListener	adjustmentValueChanged()
Component のサブクラス	FocusEvent	FocusListener	focusGained() focusLost()
	KeyEvent	KeyListener	keyPressed() keyReleased()
	MouseEvent	MouseListener	mouseClicked() mouseEntered() mouseExited() mousePressed() mouseReleased()
		MouseMotionListener	mouseDragged() mouseMoved()

ボタンを押すとラベルの文字列を切り替えるプログラムを例題 13 に示す．クラス JButton のインスタンス button がイベントソース，クラス ButtonListener のインタンス bl がイベントリスナである．クラス ButtonListener のメソッド actionPerformed() は，イベントを捕捉した際の処理が記述される．ここでは，変数 flag を用いてボタンが押されるたびにクラス JLabel のインスタンス label の文字列を切り替えている.

例題 13: Sample13.java

```
import javax.swing.*;
import java.awt.*;
```

```
import java.awt.event.*;

public class Sample13 {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Swing Sample13");
        Container contentPane = frame.getContentPane();

        JPanel panel = new JPanel();
        panel.setLayout(new BorderLayout());
        contentPane.add(panel);

        JLabel label = new JLabel("Enjoy Java programming.");
        panel.add(label, BorderLayout.NORTH);

        /* イベントソースの作成 */
        JButton button = new JButton("Push");
        panel.add(button, BorderLayout.CENTER);

        /* イベントリスナの作成と登録 */
        ButtonListener bl = new ButtonListener(label);
        button.addActionListener(bl);

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(200, 80);
        frame.setVisible(true);
    }
}

class ButtonListener implements ActionListener {  // イベントリスナの実装
    private JLabel label;
    private boolean flag = true;  // 表示文字列を切り替えるためのフラグ

    ButtonListener(JLabel l) {
        label = l;  // 文字列を表示するラベルの設定
    }

    /* ボタンが押されたときの処理(イベントハンドラ) */
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button pressed");

        if (flag == true) {
            label.setText("Enjoy Swing.");
            flag = false;
        } else {
```

```
            label.setText("Enjoy Java.");
            flag = true;
        }
    }
}
```

例題 13 では，イベントリスナクラスを専用に作成している．イベントリスナの実装には，例題 13 以外に例題 14 や例題 15 の方法がある．

例題 14: `Sample14.java`

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Sample14 {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Swing Sample14");
        Container contentPane = frame.getContentPane();

        Panel14 panel = new Panel14();
        contentPane.add(panel);

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(200, 80);
        frame.setVisible(true);
    }
}

/* イベントリスナの実装 */
class Panel14 extends JPanel implements ActionListener {
    private JLabel label;
    private boolean flag = true;

    Panel14(String title) {
        setLayout(new BorderLayout());
        label = new JLabel("Enjoy Java programming.");
        panel.add(label, BorderLayout.NORTH);

        /* イベントソースの作成 */
        JButton button = new JButton("Push");
        panel.add(button, BorderLayout.CENTER);

        /* イベントリスナの登録 */
```

```
        button.addActionListener(this);
    }

    /* ボタンが押されたときの処理(イベントハンドラ) */
    public void actionPerformed(ActionEvent e) {
        if (flag == true) {
            label.setText("Enjoy Swing.");
            flag = false;
        } else {
            label.setText("Enjoy Java.");
            flag = true;
        }
    }
}
```

例題 14 のクラス Panel4 は，クラス JPanel を継承するのと同時にインタフェース ActionListener を実装する．つまり，イベントソースを格納するコンテナがイベントリスナとなる．このため，button のリスナにクラス JPanel のインスタンス(自分自身 this)を登録している．

例題 15: Sample15.java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Sample15 {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Swing Sample15");
        Container contentPane = frame.getContentPane();

        Panel15 panel = new Panel15();
        contentPane.add(panel);

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(200, 80);
        frame.setVisible(true);
    }
}

class Panel15 extends JPanel {
    Panel15() {
        setLayout(new BorderLayout());
        final JLabel label = new JLabel("Enjoy Java programming.");
```

```
        panel.add(label, BorderLayout.NORTH);

        /* イベントソースの作成 */
        JButton button = new JButton("Push");
        panel.add(button, BorderLayout.CENTER);

        /* イベントリスナの作成と登録 */
        button.addActionListener(new ActionListener() {
            private boolean flag = true;

            /* ボタンが押されたときの処理(イベントハンドラ) */
            public void actionPerformed(ActionEvent e) {
                if (flag == true) {
                    label.setText("Enjoy Swing.");
                    flag = false;
                } else {
                    label.setText("Enjoy Java.");
                    flag = true;
                }
            }
        });
    }
}
```

例題 15 は, 匿名クラス(anonymous class)を用いた方法である. メソッド `actionPerformed()` を持つクラス `ActionListener` のインスタンスを生成するのと同時にそのインスタンスを `button` のリスナとして登録している. このようにすることで, リスナに名前を付けなくてよいので Java の GUI プログラムには多く見られる.

Java のイベントリスナはインタフェースである. よって, インタフェースで宣言されているすべてのイベントハンドラ(抽象メソッド)を, その実装クラスで定義しなければならない. たとえば, インタフェース `MouseListener` を用いる場合, 5 つのメソッドをすべて実装しなければならない. そこで, Java では, 各リスナインタフェースに対して, イベントハンドラの処理を空で実装したアダプタクラスが提供されている. たとえば, インタフェース `MouseListener` に対してはクラス `MouseAdapter` が用意されている.

10.5 グラフィクスの描画

Java で画面に描画を行う際には, グラフィックコンテキストを用いる. プラットフォーム固有のグラフィックコンテキストはクラス `Graphics` のインスタンスで保持されている. また, クラス `Graphics` には, 文字や図形の描画, フォント操作, カラー操作などを行うためのさまざまなメソッドが定義されている(実際には, `Graphics` は抽象クラスで, プラットフォームに対応したサブクラスで実装されている). 画面に文字と線を表示するプログラムを例題 16 に示す.

例題 16: `Sample16.java`

```
import javax.swing.*;
import java.awt.*;

public class Sample16 {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Swing Sample16");
        Container contentPane = frame.getContentPane();

        Canvas16 canvas = new Canvas16();
        contentPane.add(canvas);

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();  // 配置の調整
        frame.setVisible(true);
    }
}

class Canvas16 extends JPanel {
    Font font;

    Canvas16() {
        setBackground(Color.white);                    // 背景色の設定
        setPreferredSize(new Dimension(700, 500));     // 画面サイズの設定
        font = new Font("SanSerif", Font.ITALIC, 18); // フォントの作成
    }

    public void paintComponent(Graphics g) {  // paintComponent()の再定義
        super.paintComponent(g);           // JPanel による背景の描画

        g.setColor(Color.red);             // 描画色の設定
        g.setFont(font);                   // フォントの設定
        g.drawString("Java", 200, 100);    // 座標(200, 100)に文字を描画

        g.setColor(Color.blue);            // 描画色の設定
        g.drawLine(10, 10, 690, 490);      // 直線(10,10)-(690,490)を描画
    }
}
```

現在のグラフィックコンテキストを保持するクラス Graphics のインスタンスを取得するには，例題 16 のようにクラス JComponent のメソッド paintComponent()を再定義するのが簡単である．クラス JComponent のサブクラスが現実化された際に，そのクラスのメソッド paintComponent()は Graphics 型のインスタンスを引数として呼び出される(クラス JComponent のメソッド setVisible()から呼び出される)．また，ウインドウのサイズが変更されたときやウインドウの重なりが変化したときにも paintComponent() が呼び出される．よ

って，画面に文字や図形を描画する場合には，paintComponent()を再定義してGraphics型のインスタンスを取得し，そのインスタンスの適切なメソッドを呼び出せばよい．例題16では，文字や図形の描画キャンバスにクラス JPanel を利用する．グラフィックコンテキストを取得するために，JComponent の paintComponent()を再定義し，クラス Graphics のメソッド drawString()と drawLine()を呼び出すことで文字と直線を描画している．

GUI プログラムからメソッド paintComponent()を明示的に呼び出したい場合には，クラス JComponent のメソッド repaint()を用いる(repaint() → udpate() → paintComponet()の順で呼び出される)．これにより，プログラムは任意の時点で画面の再描画を要求できる．マウスをクリックすると，その位置を中心とする半径20ピクセルの円を描画するプログラムを例題17に示す．

例題 17: Sample17.java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Sample17 {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Swing Sample17");
        Container contentPane = frame.getContentPane();

        Canvas17 canvas = new Canvas17();
        contentPane.add(canvas);

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);
    }
}

class Canvas17 extends JPanel implements MouseListener {
    final static int RADIUS = 20;  // 半径の設定(定数)
    private int x = -1;       // マウスポインタのX座標
    private int y;            // マウスポインタのY座標

    Canvas17() {
        setBackground(Color.white);
        setPreferredSize(new Dimension(700, 500));
        addMouseListener(this);  // 自分自身をリスナとして登録
    }

    public void mouseEntered(MouseEvent e) { } // ウインドウに入ったとき
    public void mouseExited(MouseEvent e) { }  // ウインドウから出たとき
```

```
    public void mousePressed(MouseEvent e) { } // マウスボタンを押したとき
    public void mouseReleased(MouseEvent e) { }// マウスボタンを放したとき

    public void mouseClicked(MouseEvent e) {
    // マウスボタンをクリックしたとき
        x = e.getX();   // マウスポインタのx座標の取得
        y = e.getY();   // マウスポインタのx座標の取得
        repaint();      // 再描画の要求
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);

        if (x != -1) {  // 現実化した直後には描画を行わないため
            /* 円を描画 */
            g.setColor(Color.blue);
            g.fillOval(x - RADIUS, y - RADIUS, RADIUS * 2, RADIUS * 2);
        }
    }
}
```

10.6 メニュー

一般的なGUIプログラムにはメニューが付いている．メニューは，メニューバー(メニューを格納するためのバー形状の入れ物)，メニュー(メニュー項目を格納し管理するための入れ物)，メニュー項目(実際に選択されるもの)で構成される．メニューは入れ子にすることができる(入れ子になったメニューをサブメニューと呼ぶ)．メニュー項目が選択されるとイベント `ActionEvent` を送出する．Swing では，メニューバー，メニュー，メニュー項目を構築するために，それぞれクラス `JMenuBar`, `JMenu`, `JMenuItem` が用意されている．メニューを備えたプログラムを例題18に示す．

例題18: `Sample18.java`

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Sample18 {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Swing Sample18");
        Container contentPane = frame.getContentPane();

        /* メニューバーの生成 */
        JMenuBar menubar = new JMenuBar();
        frame.setJMenuBar(menubar);
```

```
        /* メニューの生成 */
        JMenu fileMenu = new JMenu("File");
        menubar.add(fileMenu);

        /* メニュー項目の生成 */
        JMenuItem openItem = new JMenuItem("Open...");
        fileMenu.add(openItem);

        JMenuItem saveItem = new JMenuItem("Save");
        fileMenu.add(saveItem);
        fileMenu.addSeparator();  // セパレータの追加

        JMenuItem exitItem = new JMenuItem("Exit");
        fileMenu.add(exitItem);

        /* メニュー項目が選択されたときのリスナの作成 */
        exitItem.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.exit(0);
            }
        });

        JPanel panel = new JPanel();
        panel.setBackground(Color.white);
        contentPane.add(panel);

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(700, 500);
        frame.setVisible(true);
    }
}
```

10.7 ダイアログ

ダイアログ(dialog)とは，GUI プログラムにおいて必要なときのみ表示される独立したウインドウである．ダイアログはモーダル(modal)と非モーダル(non-modal)に分けられる．モーダルダイアログは，そのウインドウを閉じないと実行が先に進めないようになっている．非モーダルダイアログはウインドウを閉じなくとも実行を先に進めることができる．`JDialog` を用いてモーダルダイアログを作成するプログラムを例題 19 に示す．

例題 19: `Sample19.java`

```
import javax.swing.*;
import java.awt.*;
```

```
import java.awt.event.*;

public class Sample19 {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Swing Sample19");
        Container contentPane = frame.getContentPane();

        /* ダイアログの作成 */
        final Dialog19 dialog = new Dialog19(frame, "Dialog",
                                             "Can you see me?");

        JButton button = new JButton("Question");
        contentPane.add(button);
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                dialog.setVisible(true);  // ダイアログの表示
            }
        });

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);
    }
}

class Dialog19 extends JDialog {
    Dialog19(Frame frame, String title, String msg) {
        super(frame, title, true);  // モーダルダイアログ
        Container contentPane = getContentPane();

        JButton yesButton = new JButton("Yes");
        contentPane.add(yesButton, BorderLayout.SOUTH);
        yesButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                setVisible(false);
            }
        });

        JLabel label = new JLabel(msg, JLabel.CENTER);
        contentPane.add(label, BorderLayout.CENTER);
        pack();
    }
}
```

JDialog のコンストラクタ呼び出しを以下のように変更すると, 非モーダルダイアログとなる.

```
super(frame, title, false);  // 非モーダルダイアログ
```

さらに，Swing では次に示す標準のダイアログがあらかじめ用意されている.

JOptionPane: 「入力」「確認」「選択」「メッセージ出力」用のダイアログ
JFileChooser: ファイル選択用のダイアログ
JColorChooser: 色選択用のダイアログ

参考図書

[1] Gosling, J., Joy, B., Steel, G. and Bracha, G., Java 言語仕様 第 2 版, 村上雅章 訳, ピアソンエデュケーション, 2000
[2] Arnold, K., Gosling, J. and Holmes, D., プログラミング言語 Java 第 3 版, 柴田芳樹 訳, ピアソンエデュケーション, 2001
[3] Campione, M., Walrath, K. and Hulm, A., Java チュートリアル 第 3 版, 安藤慶一 訳, ピアソンエデュケーション, 2001
[4] 結城浩, Java 言語プログラムレッスン(上)(下), ソフトバンクパブリッシング, 1999
[5] 高橋麻奈, やさしい Java 第 2 版, ソフトバンクパブリッシング, 2002
[6] 高橋麻奈, やさしい Java 活用編, ソフトバンクパブリッシング, 2002
[7] 林晴比古, 新 Java 入門 ビギナー編, ソフトバンクパブリッシング, 2001
[8] 林晴比古, 新 Java 入門 シニア編, ソフトバンクパブリッシング, 2002
[9] Topley, K., JFC プログラミング Vol.1, Swing コンポーネントプログラミング基礎編, 浜田真理 訳, ピアソンエデュケーション, 1999
[10] グラフィック Java2 Vol.2 Swing 編(上)(下), 福龍興業 訳, アスキー, 2000