目 次

第1章　プログラミング

1.1　プログラミング

データ： 計算機処理の対象となる情報

プログラム： 計算機処理の手順を示す情報

プログラミング： 特定の問題を解くためにプログラムを作成すること

アルゴリズム： 情報処理の (プログラムの) 基本的な考え方

プログラミング言語： プログラムを記述するための言語

機械語： 機械固有の言語

高級言語： 機械に独立な言語 (比較的日常言語に近い)
例：Java、C、C++、Pascal、FORTRAN、BASIC、Lisp、Prolog

ソースプログラム (source program)： 高級言語で書かれたプログラム

オブジェクトプログラム (object program)： 機械語で書かれたプログラム

言語プロセッサ： 高級言語プログラムを実行するためのプログラム

コンパイラ (compiler)： ソースプログラムをオブジェクトプログラムに変換するプログラム
プログラムを変換 (翻訳) する作業をコンパイル (compile) と呼ぶ

インタプリタ (interpreter)： プログラムを解釈して実行するプログラム

ソースファイル (source file)： ソースプログラムの入っているファイル (テキストファイル)
Java のソースファイルの拡張子は java 、C のソースファイルの拡張子は c 、C++のソースファイルの拡張子は cc または C である。

オブジェクトファイル (object file)： オブジェクトプログラムの入っているファイル (通常はバイナリファイル)
通常の UNIX のオブジェクトファイルの拡張子は o である。

仮想機械 (virtual machine)： 実際の計算機上で動く仮想の計算機 (プログラム)
Java は複数種の計算機上で実行できるように、各計算機上で「バイトコード」と呼ばれる機械語を解釈・実行するプログラムを用意し、この仮想機械上で与えられたプログラムが実行される

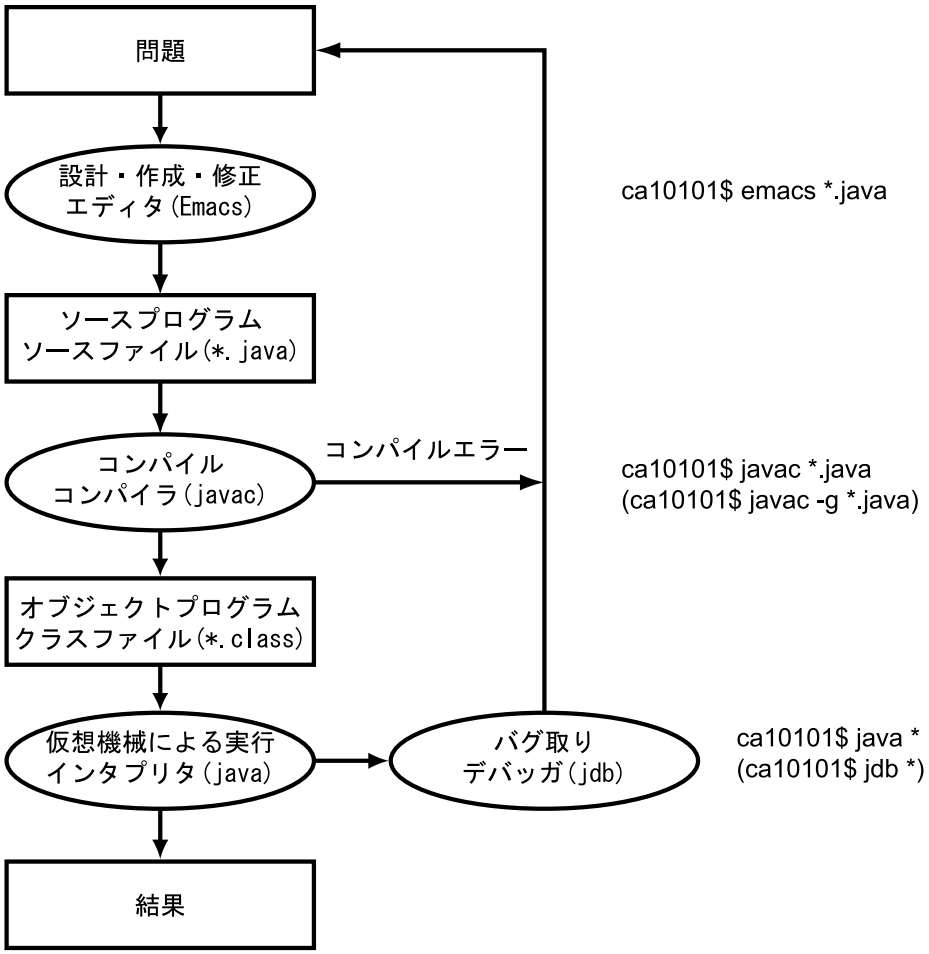


図 1.1: Java プログラミングの流れ

クラスファイル (class file)： Java の仮想機械上で、実行できるプログラムファイル
Java コンパイラによってバイトコードに変換されたオブジェクトファイル。つまり Java の仮想機械はクラスファイルのインタプリタである。クラスファイルの拡張子は class である。

エディタ： ソースファイルや文書ファイルの書き込み／編集を行うプログラム

バグ (虫, bug)： プログラムの間違い

デバッグ (虫取り, debug)： バグを修正すること

デバッガ (debugger)： デバッグ作業を支援するプログラム

パッケージ (package)： Java のクラスファイルの位置
Java では必要に応じて他のクラスファイルを利用する。`import` 命令によってクラスファイルの指定を行なう。

1.2 Javaプログラムの作成と実行

1.2.1 プログラムの作成

エディタ (Mule) を使ってプログラムを入力する。通常、Java のソースファイルは クラス名.java というファイル名にする。たとえば、以下の例であれば Renshu.java というファイル名にする。java のプログラムは、アルファベットの大文字と小文字を区別するが、基本的に英小文字 を用いる。
プログラム (Renshu.java) の例：

```
public class Renshu {
  public static void main(String[] args) {
    System.out.println("Hello, yama!");
  }
}
```

1.2.2 コンパイルと実行

Java のコンパイラとしては、情報棟では javac が利用できる。コンパイルの結果作られるクラスファイルを仮想機械であるインタプリタ java に与えるとプログラムが実行される。

javac(Java compiler)： Java プログラム用のコンパイラ
　デバッガを使う場合には、オプション-g を指定する必要がある。

　-g： デバッガを利用できるようにする

コンパイル・リンクと実行のコマンド例を示す。ここでソースプログラムには、1.2.1 節のプログラムを用いており、ファイル名は Renshu.java とする。

```
ca10101$ ls ¶                              (ソースファイルは Renshu.java)
Renshu.java
ca10101$ javac Renshu.java ¶               (コンパイルする)
ca10101$ ls -gl ¶                          (クラスファイル Renshu.class ができる)
total 2
-rw-r--r--   1 yama     teacher       418 10 11 19:19 Renshu.class
-rw-r--r--   1 yama     teacher       111 10 11 19:19 Renshu.java
ca10101$ java Renshu ¶                       (プログラムを実行する)
Hello, yama!
```

1.2.3 デバッグ

コンパイルエラー (compile error)： コンパイル時にコンパイラが指摘したエラー
　主に文法上のエラー (syntax error) が中心

jdb(java debugger)： デバッガ
　コンパイルに成功しても、プログラムが予想したように動かないことが多い。このよ

うなときにはデバッガを用いるとバグを見つけるのが楽になる。コンパイル時に -g オプションをつけておく必要がある。

```
ca10101$ cat Comperr.java ¶                (3行目の ; がない)
public class Comperr {
  public static void main(String[] args) {
    System.out.println("Hello, yama!")
  }
}
ca10101$ javac Comperr.java ¶              (コンパイルエラー)
Comperr.java:3: ';' がありません。
    System.out.println("Hello, yama!")
                                       ^
エラー 1 個
ca10101$ javac -g Renshu.java ¶            (デバッガ用のコンパイルオプション)
ca10101$ jdb Renshu ¶                      (デバッガ jdb の起動)
jdb の初期化中です...
> help ¶                                   (デバッガのコマンド一覧)
** command list **
run [class [args]]          -- アプリケーションのメインクラスの実行を開始します。

threads [threadgroup]       -- スレッドを一覧表示します
thread <thread id>          -- デフォルトスレッドを設定します
suspend [thread id(s)]      -- スレッドを中断します (デフォルト: すべて)
resume [thread id(s)]       -- スレッドを再開します (デフォルト: すべて)
where [thread id] | all     -- スレッドのスタックをダンプします
wherei [thread id] | all    -- スレッドのスタックを PC 情報といっしょにダンプします
up [n frames]               -- スレッドのスタックを上へ移動します
down [n frames]             -- スレッドのスタックを下へ移動します
kill <thread> <expr>        -- 指定された例外オブジェクトでスレッドを終了します
interrupt <thread>          -- スレッドに割り込みます

print <expr>                -- 式の値を印刷します
dump <expr>                 -- すべてのオブジェクト情報を印刷します
eval <expr>                 -- 式を評価します (印刷と同じ)
set <lvalue> = <expr>       -- フィールド/変数/配列要素に新しい値を割り当てます
locals                      -- 現行のスタックフレームのすべてのローカル変数を印刷します

classes                     -- 現時点で既知のクラスを一覧表示します
class <class id>            -- 名前付きクラスの詳細を表示します
methods <class id>          -- クラスのメソッドを一覧表示します
```

```
fields <class id>          -- クラスのフィールドを一覧表示します

threadgroups               -- スレッドグループを一覧表示します
threadgroup <name>         -- 現在のスレッドグループを設定します

stop in <class id>.<method>[(argument_type,...)]
                           -- メソッドにブレークポイントを設定します
stop at <class id>:<line> -- 行にブレークポイントを設定します
clear <class id>.<method>[(argument_type,...)]
                           -- メソッドのブレークポイントを解除します
clear <class id>:<line>    -- 行のブレークポイントを解除します
clear                      -- ブレークポイントを一覧表示します
catch <class id>           -- 指定された例外がスローされたら停止します
ignore <class id>          -- 指定された例外の 'catch' を取り消します
watch [access|all] <class id>.<field name>
                           -- フィールドへのアクセス/修正を監視します
unwatch [access|all] <class id>.<field name>
                           -- フィールドへのアクセス/修正の監視を中止します
trace methods [thread]     -- メソッドへの出入りを追跡します
untrace methods [thread]  -- メソッドへの出入りの追跡を中止します
step                       -- 現在の行を実行します
step up                    -- 現在のメソッドが呼び出し側に戻るまで実行します
stepi                      -- 現在の命令を実行します
next                       -- 1 行ステップ実行します (呼び出しをステップオーバー
します)
cont                       -- ブレークポイントから継続して実行します

list [line number|method] -- ソースコードを印刷します
use (or sourcepath) [source file path]
                           -- ソースパスを表示または変更します
exclude [class id ... | "none"]
                           -- 指定したクラスのステップまたはメソッドイベントを報
告しません
classpath                  -- ターゲット VM の classpath 情報を印刷します

monitor <command>          -- プログラムの停止時に毎回コマンドを実行します
monitor                    -- モニタを一覧表示します
unmonitor <monitor#>       -- モニタを削除します
read <filename>            -- コマンドファイルを読み込み、実行します

lock <expr>                -- オブジェクトのロック情報を印刷します
threadlocks [thread id]    -- スレッドのロック情報を印刷します
```

```
pop                          -- 現在のフレームを含むすべてのスタックをポップします
reenter                      -- ポップと同じですが、現在のフレームが再入力されます
redefine <class id> <class file name>
                             -- クラスのコードを再定義します

disablegc <expr>             -- オブジェクトのガベージコレクションを抑止します
enablegc <expr>              -- オブジェクトのガベージコレクションを許可します

!!                           -- 最後のコマンドを繰り返します
<n> <command>                -- コマンドを n 回繰り返します
help (or ?)                  -- コマンドを一覧表示します
version                      -- バージョン情報を印刷します
exit (or quit)               -- デバッガを終了します

<class id>: パッケージ修飾子または先頭か末尾に
ワイルドカード ('*') のパターンの付いたクラスのフルネーム
<thread id>: 'threads' コマンドで報告されるスレッド番号
<expr>: Java(tm) プログラミング言語式
もっとも一般的な構文がサポートされます。

user.home または user.dir では、起動コマンドを
"jdb.ini" か ".jdbrc" のいずれかに配置することができます
> exit ¶                                    (デバッガの終了)
```

第2章　Java言語

2.1　プログラムの基本構造

練習用プログラム - ファイル名: Renshu.java

＜プログラム＞

```
public class Renshu {
  public static void main(String[] args) {
    System.out.println("Hello, yama!");
  }
}
```

＜説明＞

```
// Renshu クラスの始まり
// main メソッドの始まり
// 実行文
// main メソッドの終り
// Renshu クラスの終り
```

Java のプログラムは，一部の例外 (2.10 節の注意事項参照) を除いて 半角英数字 を用いる．英字には大文字と小文字の区別があり，通常は小文字を使う．Java のプログラムはクラスの集まりであり，各クラスはメソッドの集まりである．当分の間，クラスはプログラムファイルと 1 対 1 に対応し，ファイル名とクラス名は同じになる．一般にクラス名の先頭には大文字を用いる (renshu ではなく Renshu というぐあい)．なおプログラムは‘ main ’と名づけられたメソッドから実行される．

文： 実行や宣言などの単位 (ひとかたまり)
　　文の末尾にはセミコロン (;) がつく (日本語の句点 (．) や英語のピリオド (.) と同じ)

2.2　出力文

出力文： PrintStream クラスのメソッド*
　　画面に与えられたデータを表示する実行文

```
System.out.println ( 項目 );
      (1)          (2)
```

(1)　メソッド名　画面 (標準出力) にデータを表示するメソッド
　　System.out.println - 最後に改行，System.out.print - 改行せず
(2)　項目　　定数，変数，式など

例１： 地球の表面積の計算 (地球の半径 = 6371000 m) - Surface1.java

```
public class Surface1 {
  public static void main(String[] args) {
    System.out.println(4.0*3.14159265358979*6371000.0*6371000.0);
```

*stream とはデータを入出力するための通路のようなもの.

```
  }
}
```

<実行結果>

```
ca10101$ java Surface1 ¶
5.1006447190978775E14
```

フリーフォーマット： 英語のように空白 (スペース，タブ，改行) は自由に挿入できる
変数名や数値などの単語の途中には空白を入れられない
行頭の空白は，インデント (indent) ないし字下げと呼ばれるが，プログラムを読みやすくするためにはインデントの付け方に注意を払うべきである

例２： プログラム文中の空白の効果 - Surface2.java

```
public  class
Surface2  { public  static
void  main (   String[] args  )
     {   System.out.println
(4.0   *  3.14159265358979 *
6371000.0*6371000.0); } }
```

文字列 (文字列定数)： 文字の並びのデータ
ダブルクオート (") で囲まれた部分

例３： 式と文字列 - Surface3.java

```
public class Surface3 {
  public static void main(String[] args) {
    System.out.println(4.0*3.14159265358979*6371000.0*6371000.0);
    System.out.println("4.0*3.14159265358979*6371000.0*6371000.0");
  }
}
```

<実行結果>

```
ca10101$ java Surface3 ¶
5.1006447190978775E14
4.0*3.14159265358979*6371000.0*6371000.0
```

例４： print と println の相違 (改行の位置に注意せよ) - Surface4.java

```
public class Surface4 {
  public static void main(String[] args) {
    System.out.println(4.0*3.14159265358979*6371000.0*6371000.0);
    System.out.println("4.0*3.14159265358979*6371000.0*6371000.0");
    System.out.println();
```

```
    System.out.println();
    System.out.print(4.0*3.14159265358979*6371000.0*6371000.0);
    System.out.print("4.0*3.1415");
  }
}
```

2.3 整数型，実数型 (基本データ型)

型： 値の種類

整数型 (10)，実数型 (1.414)，文字列型 ("Yamaguchi") など

整数型： 整数 (integer number) を表わす型

整数 (小数点を含まない数) を正確に表現できる*

最も短い整数 (byte)	：	8bit	-2^7	$\sim$	2^7-1
短い整数 (short)	：	16bit	-2^{15}	$\sim$	$2^{15}-1$
通常の整数 (int)	：	32bit	-2^{31}	$\sim$	$2^{31}-1$
倍長整数 (long)	：	64bit	-2^{63}	$\sim$	$2^{63}-1$

整数型定数の例： 最後に L をつけると long 型になる

0，1023，-16，2147483648L

実数型： 浮動小数点数 (floaing point number) を表わす型

小数点を含む数や非常に大きな数／小さい数を表現できる (ただし一般に近似値となる) †

短い実数 (float)	：	32bit	$\pm 1.4\times10^{-45}$	$\sim$	$\pm 3.4\times10^{+38}$
通常 (倍長) 実数 (double)	：	64bit	$\pm 4.9\times10^{-324}$	$\sim$	$\pm 1.7\times10^{+308}$

実数型定数の例： 最後に F/D をつけると float/double 型

1.0，3.141592F，6.04e23($=6.04\times10^{23}$)，1e-100D($=10^{-100}$)

例１： 整数型の範囲 (1) - Integer1.java

```
public class Integer1 {
  public static void main(String[] args) {
    System.out.println(6371000*6371000*4.0*3.14159265358979);
  }
}
```

<実行結果>

```
ca10101$ java Integer1 ¶
-2.6325471962129 43E10
```

例２： 整数型の範囲 (2) - Integer2.java

*詳細な説明は 3.3 節の「整数データ表現」を参照せよ

†詳細な説明は 4.3 節の「実数データ表現」や 4.4 節の「丸め誤差」を参照せよ

```
public class Integer2 {
  public static void main(String[] args) {
    System.out.println(6371000*6371000);
  }
}
```

例 3： 実数型の範囲 - Double1.java

```
public class Double1 {
  public static void main(String[] args) {
    System.out.println(6371000.0*6371000.0);
  }
}
```

例 4： 実数型の精度 - Double2.java

```
public class Double2 {
  public static void main(String[] args) {       // 小数点以下 17 桁
    System.out.println(1.1111111111111111);
    System.out.println(2.22222222222222222);
    System.out.println(3.33333333333333333);
    System.out.println(4.44444444444444444);
  }
}
```

2.4　変数

変数： 値を格納する名前付の器 (一般に値は変化する)

定数： 一定の値

定数変数： 値が一定として宣言された変数 (名前のついた定数)
　型修飾子として final がつき，必ず初期化を伴う．慣習として名前 (識別子) に大文字を用いることが多い．

変数宣言： ある名前を特定の型の変数として宣言する
(アクセス修飾子)　(型修飾子)　型名　変数名, 変数名 ...
　例：
```
int  x, y;
final float  PI;
public static double  a;
```

識別子： 名前のこと
文字 (A〜Z,a〜z)，下線 (_)，ドル記号 ($)，数字 (0〜9) から構成される (先頭に数字は使えない)
アルファベットでは大文字と小文字の区別がある (漢字=全角文字も使えるが避けた方が良い)

変数名： 変数を表わす識別子
例： ans, wa, sa, CP_97_10_21

型名 ： 型を表わす識別子
例： int, float, double

予約語 (キーワード)： 特別の意味を持つ識別子 (変数名やメソッド名に利用できない)
例： public, class, static, void, int, float, if, for, while

例１： 定数変数の利用 (2.2 節の例 1 参照) - Variable1.java

```
public class Variable1 {
  public static void main(String[] args) {
    final double PI = 3.14159265358979;
    final double R = 6371000.0;
    System.out.println(4.0*PI*R*R);
  }
}
```

```
<実行結果>
ca10101$ java Variable1 ¶
5.1006447190978775E14
```

例２： 定数変数の利用 (2.3 節の例 2 参照) - Variable2.java

```
public class Variable2 {
  public static void main(String[] args) {
    final int R = 6371000;
    System.out.println(R*R);
  }
}
```

例３： 地球表面の塵の体積 - Variable3.java

```
public class Variable3 {
  public static void main(String[] args) {
    final double PI = 3.14159265358979;
    final double R = 6371000.0;
    final double D = 0.001;
    System.out.println(4.0/3.0*PI*(R+D)*(R+D)*(R+D)-4.0/3.0*PI*R*R*R);
  }
}
```

2.5　代入文

代入演算子 (=)： 右辺の式を計算し，左辺の変数に結果の値を格納して，その値を返す
左辺は変数でなくてはならない

代入文： 代入演算子を含む文
変数名 = 計算式 ;

変数の初期化： 変数宣言と同時に値を代入すること
定数変数は必ず初期化を伴う
例：
```
int  x = 1;
final double  R = 6371000.0;
```

2.6　コマンド入力

Java では実行時のコマンド引数を入力データとして取り扱える．しかし，コマンド入力について正確に理解するためには文字列 (String)，配列，さらにはオブジェクト指向 (クラスとインスタンス) についての知識が必要であり，ここでは細かな説明は省略して利用する．

コマンド入力： コマンド引数によって入力データを与える
次のようにプログラムを実行すると，プログラムにデータ「0.001」を渡せる．
この際，main メソッドの args 変数 (配列) に文字列データが入る．args は配列なので args[0]，args[1]，... というようにコマンド引数は複数の文字列に分けられて扱われる．

```
ca10101$ java Assign1 0.001 ¶
```

例外処理 (Exception)： プログラムが正常に実行されない場合の処理
プログラムに与えるべきデータがなかったり，データが適切でなかったりした場合には，例外処理 (Exception) が実行される．

例１： 厚みを与えて塵の体積を計算する - Assign1.java
以下のプログラムでは Double.parseDouble(args[0]) とある部分でコマンド引数の文字列を double 型のデータに変換している．[‡]
int 型の場合には Integer.parseInt(args[0]) となる．

```
public class Assign1 {
  public static void main(String[] args) {
    final double PI = 3.14159265358979;
    final double R = 6371000.0;
    double d = Double.parseDouble(args[0]);
                                        // コマンド文字列の double 型への変換
    System.out.println(4.0/3.0*PI*(R+d)*(R+d)*(R+d)-4.0/3.0*PI*R*R*R);
```

[‡]Double.parseDouble は Java 2 SDK 1.2 から導入されたクラスメソッドのため，JDK 1.1.? 以前の Java を利用する場合には，Double.parseDouble(args[0]) の代りに Double.valueOf(args[0]).doubleValue() と書く必要がある．

```
  }
}
```

<実行結果>

```
ca10101$ java Assign1 0.001 ¶          (コマンド引数による入力)
5.10064459776E11
ca10101$ java Assign1 ¶                (引数を忘れた場合の例外処理)
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException
        at Assign1.main(Assign1.java:5)
ca10101$ java Assign1 test ¶           (double 型でない引数の例外処理)
Exception in thread "main" java.lang.NumberFormatException: test
        at java.lang.FloatingDecimal.readJavaFormatString(FloatingDecimal.java:121
        at java.lang.Double.parseDouble(Double.java:201)
        at Assign1.main(Assign1.java:5)
```

2.7 整数型，実数型 (基本データ型) の演算

二項演算子： 2 数の演算を行なう演算子
加法／減法／乗法／除法／剰余演算子

単項演算子： 1 数の演算を行なう演算子
インクリメント／デクリメント／単項マイナス演算子

加法演算子 (+)： 2 数 (定数，変数，式) の和を計算する
式 + 式

減法演算子 (-)： 2 数 (定数，変数，式) の差を計算する
式 - 式

乗法演算子 (*)： 2 数 (定数，変数，式) の積を計算する
式 * 式

除法演算子 (/)： 2 数 (定数，変数，式) の商を計算する
式 / 式

剰余演算子 (%)： 2 整数 (定数，変数，式) の剰余 (余り) を計算する
式 % 式

例 1： 塵の厚みを mm 単位で与える - Assign2.java

```
public class Assign2 {
  public static void main(String[] args) {
    final double PI = 3.14159265358979;
    final double R = 6371000.0;
    double d = Double.parseDouble(args[0]);
    d = d/1000.0;
```

```
    System.out.println(4.0/3.0*PI*(R+d)*(R+d)*(R+d)-4.0/3.0*PI*R*R*R);
  }
}
```

<実行結果>

```
ca10101$ java Assign2 1 ¶
5.10064459776E11
```

単項マイナス演算子 (-)： 数の符号を逆転する

-式

複合代入演算子 (+=，-=，*=，/=，%=)： 二項演算子と代入演算子を複合した演算子

以下の2つは同じ効果を持つ

変数 複合代入演算子 式 `a += b;`

→ 変数 = 変数 二項演算子 式 `a = a + b;`

インクリメント (increment) 演算子 (++)： 整数変数に1を加え，その値を返す

変数++，++変数

例： `sum = a + b++;` → a と b を加え sum に代入し，b に1を加える

`sum = a + ++b;` → b に1を加えて，a と b を加え sum に代入する

デクリメント (decrement) 演算子 (--)： 整数変数から1を引き，その値を返す

変数--，--変数

演算子の優先順：

単項演算子 > 二項演算子 (*，/，%，+，-) > 代入演算子

左の演算子 > 右の演算子 (代入演算子だけは，右が優先)

() で囲まれた部分は優先される

算術変換： 二項演算子や代入演算子で複数の型があると型が変換される

二項演算子： int と long なら long

int と double なら double

float と double なら double

代入演算子： 左辺の変数が long で右辺が int なら long

左辺の変数が double で右辺が int なら double

左辺の変数が double で右辺が float なら double

キャスト (cast)/型変換： キャストによって明示的にデータの型を変換できる

（型名）データ

例：
```
double d = 3.1415;
float f = (float)d;
int i = (int)d;
```

注意事項： 数式の書き方

1. 少しでも怪しいときは () を用いる
   例： `x = a / b / c;` → `x = (a / b) / c; x = a / (b / c);`
   `x = a + b * c;` → `x = a + (b * c); x = (a + b) * c;`
2. 変数と二項 (代入) 演算子の間には，なるべく空白 ( ) を入れる
   例： `x=a+++b;` → `x = a++ + b; x = a + ++b;`

2.8 文字列型 (参照データ型)

文字列型： 文字の並びを表す型 (String 型)

キーボードから入力されるデータやディスプレイに表示されるデータは，文字の並びであり，String 型となる．
コマンド引数として与えられるデータも String 型

```
public static void main(String[] args) {
                        ^^^^^^ この部分に注目!!
```

文字列型変数宣言：

```
String  name;
```

文字列型定数の例：” で囲まれた文字の並び

```
"Hello, Yama!!", "山口"
```

文字列演算子 (+)： 2つの文字列を結合する
文字列 + 文字列

例１： 文字列の結合 (1) - String1.java

```
public class String1 {
  public static void main(String[] args) {
    System.out.println("Hello, " + "yama!");
  }
}
```

<実行結果>

```
ca10101$ java String1 ¶
Hello, yama!
```

例２： 文字列の結合 (2) - String2.java
コマンド引数は args[0], args[1] ... というように一単語ずつ分離して渡される．

```
public class String2 {
  public static void main(String[] args) {
    System.out.println("Hello, " + args[0] + " and " + args[1] + "!");
  }
```

```
}
```

<実行結果>

```
ca10101$ java String2 Yasushi Taro ¶     （コマンド引数を２つ与える）
Hello, Yasushi and Taro!
ca10101$ java String2 Taro Yasushi ¶     （順序を変えてみると…）
Hello, Taro and Yasushi!
```

文字列変換： 多くのデータ型は文字列に変換される

一旦，文字列になると文字列 (文字の並び) として結合される．
加算 (+) の最初の項が文字列だと，すべての項が文字列に変換されて結合される．

文字列 + 他のデータ型

例３： 文字列変換 (1) - Conversion1.java

```
public class Conversion1 {
  public static void main(String[] args) {
    int  a = 10;
    System.out.println("a = " + a);       // 文字列 "a = " と整数 a の和
  }
}
```

<実行結果>

```
ca10101$ java Conversion1 ¶
a = 10
```

例４： 文字列変換 (2) - Conversion2.java

括弧の付きかたによって，同じ演算子 '+' であっても意味が異なる．

```
public class Conversion2 {
  public static void main(String[] args) {
    int  a = Integer.parseInt(args[0]);
    int  b = Integer.parseInt(args[1]);
    System.out.println("Result: " + a + " + " + b + " = " + (a+b));
  }
}
```

データ変換： 文字列から基本データに変換する (やや面倒)

文字列 str を int 型データに変換するにはメソッド Integer.parseInt(str) を，double 型データに変換するにはメソッド Double.parseDouble(str) を，それぞれ利用する (2.6 節の例 1 を参照のこと)．

2.9 文字型 (基本データ型)

文字型： 文字 1 つを表す型 (char 型)

Java では Unicode を採用しており，すべての文字を 2 byte で表現している．

文字型変数宣言：

```
char   変数名，変数名 ...
```

文字型定数の例： ’ で囲まれた文字 (文字列は ” で囲まれる)

```
'1', 'A', 'b', '!', 'n'
```

バックスラッシュ文字： 通常の文字以外の特殊な文字 (制御文字など)

\n	→	復帰改行 (NL)
\t	→	タブ (HT)
\b	→	バックスペース (BS)
\f	→	改行 (FF)
\r	→	復帰 (CR)
\\	→	\
\’	→	’
\"	→	"
\uxxxx	→	Unicode 文字 (xxxx は 4 桁の 16 進数)

2.10 コメント

人がプログラムの意味を理解しやすくするために，コメント (注釈) を入れられる (プログラムの実行には影響しない)
コメントは，/* で始まり， */ で終る (/**で始まるコメントは javadoc というドキュメント作成コマンドに利用される)
ある行の後半をすべてコメントとするには，// を使う

注意事項： 日本語 (Unicode，ECC では EUC コード) を用いてよい箇所

(1) コメント
(2) 文字列
(3) 識別子 (変数名, クラス名, メソッド名) - 避けた方が無難

例１： 読み易いプログラム - Comment1.java

```
public class Comment1 {
  /** 地球表面に積もった塵の体積を求める */
  public static void main(String[] args) {
    final double パイ = 3.14159265358979; /* 円周率 */
    final double R = 6371000.0;           // 半径 (radius)
    double d = Double.parseDouble(args[0]);
    d /= 1000.0;                          // 複合代入演算子
    System.out.println("体積 = " +
```

```
        (4.0/3.0*パイ*(R+d)*(R+d)*(R+d)-4.0/3.0*パイ*R*R*R) + " m^3");
  }
}
```

<実行結果>

```
ca10101$ java Comment1 1 ¶
体積 = 5.10064459776E11 m^3
```

2.11　while による繰返し (制御構造)

制御構造： 特定の条件に応じて実行 (振舞い) を変える仕組み

Java では while，do，for などによる繰返しや if による条件分岐が用意されている

繰返し： 条件が成立する間，繰返し実行する

while 文全体が，また 1 つの文を構成するとみなせる

while は先に条件判定を行なうので，場合によっては 1 回も実行されない

while（式）文

→ 式が正しい場合は，文を実行し，再び式を評価する

　式が正しくない場合には，次に進む

例 1： 単利の元利計算 (利率 5.75%の場合) - While1.java

```
public class While1 {
  public static void main(String[] args) {
    final double R = 0.0575;   // 5.75%
    System.out.println("Term " + 1 + ": " + (1.0+R*1));
    System.out.println("Term " + 2 + ": " + (1.0+R*2));
    System.out.println("Term " + 3 + ": " + (1.0+R*3));
    System.out.println("Term " + 4 + ": " + (1.0+R*4));
    System.out.println("Term " + 5 + ": " + (1.0+R*5));
  }
}
```

<実行結果>

```
ca10101$ java While1 ¶
Term 1: 1.0575
Term 2: 1.115
Term 3: 1.1725
Term 4: 1.23
Term 5: 1.2875
```

例 1'： 単利の利息計算 (利率 5.75%の場合) - While2.java

```
public class While2 {
  public static void main(String[] args) {
    final double R = 0.0575;              // 5.75%
    int    i = 1;
    System.out.println("Term " + i + ": " + (1.0+R*i));  i++;
    System.out.println("Term " + i + ": " + (1.0+R*i));  i++;
    System.out.println("Term " + i + ": " + (1.0+R*i));  i++;
    System.out.println("Term " + i + ": " + (1.0+R*i));  i++;
    System.out.println("Term " + i + ": " + (1.0+R*i));  i++;
  }
}
```

例２： 繰返しの利用 ({ と } については 2.12 節を参照せよ) - While3.java

```
public class While3 {
  public static void main(String[] args) {
    final double R = 0.0575;              // 5.75%
    int    i = 1;
    while (i < 6) {
      System.out.println("Term " + i + ": " + (1.0+R*i));
      i++;
    }
  }
}
```

2.12 複文 (制御構造)

複文： 複数の宣言と複数の文の並びを { } で囲い，全体を１つの文として扱う
たとえば，複文を用いることで，while 文の実行文に複数の文を書ける
間違いを少なくするために { } をできるだけ利用する方がよい

インデント (indent)： 行頭の空白
複文ブロックを構成する文には，同じインデントをつけると見やすくなる

例１： 複文の { } を忘れると… - While4.java

```
public class While4 {
  public static void main(String[] args) {
    final double R = 0.0575;              // 5.75%
    int    i = 1;
    while (i < 6)         // 計算機の実行にはインデントよりも { } が重要
      System.out.println("Term " + i + ": " + (1.0+R*i));
      i++;
  }
```

```
}
```

<実行結果>

```
ca10101$ java While4 ¶
Term 1: 1.0575
Term 1: 1.0575
  :
Term 1: 1.0575
^C                                （暴走して止まらなくなるので <Ctrl-C>で止める）
```

2.13　真理値型 (基本データ型)

真理値型： 真偽 (true，false) を表す型 (boolean 型)

関係演算や等値演算の結果は，真理値型の値となる.

真理値型変数宣言：

```
boolean  変数名，変数名 ...
```

真理値型定数： true または false

関係演算子： 右辺と左辺の値を比較し，大小関係の真理値 (真偽) を返す

- a > b　：a>b ならば真，そうでなければ偽
- a < b　：a<b ならば真，そうでなければ偽
- a >= b　：a≥b ならば真，そうでなければ偽
- a <= b　：a≤b ならば真，そうでなければ偽

等値演算子： 右辺と左辺の値を比較し，等値関係の真理値 (真偽) を返す

- a == b　：a＝b ならば真，そうでなければ偽
- a != b　：a≠b ならば真，そうでなければ偽

論理演算子： 真理値 (真偽) の間の演算

- X & Y　：X が真かつ Y が真ならば真，そうでなければ偽 (論理積)
- X && Y　：論理積 - &と同じだが，左辺が偽ならば右辺を評価せずに偽を返す
- X | Y　：X が真または Y が真ならば真，そうでなければ偽 (論理和)
- X || Y　：論理和 - |と同じだが，左辺が真ならば右辺を評価せずに真を返す
- !X　：X が偽ならば真，そうでなければ偽 (論理否定)

真理値表

X	Y	X&Y	X&&Y	X|Y	X||Y	!X
真	真	真	真	真	真	偽
真	偽	偽	偽	真	真	偽
偽	真	偽	偽	真	真	真
偽	偽	偽	偽	偽	偽	真

例：

`!(a == b)`	a と b は等しくない　`a != b`
`!(a > 0)`	a は正ではない　`a <= 0`
`(a > 0) && (a % 2 == 0)`	a は正の偶数である
`(a > 0) && (a < 2)`	a は 0 より大きく 2 より小さい (数学のように $0 < a < 2$ と書いてはならない))

無限ループ： いつまでも終らないループ

```
while (true) ...;
```

例１： 真理値型と関係演算 - Bool1.java

```
public class Bool1 {
  public static void main(String[] args) {
    boolean b1 = (1 > 0);
    boolean b2 = (0 > 1);
    System.out.println("True (1 > 0) is " + b1);
    System.out.println("False (0 > 1) is " + b2);
  }
}
```

例２： 真理値型と論理演算 - Bool2.java

```
public class Bool2 {
  public static void main(String[] args) {
    System.out.println("A" + "\t" + "B" + "\t" +
                "A & B" + "\t" + "A && B" + "\t" +
                "A | B" + "\t" + "A || B" + "\t" + "!A");
    System.out.println("" + true + "\t" + true + "\t" +
                (true & true) + "\t" + (true && true) + "\t" +
                (true | true) + "\t" + (true || true) + "\t" + (!true));
    System.out.println("" + true + "\t" + false + "\t" +
                (true & false) + "\t" + (true && false) + "\t" +
                (true | false) + "\t" + (true || false) + "\t" + (!true));
    System.out.println("" + false + "\t" + true + "\t" +
                (false & true) + "\t" + (false && true) + "\t" +
                (false | true) + "\t" + (false || true) + "\t" + (!false));
    System.out.println("" + false + "\t" + false + "\t" +
                (false & false) + "\t" + (false && false) + "\t" +
                (false | false) + "\t" + (false || false) + "\t" + (!false));
  }
}
```

2.14 for による定数回の繰返し (制御構造)

繰返し： 条件が成立する間，繰返し実行する

for は繰返しの回数が明示的に書けるので，定数回の繰返しには for の利用が好ましい)

for 文全体が，また 1 つの文を構成するとみなせる

for（式 1; 式 2; 式 3）文
→ まず式 1 を評価する (最初の 1 回だけ)
式 2 が成立するならば，文を実行し，式 3 を評価する
式 2 が成立しないならば，次に進む

以下のプログラムと全く等価である

```
式1;
while (式2) {
  文
  式3;
}
```

例 1： for を使った元利計算 (単利) - For1.java

```
public class For1 {
  public static void main(String[] args) {
    final double R = 0.0575;              // 5.75%
    for (int i = 1; i < 101; i++)
      System.out.println("Term " + i + ": " + (1.0+R*i));
  }
}
```

<実行結果>

```
ca10101$ java For1 ¶
  :                                    (全部で 100 行表示される)
Term 98: 6.635000000000001             (浮動小数点数の誤差)
Term 99: 6.6925
Term 100: 6.75
```

例 2： for を使った元利計算 (複利) - For2.java

```
 1: public class For2 {
 2:   public static void main(String[] args) {
 3:     final double R = 0.0575;              // 5.75%
 4:     double x = 1.0;
 5:     for (int i = 1; i < 101; i++) {
 6:       x = x*(1.0+R);
```

```
 7:       System.out.println("Term " + i + ": " + x);
 8:     }
 9:   }
10: }
```

<実行結果>

```
ca10101$ java For2 ¶
  :
Term 98: 239.5945045283517
Term 99: 253.37118853873196
Term 100: 267.94003187970907
```

（例１とは結果が大きく違うことに注意せよ）

例２'： 複合演算子の利用 (例２の 6 行目は次のように書いても良い)

```
 6:     x *= 1.0+R;
```

例３： (当初/元金の) 一定割合 (Y) で返済 (減少) する場合 - For3.java

```
public class For3 {
  public static void main(String[] args) {
    final double R = 0.0575;              // 5.75%
    final double Y = 0.1;                 // 10.0%
    double x = 1.0;
    for (int i = 1; i < 101; i++) {
      x *= 1.0+R;
      x -= Y;
      System.out.println("Term " + i + ": " + x);
    }
  }
}
```

2.15 変数宣言とスコープ

変数宣言： 識別子 (名前) を特定の型の変数名として有効化する

スコープ： (変数名などの) 識別子が有効な範囲
変数名の場合，変数宣言以降で，複文ブロック (中括弧 { }) の内側がスコープとなる.

例１： 2.14 節の例２ - For4.java
7 行目と 8 行目を入れ換えるとコンパイルできない.

```
 1: public class For4 {
 2:   public static void main(String[] args) {
 3:     final double R = 0.0575;              // 5.75%
 4:     double x = 1.0;
```

```
 5:     for (int i = 1; i < 101; i++) {
 6:       x = x*(1.0+R);
 7:     }
 8:     System.out.println("Term " + i + ": " + x);
 9:   }
10: }
```

例 1'： スコープの変更

例 1 の i の変数宣言を変えるとコンパイルできる (プログラムとしての意味はない).

```
 4:     double x = 1.0;
 4':    int i;
 5:     for (i = 1; i < 101; i++) {
```

2.16　do による繰返し (制御構造)

繰返し： 条件が成立する間，繰返し実行する

do 文全体が，また 1 つの文を構成するとみなせる

do 文 while（式）

→ まず文を実行し，次に式を評価する

式が正しい場合には，再び文を実行し，式の評価を行なう

正しくない場合には，次に進む

注意： (while と do の違い)

while は，最初に条件が成立しなければ，1 回も文を実行しない

do は，条件判定が後にあるので，必ず 1 回は文を実行する

例 1： 返済の計算 (残りが負になったら終了する) - Do1.java

```
 1: public class Do1 {
 2:   public static void main(String[] args) {
 3:     final double R = 0.0575;              // 5.75%
 4:     final double Y = 0.1;                 // 10.0%
 5:     int    i = 1;
 6:     double x = 1.0;
 7:     do {
 8:       x *= 1.0+R;
 9:       x -= Y;
10:       System.out.println("Term " + i + ": " + x);
11:       i++;
12:     } while (x > 0.0);
13:   }
14: }
```

例２： 返済所要期数だけを表示する (例１の 12〜14 行目を変更する)
判定の直前に n++ と n の値が 1 つ増えているので，表示の際に 1 つ減らす必要がある

```
10:       i++;
11:     } while (x > 0.0);
12:     System.out.println("Terms: " + (i-1));
```

2.17 繰返しの応用

1. $\sum_{i=1}^{n} i = 1 + 2 + \cdots + n$ の計算

例１： while を用いる場合

```
int sum = 0;
int i = 1;
while (i <= 10) {
  sum += i;  i++;
}
```

例２： do を用いる場合

```
int sum = 0;
int i = 1;
do {
  sum += i;  i++;
} while (i <= 10);
```

例３： for を用いる場合

```
int sum = 0;
for (int i = 1; i <= 10; i++)
  sum += i;
```

2. 繰返しの入れ子：九九表

```
for (int i = 1; i < 10; i++)
  for (int j = 1; j < 10; j++)
    cout << i << " x " << j << " = " << i*j << endl;
```

2.18 if による条件分岐 (制御構造)

条件文： 条件が成立した時に実行する
if 文全体が，また 1 つの文を構成するとみなせる

if（式）文 1
→ 式が正しい場合に文 1 が実行される
if（式）文 1 else 文 2
→ 式が正しい場合に文 1 が実行され，正しくない場合に文 2 が実行される

例１： 返済の計算 (最後をピッタリゼロにして終了する) - If1.java

```
 1: public class If1 {
 2:   public static void main(String[] args) {
 3:     final double R = 0.0575;              // 5.75%
 4:     final double Y = 0.1;                 // 10.0%
 5:     int    i = 1;
 6:     double x = 1.0;
 7:     double z;
 8:     do {
 9:       x *= 1.0+R;
10:       if (x >= Y) z = Y; else z = x;
11:       x -= z;
12:       System.out.println("Term " + i + ": " + x);
13:       i++;
14:     } while (x > 0.0);
15:   }
16: }
```

例１'： while 文を使う場合 (例 1 の 8 行目と 14 行目を変更する)

```
 8:     while (x > 0.0) {
 :
14:     }
```

2.19 数値データと条件式

思いついたままにプログラムすると，うまく動かないことがある
たとえば，以下のようなことに注意しなくてはいけない
(1) 計算機内部の数値データ表現 (整数型/実数型)
(2) 繰返しの意味

例１： 実数の値の判定に等値演算 == を用いてはいけない - Float1.java
実数はあくまで近似値であり正確な値ではなく，float 型と double 型では値が異なってくる．

```
public class Float1 {
  public static void main(String[] args) {
    double x = 0.0;
    for (int i = 0; i < 10; i++)
      x += 0.1;
    if (x == 1.0)
      System.out.println("1.0 == 0.1 * 10 == " + x);
    else
      System.out.println("1.0 != 0.1 * 10 == " + x);
```

```
  }
}
```

例２： 例１を正しく実行する (微小な範囲内に収まるか否かを判定する) - Float2.java

```
public class Float2 {
  public static void main(String[] args) {
    final double EPS = 1.0E-14;
    double x = 0.0;
    for (int i = 0; i < 10; i++)
      x += 0.1;
    if ((x > (1.0-EPS)) && (x < (1.0+EPS)))
      System.out.println("1.0 == 0.1 * 10 == " + x);
    else
      System.out.println("1.0 != 0.1 * 10 == " + x);
  }
}
```

例３： 利率が 4〜7 %の間で変動した場合の返済所要期数 - DLoop1.java
4 %から始めて 0.01 %刻みで 7 %まで計算する ＝ 7.01 %になったら止める
2.16 節「do による繰返し」の例２のプログラムを参照せよ
実数型の数値は近似表現であり，正確な値ではない

```
public class DLoop1 {
  public static void main(String[] args) {
    final double Y = 0.1;                  // 10.0%
    double r = 0.04;                       // 4.00%
    while (r != 0.0701) {                  // 誤差のため 0.0701 にならない
      int i = 1;
      double x = 1.0;
      do {
        x *= 1.0+r;
        x -= Y;
        i++;
      } while (x > 0.0);
      System.out.println("Rate: " + r + ", Term: " + (i-1));
      r += 0.0001;
    }
  }
}
```

<実行結果>

```
ca10101$ java DLoop1 ¶
  :
```

```
Rate: 0.0998000000000017, Term: 66
Rate: 0.09990000000000171, Term: 73      (数値誤差が溜っている)
^C                                (暴走するので<Ctrl-C>で止める)
```

例 4： 利率が 4〜7 %の間で変動した場合の返済所要期数 - DLoop2.java
4 %から始めて 0.01 %刻みで 7 %まで 301 回計算する.
回数が定数であるから，for を使うことが望ましい.

```
public class DLoop2 {
  public static void main(String[] args) {
    final  double Y = 0.1;                  // 10.0%
    for (int n = 400; n < 701; n++) {
      int i = 1;
      double r = n/10000.0;
      double x = 1.0;
      do {
        x *= (1.0+r);
        x -= Y;
        i++;
      } while (x > 0.0);
      System.out.println("Rate: " + r + ", Term: " + (i-1));
    }
  }
}

<実行結果>
ca10101$ java DLoop2 ¶
  :
Rate: 0.0698, Term: 18
Rate: 0.0699, Term: 18
Rate: 0.07, Term: 18                    (数値誤差もない)
```

2.20 入力文 (キーボード入力と例外処理)

Java ではキーボードから入力された文字列を解析して内部データにする際の様々な問題をパッケージや例外処理を用いて解決している．しかし，パッケージや例外処理について正確に理解するためにはオブジェクト指向 (クラスとインスタンス) についての知識が必要であり，ここでは細かな説明を抜いて利用する.

入力文： キーボードからの入力文字列 (文字の並び) をデータとして解釈する
Java では入力を java.io.* パッケージを用いて処理する．概ね次のような形式になる.

```
import java.io.*;                       // java.io.* パッケージの import
```

```
try {                                   // 例外処理の宣言
  BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
                                        // 入力用 BufferedReader の生成
  int i = Integer.parseInt(br.readLine());
                                        // 整数 (int) 型データの入力
  double d = Double.parseDouble(br.readLine());
                                        // 実数 (double) 型データの入力
}
catch (Exception e) {                   // 例外発生時の処理
  System.out.println("Error: " + e);// 例外の表示
  e.printStackTrace();                  // StackTrace(メソッド呼出状況) の表示
}
```

パッケージ (package)： Java でのプログラムのまとまり

複数のクラスをひとまとめにしたものをパッケージと呼ぶ．特定のパッケージを用いる場合には，最初にパッケージを import する．

java.io.* パッケージを import することで，キーボードから文字列を獲得するための BufferedReader クラスやそのメソッド readLine が利用できるようになる．

例外処理 (exception)： プログラムが正常に実行されない場合の処理

例外クラスを用いることで処理を中断して適当なところに実行を移せる．入力文などで想定外の情報が入力された場合には，例外処理を行なう．

基本的には try 以降の { } の内側を上から順に実行するが，例外 (Exception) が発生した場合は，発生した例外ごとに catch してエラー表示などの例外処理を行なう．

例１： 2 つの数を計算する - Calculate1.java

実行時にキーボードからの入力を受け付け，double 型の値として計算する

```
import java.io.*;                       // java.io.* パッケージの import
public class Calculate1 {
  public static void main(String[] args) {
    try {                               // 例外処理の宣言
      BufferedReader br =
        new BufferedReader(new InputStreamReader(System.in));
                                        // 入力用 BufferedReader の生成
      System.out.print("a -> ");
      double a = Double.parseDouble(br.readLine());
                                        // 実数 (double) 型データの入力
      System.out.print("b -> ");
      double b = Double.parseDouble(br.readLine());
                                        // 実数 (double) 型データの入力
      System.out.println("" + a + " + " + b + " = " + (a+b));
      System.out.println("" + a + " - " + b + " = " + (a-b));
      System.out.println("" + a + " * " + b + " = " + (a*b));
```

```
      System.out.println("" + a + " / " + b + " = " + (a/b));
    }
    catch (Exception e) {                    // 例外発生時の処理
      System.out.println("Error: " + e);
      e.printStackTrace();
    }
  }
}
```

<実行結果>

```
ca10101$ java Calculate1 ¶
a -> 1 ¶                                    (キーボードから入力)
b -> 2 ¶                                    (キーボードから入力)
1.0 + 2.0 = 3.0
1.0 - 2.0 = -1.0
1.0 * 2.0 = 2.0
1.0 / 2.0 = 0.5
```

この実行例は問題なく動作しているが，課題 (レポート) では色々な例外を発生させよ．

2.21 ファイル入出力

ファイルへの入出力はストリーム (データの通り道) を通して行なう．プログラムはストリームを介して，ファイルからデータを読み込んだり，ファイルに書き込んだりできる．文法の詳細を理解するのは，オブジェクト指向 (クラスとインスタンス) を勉強した後になる．

ストリーム： ファイルや装置などとデータを受け渡すためのデータの通り道
プログラムはストリームを介してデータを入出力する．

文字ストリーム： 文字データを読み書きするためのストリーム (Reader，Writer など)
文字データの読み書き専用のストリームで，授業のファイル入出力にはこれを利用する．

バイトストリーム： 文字以外のデータ (バイナリデータ) を扱うためのストリーム
1 バイト単位でデータの読み出しや書き込みを行なう (標準入出力は本来バイトストリーム)．

標準入力： 標準的な入力元のストリーム (InputStream System.in)
通常は tty コマンドで返されるデバイスファイルが結びつけられており，キーボードからの入力に相当する

標準出力： 標準的な出力先のプリント機能付きストリーム (PrintStream System.out)
通常は tty コマンドで返されるデバイスファイルが結びつけられており，ディスプレイへの出力に相当する

標準エラー出力： エラー出力先のプリント機能付きストリーム (PrintStream System.err)
通常は tty コマンドで返されるデバイスファイルが結びつけられており，ディスプレイへの出力に相当する

ファイル終端記号： ファイルの終端を表す文字コード
UNIX では <ctrl-D>，Windows では <ctrl-Z>が用いられる

`BufferedReader br = new BufferedReader(new InputStreamReader(System.in))`
標準入力からのバッファ付き文字ストリーム (br) を作る

`BufferedReader br = new BufferedReader(new FileReader(str))`
ファイル (str - ファイル名の文字列) からのバッファ付き文字ストリーム (br) を作る

`String str = br.readLine()`
バッファ付き文字ストリーム (br) から 1 行分の文字列を読み込む
戻し値： 読み出した文字列，ファイル終端の場合は null)

`int i = br.read()`
バッファ付き文字ストリーム (br) から 1 文字 (2 バイト) を読み込む
戻し値： 読み出した文字 (2 バイト)，ファイル終端の場合は-1)

`br.close()`
バッファ付き文字ストリーム (br) を閉鎖する (以後 br は使えなくなる)

`System.out.print(str)`
標準出力 (プリント機能付きバイトストリーム) へ文字列 str を書き出す

`System.out.println(str)`
標準出力 (プリント機能付きバイトストリーム) へ文字列を書き出して改行する

`System.out.write(i)`
標準出力 (プリント機能付きバイトストリーム) へ 1 文字 (2 バイト: i - int 型) を書き出す

`PrintWriter pw = new PrintWriter(new FileWriter(str))`
ファイル (str - ファイル名の文字列) へのプリント機能付き文字ストリーム (pw) を作る

`pw.print(str)`
プリント機能付き文字ストリーム (pw) へ文字列 str を書き出す

`pw.println(str)`
プリント機能付き文字ストリーム (pw) へ文字列 str を書き出して改行する

`pw.write(i)`
プリント機能付き文字ストリーム (pw) へ 1 文字 (2 バイト: i - int 型) を書き出す

`pw.close()`
プリント機能付き文字ストリーム (pw) を閉鎖する (以後 pw は使えなくなる)

出力リダイレクション ('>' '>>' '>&')：出力先を他のファイルに切替えること

ca10101$ コマンド > ファイル
: ファイルを新たに作成して (ファイルが既に存在する場合、もとのファイルは消去される) 標準出力を蓄える

ca10101$ コマンド >> ファイル
: 既存のファイル (無いときは新たに作成される) の最後に標準出力を蓄える

ca10101$ コマンド >& ファイル
: ファイルを新たに作成して (従来あったファイルは消去される) 標準エラーを蓄える

入力リダイレクション ('<')：入力元を標準入力から他のファイルに切替えること

ca10101$ コマンド < ファイル
: コマンドにファイルの内容を入力する

例 1：標準入出力を介したデータの読み込みと書き出し - Copy1.java
標準入力から 1 行文の文字列データを読み込んで，標準出力に書き出す.

```
import java.io.*;                          // java.io.* パッケージの import
public class Copy1 {
  public static void main(String[] args) {
    try {                                  // 例外処理の宣言
      BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
                                           // 標準入力 BufferedReader 生成
      String str;
      while ((str = br.readLine()) != null) // 標準入力から 1 行読み込み
        System.out.println(str);           // 標準出力へ 1 行書き出し
      br.close();                          // 標準入力 BufferedReader 閉鎖
    }
    catch (Exception e) {                  // 例外発生時の処理
      System.out.println("Error: " + e);
      e.printStackTrace();
    }
  }
}
```

そのまま実行すると，キーボードから読み込んでディスプレイに表示する．入出力リダイレクションを用いると，データをファイルに書き出したり，データをファイルから読み込んだりできる．ただし，ファイル入出力とキーボード入力やディスプレイ表示は両立しない (独立並行に利用できない).

```
ca10101$ java Copy1 ¶
Yamaguchi Yasushi ¶                    (キーボードからの入力)
Yamaguchi Yasushi                      (同じものが画面に表示)
Computer Programming 1 ¶               (キーボードからの入力)
```

```
Computer Programming 1                    (同じものが画面に表示)
<ctrl-D>                                  (<ctrl-D>を入力すると終了)
ca10101$ java Copy1 > gomi ¶
Yamaguchi Yasushi ¶                       (キーボードからの入力)
Computer Programming 1 ¶                  (キーボードからの入力)
<ctrl-D>                                  (<ctrl-D>を入力すると終了)
ca10101$ cat gomi ¶
Yamaguchi Yasushi
Computer Programming 1
ca10101$ java Copy1 < gomi ¶
Yamaguchi Yasushi
Computer Programming 1
ca10101$ java Copy1 < gomi > junk ¶
ca10101$ cat junk ¶
Yamaguchi Yasushi
Computer Programming 1
```

例２： 標準入出力を介したデータの読み込みと書き出し - Copy2.java
機能は例１と同じだが，入出力の単位が１行ではなく１文字になっている．

```
import java.io.*;                         // java.io.* パッケージの import
public class Copy2 {
  public static void main(String[] args) {
    try {                                 // 例外処理の宣言
      BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
                                          // 標準入力 BufferedReader 生成
      int i;
      while ((i = br.read()) != -1)       // 標準入力から１文字読み込み
        System.out.write(i);              // 標準出力へ１文字書き出し
      br.close();                         // 標準入力 BufferedReader 閉鎖
    }
    catch (Exception e) {                 // 例外発生時の処理
      System.out.println("Error: " + e);
      e.printStackTrace();
    }
  }
}
```

例３： ファイルからの読み込みとファイルへの書き出し - Copy3.java
コマンド引数で指定したファイル (args[0]) からデータを読み込み，行番号を付けて
コマンド引数で指定したファイル (args[1]) へデータを書き出す
最後に全体の行数を画面に出力する

```
import java.io.*;                         // java.io.* パッケージの import
```

```
public class Copy3 {
  public static void main(String[] args) {
    try {                                   // 例外処理の宣言
      BufferedReader br = new BufferedReader(new FileReader(args[0]));
                                  // ファイル args[0] の BufferedReader 生成
      PrintWriter pw = new PrintWriter(new FileWriter(args[1]));
                                  // ファイル args[1] への PrintWriter 生成
      int n = 0;
      String str;
      while ((str = br.readLine()) != null)   // ファイルから 1 行読み込み
        pw.println("" + (++n) + ":\t" + str); // ファイルへ出力 (行番号付)
      br.close();                           // ファイル BufferedReader 閉鎖
      pw.close();                           // ファイルへの PrintWriter 閉鎖
      System.out.println("Total: " + n + " lines");
    }
    catch (Exception e) {                   // 例外発生時の処理
      System.out.println("Error: " + e);
      e.printStackTrace();
    }
  }
}
```

2.22 メソッド

メソッド： 一定の手続きをまとめたもの

複数のデータ (引数) を受取って，何らかの作業を行ない，値を 1 つ戻す
→同じことを何回も書かずにすむ & 複雑な手続きをメソッドに任せられる

メソッド定義： メソッドの実行手続き (作業) を定義する

メソッド呼出： 実際にメソッドにデータを渡して実行する

引数： メソッドに渡すデータ

仮引数： メソッド定義での引数 (変数)

実引数： 実際にメソッド実行に際して渡されるデータ

戻し値： メソッドの戻すデータ (0 ないし 1 個のみ)

複数個の値を戻したい場合には配列やオブジェクトを用いる

メソッド定義： メソッドの実行手続きを定める

static	型名	メソッド名	(引数並び)	{ メソッド本体 }
(1)	(2)	(3)	(4)	(5)

(1) 型修飾子　　本節でのメソッドはクラス (static) メソッドである
(2) 型名　　メソッドの戻す値の型 (戻し値が 0 個の時は void)
(3) メソッド名　　メソッドの名前 (メソッド呼出に用いられる)
(4) 引数並び　　メソッドに与える値の型と仮引数の組
(3) メソッド本体　メソッドの実行内容 (文の並び)

メソッド定義における引数並びでは，各引数がメソッド内の局所的な変数として利用される．型名と変数名が組となって並べられ，呼出にあたっては実引数の値に初期化されて実行される．

return 文： メソッドの実行を終了し，戻す値を指定する

```
return 式 ;
```

メソッド呼出 (実行)： 値を与えてメソッドを実行する

メソッド名（実引数並び);

クラス変数： メソッドの外で定義された変数

クラス内のすべてのメソッドから共通に利用できる．
型修飾子として static がつき，Java では static field と呼ばれる．詳細はオブジェクト指向 (クラスとインスタンス) で説明する．

局所変数： メソッドの内で定義された変数

当該のメソッドの中 (仮引数と同じ範囲) でしか利用できない．

例 1： 地球上の塵の体積の計算 - Method1.java

```
public class Method1 {
  static double d;                          // クラス変数 d（塵の厚さ）
  public static void main(String[] args) {
    d = Double.parseDouble(args[0]);
    d = d/1000.0;
    printv();                               // メソッド呼出 引数 0
  }
  static void printv() {                    // メソッド定義 引数 0 戻し値 0
    final double PI = 3.14159265358979;     // 局所 (定数) 変数 PI（円周率）
    final double R = 6371000.0;             // 局所 (定数) 変数 R（半径）
    System.out.println(4.0/3.0*PI*(R+d)*(R+d)*(R+d)-4.0/3.0*PI*R*R*R);
  }
}
```

例 2： 厚さを変えた塵の体積の計算 (1) - Method2.java

```
  public static void main(String[] args) {// main 以外は例 1 と同じ
    double dn = Double.parseDouble(args[0]);
                                            // 局所変数 dn（塵の厚さ mm 単位）
    dn = dn/1000.0;
```

```
    d = dn;  printv();                    // メソッド呼出 引数0
    d = 10.0*dn;  printv();               // メソッド呼出 引数0
    d = 100.0*dn;  printv();              // メソッド呼出 引数0
  }
```

例3： 厚さを変えた塵の体積の計算 (2) - Method3.java

```
public class Method3 {                    // クラス変数なし
  public static void main(String[] args) {
    final double R = 6371000.0;           // 局所(定数)変数 R (半径)
    double dn = Double.parseDouble(args[0]);
                                          // 局所変数 dn (塵の厚さmm単位)
    dn = dn/1000.0;
    printv(dn, R);                        // メソッド呼出 引数2 dn, R
    printv(10.0*dn, R);                   // メソッド呼出 引数2 10.0*dn, R
    printv(100.0*dn, R);                  // メソッド呼出 引数2 100.0*dn, R
  }
  static void printv(double d, double x) {// メソッド定義 引数2 戻し値0
    final double PI = 3.14159265358979;   // 局所(定数)変数 PI (円周率)
    System.out.println(4.0/3.0*PI*(x+d)*(x+d)*(x+d)-4.0/3.0*PI*x*x*x);
  }
}
```

例4： 厚さを変えた塵の体積の計算 (3: 戻し値の利用) - Method4.java

```
public class Method4 {                    // クラス変数なし
  public static void main(String[] args) {
    final double R = 6371000.0;           // 局所(定数)変数 R (半径)
    double dn = inputmm(args[0]);         // 局所変数 dn (塵の厚さm単位)
                                          // メソッド呼出 引数1 args[0]
    printv(dn, R);                        // メソッド呼出 引数2 dn, R
    printv(10.0*dn, R);                   // メソッド呼出 引数2 10.0*dn, R
    printv(100.0*dn, R);                  // メソッド呼出 引数2 100.0*dn, R
  }
  static void printv(double d, double x) {// メソッド定義 引数2 戻し値0
    System.out.println(vol(d, x));        // メソッド呼出の入れ子
  }
  static double inputmm(String str) {     // メソッド定義 引数1 double型
    double d = Double.parseDouble(str);
                                          // 局所変数 d (strを数値に変換)
    return d/1000.0;                      // 戻し値 (塵の厚さm単位)
  }
  static double vol(double d, double x) { // メソッド定義 引数2 double型
    final double PI = 3.14159265358979;   // 局所(定数)変数 PI (円周率)
```

```
    return 4.0/3.0*PI*(x+d)*(x+d)*(x+d)-4.0/3.0*PI*x*x*x;
  }                                          // 戻し値（塵の体積）
}
```

2.23 再帰呼出

再帰呼出 (recursion, recursive call)： メソッドが直接/間接に自らを呼出すこと
漸化式などの形式で与えられる計算の表現に適している
必ず再帰呼出の終了条件 (漸化式の初期条件) を伴う

自己再帰 (self recursion)： メソッドが直接に自らを呼出すこと

末尾再帰 (tail recursion)： メソッドが実行文の末尾で再帰呼出を行なうこと
反復計算に変換可能である (再帰呼出は遅いので可能なら反復計算にすべき)

例１： factorial(階乗) の計算 (1) - Factorial1.java
階乗は以下のように定義できる.

$$n > 0 \quad \text{のとき} \quad n! = n \times (n-1)!$$
$$n = 0 \quad \text{のとき} \quad n! = 1$$

これをそのまま Java のメソッドにすると，次のようになる (末尾再帰の形になっているので，反復計算で書ける).

```
public class Factorial1 {
  public static void main(String[] args) {
    int  n = Integer.parseInt(args[0]);
    System.out.println("Factorial: " + n + "! = " + factorial(n));
  }
  static int factorial(int n) {
    if (n > 0)
      return  n * factorial(n-1);
    else
      return  1;
  }
}
```

例２： factorial(階乗) の計算 (2) - Factorial2.java
例１のプログラムに 10, 13, 14, 15 などの数字を入力してみると，int 型の表現範囲が意外に狭いことが分かる．double 型に変えてみると…．

```
  // 殆ど例１と同じ，メソッド factorial の戻し値の型を変える
  static double factorial(int n) {
    if (n > 0)
      return  n * factorial(n-1);
```

```
    else
      return  1.0;
  }
```

例 3： 2 進数表示 (1) - Binary1.java
数を 2 で繰り返し割った際の剰余が 2 進数となる繰り返しに while を用いると，次のようなプログラムになる．

```
public class Binary1 {
  public static void main(String[] args) {
    int  i = Integer.parseInt(args[0]);
    printBinary(i);
    System.out.println();
  }
  static void printBinary(int i) {
    while (i > 0) {                        // 0 より大きければ
      System.out.print(i % 2);             // 当該桁を表示して
      i /= 2;                              // さらに上位の計算をする
    }
  }
}
```

例 4： 2 進数表示 (2) - Binary2.java
例 3 では表示しては計算を進めるために文字の順序が逆になる．計算を進めてから表示するという順序にするために，再帰呼出が利用できる．このプログラムは末尾再帰の形になっていないので，単純な反復計算では書けない．

```
  // メソッド printBinary 以外は例 3 に同じ
  static void printBinary(int i) {
    if (i > 1)                             // 1 より大きければ
      printBinary(i / 2);                  // 上位の桁を表示して
    System.out.print(i % 2);               // 当該桁を表示する
  }
```

2.24　配列

県と面積の対応表 (平成 10 年現在) – ある県 (i 番) の面積 (area km^2) が知りたい

県	(番)	0	1	2	3	4	5	6
面積	(km^2)	6093.78	6408.28	6363.16	3767.09	4995.72	2102.14	2415.11
県	(名)	Ibaraki	Tochigi	Gunma	Saitama	Chiba	Tokyo	Kanagawa

少なくとも 7 個のデータを格納できる場所を用意する必要がある
単に変数 area? を 7 個用意するのでは，ある i に対応する area? を直接には求められない

例１： i 番の県の面積を表示する - Array1.java

```
public class Array1 {
  public static void main(String[] args) {
    double area0 = 6093.78, area1 = 6408.28, area2 = 6363.16,
       area3 = 3767.09, area4 = 4995.72, area5 = 2102.14, area6 = 2415.11;
    int  i = Integer.parseInt(args[0]);
    if (i == 0) System.out.println("Area of #0: " + area0 + " km^2");
    else if (i == 1) System.out.println("Area of #1: " + area1 + " km^2");
    else if (i == 2) System.out.println("Area of #2: " + area2 + " km^2");
    else if (i == 3) System.out.println("Area of #3: " + area3 + " km^2");
    else if (i == 4) System.out.println("Area of #4: " + area4 + " km^2");
    else if (i == 5) System.out.println("Area of #5: " + area5 + " km^2");
    else if (i == 6) System.out.println("Area of #6: " + area6 + " km^2");
  }
}
```

配列 (array)： 表のように順序づけられたデータ (i 番目) を表現する

配列宣言： 配列 (指定された型の値を複数格納できる) 変数の宣言
通常は同時に配列領域を作成して配列変数に代入する

```
型名 []  変数名 ,  変数名 ...
型名  変数名 [] ,  変数名 [] ...
```

配列領域： 配列データの実体
複数の値を格納する場所

配列領域の作成： 配列データの実体の作成
配列変数宣言だけでは，複数の値を保持する (メモリ) 領域は確保されない．特定の型の値をサイズ個数分保持するための，配列領域の作成や配列の初期化が必要となる

```
型名 []  変数名 = new 型名 [サイズ] ;
型名  変数名 [] = new 型名 [サイズ] ;
```

配列の初期化： 配列宣言と同時に値を代入すること
初期値の個数が配列の大きさ (サイズ) になる

```
型名 [] 変数名  = { 初期値 1, 初期値 2, ... } ;
型名  変数名 [] = { 初期値 1, 初期値 2, ... } ;
```

配列の大きさ (サイズ)： 配列に格納可能なデータの個数
配列領域を作成する際に定まり，length フィールドによって調べられる

```
変数名.length
```

配列要素： 添字 (index) によって区別される配列の各変数

添字は整数型の定数または変数で，0 以上 (サイズ-1) 以下のみ有効，それ以外はエラーとなる

変数名 [添字]

例２： 配列を用いて i 番の県の面積を表示する - Array2.java

```
public class Array2 {
  public static void main(String[] args) {
    double[] area = new double[7];          // 配列宣言と配列領域の作成
    area[0] = 6093.78; area[1] = 6408.28; area[2] = 6363.16; area[3] = 3767.09;
    area[4] = 4995.72; area[5] = 2102.14; area[6] = 2415.11;
                                            // 配列要素への値の設定
    int  i = Integer.parseInt(args[0]);
    System.out.println("Area of #" + i + ": " + area[i] + " km^2");
  }
}
```

例３： 例 2 で配列の初期化を利用する - Array3.java
同時に文字列の配列 (String name[]) も利用する

```
public class Array3 {
  public static void main(String args[]) {
        // args も String 型の配列で，複数の文字列データを受取れる
    double area[] = {6093.78, 6408.28, 6363.16, 3767.09,
                     4995.72, 2102.14, 2415.11};
                     // double 型の配列宣言と初期化，[] が変数名の後
    String[] name = {"Ibaraki", "Tochigi", "Gunma", "Saitama",
                     "Chiba", "Tokyo", "Kanagawa"};
                     // String 型の配列宣言と初期化，[] が型名の後
    int  i = Integer.parseInt(args[0]);  // 文字列データを 1 つ渡す
    System.out.println("Area of " + name[i] + ": " + area[i] + " km^2");
  }
}
```

2.25 配列と繰返し

添字に変数を用いて，繰返しを効果的に利用する

例１： 関東地方全体に対する i 番の県の面積の割合 - Iteration1.java
最初に total = 0.0 とし，これに全ての県の面積を加えて，総面積を得る

```
public class Iteration1 {
  public static void main(String[] args) {
```

```
    double[] area = {6093.78, 6408.28, 6363.16, 3767.09,
                     4995.72, 2102.14, 2415.11};
    double total = 0.0;                    // double型変数 total の初期化
    for (int j = 0; j < area.length; j++) // 反復による面積の総和の計算
      total += area[j];
    int  i = Integer.parseInt(args[0]);
    System.out.println("Ratio of #" + i + " = " + (area[i]/total));
  }
}
```

例２： i 番の県の順位を調べる - Iteration2.java
最初に rank = 1 位と仮定して，より大きな県があれば順位を一つずつ下げる

```
public class Iteration2 {
  public static void main(String[] args) {
    double[] area = {6093.78, 6408.28, 6363.16, 3767.09,
                     4995.72, 2102.14, 2415.11};
    int  i = Integer.parseInt(args[0]);
    int rank = 1;                          // 仮の順位として 1 位
    for (int j = 0; j < area.length; j++) // 反復によって全てと比較
      if (area[j] > area[i])  rank++;      // 相手が大きければ順位を下げる
    System.out.println("Rank of #" + i + " = " + rank);
  }
}
```

例３： 各県の面積の割合と順位を調べる - Iteration3.java
二重ループ (for 文の入れ子) を用いて，各県の面積の割合と順位を調べる

```
public class Iteration3 {
  public static void main(String[] args) {
    double[] area = {6093.78, 6408.28, 6363.16, 3767.09,
                     4995.72, 2102.14, 2415.11};
    double total = 0.0;                      // 総面積の計算
    for (int i = 0; i < area.length; i++)
      total += area[i];
    for (int i = 0; i < area.length; i++) { // 全ての県について反復計算
      int rank = 1;                          // 順位の計算
      for (int j = 0; j < area.length; j++)
        if (area[j] > area[i])  rank++;
      System.out.println("Rank of #" + i + " = " + rank +
                         ", \t Ratio = " + (area[i]/total));
    }
  }
}
```

例4： ファイルからの読込みデータに例3のプログラムを実行する - Iteration4.java
ファイルの先頭行は県数, それ以降に1行ずつ面積, が書いてある (CFIVEから KantoArea.dat と JapanArea.dat をコピーせよ)

```
import java.io.*;                          // java.io.* パッケージの import
public class Iteration4 {
  public static void main(String[] args) {
    try {                                  // 例外処理の宣言
      BufferedReader br = new BufferedReader(new FileReader(args[0]));
                                 // ファイル args[0] の BufferedReader 生成
      int n = Integer.parseInt(br.readLine());
                                           // 県数の読み込み
      double[] area = new double[n];       // 配列宣言と配列領域作成
      double total = 0.0;
      for (int i = 0; i < n; i++) {        // 面積の読み込みと総面積の計算
        area[i] = Double.parseDouble(br.readLine());
        total += area[i];
      }
      br.close();                          // ファイル BufferedReader 閉鎖
      for (int i = 0; i < n; i++) {        // 全ての県について反復計算
        int rank = 1;                      // 順位の計算
        for (int j = 0; j < n; j++)
          if (area[j] > area[i])  rank++;
        System.out.println("Rank of #" + i + " = " + rank +
                           ", \t Ratio = " + (area[i]/total));
      }
    }
    catch (Exception e) {                  // 例外発生時の処理
      System.out.println("Error: " + e);
      e.printStackTrace();
    }
  }
}
```

2.26 break による繰返し (および switch) からの復帰

復帰： 繰返し (while, do〜while, for) や switch から復帰する
繰返しループまたは switch を中断し, 1つ外に抜け出す ({} の後に実行が移る)

```
  break ;
```

2.27 continue による繰返しの中断

中断： 繰返し (while，do〜while，for) を中断して，次の繰返しステップに移る
while，do の場合　条件判定式に移る
for の場合　　式 3 に移る

```
continue ;
```

2.28 switch による条件判定

条件判定： 整数値による複数条件分岐
式の値 (整数) に一致する case 定数式 (整数): 以降の文をすべて実行する
一致するものがなければ，default: 以降の文をすべて実行する
switch から抜け出すには，break で明示しなくてはならない (break を用いないと以降の文がすべて実行される)

```
switch（式）{
case 定数式：　文の並び（普通は最後に break;）
case 定数式：　文の並び（普通は最後に break;）
  :
default ：　文の並び
}
```

例１： 2.24 節の例 1 のプログラムの書き換え - Switch1.java

```
public class Switch1 {
  public static void main(String[] args) {
    double area0 = 6093.78, area1 = 6408.28, area2 = 6363.16,
       area3 = 3767.09, area4 = 4995.72, area5 = 2102.14, area6 = 2415.11;
    int  i = Integer.parseInt(args[0]);
    switch (i) {                    // switch による分岐
    case 0:                         // i == 0 の場合(breakがあることに注意)
      System.out.println("Area of #0: " + area0 + " km^2");  break;
    case 1:                         // i == 1 の場合(breakがあることに注意)
      System.out.println("Area of #1: " + area1 + " km^2");  break;
    case 2:                         // i == 2 の場合(breakがあることに注意)
      System.out.println("Area of #2: " + area2 + " km^2");  break;
    case 3:                         // i == 3 の場合(breakがあることに注意)
      System.out.println("Area of #3: " + area3 + " km^2");  break;
    case 4:                         // i == 4 の場合(breakがあることに注意)
      System.out.println("Area of #4: " + area4 + " km^2");  break;
    case 5:                         // i == 5 の場合(breakがあることに注意)
      System.out.println("Area of #5: " + area5 + " km^2");  break;
    case 6:                         // i == 6 の場合(breakがあることに注意)
      System.out.println("Area of #6: " + area6 + " km^2");  break;
```

```
    default:                        // 上記以外の場合
      System.out.println("Out of range: i = " + i);
    }
  }
}
```

例2： 入力された整数の比較 - Switch2.java
コマンド引数として -1, 0, 1, 2, 3, 4, 5 などを与えて，break の有無による動作の違いを確認せよ

```
public class Switch2 {
  public static void main(String[] args) {
    int  i = Integer.parseInt(args[0]);
    switch (i) {                    // switch による分岐
    case 0:                         // i == 0 の場合(breakがないことに注意)
      System.out.println("" + i + " is less than one.");
    case 1:                         // i == 1 の場合(breakがないことに注意)
      System.out.println("" + i + " is less than two.");
    case 2:                         // i == 2 の場合(breakがないことに注意)
      System.out.println("" + i + " is less than three.");
    case 3:                         // i == 3 の場合(breakがあることに注意)
      System.out.println("" + i + " is less than four.");  break;
    case 4:                         // i == 4 の場合(breakがあることに注意)
      System.out.println("" + i + " is four.");  break;
    default:                        // 上記以外の場合
      System.out.println("" + i + " is negative or more than four.");
    }
  }
}
```

例3： ファイル内の数文字, 空白文字, 行数(改行文字), その他の文字を数える - Switch3.java
数文字の個数データは配列 (添字は 0 から 9) に蓄えられる．文字は 2byte のビット列 (文字コード) によって表現されており，'0' から'9' までの文字コードは連続して 1 つずつズレているので，'0' を引いてやれば適切な添字番号が得られる

```
import java.io.*;                           // java.io.* パッケージの import
public class Switch3 {
  public static void main(String[] args) {
    try {                                   // 例外処理の宣言
      BufferedReader br = new BufferedReader(new FileReader(args[0]));
                                  // ファイル args[0] の BufferedReader 生成
      int i;
      int nline = 0, nwhite = 0, nother = 0;
      int[] ndigit = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
```

```
                                          // 文字数初期化,ndigit は数文字
      while ((i = br.read()) != -1) {     // 標準入力から 1 文字読み込み
        switch (i) {                      // 文字による分岐
          case '0':  case '1':  case '2':  case '3':  case '4':
          case '5':  case '6':  case '7':  case '8':  case '9':
            ndigit[i-'0']++;  break;      // '0'-'9' の文字コードは連続
          case '\n':
            nline++;                      // 改行数
          case ' ':  case '\t':
            nwhite++;  break;             // 空白文字 (改行, 空白, タブ) 数
          default:
            nother++;  break;             // その他の文字, break は不要
        }
      }
      br.close();                         // ファイル BufferedReader 閉鎖
      for (i = 0; i < 10; i++)
        System.out.println("# of " + i + "\'s: " + ndigit[i]);
      System.out.println("# of lines: " + nline);
      System.out.println("# of whites: " + nwhite);
      System.out.println("# of other characters: " + nother);
    }
    catch (Exception e) {                 // 例外発生時の処理
      System.out.println("Error: " + e);
      e.printStackTrace();
    }
  }
}
```

2.29 クラスとインスタンス (1) - 複合データ型

Java はオブジェクト指向言語として有名である.
Java のオブジェクトには, 様々な特徴があるが, ここではオブジェクトの最も基本的な側面を学習する.

オブジェクト (object)： 一連の情報 (データとプログラム) をひとまとめにした「もの」
Java プログラミングの観点から見ると, 処理対象の情報であるデータは変数に対応し, 処理手順の情報であるプログラムはメソッドに対応する. すなわち, オブジェクトは変数群とメソッド群を持つ. オブジェクトには, 型に相当する「クラス」と実体 (定数や変数) に相当する「インスタンス」がある.

クラス (class)： 複数のデータとメソッドをひとまとめにしたオブジェクトの型
「人間 (一般)」,「教官 (一般)」,「犬 (一般)」に対応する概念

インスタンス (instance)： あるクラスに属するオブジェクトの実体 (個々のデータ)
担当教官の「山口 泰」や隣の犬の「コロ」に対応する概念

クラス宣言： クラス名/クラスの内容 (変数およびメソッド) を宣言する
クラス宣言されたクラス名は型名となる

```
クラス修飾子　class　クラス名 {
  変数 (フィールド) 宣言並び
  コンストラクタ宣言並び
  メソッド宣言並び
}
```

クラス修飾子： クラスの性質を指定する

`public`	他のパッケージ (ファイル) から参照できる
`abstract`	インスタンス化できない (「継承」で後述)
`final`	拡張できない (「継承」で後述)

変数宣言： オブジェクトの持つ変数を宣言する

```
アクセス修飾子　変数修飾子　型名　変数名，変数名;
```

アクセス修飾子： 変数やメソッドのアクセスに関する性質を指定する

`public`	他の全てのクラスから参照できる
`protected`	サブクラスないし同一パッケージから参照できる
(省略)	同一パッケージから参照できる
`private`	他のクラスからは参照できない

変数修飾子： 変数の性質を指定する

`static`	クラス変数 (`static` のないものは，インスタンス変数)
`final`	定数変数

クラス変数： クラスに属す全メソッドによって共有されるユニークな変数
クラス宣言中で `static` 修飾子のついた変数はクラス変数となり，当該クラスの全てのメソッドおよび全てのインスタンスによって共有される
クラス変数は，次のようにしてアクセスする

```
クラス名. 変数名
インスタンス. 変数名
```

メソッドやコンストラクタ内では，変数名のみでクラス変数に直接アクセスできる (2.22 節「メソッド」で説明したクラス変数は，この用法であった)

```
変数名
```

インスタンス変数： 各インスタンスが (複数個) 持つ固有の変数
クラス宣言中で `static` 修飾子がついていない変数はインスタンス変数となる
各インスタンス (オブジェクトの実体) は，複数のデータから構成されているが，この複数のデータはインスタンス変数に保持される．言い換えると，各インスタンス固有

の性質 (属性) を保持する変数と考えてもよい．たとえば，特定の人の「名前」「身長」「体重」などが相当する
インスタンス変数は，次のようにしてアクセスする (「山口泰. 名前」とか「山田太郎. 身長」のような記法)

インスタンス . 変数名

インスタンスメソッドやコンストラクタ内では，変数名のみでインスタンス変数に直接アクセスできるが，this によって明示することもできる．
this はインスタンスが自分自身を示す予約語で，そのためインスタンスメソッドないしコンストラクタ内でのみ利用される§

変数名
this. 変数名

メソッド宣言： オブジェクトの持つメソッドを宣言する

アクセス修飾子 メソッド修飾子 型名 メソッド名（仮引数並び）{メソッド本体}

メソッド修飾子： メソッドの性質を指定する

static	クラスメソッド（static のないものは，インスタンスメソッド）
final	オーバライドできないメソッド (「継承」で後述)
abstract	当該クラスでは実装せず，サブクラスで実装されるメソッド (「継承」で後述)
synchronized	実行時にインスタンスをロックするメソッド (「マルチスレッド」で後述)

クラスメソッド： クラス内で共有されるメソッド (2.22 節で学んだメソッド)
クラス宣言中で static 修飾子のついたメソッドはクラスメソッドとなり，当該クラスの全てのメソッドおよび全てのインスタンスから呼び出され，ユニークな作業を実施して，値を (高々1 つ) 戻す
クラスメソッドは，次のようにクラス名とメソッド名，またはインスタンスとメソッド名，の組合せによって呼び出される

クラス名 . メソッド名 (実引数並び)
インスタンス . メソッド名 (実引数並び)

同一クラスのメソッドやコンストラクタ内では，メソッド名のみでクラスメソッドに直接アクセスできる (2.22 節で説明された「メソッド」は，クラスメソッドであった)

メソッド名 (実引数並び)

インスタンスメソッド： 各インスタンス固有のメソッド
クラス宣言中で static 修飾子がついていないメソッドはインスタンスメソッドとなり，当該クラスのインスタンスを介して呼び出され，そのインスタンス固有の作業を実施して，値を (高々1 つ) 戻す
インスタンスメソッドは，次のようにインスタンスとメソッド名の組合せによって呼び出される

§インスタンスメソッドとコンストラクタについては，もう少し後に説明がある．

インスタンス . メソッド名 (実引数並び)

インスタンスメソッドやコンストラクタ内では，メソッド名のみでインスタンスメソッドを直接呼び出せるが，this によって明示することもできる

メソッド名 (実引数並び)
this . メソッド名 (実引数並び)

コンストラクタ (constructor)： インスタンス化の際に呼ばれるメソッド
クラスと同じ名前を持ち，インスタンス変数などの初期化に利用される．インスタンスメソッドの一種と考えられる

コンストラクタ宣言： オブジェクトの持つコンストラクタを宣言する

アクセス修飾子　クラス名　(仮引数並び) {コンストラクタ本体}

インスタンス化 (instantiation)： インスタンスの生成 (オブジェクトの実体化)
あるクラスに属するインスタンスを実際に 1 つ生成 (実体化) すること．生成されたインスタンスは，当該の型 (クラス) の変数に代入されたり，オペランド (演算の対象) や他のメソッドの引数として利用されたりする

new クラス名　(実引数並び)

例 1： 県のデータ - Prefecture1.java

```
public class Prefecture1 {
  String name;                              // インスタンス変数宣言 name
  double area;                              // インスタンス変数宣言 area
  int    male;                              // インスタンス変数宣言 male
  int    female;                            // インスタンス変数宣言 female
  int population() {                        // インスタンスメソッド宣言
    return  male + female;
  }
  double popDensity() {                     // インスタンスメソッド宣言
    return  population() / area;
  }

  public static void main(String[] args) {
    Prefecture1 p = new Prefecture1();      // インスタンス化 & 代入
    p.name = "Kanagawa";                    // インスタンス変数への代入
    p.area = 2415.11;                       // インスタンス変数への代入
    p.male = 4219000;                       // インスタンス変数への代入
    p.female = 4079000;                     // インスタンス変数への代入
    System.out.println("Area of " + p.name + " = " + p.area + " [km^2]");
                                            // インスタンス変数へのアクセス
    System.out.println("Pop. of " + p.name + " = " +
```

```
                        p.population() + " [persons]");
                                                    // インスタンスメソッドの呼出
    System.out.println("Pop. Density of " + p.name + " = " +
                        p.popDensity() + " [persons/km^2]");
  }
}
```

参照データ型変数： 配列やオブジェクトの変数
参照データ型変数の場合，変数を宣言しただけでは実体は存在しないので，配列領域やインスタンスなどの実体を生成する必要がある¶

多重定義，オーバロード (overload)： 同じ名前のメソッド/コンストラクタを複数作ること
メソッド名だけでなく引数の型の並びの違いでメソッド/コンストラクタを区別する

シグニチャ(signature)： メソッド名と引数の型の並び
メソッドはシグニチャで区別される

デフォルトコンストラクタ (default constructor)： 標準で作られるコンストラクタ
コンストラクタ宣言が省略されると，引数を持たないコンストラクタが自動的に 1 つ作られる (各インスタンス変数の初期値は `null` または 0 となる) ‖
コンストラクタ宣言があるとデフォルトコンストラクタは作られない

例２： 県のデータ (オブジェクトの配列とコンストラクタのオーバロード) - `Prefecture2.java`
配列データと県データを別々に実体化していることや，オーバロードされたコンストラクタの引数の違いなどに注目せよ

```
public class Prefecture2 {
  String name;
  double area;
  int    male;
  int    female;
  Prefecture2() {                          // 引数なしコンストラクタ
  }                                        // デフォルトの代用
  Prefecture2(String n, double a) {        // 2引数のコンストラクタ
    name = n;  area = a;
  }
  Prefecture2(String name, double area, int male, int female) {
    // 4引数のコンストラクタ，引数とインスタンス変数の区別に注意せよ
    this.name = name;  this.area = area;
    this.male = male;  this.female = female;
  }
  int population() {
    return male + female;
```

¶詳細は 2.31 節「基本データ型と参照データ型」を参照せよ.
‖`null` については 2.31 節「基本データ型と参照データ型」を参照せよ.

```
  }
  double popDensity() {
    return population() / area;
  }
  String toStr() {
    return name + ":\n  Area  = " + area + " [km^2]\n  Pop.  = " +
           population() + " (" + male + "(m)/" + female + "(f)) [persons]" +
           "\n  Dens. = " + popDensity() + " [persons/km^2]";
  }
  public static void main(String[] args) {
    Prefecture2[] p = new Prefecture2[3]; // 配列領域の生成
    p[0] = new Prefecture2("Chiba", 4995.72, 2937000, 2895000);
                        // インスタンス化，4 引数コンストラクタの利用
    p[1] = new Prefecture2("Tokyo", 2102.14);
                        // インスタンス化，2 引数コンストラクタの利用
    p[1].male = 5795000;
    p[1].female = 5844000;
    p[2] = new Prefecture2();
                        // インスタンス化，引数なしコンストラクタの利用
    p[2].name = "Kanagawa";
    p[2].area = 2415.11;
    p[2].male = 4219000;
    p[2].female = 4079000;
    for (int i = 0; i < p.length; i++)
      System.out.println(p[i].toStr());
  }
}
```

2.30 クラスとインスタンス (2) - 情報隠蔽

オブジェクト指向言語では，情報隠蔽は重要な機能の 1 つである．

アクセス (access)： データを読み書きすること

情報隠蔽 (encapusulation)： オブジェクトへのアクセスを制限し，利用方法を定めること
一般にインスタンス変数やクラス変数を,「インスタンス. 変数名」や「クラス名. 変数名」という形式で読み出したり書き換えたりせずに，メソッドを介してのみデータを読み書きする．
このようにすることによって，外部からインスタンス変数やクラス変数を隠すことが可能となる

`public` **クラス：** 他のファイルやパッケージから利用可能なクラス
`public` クラスは 1 ファイルに 2 つ以上は作れない．

また public クラスのクラス名とファイル名は一致しなくてはならない.
(これまで書いてきたプログラムは，すべて 1 つの public クラスのみから構成されていた)

アクセス修飾子： 変数，メソッドなどのアクセスに関する性質を指定する

public	他の全てのクラスから参照できる
protected	サブクラスないし同一パッケージから参照できる
(省略)	同一パッケージから参照できる
private	他のクラスからは参照できない

クラスライブラリ (class library)： Java の実行プログラム (=クラスファイル) の集まり
コンパイル結果は，クラスごとに (1 つ以上の) クラスファイルになる
(コンパイル時や実行時に) プログラム中で新たなクラス (型) が必要になると，当該のクラス名のクラスファイルを自動的に読み込む (そのためにクラス名とファイル名を一致させ，public クラスとする)

クラスパス (class path)： クラスライブラリを探す場所 (ディレクトリ)

パッケージ (package)： クラスライブラリの単位
複数のクラスファイルをひとまとめにしてパッケージとし，特定のディレクトリ (パッケージ名とディレクトリ名は同じにする) に配置する

例１： 県のデータのクラスライブラリ - Prefecture3.java
順位や割合を計算するメソッド print については，2.25 節の「配列と繰返し」の例３などを復習せよ
このプログラムだけでは走らない (例２の PDemo3.java から利用される)

```
public class Prefecture3 {
  private String name;
  private double area;
  private int    male;
  private int    female;
  private static double totalArea = 0.0;  // クラス変数 (総面積)
  private static double totalPop = 0.0;   // クラス変数 (総人口)
  public Prefecture3(String name, double area, int male, int female) {
    this.name = name;  this.area = area;
    this.male = male;  this.female = female;
  }
  public String name() {                  // name の読出し
    return name;
  }
  public void name(String str) {          // name の書換え (オーバロード)
    name = str;
  }
  public double area() {                  // area の読出し
```

```
    return area;
  }
  public void area(double x) {            // areaの書換え(オーバロード)
    area = x;
  }
  public int population() {
    return male + female;
  }
  public double popDensity() {
    return population() / area;
  }
  public String toString() {
    return name + ":\n  Area  = " + area + " [km^2]\n  Pop.  = " +
           population() + " (" + male + "(m)/" + female + "(f)) [persons]" +
           "\n  Dens. = " + popDensity() + " [persons/km^2]";
  }
  public static void calculateTotal(Prefecture3[] p) {
    totalArea = 0.0;                      // totalAreaとtotalPopの計算
    totalPop  = 0.0;
    for (int i = 0; i < p.length; i++) {
      totalArea += p[i].area;
      totalPop  += p[i].population();
    }
  }
  public void print(Prefecture3[] p) {    // 順位や割合の計算
    System.out.println(this.toString());
    int rank = 1;
    for (int i = 0; i < p.length; i++)
      if (p[i].area > this.area)  rank++;
    System.out.println("  Area Rank = " + rank + ",\t Area Ratio = " +
                       (this.area / totalArea));
    rank = 1;
    for (int i = 0; i < p.length; i++)
      if (p[i].population() > this.population())  rank++;
    System.out.println("  Pop. Rank = " + rank + ",\t Pop. Ratio = " +
                       (this.population() / totalPop));
    rank = 1;
    for (int i = 0; i < p.length; i++)
      if (p[i].popDensity() > this.popDensity())  rank++;
    System.out.println("  Pop. Density Rank = " + rank);
  }
}
```

例２： 例１のクラスライブラリの利用 - PDemo3.java

コンパイル時には，例１の Java ファイル Prefecture3.java，ないしは，クラスファイル Prefecture3.class が必要で，Prefecture3.java の場合には自動的にコンパイルされる．実行時には Prefecture3.class が必要となる．コメントアウトしている部分のコメントを外して，コンパイルするとどうなるだろうか？

```
public class PDemo3 {
  public static void main(String[] args) {
    Prefecture3[] p = new Prefecture3[3]; // 配列領域の生成
    p[0] = new Prefecture3("Chiba", 4995.72, 2937000, 2895000);
    p[1] = new Prefecture3("Tokyo", 2102.14, 5795000, 5844000);
    p[2] = new Prefecture3("Kanagawa", 2415.11, 4219000, 4079000);
                                          // インスタンス化 (3 回)
    Prefecture3.calculateTotal(p);        // 総面積と総人口の計算
    for (int i = 0; i < p.length; i++)    // 県毎の順位や割合の計算
      p[i].print(p);
    System.out.println("Area of #" + 0 + " is " + p[0].area());
                                          // area の読出し
    // System.out.println("Area of #" + 0 + " is " + p[0].area);
    p[0].area(0.0);                       // area の書換え
    // p[0].area = 0.0;
    System.out.println("Area of #" + 0 + " is " + p[0].area());
    System.out.println("Name of #" + 2 + " is " + p[2].name());
                                          // name の読出し
    // System.out.println("Name of #" + 2 + " is " + p[2].name);
    p[2].name("Yokohama");                // name の書換え
    // p[2].name = "Yokohama";
    System.out.println("Name of #" + 2 + " is " + p[2].name());
    Prefecture3.calculateTotal(p);        // 総面積と総人口の計算
    for (int i = 0; i < p.length; i++)    // 県毎の順位や割合の計算
      p[i].print(p);
  }
}
```

例３： 県データのクラスライブラリ (完成版) - Prefecture.java

このプログラムは，今後，何回も利用するの注意すること
このプログラムだけでは走らない (たとえば例４の PDemo4.java から利用される)

```
import java.io.*;
import java.util.*;
public class Prefecture {
  protected String name;                  // protected(継承用)
  protected double area;                  // protected(継承用)
  protected int    male;                  // protected(継承用)
```

```
protected int    female;                // protected(継承用)
private static double totalArea = 0.0;  // クラス変数(総面積)
private static double totalPop = 0.0;   // クラス変数(総人口)
public Prefecture(String name, double area, int male, int female) {
  this.name = name;  this.area = area;
  this.male = male;  this.female = female;
}
public String name() {                  // nameの読出し(オーバロードなし)
  return name;
}
public double area() {                  // areaの読出し(オーバロードなし)
  return area;
}
public int population() {
  return male + female;
}
public double popDensity() {
  return population() / area;
}
public String toString() {
  return name + ":\n  Area  = " + area + " [km^2]\n  Pop.  = " +
         population() + " (" + male + "(m)/" + female + "(f)) [persons]" +
         "\n  Dens. = " + popDensity() + " [persons/km^2]";
}
public void print(Prefecture[] p) {    // 順位や割合の計算
  System.out.println(this.toString());
  int rank = 1;
  for (int i = 0; i < p.length; i++)
    if (p[i].area > this.area)  rank++;
  System.out.println("  Area Rank = " + rank + ",\t Area Ratio = " +
                     (this.area / totalArea));
  rank = 1;
  for (int i = 0; i < p.length; i++)
    if (p[i].population() > this.population())  rank++;
  System.out.println("  Pop. Rank = " + rank + ",\t Pop. Ratio = " +
                     (this.population() / totalPop));
  rank = 1;
  for (int i = 0; i < p.length; i++)
    if (p[i].popDensity() > this.popDensity())  rank++;
  System.out.println("  Pop. Density Rank = " + rank);
}
public static Prefecture[] loadFile(String file) {//ファイルから読込み
```

```
    Prefecture[] p = null;
    try {
      BufferedReader br = new BufferedReader(new FileReader(file));
      int n = Integer.parseInt(br.readLine());
      p = new Prefecture[n];
      totalArea = 0.0;
      totalPop  = 0.0;
      for (int i = 0; i < n; i++) {
        StringTokenizer st = new StringTokenizer(br.readLine());
        String name = st.nextToken();
        double area = Double.parseDouble(st.nextToken());
        int male   = Integer.parseInt(st.nextToken());
        int female = Integer.parseInt(st.nextToken());
        p[i] = new Prefecture(name, area, male, female);
        totalArea += area;
        totalPop  += male + female;
      }
      br.close();
    }
    catch (Exception e) {
      System.out.println("Error: " + e);
      e.printStackTrace();
    }
    return p;
  }
}
```

例４： 例３のクラスライブラリの利用 - PDemo.java
コンパイルと実行には例３のクラスファイル Prefecture.class が必要
データファイルの先頭行は県数，それ以降に１行ずつ県名，面積，男性数，女性数が書いてある (CFIVE から Kanto.dat と Japan.dat をコピーせよ)

```
public class PDemo {
  public static void main(String[] args) {
    Prefecture[] p = Prefecture.loadFile(args[0]);
    for (int i = 0; i < p.length; i++)
      p[i].print(p);
  }
}
```

2.31　基本データ型と参照データ型

基本データ型 (primitive data type)： Java 仮想機械の最も基本とするデータ型

整数型 (byte，short，int，long)，実数型 (float，double)，真理値型 (boolean)，文字型 (char) の 8 種類のみが基本データ型であり，これ以外のデータは，すべて参照データ型 (オブジェクト) となる

基本データ型変数： 基本データ型の変数

変数自体がデータを格納する

参照データ型 (reference data type)： Java のオブジェクトデータの型

基本データ型以外のデータで，フィールドとメソッドを持つ Java のオブジェクトデータ

参照データ型変数： 参照データ型の変数

参照データ型の場合，変数はデータの実体を格納せずに，メモリ上の別の場所にあるデータの実体を参照する．つまり，データの実体は別の場所に存在するはずであり，新しいデータを作る場合には new 演算子を用いて，データの実体を生成する必要がある．参照データ型の場合には，同一のデータ (実体) を複数の変数で共有できる．この応用として，別メソッドの変数と実体を共有することで，メソッド呼び出しに伴う副作用で値を書き換えられる．

null**(ナル)：** 参照データ型変数が何も参照しない場合の値

実体が生成されておらず，何も参照するものがない場合には，参照データ型変数には null という値が入る

例１： 参照データ型変数による実体の共有 - Reference1.java

```
public class Reference1 {
  String name;
  double area;
  int    population;
  Reference1(String n, double a, int p) {
    name = n;  area = a;  population = p;
  }

  public static void main(String[] args) {
    int origI = -1;
    int copyI = origI;                    // 基本データ型変数の代入 = 複製

    int[] origA = {3, 2, 1};
    int[] copyA = origA;                  // 参照データ型変数の代入 = 共有

    Reference1 origR = new Reference1("Tokyo", 2102.14, 11639000);
    Reference1 copyR = origR;             // 参照データ型変数の代入 = 共有
```

```
    System.out.println("origI = " + origI + ", copyI = " + copyI);
    System.out.println("origA = (" + origA[0] + ", " +
                       origA[1] + ", " + origA[2] + ")");
    System.out.println("copyA = (" + copyA[0] + ", " +
                       copyA[1] + ", " + copyA[2] + ")");
    System.out.println("origR = (" + origR.name + ", " +
                       origR.area + ", " + origR.population + ")");
    System.out.println("copyR = (" + copyR.name + ", " +
                       copyR.area + ", " + copyR.population + ")");

    origI = 0;  copyI = 1;                  // 基本データ型の実体の書き換え
    origA[0] = 0;  copyA[2] = 5;            // 参照データ型の実体の書き換え
    origR.name = "Kanagawa";  copyR.population = 0;
    System.out.println("origI = " + origI + ", copyI = " + copyI);
    System.out.println("origA = (" + origA[0] + ", " +
                       origA[1] + ", " + origA[2] + ")");
    System.out.println("copyA = (" + copyA[0] + ", " +
                       copyA[1] + ", " + copyA[2] + ")");
    System.out.println("origR = (" + origR.name + ", " +
                       origR.area + ", " + origR.population + ")");
    System.out.println("copyR = (" + copyR.name + ", " +
                       copyR.area + ", " + copyR.population + ")");
  }
}
```

例２：参照データ型変数による副作用の利用 - Reference2.java
別メソッドの変数との実体を共有することによって，メソッド呼び出しに伴って２つ以上の値を返す (書き換える) ことができる

```
public class Reference2 {
  String name;
  double area;
  int    population;
  Reference2(String n, double a, int p) {
    name = n;  area = a;  population = p;
  }
  static void sideEffect(int i, int[] a, Reference2 r) {
    i = 0;  a[0] = 0;  r.name = "Kanagawa";
  }
  public static void main(String[] args) {
    int i = 1;
    int[] a = {3, 2, 1};
    Reference2 r = new Reference2("Tokyo", 2102.14, 11639000);
```

```
    System.out.println("i = " + i);
    System.out.println("a = (" + a[0] + ", " + a[1] + ", " + a[2] + ")");
    System.out.println("r = (" + r.name + ", " +
                       r.area + ", " + r.population + ")");

    sideEffect(i, a, r);                    // 副作用による値の変更
    System.out.println("i = " + i);
    System.out.println("a = (" + a[0] + ", " + a[1] + ", " + a[2] + ")");
    System.out.println("r = (" + r.name + ", " +
                       r.area + ", " + r.population + ")");
  }
}
```

例3：副作用による複数データの受け戻し - Reference3.java

```
public class Reference3 {
  static void calculate(double[] a, double[] b, double[] x, double[] y) {
    for (int i = 0; i < 3; i++) {           // 参照データ x,y の書き換え
      x[i] = a[i] + b[i];
      y[i] = a[i] - b[i];
    }
  }
  public static void main(String[] args) {
    double[] vec1 = {1.0, 0.0, 0.0};
    double[] vec2 = {0.0, 0.0, 1.0};
    double[] ans1 = new double[3];
    double[] ans2 = new double[3];
    calculate(vec1, vec2, ans1, ans2);      // 複数データの受け戻し
    System.out.println("ans1 = (" + ans1[0] + ", " +
                       ans1[1] + ", " + ans1[2] + ")");
    System.out.println("ans2 = (" + ans2[0] + ", " +
                       ans2[1] + ", " + ans2[2] + ")");
    calculate(vec1, vec2, vec1, vec2);      // ただし注意が必要
    System.out.println("vec1 = (" + vec1[0] + ", " +
                       vec1[1] + ", " + vec1[2] + ")");
    System.out.println("vec2 = (" + vec2[0] + ", " +
                       vec2[1] + ", " + vec2[2] + ")");
  }
}
```

2.32 クラスとインスタンス (3) - 継承

Java ではクラスの継承が利用できる．一般的な機能を持ったクラスを用意しておけば，そのクラスを継承 (拡張) したクラスを作ることで，新しいプログラムを簡単に作り出せる．

継承 (inheritance)： 他のクラスの性質 (変数やメソッド) を引き継ぐこと
他のクラスで定義された変数とメソッドを含み，新たに変数とメソッドを加えられる
クラス A を継承する新しいクラス B は，次のように定義される

```
class クラスB extends クラスA { }
```

このため，継承と同じ意味で「拡張」と言うこともある
ただし，`final` クラスは継承 (拡張) できない (クラス A は `final` であってはならない)

サブクラス (subclass)： 継承するクラスのこと
クラス A を継承 (拡張) したクラス B は，クラス A のサブクラス になる

スーパークラス (superclass)： 継承されるクラスのこと
クラス A を継承 (拡張) した クラス B のスーパークラス は，クラス A になる

多重継承 (multiple inheritance)： 複数のクラスを同時に継承すること
Java では多重継承は許されていない (`extends` の後にはクラス名を 1 つしか書けない)

コンストラクタの連鎖： スーパークラスのコンストラクタの呼出
サブクラスがインスタンス化される場合には，スーパークラスのコンストラクタも起動されなくてはならない．
サブクラスのコンストラクタに何も書かれていなければ，スーパークラスのデフォルトコンストラクタが起動される (逆に言えばデフォルトコンストラクタはスーパークラスのデフォルトコンストラクタ呼出のみを実行する)
スーパークラスのコンストラクタを明示的に呼び出す場合には，サブクラスの コンストラクタ定義の最初に 次のように書く．

```
super(実引数並び)
```

同一クラスのシグニチャの異なるコンストラクタを呼び出す場合には，次のようにする

```
this(実引数並び)
```

シャドウ (shadow)： サブクラスでスーパークラスと同一名の変数を作ること
シャドウする変数は普通にアクセスできる

```
変数名
this. 変数名
```

シャドウされた変数には，以下のようにアクセスする (スーパークラスの変数であることを明示的に書いてアクセスするか，スーパークラスにキャストしてから変数にアクセスする)

super.変数名
((スーパークラス名)this).変数名

オーバライド (override)： サブクラスでスーパークラスと同一シグニチャのインスタンスメソッドを作ること
オーバライドするメソッドは，普通に呼び出される

メソッド名 (実引数並び)
this.メソッド名 (実引数並び)

オーバライドされたメソッドを用いるには，以下のように呼び出す (シャドウ変数のようなキャストは使えない)

super.メソッド名 (実引数並び)

クラスメソッドやfinalメソッドはオーバライドできない (スーパークラスでstaticやfinal宣言されていてはならない)

動的メソッド検索 (dynamic dispatch)： メソッドを実行時に決定すること
実際に起動されるメソッドが実行時でないと定まらないこともありうる

抽象クラス (abstract class)： インスタンス化されないクラス
実際にはサブクラスがインスタンス化される
抽象クラスの定義は，次のようになる

abstract class クラス名 { }

抽象メソッド (abstract method)： 定義されないメソッド
抽象クラスにおいてのみ定義され，メソッド名，引数並びと戻し値の型などは決まっているが，動作が決まっていないメソッド
抽象メソッドとなりうるのはインスタンスメソッドのみで，実際の動作はサブクラスで定義される．サブクラスで抽象メソッドを定義することを,「抽象メソッドを実装する」と言う
抽象メソッドの宣言は，次のようになる (メソッド本体がないことに注意せよ)

abstract 型名 メソッド名 (仮引数並び) ;

Object **クラス：** 最上位のクラス
すべてのクラスはObjectクラスのサブクラスとなる．明示的に拡張しない (クラス宣言にextendsが含まれない) クラスは，Objectクラスのサブクラスとなる．

例１： 2乗根を求めるプログラム - Function1.java
二分探索法 (詳細は4.1節「二分探索法」を参照せよ) を用いて2乗根を計算している．なお，以下のプログラムでは，解を求める関数が単調増加関数であることを前提としている．

```
public class Function1 {
  private static final double EPS = 1.0e-14; // 解探索の打切り幅
  private double c, lower, upper;           // 定数と探索区間の両端
  public Function1(double c) {              // コンストラクタ - 定数 c
    this.c = c;
  }
  public double compute(double x) {         // 2次関数 (x^2 - c) の計算
    return x*x - c;
  }
  private void bound() {                    // 探索区間の設定
    if (c < 0) {                            // c<0 の場合, 解なし
      System.out.println("Error: c (= " + c + " ) must be positive.");
      lower = upper = 0.0;
    }
    else if (c < 1.0) {                     // 0<c<1 の場合, 区間 [c,1]
      lower = c;  upper = 1.0;
    }
    else {                                  // 1<c の場合, 区間 [1,c]
      lower = 1.0;  upper = c;
    }
  }
  public double solve() {                   // 二分探索法による求解
    bound();
    while ((upper-lower) > EPS*Math.max(1.0, Math.abs(lower))) {
      double mid = (lower+upper)/2.0;
      if (compute(mid) > 0.0)
        upper = mid;
      else
        lower = mid;
    }
    return (lower+upper)/2.0;
  }
  public static void main(String[] args) {
    Function1 f = new Function1(Double.parseDouble(args[0]));
    System.out.println("root: " + f.solve());
  }
}
```

例2： 2乗根と3乗根を求めるプログラム - Function2.java

クラス変数 EPS，インスタンス変数 c，lower，upper，二分探索のメソッド solve() などは共用可能なので，3乗根を計算するためのクラス CubicFunction2 を Function2 のサブクラスとする．

```
public class Function2 { // 継承される変数やメソッドはprotectedに変更
  protected static final double EPS = 1.0e-14; // 解探索の打切り幅
  protected double c, lower, upper;          // 定数と探索区間の両端
  public Function2(double c) {               // コンストラクタ - 定数 c
    this.c = c;
  }
  public double compute(double x) {          // 2次関数 (x^2 - c) の計算
    return x*x - c;
  }
  protected void bound() {                   // 探索区間の設定
    if (c < 0) {                             // c<0 の場合, 解なし
      System.out.println("Error: c (= " + c + " ) must be positive.");
      lower = upper = 0.0;
    }
    else if (c < 1.0) {                      // 0<c<1 の場合, 区間 [c,1]
      lower = c;  upper = 1.0;
    }
    else {                                   // 1<c の場合, 区間 [1,c]
      lower = 1.0;  upper = c;
    }
  }
  public double solve() {                    // 二分探索法による求解
    bound();
    while ((upper-lower) > Math.max(EPS, EPS*Math.abs(lower))) {
      double mid = (lower+upper)/2.0;
      if (compute(mid) > 0.0)
        upper = mid;
      else
        lower = mid;
    }
    return (lower+upper)/2.0;
  }
  public static void main(String[] args) {
    Function2 fs = new Function2(Double.parseDouble(args[0]));
    Function2 fc = new CubicFunction2(Double.parseDouble(args[0]));
    System.out.println("square root: " + fs.solve());
    System.out.println("cubic root: " + fc.solve());
  }
}

class CubicFunction2 extends Function2 {   // 3次関数クラス
  public CubicFunction2(double c) {         // コンストラクタの明示的呼出
```

```
    super(c);
  }
  public double compute(double x) {        // compute のオーバライド
    return x*x*x - c;                      // 3 次関数 (x^3 - c) の計算
  }
  protected void bound() {                 // bound のオーバライド
    if (c > 0.0)                           // 探索区間の設定
      if (c > 1.0) {                       // 1<c の場合, 区間 [1,c]
        lower = 1.0;  upper = c;
      }
      else {                               // 0<c<1 の場合, 区間 [c,1]
        lower = c;  upper = 1.0;
      }
    else
      if (c < -1.0) {                      // c<-1 の場合, 区間 [c,-1]
        lower = c;  upper = -1.0;
      }
      else {                               // -1<c<0 の場合, 区間 [-1,c]
        lower = -1.0;  upper = c;
      }
  }
}
```

例 3： 2 乗根と 3 乗根を求めるプログラム - Function3.java

クラス変数 EPS, インスタンス変数 c, lower, upper, 二分探索のメソッド solve() などは一般の関数に利用できる. そこで, 抽象クラス Function3 を作り, 2 乗根を計算するクラス SquareFunction3 と 3 乗根を計算するクラス CubicFunction3 を Function3 のサブクラスとする. 抽象クラス Function3 では, 2 つのメソッド compute(x), bound() は抽象メソッドであり, サブクラスにおいて定義される.

```
public abstract class Function3 {                  // 抽象クラス Function3
  protected static final double EPS = 1.0e-14; // 解探索の打切り幅
  protected double c, lower, upper;        // 定数と探索区間の両端
  public abstract double compute(double x);// 抽象メソッド compute
  protected abstract void bound();         // 抽象メソッド bound
  public Function3(double c) {             // コンストラクタ - 定数 c
    this.c = c;
  }
  public double solve() {                  // 二分探索法による求解
    bound();
    while ((upper-lower) > Math.max(EPS, EPS*Math.abs(lower))) {
      double mid = (lower+upper)/2.0;
      if (compute(mid) > 0.0)
```

```
        upper = mid;
      else
        lower = mid;
    }
    return (lower+upper)/2.0;
  }
  public static void main(String[] args) {
    Function3 fs = new SquareFunction3(Double.parseDouble(args[0]));
    Function3 fc = new CubicFunction3(Double.parseDouble(args[0]));
    System.out.println("square root: " + fs.solve());
    System.out.println("cubic root: " + fc.solve());
  }
}

class SquareFunction3 extends Function3 {  // 2次関数クラス
  public SquareFunction3(double c) {       // コンストラクタの明示的呼出
    super(c);
  }
  public double compute(double x) {        // computeの定義
    return x*x - c;                        // 2次関数 (x^2 - c) の計算
  }
  protected void bound() {                 // boundの定義
    if (c < 0) {                           // 探索区間の設定
      System.out.println("Error: c (= " + c + " ) must be positive.");
      lower = upper = 0.0;
    }
    else if (c < 1.0) {
      lower = c;  upper = 1.0;
    }
    else {
      lower = 1.0;  upper = c;
    }
  }
}

class CubicFunction3 extends Function3 {   // 3次関数クラス
  public CubicFunction3(double c) {        // コンストラクタの明示的呼出
    super(c);
  }
  public double compute(double x) {        // computeの定義
    return x*x*x - c;                      // 3次関数 (x^3 - c) の計算
  }
```

```
  protected void bound() {                    // bound の定義
    if (c > 0.0)                              // 探索区間の設定
      if (c > 1.0) {
        lower = 1.0;  upper = c;
      }
      else {
        lower = c;  upper = 1.0;
      }
    else
      if (c < -1.0) {
        lower = c;  upper = -1.0;
      }
      else {
        lower = -1.0;  upper = c;
      }
  }
}
```

例４： 解の計算 (完成版) - Function.java

二分探索法で解を求める抽象クラス Function . このプログラムだけでは走らないので，例５のプログラム FDemo4.java とともに走らせること.

```
public abstract class Function {            // 抽象クラス Function
  protected static final double EPS = 1.0e-14; // 解探索の打切り幅
  protected double c, lower, upper;         // 定数と探索区間の両端
  public abstract double compute(double x);// 抽象メソッド compute
  protected abstract void bound();          // 抽象メソッド bound
  public Function(double c) {               // コンストラクタ - 定数 c
    this.c = c;
  }
  public String description(double d) {     // 関数 f(d) の表示文字列
    return "f( " + d + " )";
  }
  public double solve() {                   // 二分探索法による求解
    bound();
    while ((upper-lower) > Math.max(EPS, EPS*Math.abs(lower))) {
      double mid = (lower+upper)/2.0;       // 解の絶対値が大きい際の処置
      if (compute(mid) > 0.0)
        upper = mid;
      else
        lower = mid;
    }
    return (lower+upper)/2.0;
```

```
  }
}
```

例5： 解の計算 (完成版) - FDemo4.java

例4の二分探索法で解を求める抽象クラス Function を拡張して，2乗根を計算するクラス SquareFunction，3乗根を計算するクラス CubicFunction，正接 ($\tan x$) の解を計算するクラス Tangent を定義している．2つの抽象メソッド compute(x)，bound() は，それぞれのサブクラスにおいて定義されている．

```
public class FDemo4 {
  public static void main(String[] args) {
    Function[] f = new Function[3];
    f[0] = new SquareFunction(Double.parseDouble(args[0]));
    f[1] = new CubicFunction(Double.parseDouble(args[0]));
    f[2] = new Tangent(Double.parseDouble(args[0]));
    for (int i = 0; i < 3; i++)
      System.out.println(f[i].description(f[i].solve()) + " = " + args[0]);
            // 動的メソッド探索 - 配列 f[i] には Function の配列であるが,
            // 実際には様々なサブクラスのインスタンスが格納されており,
            // 実行時にメソッド description, compute, bound が定まる
  }
}

class SquareFunction extends Function {  // 2次関数クラス
  public SquareFunction(double c) {       // コンストラクタの明示的呼出
    super(c);
  }
  public String description(double d) {    // description のオーバライド
    return "( " + d + " )^2";
  }
  public double compute(double x) {        // compute の定義
    return x*x - c;                        // 2次関数 (x^2 - c) の計算
  }
  protected void bound() {                 // bound の定義
    if (c < 0) {                           // 探索区間の設定
      System.out.println("Error: c (= " + c + " ) must be positive.");
      lower = upper = 0.0;
    }
    else if (c < 1.0) {
      lower = c;  upper = 1.0;
    }
    else {
      lower = 1.0;  upper = c;
```

```
    }
  }
}

class CubicFunction extends Function {   // 3次関数クラス
  public CubicFunction(double c) {        // コンストラクタの明示的呼出
    super(c);
  }
  public String description(double d) {   // descriptionのオーバライド
    return "( " + d + " )^3";
  }
  public double compute(double x) {       // computeの定義
    return x*x*x - c;                     // 3次関数 (x^3 - c) の計算
  }
  protected void bound() {                // boundの定義
    if (c > 0.0)                          // 探索区間の設定
      if (c > 1.0) {
        lower = 1.0;  upper = c;
      }
      else {
        lower = c;  upper = 1.0;
      }
    else
      if (c < -1.0) {
        lower = c;  upper = -1.0;
      }
      else {
        lower = -1.0;  upper = c;
      }
  }
}

class Tangent extends Function {          // 正接関数(tan)クラス
  public Tangent(double c) {              // コンストラクタの明示的呼出
    super(c);
  }
  public String description(double d) {   // descriptionのオーバライド
    return "tan( " + d + " )";
  }
  public double compute(double x) {       // computeの定義
    return Math.tan(x) - c;               // 正接 (tan x) の計算
  }
```

```
  protected void bound() {                  // bound の定義
    lower = - (Math.PI / 2.0) + EPS;        // 探索区間の設定
    upper = (Math.PI / 2.0) - EPS;          // (-PI/2 + epsilon, PI/2 - epsilon)
  }
}
```

2.33 アプレットとグラフィクス

応用プログラム，アプリケーション (application)： 特定の目的を持ったプログラム

一般的には，基本プログラム (オペレーティングシステム) との対比として，ワープロや表計算などの特定用途のプログラムを意味する．
Java の場合には，アプレットの対比として，特に制限のないプログラムを意味し，以下のコマンドによって実行されるものをアプリケーションと呼ぶ

```
ca10101$ java クラスファイル名¶
```

ただし,「クラスファイル名」と「クラス名」は同じであり，そのクラスは public クラスでなくてはならない．さらに public なクラスメソッドである main から実行が開始される

アプレット (applet)： 小さなアプリケーション

WWW ページから実行可能なプログラムで，アプレットビューワや WWW ブラウザ上で起動される．グラフィクスのプログラムなどを簡単に実現できる一方で，ファイル出力ができないなどの制限がある
アプレットプロブラムは Applet クラスのサブクラスとして実現し,「クラスファイル名」と「クラス名」は同じとする．public なインスタンスメソッドである init，start，stop，destroy，paint などが状況に応じて実行される

APPLET **タグ (tag)：** アプレット実行のための HTML タグ

WWW ページから実行するアプレットのクラスファイルを指定するための HTML タグ (WWW ページ記述用言語の予約語)
最も簡単な APPLET タグは以下の通り

```
<APPLET CODE="クラスファイル名" WIDTH=幅ピクセル数 HEIGHT=高さピクセル数>
</APPLET>
```

より詳細には，次の通り ([] 内は省略可能部分)

```
<APPLET [CODEBASE="クラスファイルのある URL"]
         CODE="クラスファイル名"
        [ALT="実行不可能な場合の表示文字列"]
         WIDTH=幅ピクセル数
         HEIGHT=高さピクセル数
        [ALIGN=配置指定]
        [HSPACE=横枠空白ピクセル数]
```

```
        [VSPACE=縦枠空白ピクセル数]>
[<PARAM NAME="パラメタ名" VALUE="値文字列">] ...
</APPLET>
```

アプレットビューワやWWWブラウザが，APPLETタグを含んだWWWページ(HTMLファイル)を読み込むと，指定されたクラスファイル(アプレット)が実行される

アプレットビューワ (appletviewer)： アプレット実行プログラム (Java仮想機械)
以下のコマンドを実行すると，アプレットビューワがWWWページ(HTMLファイル)を読み込んで，HTMLファイル内のAPPLETタグによって指定されたクラスファイルが実行される

```
ca10101$ appletviewer WWWページURL ¶
```

WWWブラウザ： WWWページを見るためのブラウザ
Safari や Mozilla などのWWWブラウザもアプレットを実行するためのJava仮想機械を内蔵している
たとえば，以下のコマンドを実行すると，SafariがWWWページ(HTMLファイル)を読み込んで，当該のページが表示されるとともに，APPLETタグで指定されたアプレットクラスファイルが実行される(WWWページの一部としてアプレットが実行される)

```
ca10101$ open WWWページURL ¶
```

`init` アプレット初期化メソッド
アプレットビューワないしWWWブラウザによってアプレットがメモリ上に読み込まれた際に，最初に呼び出されるメソッド

`start` アプレット起動メソッド
アプレットビューワないしWWWブラウザによってアプレットの実行が開始される際に，呼び出されるメソッド

`stop` アプレット停止メソッド
アプレットビューワないしWWWブラウザによってアプレットの実行が停止される際に，呼び出されるメソッド

`destroy` アプレット破棄メソッド
アプレットビューワないしWWWブラウザによってアプレットがメモリから捨てられる際に，呼び出されるメソッド

`paint` 描画メソッド
画面上でComponentが隠された後に，再度描画し直す必要があるときなどに，自動的に呼び出されるメソッド

グラフィクス (graphics)： 図形や文字などの描画
語源(graph)的には線図形を描く手法を指すが，計算機的な用語(コンピュータグラフィクス)では，塗りつぶしや画像なども含んだ描画全般を指す

画素，ピクセル (pixel)，ラスタ (raster)： 計算機の扱う画像の 1 点

現在の計算機によるグラフィクスは，画像を格子状に細かく分けて，各格子点に色をつけることで画像を構成する．この格子点をピクセル (画素)，またはラスタ (raster) と呼ぶ．たとえば，教育棟の NC(ネットワークコンピュータ) の画面は，1024(横) × 768(縦) のピクセルから構成されている

ピクセルの座標は，左上を原点 (0, 0) とし，x 軸は右向き，y 軸は下向きになる

`Graphics` **クラス：** 描画のためのクラス

`Graphics` クラスのオブジェクト (インスタンス) は，描画領域，描画のための色やフォントなどの情報を保持するともに，描画のためのインタンスメソッド提供する．`Applet` は `Graphics` オブジェクトを自動的に生成するので，それを用いて描画を行なう

例１： アプレット起動用 WWW ページ (最も簡単なアプレット) - `HelloApplet.html`

例 2 のプログラム (`HelloApplet.class`) を呼び出すための HTML ファイル．クラスファイル `HelloApplet.class` を同じディレクトリに置いて，アプレットビューワや WWW ブラウザで見ると，当該 WWW ページ内でアプレットが実行される

```
<HTML>
<HEAD><TITLE> Hello Applet </TITLE></HEAD>
<BODY>
<H2>Source Code</H2>
<CODE><PRE>
import java.applet.*;
import java.awt.*;
public class HelloApplet extends Applet {
  public void paint(Graphics g) {
    g.drawString("Hello Applet!!", 40, 100);  // 文字列の表示 - 位置 (40,100)
  }
}
</PRE></CODE>
<H2>Result</H2>
<APPLET CODE="HelloApplet" WIDTH=200 HEIGHT=200>
</APPLET>
</BODY>
</HTML>
```

例２： 最も簡単なアプレットプログラム - `HelloApplet.java`

例 1 の HTML ファイルを用いるとアプレットビューワなどで実行できる

```
import java.applet.*;
import java.awt.*;
public class HelloApplet extends Applet {
  public void paint(Graphics g) {
    g.drawString("Hello Applet!!", 40, 100);  // 文字列の表示 - 位置 (40,100)
```

```
  }
}
```

例３： アプレット起動用 WWW ページ (アプレットの動作) - AppletLifeCycle.html
例 4 のプログラム (AppletLifeCycle.class) を呼び出すための HTML ファイル

```
<HTML>
<HEAD><TITLE> Applet Life Cycle </TITLE></HEAD>
<BODY>
<H2>Result</H2>
<APPLET CODE="AppletLifeCycle" WIDTH=600 HEIGHT=100>
</APPLET>
</BODY>
</HTML>
```

例４： アプレットの動作 - AppletLifeCycle.java
アプレットビューワでは，メニュから「アプレット→再起動」,「アプレット→再読込み」,「アプレット→中止」,「アプレット→開始」としたり，他のウィンドウを重ねたりすること.
WWW ブラウザ (Safari) では，WWW ページの読み込み (「ファイル→ファイルを開く…」)，再読み込み (ウィンドウ内のツールバーで「再読込み (渦巻印)」)，他の WWW ページへの移動・復帰 (「←」「→」) したり，他のウィンドウを重ねたりすること.

```
import java.applet.*;
import java.awt.*;
public class AppletLifeCycle extends Applet {
  private String str = "";
  public void init() {
    str += "init;  ";
  }
  public void start() {
    str += "start;  ";
  }
  public void stop() {
    str += "stop;  ";
  }
  public void destroy() {
    str += "destroy;  ";
    System.out.println("destroy");
  }
  public void paint(Graphics g) {
    str += "p;  ";
    g.drawString(str, 10, 50);
  }
}
```

例5： アプレット起動用WWWページ(アプレットの情報) - AppletInformation.html
例6のプログラム(AppletInformation.class)を呼び出すためのHTMLファイル
PARAMタグによってアプレットにパラメタを渡している

```
<HTML>
<HEAD><TITLE> Applet Information </TITLE></HEAD>
<BODY>
<H2>Result</H2>
<APPLET CODE="AppletInformation" WIDTH=400 HEIGHT=300>
<PARAM NAME="Message" VALUE="Argument for mesg">
<PARAM NAME="Text" VALUE="Argument for text">
</APPLET>
</BODY>
</HTML>
```

例6： アプレットの情報 - AppletInformation.java
getParameterメソッドによってWWWページからパラメタ情報を読み込める(アプリケーションのコマンド引数に似ている)．例5のPARAMタグで指定されたNAME文字列に対応するVALUE文字列を獲得する

```
import java.applet.*;
import java.awt.*;
public class AppletInformation extends Applet {
  private int ptimes = 0;
  public void paint(Graphics g) {
    g.drawString("Message:" + getParameter("Message"), 10, 50);
    g.drawString("Text: " + getParameter("Text"), 10, 100);
                        // WWWページからのパラメタ情報の獲得
    g.drawString("Foreground: " + getForeground(), 10, 150);
    g.drawString("Background: " + getBackground(), 10, 200);
                        // 描画色と背景色の獲得
    g.drawString("Dimension: " + getSize(), 10, 250);
                        // アプレット(表示領域)の獲得
    showStatus("Paint times: " + (++ptimes));
                        // paintメソッドの呼出し回数の表示（ステータス行）
  }
}
```

例7： ×線の表示 - Crossline.java
アプレットの表示領域において対角線(2本)を描画する．表示領域の大きさはgetSizeメソッドで調べている

```
import java.applet.*;
import java.awt.*;
```

```
/*
  <APPLET CODE="Crossline" WIDTH=300 HEIGHT=200>
  </APPLET>
*/
public class Crossline extends Applet {
  public void paint(Graphics g) {
    Dimension d = getSize();
      // 幅と高さの２つの値を戻すために Dimension オブジェクトを用いている
      // 幅：d.width , 高さ：d.height
    g.drawLine(0, 0, d.width-1, d.height-1);
    g.drawLine(0, d.height-1, d.width-1, 0);
  }
}
```

上記の Java プログラムではコメント部分に PARAM タグを書いてあるので，次のようにしてアプレットを起動できる (HTML ファイルは不要)

```
ca10101$ appletviewer Crossline.java ¶
```

例８： 円の描画 - Circle.java

曲線は細かな線分の集合として描画する．中心 (x_o, y_o)，半径 r の円は，次式で表せる．

$$x = r\cos\theta + x_o,\ y = r\sin\theta + y_o \quad (\theta \in [-\pi, \pi])$$

そこで，円上の頂点 (以下の例では 128 個) を線分列で結んで表示する PARAM タグで与えるパラメタ Corners によって頂点の個数を指定するので，再コンパイルせずに頂点の個数を変更できる．VALUE の数値を変えてみること

```
import java.applet.*;
import java.awt.*;
/*
  <APPLET CODE="Circle" WIDTH=300 HEIGHT=300>
  <PARAM NAME="Corners" VALUE="128">
  </APPLET>
*/
public class Circle extends Applet {
  private int nc;
  public void init() {
    nc = Integer.parseInt(getParameter("Corners"));
         // PARAM タグで指定された頂点個数の読込み
  }
  public void paint(Graphics g) {
    Dimension d = getSize();
    int radius;        // 半径：表示領域の幅と高さのうち狭い方の 40%
```

```
    if (d.width < d.height)
      radius = (int)(d.width * 0.4);
    else
      radius = (int)(d.height * 0.4);
    int oldX = radius + (d.width / 2);
    int oldY = d.height / 2;
    for (int i = 1; i <= nc; i++) {
      double theta = 2.0 * Math.PI * i / nc;
      int newX = (int)(radius * Math.cos(theta)) + (d.width / 2);
      int newY = (int)(radius * Math.sin(theta)) + (d.height / 2);
      g.drawLine(oldX, oldY, newX, newY);
                        // (oldX, oldY) から (newX, newY) へ線分描画
      oldX = newX;
      oldY = newY;
    }
  }
}
```

2.34　イベント処理

グラフィクスを用いた対話的な (interactive; 人間と計算機との対話を実現する) プログラムでは，人間の入力に応じて動作が変わる．人間の入力のように，いつ生じるかわからない状態変更を処理する手法として，イベントの考え方が用いられる．

イベント (event)： (入力などによって生じた) 状態変更

マウスのボタンをクリックしたり，キーボードから文字を入力したりすると，その状態の変更がイベントとして通知される．グラフィクスを用いた対話的なプログラムでは，イベントの通知を受けて，別の作業へと実行が移る．このようにイベントに応じてプログラムが実行される方式をイベント駆動型 (event driven) のプログラムと呼ぶ

イベントソース (event source)： イベントが生じたオブジェクト

イベントは，様々な場所で，様々な機会に生じる．イベントが生じた場所 (オブジェクト) がイベントソースになる．たとえば，ある `Applet` 内でマウスボタンがクリックされた場合には，その `Applet` がイベントソースとなり，マウス (がクリックされたという) イベントが通知される

イベントリスナ，`EventListener`： イベントを受けとるオブジェクト (インタフェース)

イベントが生じると，イベントソースから，登録されている `EventListener`(複数) にイベントが通知され，該当する `EventListener` のメソッドが起動される．すなわち，予め `Applet` オブジェクト，より正確には `Component` オブジェクト，に `add`*Event*`Listener` メソッドを用いて `EventListener` を登録しておく必要がある．Java では `EventListener` はインタフェースとして実現されている

イベントアダプタ，EventAdapter**：**イベントを受けとるオブジェクト (クラス)

インタフェースである EventListener に代わるクラスとして，EventAdapter がある．この授業では，インタフェースについて学習していないので，EventAdapter を用いてイベント処理プログラムを作成する．イベントが生じると，イベントソースである Applet に登録されているすべての EventAdapter にイベントが通知され，イベントの種類に応じたメソッド起動される．ただし，本来の使い方とは若干異なることに注意すること

マウスイベント，MouseEvent**：**マウス関連のイベントオブジェクト

Java では，イベント自体も継承を用いたオブジェクト (クラス) として定義されている．この授業ではマウス関連のイベントである MouseEvent のみを学ぶ．

MouseEvent には，以下の種類がある

MouseEvent の種類	状態変更	EventListener/Adapter	起動メソッド
MOUSE_CLICKED	ボタンのクリック	MouseListener/Adapter	mouseClicked
MOUSE_ENTERED	カーソルの進入	MouseListener/Adapter	mouseEntered
MOUSE_EXITED	カーソルの退出	MouseListener/Adapter	mouseExited
MOUSE_PRESSED	ボタンのプレス	MouseListener/Adapter	mousePressed
MOUSE_RELEASED	ボタンのリリース	MouseListener/Adapter	mouseReleased
MOUSE_DRAGGED	マウスのドラッグ	MouseMotionListener/Adapter	mouseDragged
MOUSE_MOVED	カーソルの移動	MouseMotionListener/Adapter	mouseMoved

また MouseEvent は修飾子によって，イベントが発生したボタン (左中右) やその時のキーボードの状態を調べられる．修飾子は getModifiers メソッドによって獲得できる．また MouseEvent が発生した際のマウスカーソルの位置は，getX メソッドと getY メソッドによって調べられる

例１：マウスイベントによる背景色の変更 - BackgroundApplet.java

Applet 内でマウスボタンをクリックすると背景色が変化する．ここで MouseEvent のイベントソースは Applet クラスを拡張した BackgroundApplet であり，EventListener は MouseAdapter クラスを拡張した BAMouseAdapter である．発生した MouseEvent は，まず BAMouseAdapter に通知され，mousePressed メソッドないし mouseReleased メソッドが起動される．これらのメソッドは BackgroundApplet の同名のメソッド mousePressed ないし mouseReleased に MouseEvent をそのまま通知する．BAMouseAdapter は，通知先の BackroundApplet オブジェクトをインスンタンス変数 ba に保持している

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/*
  <APPLET CODE="BackgroundApplet" WIDTH=200 HEIGHT=200>
  </APPLET>
*/
public class BackgroundApplet extends Applet {
  public void init() {
```

```
    setBackground(Color.green);  // 背景色を緑に指定
    addMouseListener(new BAMouseAdapter(this));
  }    // BAMouseAdapter のインスタンスを生成し，MouseListener として登録
  protected void mousePressed(MouseEvent me) {
    setBackground(Color.red);    // 背景色を赤に指定
    repaint();                   // 画像更新の要求 (update の起動)
  }
  protected void mouseReleased(MouseEvent me) {
    setBackground(Color.green);  // 背景色を緑に指定
    repaint();                   // 描画更新の要求 (update の起動)
  }
}
class BAMouseAdapter extends MouseAdapter {
  private BackgroundApplet ba;   // イベントを通知する Applet
  public BAMouseAdapter(BackgroundApplet ba) {
    this.ba = ba;                // ba(通知先 Applet) の初期化
  }
  public void mousePressed(MouseEvent me) {
    ba.mousePressed(me);         // ba へ mousePressed の通知
  }
  public void mouseReleased(MouseEvent me) {
    ba.mouseReleased(me);        // ba へ mouseReleased の通知
  }
}
```

例 2： マウスイベントによる背景色の変更 (インタフェース版) - BackgroundApplet1.java
例 1 とまったく同じプログラムだが，MouseAdapter クラスではなく MouseLinstener インタフェースを用いている．インタフェース (詳細は 2.35 節「インタフェース」を見よ) によって，BackgroundApplet は，イベントソースかつ描画領域であるとともに，同時に MouseListener すなわち EventListener となる．このためプログラムは非常にすっきりしている．ただし，インタフェースのメソッドはすべて抽象メソッドであるため，何もしないメソッドにも定義が必要となる

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/*
  <APPLET CODE="BackgroundApplet1" WIDTH=200 HEIGHT=200>
  </APPLET>
*/
public class BackgroundApplet1 extends Applet implements MouseListener {
  public void init() {
    setBackground(Color.green);  // 背景色を緑に指定
```

```
    addMouseListener(this);       // 自分自身を MouseListener として登録
  }
  public void mousePressed(MouseEvent me) {
    setBackground(Color.red);     // 背景色を赤に指定
    repaint();                    // 画像更新の要求 (update の起動)
  }
  public void mouseReleased(MouseEvent me) {
    setBackground(Color.green);   // 背景色を緑に指定
    repaint();                    // 描画更新の要求 (update の起動)
  }
  public void mouseEntered(MouseEvent me) { }
  public void mouseExited(MouseEvent me) { }
  public void mouseClicked(MouseEvent me) { }
}                           // その他の何もしないメソッドについても定義が必要
```

例３： カーソル位置の獲得と描画 - MarkerApplet.java

getModifiers メソッドによって修飾子を獲得できる．修飾子は int 型の値であり，各種のマスク値を比較することで，イベントの生じたボタンの種類やキーボードの状態を調べられる．また，getX メソッドと getY メソッドを用いると，イベント発生時のカーソル位置を調べられる．このプログラムはボタンが押されるたびに，カーソル位置に正方形を描画する．押されたボタンによって正方形の色が変わる．前の例のように repaint メソッドによって update していないので，一度描いた正方形は消えずに画面上に残っている

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/*
  <APPLET CODE="MarkerApplet" WIDTH=200 HEIGHT=200>
  </APPLET>
*/
public class MarkerApplet extends Applet {
  static final int SIZE = 10;     // 描画する正方形の一辺の長さ
  public void init() {
    addMouseListener(new MAMouseAdapter(this));
  }                           // MAMouseAdapter を MouseListener として登録
  protected void mousePressed(MouseEvent me) {
    int modifiers = me.getModifiers(); // 修飾子 (modifiers) の獲得
    Graphics g = getGraphics();  // Graphics の獲得
    if ((modifiers & me.BUTTON3_MASK) != 0)
      g.setColor(Color.blue);     // ボタン 3(右ボタン) なら描画色を青に指定
    // else if ((modifiers & me.BUTTON2_MASK) != 0)
    //  g.setColor(Color.green);   // ボタン 2(中ボタン) は Mac ではうまくな
```

```
い
    else
      g.setColor(Color.red);     // その他 (左ボタン) なら描画色を赤に指定
    int x = me.getX();           // カーソルの x 座標
    int y = me.getY();           // カーソルの y 座標
    g.fillRect(x - (SIZE/2), y - (SIZE/2), SIZE, SIZE);
  }                              // 正方形の描画
}
class MAMouseAdapter extends MouseAdapter {
  private MarkerApplet ma;       // イベントを通知する Applet
  public MAMouseAdapter(MarkerApplet ma) {
    this.ma = ma;                // ma(通知先 Applet) の初期化
  }
  public void mousePressed(MouseEvent me) {
    ma.mousePressed(me);         // ma へ mousePressed の通知
  }
}
```

例 4： マウスによる線分の描画 - RubberbandApplet.java

MouseAdapter だけではなく MuoseMotionAdapter を用いることで，マウスをドラッグ (ボタンをプレスしながら移動) する間も，マウスの動きに合わせて線分を描ける．このような線分は，一般にラバーバンド (早い話がゴム紐) と呼ばれる

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/*
  <APPLET CODE="RubberbandApplet" WIDTH=200 HEIGHT=200>
  </APPLET>
*/
public class RubberbandApplet extends Applet {
  private int startX;            // ゴム紐の始点の x 座標
  private int startY;            // ゴム紐の始点の y 座標
  public void init() {
    addMouseListener(new RAMouseAdapter(this));
    addMouseMotionListener(new RAMouseMotionAdapter(this));
  }                    // RAMouseAdapter と RAMouseMotionAdapter の登録
  protected void mousePressed(MouseEvent me) {
    startX = me.getX();          // 始点 x 座標の変更 (ボタンプレス時)
    startY = me.getY();          // 始点 y 座標の変更 (ボタンプレス時)
  }
  protected void mouseDragged(MouseEvent me) {
    Graphics g = getGraphics();  // 描画用 Graphics オブジェクトの獲得
```

```
    update(g);                    // 画面の消去
    int x = me.getX();            // カーソル位置の x 座標
    int y = me.getY();            // カーソル位置の y 座標
    g.drawLine(startX, startY, x, y); // 線分の描画
  }
}
class RAMouseAdapter extends MouseAdapter {
  private RubberbandApplet ra;   // イベントを通知する Applet
  public RAMouseAdapter(RubberbandApplet ra) {
    this.ra = ra;                 // ra(通知先 Applet) の初期化
  }
  public void mousePressed(MouseEvent me) {
    ra.mousePressed(me);          // ra へ mousePressed の通知
  }
}
class RAMouseMotionAdapter extends MouseMotionAdapter {
  private RubberbandApplet ra;   // イベントを通知する Applet
  public RAMouseMotionAdapter(RubberbandApplet ra) {
    this.ra = ra;                 // ra(通知先 Applet) の初期化
  }
  public void mouseDragged(MouseEvent me) {
    ra.mouseDragged(me);          // ra へ mouseDragged の通知
  }
}
```

例 5： マウスを用いたお絵描き - DrawApplet.java
マウスの移動に伴って，次々と短い線分を描画することで，自由な線が描ける．

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/*
  <APPLET CODE="DrawApplet" WIDTH=200 HEIGHT=200>
  </APPLET>
*/
public class DrawApplet extends Applet {
  private int oldX;               // 直前カーソル位置の x 座標
  private int oldY;               // 直前カーソル位置の y 座標
  public void init() {
    addMouseListener(new DAMouseAdapter(this));
    addMouseMotionListener(new DAMouseMotionAdapter(this));
  }                    // DAMouseAdapter と DAMouseMotionAdapter の登録
  protected void mousePressed(MouseEvent me) {
```

```
    oldX = me.getX();            // 直前 x 座標の設定 (ボタンプレス時)
    oldY = me.getY();            // 直前 y 座標の設定 (ボタンプレス時)
  }
  protected void mouseDragged(MouseEvent me) {
    int newX = me.getX();        // カーソル位置の x 座標
    int newY = me.getY();        // カーソル位置の x 座標
    getGraphics().drawLine(oldX, oldY, newX, newY); // 線分の描画
    oldX = newX;                 // 直前 x 座標の更新
    oldY = newY;                 // 直前 y 座標の更新
  }
}
class DAMouseAdapter extends MouseAdapter {
  private DrawApplet da;         // イベントを通知する Applet
  public DAMouseAdapter(DrawApplet da) {
    this.da = da;                // da(通知先 Applet) の初期化
  }
  public void mousePressed(MouseEvent me) {
    da.mousePressed(me);         // da へ mousePressed の通知
  }
}
class DAMouseMotionAdapter extends MouseMotionAdapter {
  private DrawApplet da;         // イベントを通知する Applet
  public DAMouseMotionAdapter(DrawApplet da) {
    this.da = da;                // da(通知先 Applet) の初期化
  }
  public void mouseDragged(MouseEvent me) {
    da.mouseDragged(me);         // da へ mouseDragged の通知
  }
}
```

2.35 インタフェース - 継承

Java で継承できるクラスは唯 1 つに限られている．すなわち，自分のスーパークラス (親) は，高々1 つしか許していない．(extends の後ろにはクラス名を 1 つしか書けない)．この制約によって継承関係の木構造を保証し，継承に関わる複雑な問題 (たとえば 2 つのスーパークラスに同じ名前のインスタンス変数がある場合，どちらを優先するかなどの問題) を回避している．しかし，実際問題として，2 つの性質を継承するクラスは非常に必要性が高い．Java では，インタフェースと呼ばれる一種の抽象クラスが用いられる．

多重継承 (multiple inheritance)： 複数のクラスを継承すること
Java では許されていない．

インタフェース (interface)： 非常に限定された抽象クラス (のようなもの)

public，static，final なクラス定数変数と public，abstract な抽象メソッドのみで構成される．実際のメソッドはインタフェースを実装するクラスで定義される．

インタフェース宣言： インタフェース名/インタフェースの内容 (クラス定数変数および抽象メソッド) を宣言する
インタフェース宣言されたインタフェース名は型名となる

```
インタフェース修飾子　interface　インタフェース名 {
  クラス定数変数 (フィールド) 宣言並び
  抽象メソッド宣言並び
}
```

インタフェース修飾子： インタフェースの性質を指定する
public　他のパッケージ (ファイル) から参照できる

クラス定数変数宣言： インタフェースの持つクラス定数変数を宣言する

変数修飾子　型名　変数名 ＝ 初期値 ；

変数修飾子： 変数の性質を指定する
インタフェースの変数は，public，static，final なクラス変数定数に限られ，通常は変数修飾子は省略される．

抽象メソッド宣言： インタフェースの持つ抽象メソッドを宣言する

メソッド修飾子 型名 メソッド名（仮引数並び）；

メソッド修飾子： メソッドの性質を指定する
インタフェースのメソッドは，public，abstract な抽象メソッドに限られ，通常はメソッド修飾子は省略される．

インタフェースの実装： インタフェースは他のクラスで実装される
あるクラスが，インタフェースの性質を継承することを,「インタフェースを実装する」と言う．インタフェース単独では，オブジェクトを作ることはできず，他のクラスで実装されることによってインスタンス化される．この際，インタフェースが実装されるクラスにおいては，全ての抽象メソッドが実装されなくてはならない．
インタフェース A, B を実装するクラス X は，次のように定義される

```
class　クラス X　implements　インタフェース A，インタフェース B　{ }
```

クラスを継承する際に extends と書いた代りに implements と書くが，クラスの継承とは異なり，複数のインタフェースを実装できる．ただし，以下の条件を満たさないとコンパイルエラーとなる．

- 2 つのインタフェースが同じ名前のクラス定数変数を持ってはならない
- 2 つのインタフェースが同じシグネチャの抽象メソッドを持つ場合，戻し値が同じでなくてはならない

インタフェースの継承： 他のインタフェースを継承して新しいインタフェースを定義できる
インタフェースA, Bを継承するインタフェースXは，次のように定義される

```
interface インタフェースX extends インタフェースA, インタフェースB { }
```

クラスの継承とは異なり，インタフェースは複数のインタフェースを継承できる．ただし，以下の条件を満たさないとコンパイルエラーとなる．

- 2つのスーパインタフェースが同じ名前のクラス定数変数を持ってはならない
- 2つのスーパインタフェースが同じシグネチャの抽象メソッドを持つ場合，戻し値が同じでなくてはならない

この授業ではインタフェースの継承は扱わない．

例1： ソーティング可能インタフェース - Sortable.java
2つのオブジェクト (インスタンス) の大小関係が比較できれば，ソーティング可能である5.2.1節の例3と5.3.4節の例1を比較すると分かるように，greaterThanというメソッドが異なるだけで，実際のソーティングのプログラム (sortメソッド) は殆んど同じである．しかし，比較するデータの型 (greaterThanメソッドの引数の型) が，それぞれ
Prefecture型とdouble型であるため，sortメソッドの引数や変数tmpもPrefecture型とdouble型になっている．
そこで，ソーティング可能なオブジェクトに共通の性質として，Sortableというインタフェースを提供する．

```
public interface Sortable {
  boolean greaterThan(Sortable another);
}
```

例2： 最小値選択法のプログラム - SimpleSort.java
ソート対象となるデータをSortable型としてプログラムを作る．

```
public class SimpleSort {
  public static void sort(Sortable[] s) { // Sortable型データのソート
    for (int i = 0; i < s.length-1; i++) {
      int min = i;
      for (int j = i+1; j < s.length; j++)
        if (s[min].greaterThan(s[j]))     // greaterThanメソッドで比較
          min = j;
      Sortable tmp = s[i];  s[i] = s[min];  s[min] = tmp;
    }
  }
}
```

例３： ソーティング時間の計測プログラム - SortingTime.java
インタフェース Sortable を実装している．Sortable.class，SimpleSort.class とともにコンパイルすること．コメントされている部分を切替えることによって，ソーティング方法を変更できる．

```
public class SortingTime implements Sortable {
  private double value;
  SortingTime(double value) {
    this.value = value;
  }
  public boolean greaterThan(Sortable another) {
    return  this.value > ((SortingTime)another).value;
  }
  public static void main(String[] args) {
    final int TIMES = 10;
    long t = 0;
    SortingTime[] st = new SortingTime[Integer.parseInt(args[0])];
    for (int i = 0; i < TIMES; i++) {
      for (int j = 0; j < st.length; j++)
        st[j] = new SortingTime(Math.random());
      System.out.print("Start sorting #" + i + " ... ");
      t -= System.currentTimeMillis();
      SimpleSort.sort(st);
      // BubbleSort.sort(st);
      // QuickSort.sort(st);
      // MergeSort.sort(st);
      t += System.currentTimeMillis();
      System.out.println("... finished.");
    }
    System.out.println("Average time for sorting " + st.length +
                       " elements: " + (t/TIMES) + " [msec]");
  }
}
```

例４： Prefecture データのソーティング - SortPrefecture.java
クラス Prefecture を拡張し，インタフェース Sortable を実装している．Sortable.class，SimpleSort.class，Prefecture.class とともにコンパイルすること．

```
public class SortPrefecture extends Prefecture implements Sortable {
  public SortPrefecture(Prefecture p) {
    super(p.name, p.area, p.male, p.female);
  }
  public boolean greaterThan(Sortable another) {
    return  this.area > ((SortPrefecture)another).area;
```

```
    // return  this.name.compareTo(((SortPrefecture)another).name) > 0;
    // return  this.popDensity() > ((SortPrefecture)another).popDensity();
    // 以上のコメントを切替えることで，ソーティングのキーを変更できる
  }
  public static void main(String[] args) {
    Prefecture[] p = Prefecture.loadFile(args[0]);
    // loadFile メソッドは Prefecture オブジェクトの配列を作るので,
    // ソーティング可能な SortPrefecture オブジェクトの配列を作り直す
    SortPrefecture[] sp = new SortPrefecture[p.length];
    for (int i = 0; i < sp.length; i++)
      sp[i] = new SortPrefecture(p[i]);
    SimpleSort.sort(sp);
    // BubbleSort.sort(sp);
    // QuickSort.sort(sp);
    // MergeSort.sort(sp);
    // 以上のコメントを切替えることで，ソーティングアルゴリズムを変更できる
    System.out.println("Name \t\t  Area \t\tPopulation \t  Pop. Density");
    for (int i = 0; i < sp.length; i++)
      System.out.println(sp[i].name() + "     \t " + sp[i].area() + " \t " +
                         sp[i].population() + " \t " + sp[i].popDensity());
  }
}
```

第3章　情報量と文字コード

3.1　n進数

n進数： nを1単位(桁)とする数(各桁にn種類の文字が使われる)

10進数： 各桁が0〜9の10種類の数字によって表現される数

2進数： 各桁が0か1の2種類の数字によって表現される数
計算機内部では，情報はON/OFF (1/0)のビット列で表現されるので2進数が便利である．

16進数： 各桁が0〜9と`A`〜`F`の16種類の数字によって表現される数
16進数の1桁が2進数の4桁に対応することから，よく用いられる．2桁で丁度8ビット(1バイト)を表現できる．

3.1.1　n進数から10進数への変換

10進数の場合：

$$\begin{aligned} A_nA_{n-1}\cdots A_2A_1A_{0(10)} &= A_n\times 10^n + A_{n-1}\times 10^{n-1}\cdots + A_2\times 10^2 + A_1\times 10^1 + A_0\times 10^0 \\ &= \sum_{i=0}^{n} A_i\times 10^i \end{aligned}$$

例：　$1024_{(10)} = 1\times 10^3 + 0\times 10^2 + 2\times 10^1 + 4\times 10^0$

2進数の場合：

$$\begin{aligned} A_nA_{n-1}\cdots A_2A_1A_{0(2)} &= A_n\times 2^n + A_{n-1}\times 2^{n-1}\cdots + A_2\times 2^2 + A_1\times 2^1 + A_0\times 2^0 \\ &= \sum_{i=0}^{n} A_i\times 2^i \end{aligned}$$

例：　$1011_{(2)} = 1\times 2^3 + 0\times 2^2 + 1\times 2^1 + 1\times 2^0 = 11_{(10)}$

16進数の場合：

$$\begin{aligned} A_nA_{n-1}\cdots A_2A_1A_{0(16)} &= A_n\times 16^n + A_{n-1}\times 16^{n-1}\cdots + A_2\times 16^2 + A_1\times 16^1 + A_0\times 16^0 \\ &= \sum_{i=0}^{n} A_i\times 16^i \end{aligned}$$

例：　$\mathtt{1B7}_{(16)} = 1\times 16^2 + 11\times 16^1 + 7\times 16^0 = 439_{(10)}$

3.1.2　10進数から2進数への変換

10進数を，2で繰り返し割って，余り (0または1) をとる．

例：

2	25	
2	12	$\cdots 1$
2	6	$\cdots 0$
2	3	$\cdots 0$
2	1	$\cdots 1$
	0	$\cdots 1$

†　$25_{(10)} = 11001_{(2)} (= 16 + 8 + 1)$

3.2　ビット (bit) とバイト (byte)

ビット/バイト： 情報量の単位

ビット： 1ビット $=$ 2進数の1桁分の情報量

n ビットは2進数 n 桁分 (2^n 種類) の情報量

バイト： 1バイト $=$ 8ビット $(= 2^8 = 256$ 種類の情報量)

キロバイト (Kbyte)： 1,024バイト $= 2^{10}$ バイト (約1,000バイト)

メガバイト (Mbyte)： 1,024キロバイト (約1,000キロバイト $=$ 1,000,000バイト)

ワード (word)： 計算機 (CPU) が1度に処理できる情報量 (計算機 (CPU) に固有の値)

例：16ビットパソコン，32ビットパソコン

3.3　整数データ表現

整数データは，計算機内のビット列を2進数として解釈して扱われる．すでに説明したように n ビットであれば，2^n 個の整数値が表現できる．したがって，8ビットであれば 2^8 個，16ビットであれば 2^{16} 個，32ビットであれば 2^{32} 個となる．

符号なし整数 (unsigned)： 符号を持たない，正の整数のみの表現

n ビットで0から $2^n - 1$ までを表現する

$00\ldots0_{(2)}$	$\cdots$	$01\ldots1_{(2)}$	$10\ldots0_{(2)}$	$\cdots$	$11\ldots1_{(2)}$
0	$\cdots$	$2^{n-1}-1$	2^{n-1}	$\cdots$	2^n-1

2の補数表現： 一般的な符合付き整数表現

最上位ビットが1の場合，残りの $n-1$ ビットから 2^{n-1} を引いた負の値となる (別の言い方をすると符号なし整数の値から 2^n を引いた値を表すものとする)

n ビットで -2^{n-1} から $2^{n-1}-1$ までを表現する

$10\ldots0_{(2)}$	$\cdots$	$11\ldots1_{(2)}$	$00\ldots0_{(2)}$	$\cdots$	$01\ldots1_{(2)}$
-2^{n-1}	$\cdots$	-1	0	$\cdots$	$2^{n-1}-1$

1 の補数表現： 最上位ビットを単なる符号とみなす整数表現

n ビットで $-2^{n-1}+1$ から $2^{n-1}-1$ までを表現する (通常は利用されない)

$10\ldots0_{(2)}$	$\cdots$	$11\ldots1_{(2)}$	$00\ldots0_{(2)}$	$\cdots$	$01\ldots1_{(2)}$
$-2^{n-1}+1$	$\cdots$	-0	0	$\cdots$	$2^{n-1}-1$

オーバフロー (overflow) 数値の表現範囲を越えて正確な値でなくなること

たとえば，n ビットの符合付き整数 (2 の補数表現) で，$2^{n-1}-1$ に 1 を加えると，一般に -2^{n-1} となってしまう

3.4 文字コード

文字コード： 計算機内では文字も 2 進数で表現される

半角文字コード (英字，数字，カナ)： 1 バイト (256 種類)
漢字コード： 2 バイト (65536 種類)

例： 新聞 n 枚分の情報量 $= 2 \times$ 新聞 1 枚当たりの文字数 $\times\, n$ (バイト)

ASCII(American Standard Code for Information Interchange) コード： アメリカ標準協会の制定した英数字の文字コード

JIS(Japan Industrial Standard) コード： 日本工業規格の英数字と漢字の文字コード

漢字の文字コードには，JIS コード，SJIS(Shift JIS) コード，EUC(Extended UNIX Code) コード，UTF-8(UCS Transformation Format) コードなどがある．

漢字コード変換： 漢字コードを変えること

次の 2 通りの方法がある．

1. nkf コマンド：
   オプションによって JIS(-j)，EUC(-e)，SJIS(-s) を切り替える．
   標準出力に出力されるので通常はリダイレクションを用いる．

   ```
   ca10101$ nkf -j original > original.jis ¶   (JIS コードへの変換)
   ca10101$ nkf -e original > original.euc ¶   (EUC コードへの変換)
   ca10101$ nkf -s original > original.sjis ¶  (SJIS コードへの変換)
   ca10101$ nkf -w original > original.utf ¶   (UTF-8 コードへの変換)
   ```

2. Emacs の漢字コード変換：
   Emacs のウィンドウ左下の `[--]?.:` の ? でコードがわかる．
   以下の操作でセーブ時の漢字コードを指定する．

   ```
   C-x <enter> f → Coding system for visited file (default, nil): ????
   ¶
                                                    (漢字コードの指定)
   ```

   ここで ????部には，以下のいずれかを指定する．

   ```
   euc-japan : EUC ,   junet : JIS ,   sjis : SJIS
   ```

   (指定するときにスペースキーを打てば可能なものの一覧が表示される)

3.4.1 JIS半角(1バイト)文字コード表

		下 0	1	2	3	位 4	5	4 6	7	ビ 8	9	ッ A	B	ト C	D	E	F
	0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
	1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
	2		!	"	#	$	%	&	’	(	)	*	+	,	−	.	/
上	3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
	4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
位	5	P	Q	R	S	T	U	V	W	X	Y	Z	[	¥	]	^	_
	6	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
4	7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	
	8																
ビ	9																
	A		。	「	」	、	・	ヲ	ァ	ィ	ゥ	ェ	ォ	ャ	ュ	ョ	ッ
ッ	B	−	ア	イ	ウ	エ	オ	カ	キ	ク	ケ	コ	サ	シ	ス	セ	ソ
	C	タ	チ	ツ	テ	ト	ナ	ニ	ヌ	ネ	ノ	ハ	ヒ	フ	ヘ	ホ	マ
ト	D	ミ	ム	メ	モ	ヤ	ユ	ヨ	ラ	リ	ル	レ	ロ	ワ	ン	゛	゜
	E																
	F																

JISコード(16進)：	21	37	4A	63	B1	DE
JISコード(10進)：	33	55	74	99	177	222
文　字　：	!	7	J	c	ア	゛

第4章　数値計算と数値誤差

数値計算は，代数演算とは，かなり様子が異なる．代数では等式を一定の規則のもとに変形して解析的に求める．これに対して，一般に数値計算では，一般に反復計算によって (近似的に) 解を求める．この際，2 進数 (による近似) 表現に由来する有効桁数や計算方法による誤差などに注意すべきである．

4.1　二分探索法

中間値の定理を応用する方法．方程式 $f(x) = 0$ の解を計算する際に，$f(a)$ と $f(b)$ の符号が異なれば，区間 $[a, b]$ の間に解が存在する．図 4.1 を見て分かるように，a と b の中央の値である $m = \underline{(a + b)/2}$ における $\underline{f(m)}$ の符号を調べることによって，解の存在する区間を $\underline{[a, m]}$ ないし $\underline{[m, b]}$ の一方に狭められる．このいずれかを新しい区間として，さらに区間を狭めることができる．この操作を反復することで，解の存在範囲を狭めていく．解の存在範囲が十分に小さくなった ($\underline{b - a \leq \varepsilon}$) ところで，最終的に解が求まったものとする．この方法では，予め解の存在する区間 $[a, b]$ ($f(a)$ と $f(b)$ の符号が異なる 2 つの値 a, b) が与えられなくてはならない．

例１： 二分探索法による $x^3 - 1 = 0$ の解 - `Bisect.java`

```
public class Bisect {
   static double funct(double x) {
      return x*x*x - 1.0;
   }
   public static void main(String[] args) {
      final double EPS = 1.0e-13;
      double a = Double.parseDouble(args[0]);
      double b = Double.parseDouble(args[1]);
      double fa = funct(a);
      double fb = funct(b);
      if (fa*fb >= 0.0) {
         System.out.println("Bad numbers result in the same sign:");
         System.out.print("  funct(" + a + ") = " + fa + ", ");
         System.out.println("funct(" + b + ") = " + fb);
      }
      else { // 以下の while ループが二分探索法
         while (              ) {
            double m =
```

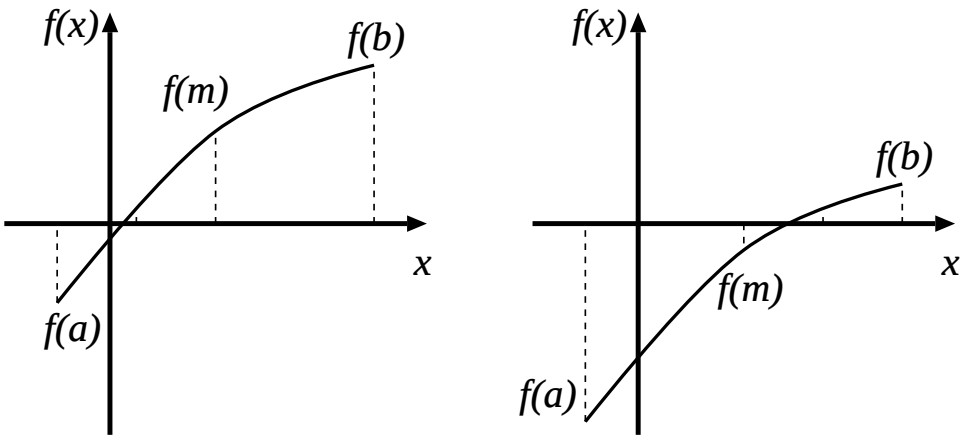


図 4.1: 二分探索法の原理

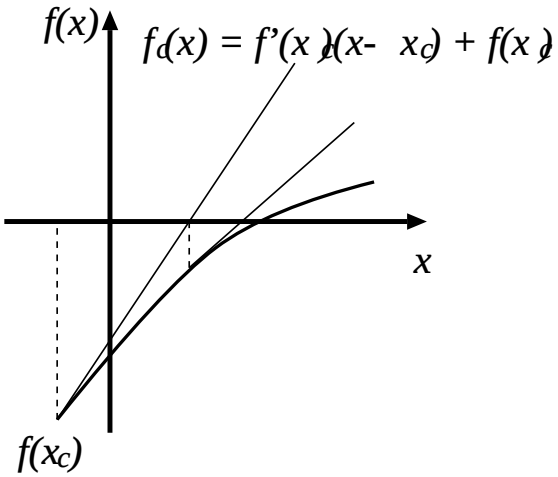


図 4.2: ニュートン法の原理

```
            double fm = [          ]
            // 次の 1 行は探索過程を示すためのもの
            System.out.println("funct(" + m + ") = " + fm);
            if (fa*fm > 0.0) {
               [ ] = m;  [ ] = fm;
            }
            else {
               [ ] = m;  [ ] = fm;
            }
         }
         System.out.println("The answer is " + ((a+b)/2.0));
      }
   }
}
```

4.2 ニュートン法

関数 $f(x)$ を 1 次近似し，その解を求めることを繰り返すことで，速く解を求める方法．方程式 $f(x) = 0$ の解の計算を考える．今，図 4.2 のように $x = x_c$ において，関数値 $f(x_c)$ と導関数値 $f'(x_c) = \frac{df(x_c)}{dx}$ の値が得られるとする．このとき，$x = x_c$ における $f(x)$ の 1 次近似 $f_c(x)$ は，

$$f_c(x) = f'(x_c)(x - x_c) + f(x_c)$$

で与えられる．この 1 次近似式 $f_c(x) = 0$ の解は，

$$x = x_c - \frac{f(x_c)}{f'(x_c)}$$

となる．これを新しい近似解 x_c として，反復することで $f(x) = 0$ の解に急速に近づいてゆく．x_c と $f_c(x) = 0$ の解の差が十分に小さくなった ($|x - x_c| \leq \varepsilon$) ところで，最終的に解が求まったものとする．関数の値とともに，その導関数の値が計算できないと，ニュートン法は利用できない．また，最初に適切な第一近似 (first guess) が与えられないと，所望の解に収束しないことがある．

例 1： ニュートン法による $x^3 - 1 = 0$ の解 - `Newton.java` *

```
public class Newton {
   static double funct(double x) {
      return  x*x*x - 1.0;
   }
   static double deriv(double x) {
      return  3.0*x*x;
   }
   public static void main(String[] args) {
      final double EPS = 1.0e-13;
      double x = Double.parseDouble(args[0]);
      double xc;
      do { // この do ループがニュートン法
         xc = 
         double f = 
         double d = 
         // 次の 1 行は探索過程を示すためのもの
         System.out.println("funct(" + xc + ") = " + f);
         x = 
      } while (Math.abs                );
      System.out.println("The answer is " + x);
   }
}
```

*`Math.abs(x)` は，x の絶対値 $|x|$ を計算するメソッド．

4.3　実数データ表現 (浮動小数点数表現)

計算機内の数値データは，有限のビット列で表現される．大きな数や小さな数を表現するためには，符号部と指数部，仮数部をもった実数表現が利用される．

実数表現 符号部，指数部，仮数部からなる表現方法
浮動小数点数表現と呼ばれることもある

IEEE 754 実数表現に関する規格 (標準)
多くの計算機で利用されている実数表現で，以下のような表現形式である

$$(-1)^{s_{(2)}} 1.d_1 d_2 \cdots d_{m(2)} \times 2^{d'_1 d'_2 \cdots d'_{c(2)} - b} \quad (s, d_i, d'_i \in \{0, 1\})$$

$s_{(2)}$ が符号，$1.d_1 d_2 \cdots d_{m(2)}$ が仮数，$d'_1 d'_2 \cdots d'_{c(2)} - b$ が指数となる．また b はバイアスと呼ばれ，指数が負値を採れるようにするためのものである．
32 ビットで表現する単精度と 64 ビットで表現する倍精度がある．

	符号部	指数部	仮数部	m	c	b
単精度 (32 ビット)	第 0 ビット	第 1〜8 ビット	第 9〜31 ビット	23	8	127
倍精度 (64 ビット)	第 0 ビット	第 1〜11 ビット	第 12〜63 ビット	52	11	1023

4.4　丸め誤差

$0.1_{(10)}$ を表そうとすると，

$$0.1_{(10)} = 2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + \cdots = 0.000110011\cdots_{(2)} \times 2^0 = 1.10011\cdots_{(2)} \times 2^{-4}$$

となり，有限桁では近似的にしか表現できない．(10 進の小数で 1/3 や 1/7 を表す場合に相当している)

単精度表現 仮数部 23 ビット，指数部 8 ビット
仮数の表現範囲は，$1.0\ldots0_{(2)} = 1$ から $1.1\ldots1_{(2)} = 2 - 2^{-23} \simeq 2 - 1.2 \times 10^{-7}$ であり，有効桁数は 24 ビットとなる．これは 10 進数で 7 桁程度の精度しかないことを意味している
指数の表現範囲は，指数部のビットがすべて 0 か 1 の場合は特別な意味を持っているので，$2^{00000001_{(2)}-127} = 2^{-126} \simeq 1.2 \times 10^{-38}$ から $2^{11111110_{(2)}-127} = 2^{127} \simeq 1.7 \times 10^{38}$ となる．仮数部の最大値は概ね 2 であるから，最大値の限界は $2^{128} \simeq 3.4 \times {}^{38}$ となる．

倍精度表現 仮数部 52 ビット，指数部 11 ビット
仮数の表現範囲は，$1.0\ldots0_{(2)} = 1$ から $1.1\ldots1_{(2)} = 2 - 2^{-52} \simeq 2 - 2.2 \times 10^{-16}$ であり，有効桁数は 53 ビット (10 進数で 16 桁程度) となる．指数の表現範囲は，$2^{0\ldots01_{(2)}-1023} = 2^{-1022} \simeq 2.2 \times 10^{-308}$ から $2^{1\ldots10_{(2)}-1023} = 2^{1023} \simeq 9.0 \times 10^{307}$ となる．最大値の限界は $2^{1024} \simeq 1.7 \times {}^{308}$ となる．

例１： 0.1 の計算機内表現 - `PointOne.java`

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

```
    float  f = 0.1f;                // 0.1f は float 型の 0.1 を意味する
    double d = 0.1;
    System.out.println("f \t\t = " + f);
    System.out.println("d \t\t = " + d);
    System.out.println("f*1.0e9 \t = " + (f*1.0e9));
    System.out.println("d*1.0e9 \t = " + (d*1.0e9));
    System.out.println("(f-d)*1.0e18 \t = " + ((f-d)*1.0e18));
    System.out.println("(f-0.1)*1.0e18 \t = " + ((f-0.1)*1.0e18));
    System.out.println("(f-0.1f)*1.0e18  = " + ((f-0.1f)*1.0e18));
  }
}
```

4.5 桁落ち

計算機の数値計算では，誤差が拡大する可能性がある．大きく分けると桁落ち誤差と情報落ち誤差がある．

桁落ち誤差 近い数値間の減算による誤差

ほぼ同じような数値の差をとると，有効桁数が減少する．

例１： 微分値の数値近似 - Derivative.java

数学的な定義 ($\frac{df}{dx}(x) = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon}$) から言えば，eps$\to 0$ の極限が微分値となるはず．

```
public class Derivative {
  static float funct(float x) {
    return  x*x/2.0f;
  }
  public static void main(String[] args) {
    final float X = 1.0f;
    float eps = 0.1f;
    for (int i = 0; i < 10; i++) {
      System.out.println("eps = " + eps + ",\t f'(" + X + ") = " +
                         ((funct(X+eps)-funct(X))/eps));
      eps *= 0.1f;
    }
  }
}
```

4.6 情報落ち

情報落ち 大きさの異なる数値間の加減算による誤差

大きさの異なる数値の間で加算や減算を行なうと，小さな数値は大きな数値の有効桁

範囲外になってしまい，値が無視されてしまう．

例１： 1 の分割 - Partition.java

```
public class Partition {
  public static void main(String[] args) {
    int n = 1;
    for (int i = 0; i < 10; i++) {
      float sum = 0.0f;
      float d = 1.0f / n;
      for (int j = 0; j < n; j++)
        sum += d;
      System.out.println("n = " + n + ",\t d = " + d);
      System.out.println("   sum = " + sum);
      n *= 10;
    }
  }
}
```

4.7　打切り誤差

一般に計算機は直接に連続関数を扱えないため，何らかの近似計算を行なうことになる．たとえば，e^x $(x \in R)$ を計算しようとすると，以下の計算式などに頼らざるを得ない．

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} = 1 + x + \frac{x^2}{2!} + \cdots$$

しかし，無限回の計算を行なうわけには行かないので，有限回 (n 回) で打ち切らざるをえない．つまり，

$$e^x = \sum_{i=0}^{n} \frac{x^i}{i!}$$

と近似せざるをえない．このときの誤差の主要な成分は，

$$\frac{x^{n+1}}{(n+1)!}$$

と考えられる．特に $x = 1$ の時は，自然対数の底 e を求める計算となるが，それを確認してみよう．

例１： 指数計算 (自然対数の底) - Exponent.java †

```
public class Exponent {
  public static void main(String[] args) {
    double x = Double.parseDouble(args[0]);
    double xi = 1.0, fi = 1.0, ex = 1.0;
```

†Math.exp(x) は，e^x を計算するメソッド．

```
        System.out.println("From Math -> " + Math.exp(x));
        System.out.println(" i \t Ans. \t\t\t Err. \t\t\t Est.");
        for (int i = 1; i <= 20; i++) {
          xi *= x;
          fi *= i;
          ex += xi/fi;
          System.out.println(" " + i + "\t" + ex + "\t" + (Math.exp(x)-ex) +
                             "\t" + ((xi*x)/(fi*(i+1))));
        }
      }
    }
```

4.8 誤差の見積り

一般に誤差を見積もることは非常に難しい．次式を念頭に入れて，積分計算を行なってみよう．

$$\int_0^{\pi} \sin x dx = [-\cos x]_0^{\pi} = 2$$

純粋に数学として考えれば，分割を細かくすれば良いはずだが，結果はどうなるだろうか？

4.8.1 台形公式

積分を区分線形 (piecewise linear) 近似する方法．全積分区間をを n 等分し，各部分区間 Δx の積分を台形の面積として計算して総和をとる．

$$\begin{aligned}\int_{x_s}^{x_e} f(x)dx &\approx \sum_{i=0}^{n-1} \frac{1}{2}\{f(x_s+i\Delta x)+f(x_s+(i+1)\Delta x)\}\Delta x \\ &= \underline{\Delta x}\left\{\frac{f(x_s)+f(x_e)}{\underline{\quad 2 \quad}} + \sum_{i=1}^{n-1}\underline{f(x_s+i\Delta x)}\right\}\end{aligned}$$

ただし，$\Delta x = \dfrac{x_e - x_s}{\underline{\quad n \quad}}$ とする．

例１： $\int_0^{\pi} \sin x dx$ の計算[‡]

```
public class Trapezoid {
   static float funct(float x) {
      return (float)(Math.sin((double)x));
   }
   public static void main(String[] args) {
      final float xs = 0.0f;
      final float xe = (float)Math.PI;
      int n = 10;
      for (int j = 1; j < 9; j++) {
```

[‡] `Math.sin(x)` は，$\sin x$ を計算するメソッド．

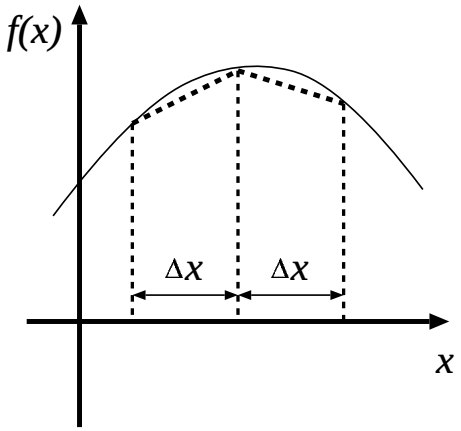


図 4.3: 台形公式による近似積分

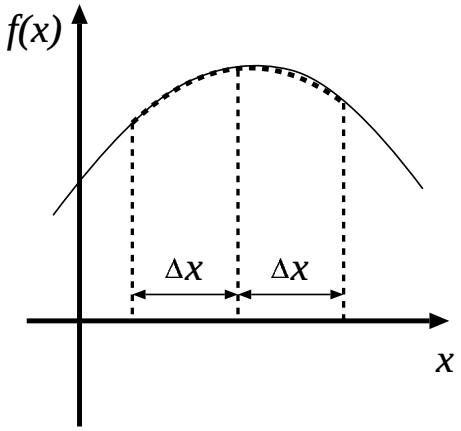


図 4.4: シンプソン公式による近似積分

```
            // 以下の 5 行が台形公式による積分計算
            float dx =
            float sum =
            for (int i = 1; i < n; i++)
               sum +=
            sum *=
            System.out.println("n = " + n + ", result = " + sum);
            n *= 10;
        }
    }
}
```

4.8.2　シンプソン公式

積分を 2 次式で近似する方法．線形近似 (台形公式) の場合の 1 区間 (Δx) の積分は

$$\frac{1}{2}\{f(x) + f(x + \Delta x)\}\,\Delta x$$

となるが，2 次近似の場合，1 区間 $(2\Delta x)$ の積分は

$$\frac{1}{3}\left\{f(x)+4f(x+\Delta x)+f(x+2\Delta x)\right\}\Delta x$$

となる．ただし，$\Delta x=\dfrac{x_e-x_s}{2n}$ は 1 区間の半分である．したがって，

$$\begin{aligned}\int_{x_s}^{x_e} f(x)dx &\approx \sum_{i=0}^{n-1}\frac{1}{3}\left\{f(x_s+2i\Delta x)+4f(x_s+(2i+1)\Delta x)+f(x_s+(2i+2)\Delta x)\right\}\Delta x\\ &= \frac{\Delta x}{3}\left\{f(x_s)+4f(x_s+\Delta x)+f(x_e)+\sum_{i=1}^{n-1}\left(2f(x_s+2i\Delta x)+4f(x_s+(2i+1)\Delta x)\right)\right\}\end{aligned}$$

となる．

第5章　サーチとソート

複数のデータから特定のデータを見つけ出す操作を「サーチ (search)」，複数のデータを特定の順序で並べ変える操作を「ソート (sort)」と言う．サーチやソートは，複数データに対する基本的な操作であり，様々なアルゴリズムが知られている．ここでは，サーチやソートを題材として，計算量などについて考えてみたい．
参考までに，Java2 SDK 1.4 には，サーチやソートのための `Arrays` クラスや，`Comparable`, `Comparator` などのインタフェースがある．

5.1　サーチ

サーチ (search)： 複数データから特定の条件を満たすものを探すこと
　サーチには様々な手法 (アルゴリズム＝計算法) がある

キー (key)： サーチやソートする際の判定属性

線形サーチ (linear search)： n 個のデータで比較回数 $O(n)$ のアルゴリズム

バイナリサーチ (binary search)： n 個のデータで比較回数 $O(\log n)$ のアルゴリズム
　線形サーチよりは効率が良いが，予めデータがソートされていなくてはならない

以下では，県の配列データから県名をキーとしてサーチし，その県の情報を表示するプログラムを考える．

5.1.1　線形サーチ

配列 `p[i]` を端から順々に調べ，条件に合うものが見つかるまで繰り返す．
線形サーチは，サイズ n の配列に対して，平均 $n/2$ 回の判定でサーチが終了する．これは n の次数 (order) で言えば，n に比例した計算であるので，線形サーチの計算量は $O(n)$ となる．

例１： 線形サーチ - `LinearSearch1.java`
　2.29 節「クラスとインスタンス」の例５のクラスライブラリが必要なので，クラスファイル (Prefecture.class) を当該ディレクトリにコピーすること
　また対象データとして CFIVE の `Japan.dat` を用いよ

```
public class LinearSearch1 {
  public static void main(String[] args) {
    Prefecture[] p = Prefecture.loadFile(args[0]);
    Prefecture cp = search(p, args[1]);
    if (cp != null)
```

```
      cp.print(p);
    else
      System.out.println("Invalid name -> " + args[1]);
  }
  static Prefecture search(Prefecture[] p, String str) {
    for (int i = 0; i < p.length; i++)
      if (str.equals(p[i].name()))  return p[i];
      // if (p[n].name().equals(str))  return p[i]; でも良い
    return null;
  }
}
```

<実行結果>

```
ca10101$ java LinearSearch1 Japan.dat Tokyo ¶
Tokyo:
  Area  = 2102.14 [km^2]
  Pop.  = 11639000 (5795000(m)/5844000(f)) [persons]
  Dens. = 5536.738751938501 [persons/km^2]
  Area Rank = 45,        Area Ratio = 0.005730467501676294
  Pop. Rank = 1,         Pop. Ratio = 0.09292466387762272
  Pop. Density Rank = 1
```

5.1.2 バイナリサーチ

検索対象データがキーに関してソートされている (たとえばデータが県名のアルファベット順で並べられている) 場合には，バイナリサーチによって効率良く見つけ出せる．
バイナリサーチは，サイズ n の配列に対して，1 回の判定ごとに配列のサイズが $n/2$ になる．したがって，最終的には $\log_2 n$ 回の判定を繰り返せば，サーチが終了するはずである．したがって，バイナリサーチの計算量は $O(\log n)$ となる (情報に関する議論では log の底は 2 である)．

1. 下限の lower と上限 upper を含まずに，lower と upper の間に求めるデータがあると仮定するので，lower と upper の差が 2 以上でなければ求めるデータは存在しない
2. データがある場合は，中間の middle と探しているデータのキーを比較する
   (a) middle が求めるものならば，それを答として返す
   (b) middle より前にあるはずならば，lower と middle の間で (端は含めない) 探す
   (c) middle より後ろにあるはずならば，middle と upper の間で (端は含めない) 探す

例１： バイナリサーチ (再帰版) - BinarySearch1.java
2.29 節「クラスとインスタンス」の例 5 のクラスライブラリが必要である
対象データとしては CFIVE の SortedJapan.dat を用いよ

```
public class BinarySearch1 {
  public static void main(String[] args) {
    Prefecture[] p = Prefecture.loadFile(args[0]);
    Prefecture cp = search(p, args[1]);
    if (cp != null)
      cp.print(p);
    else
      System.out.println("Invalid name -> " + args[1]);
  }
  static Prefecture search(Prefecture[] p, String str) {
    return bsearch(p, str, -1, p.length);
              // 下限 lower と上限 upper は含まないので，1 つズレた値を用いる
  }
  static Prefecture bsearch(Prefecture[] p, String str, int lower, int upper) {
    if ((upper - lower) < 2)
      return null;
    else {
      int middle = (lower + upper) / 2;
      int diff = str.compareTo(p[middle].name());
      if (diff == 0)      return p[middle];
      else if (diff < 0)  return bsearch(p, str, lower, middle);
      else                return bsearch(p, str, middle, upper);
    }
  }
}
```

<実行結果>

```
ca10101$ java LinearSearch1 SortedJapan.dat Tokyo ¶
Tokyo:
  Area  = 2102.14 [km^2]
  Pop.  = 11639000 (5795000(m)/5844000(f)) [persons]
  Dens. = 5536.738751938501 [persons/km^2]
  Area Rank = 45,        Area Ratio = 0.005730467501676294
  Pop. Rank = 1,         Pop. Ratio = 0.09292466387762272
  Pop. Density Rank = 1
```

5.2 ソート

ソート (sort)： データを特定の順に並べること
　ソートには様々な手法 (アルゴリズム＝計算法) がある

最小値選択法，バブルソート： $O(n^2)$ のアルゴリズム

クィックソート，マージソート： $O(n \log n)$ のアルゴリズム

以下ではキーが昇順になるように，配列データを並べ変えるプログラムを考える．

5.2.1 最小値選択法

配列 p[i] の中から最小のものを探して前に持ってくることを繰返す
単純ソートと呼ばれることもある

1. まず先頭 (p[0]) と残りの要素 (p[1]〜p[p.legnth-1]) を比較して，p[0] より小さい要素があったら交換して，最終的に最も小さい値を p[0] に格納する
2. 次に p[1] と残りの要素 (p[2]〜p[p.legnth-1]) を比較して，次に小さい値を p[1] に格納する
3. 同様に p[i] と残りの要素 (p[i+1]〜p[p.legnth-1]) を比較して，その中で最も小さい値を p[i] に格納する
4. i = p.legnth-2 まで繰返せば，最後の 1 つは最大値で p[p.legnth-1] に格納される (ソート完了)

ソートの過程 (図では下が前であり，左から右へと変化していく)

4	4	4	4	**4**	4	4	4	**4**	4	4	**4**	...
6	6	6	**6**	6	6	6	**6**	6	6	**6**	6	...
1	1	**1**	2	2	2	**2**	3	3	**3**	5	5	...
3	**3**	3	3	3	**3**	5	5	5	**5**	**3**	**3**	...
5	5	5	5	5	**5**	**3**	**2**	**2**	2	2	2	...
2	**2**	**2**	**1**	**1**	1	1	1	1	1	1	1	...

例１： 最小値選択法 (県データの面積順 (昇順) ソート) - SimpleSort1.java
このプログラム (この後のソートのプログラム) にも 2.29 節「クラスとインスタンス」の例 5 のクラスライブラリが必要である
対象データとしては CFIVE の Japan.dat を用いよ

```
public class SimpleSort1 {
  public static void main(String[] args) {
    Prefecture[] p = Prefecture.loadFile(args[0]);
    sort(p);
    System.out.println("Name \t\t  Area \t\tPopulation \t  Pop. Density");
    for (int i = 0; i < p.length; i++)
      System.out.println(p[i].name() + "     \t " + p[i].area() + " \t " +
                         p[i].population() + " \t " + p[i].popDensity());
  }
  static boolean greaterThan(Prefecture p0, Prefecture p1) {
    return  p0.area() > p1.area();
  }
```

```
  static void sort(Prefecture[] p) {
    for (int i = 0; i < p.length-1; i++)
      for (int j = i+1; j < p.length; j++)
        if (greaterThan(p[i], p[j])) {
          Prefecture tmp = p[i];  p[i] = p[j];  p[j] = tmp;
        }
  }
}
```

注意： 変数置換

変数置換 (データの入替え) を行なう場合には，データを仮に保持する変数を用意して代入を 3 回行なう必要がある

a と b の入替え

× `a = b; b = a;` `// a,b ともに元の b の値になる`

○ `tmp = a; a = b; b = tmp;`

例 2： 最小値選択法 (県データの名前順ソート)

ソートのキーを面積ではなく名前にして，名前の順 (昇順) にデータをソートする

```
/* クラスメソッド greaterThan 以外は例 1 と同じ */
  static boolean greaterThan(Prefecture p0, Prefecture p1) {
    return  p0.name().compareTo(p1.name()) > 0;
  }
```

例 3： 変数置換の回数を減らした最小値選択法

最少データの番号 (添字 min) を記録して，置換は最後の 1 回のみにする

```
/* クラスメソッド sort 以外は例 1 と同じ */
  static void sort(Prefecture[] p) {
    for (int i = 0; i < p.length-1; i++) {
      int min = i;
      for (int j = i+1; j < p.length; j++)
        if (greaterThan(p[min], p[j]))  min = j;
      Prefecture tmp = p[i];  p[i] = p[min];  p[min] = tmp;
    }
  }
```

5.2.2 バブルソート

配列の隣同士の要素 p[j] と p[j+1] を昇順にすることを繰返す

1. 先頭 p[0] から始めて，最後の 1 つ前 p[p.length-2] まで，その配列要素 p[j] とその次の配列要素 p[j+1] と比較し，小さい方を前に大きい方を後にすることを繰り返す. 結果的に最大のものが最後 p[p.length-1] に格納される

2. 先頭 p[0] から最後の２つ前 p[p.length-3] まで比較して，次に大きいものを最後の１つ前 p[p.length-2] に格納する

3. 同様に先頭 p[0] から p[i-1] まで比較すると，そのなかで最も大きいものが p[i] に格納される

4. i = 1 まで繰返せば，最後の１つが最小となり，p[0] に格納される (ソート完了)

ソートの過程 (図では下が前であり，左から右へと変化していく)

4	4	4	4	**4**	6	6	6	6	6	6	6	...
6	6	6	**6**	**6**	4	4	4	**4**	5	5	5	...
1	1	**1**	**5**	5	5	5	**5**	**5**	4	4	**4**	...
3	**3**	**5**	1	1	1	**1**	**3**	3	3	**3**	**3**	...
5	**5**	3	3	3	**3**	**3**	1	1	**1**	**2**	2	...
2	2	2	2	2	**2**	2	2	2	**2**	1	1	...

例１： バブルソート

```
/* クラスメソッド sort 以外は例１と同じ */
   static void sort(Prefecture[] p) {
      for (int i = [          ]; i > [ ]; i--)
         for (int j = [ ]; j < [ ]; j++)
            if (greaterThan(p[j], p[j+1])) {
               Prefecture tmp = p[j];  p[j] = [      ];  [      ] = tmp;
            }
   }
```

5.3　ソーティング計算量の下限

ソーティング対象のデータ量が n であるとすると，その並び方のバラエティ(順列) は $n!$ となる．ソーティング計算の基礎は，2つのデータを比較して大小を判定して，何らかの操作を行うことである．すなわち1回の基本操作は，大小いずれか (2つに1つの可能性) に応じて処理を進めることにある．したがって，全体を並び替えるには $\log n! \approx n \log n$ の操作が必要となる．

別の言い方をするならば，当初のデータの順列を解析して，適切な順序 (昇順や降順) に並び替える作業がソーティングである．1回の基本操作では，データの順列に関して 1 [bit] の情報量を獲得して，並び替え操作を行っている．データ量 n の順列は，$\log n! \approx n \log n$ [bit] の情報量を持っているので，ソーティングの計算量の下限は $O(n \log n)$ となる．

5.3.1　挿入ソート

最小値選択法やバブルソートでは，実際のところ，全ての配列要素の組について比較を行なっている．つまり，配列要素の個数が n とすると，比較の回数は $n(n-1)/2$ 回になる．これは次数 (order) で言えば，n の2乗 (n^2) に比例した計算であり，計算量は $O(n^2)$ となる．

挿入ソートでは，既に n 個のソート済のデータがあるとして，新たにデータを 1 つ挿入することを考える．新しいデータを挿入すべき場所は，バイナリサーチによって，$O(\log n)$ の手間で求められる．それを n 回繰り返すと，全てのデータをソートしたものが得られる．したがって，挿入ソートの計算量は $O(n \log n)$ となる．これはソーティングに必要な計算量の下限に相当する手間で，ソーティングを実行できることを示している．
しかし実際には，配列の途中にデータを自由に挿入することは困難である．配列のサイズが $2^n - 1$ であれば可能であるが，それは現実的ではない．したがって，1 回挿入するたびに，配列内のデータをズラす操作が必要となる．挿入 1 回ごとに，ズラす操作は平均 $n/2$ 回必要であり，全体で n 個のデータを挿入することから，ズラす操作は全体として $O(n^2)$ 回必要となる．

5.3.2 クイックソート

データの前半に小さいデータ，データの後半に大きいデータを，それぞれまとめ，前半と後半を個別にソートする．クラスメソッド qsort の再帰呼出によって実現する．

理想的なクイックソート： 現実には不可能なクイックソート

次のような作業 x があると仮定する．① n 個のデータに対して操作を施したとき，② 中央値 (メディアン) を判定するとともに，③ データの前半には中央値以下のデータ，後半には中央値以上のデータを分離する．この作業 x を実施した後には，$n/2$ 個のデータからなる前半部と後半部を，それぞれソートすれば良い．そこで，それぞれの部分 (前半部と後半部) に作業 x を再帰的に施す．新たな作業 x は $n/2$ 個のデータに対する操作が 2 回なので，結局 n 個のデータに対して操作が必要となる．ただし，1 回の作業 x ごとにソートすべきデータの個数が $1/2$ になるので，作業 x を $\log n$ 回繰り返すと全てのデータがソートできる．つまり，計算量は $O(n \log n)$ となるはずである．しかし，実際に効率良く②を実現する作業 x は存在しない．

クラスメソッド qsort**：** lower から upper までをソートする．

1. データが 2 つ以上 (lower < upper) のときソートする．
2. データの中央位置 (lower+upper)/2 の値 pm を基準として，先頭ならびに末尾から順時比較し (それぞれ i と j)，pm より小さいデータが前，pm より大きいデータが後ろになるようにする．そうでないものが見つかったら，互いに置換する．この作業を i <= j が成り立つ限り繰り返す．終了した時点では j < i（殆んどの場合は j = i-1）の状態になり，lower〜j には pm 以下のデータ，i〜upper には pm 以上のデータがあることが保証される．
3. lower〜j と i〜upper の範囲をそれぞれ qsort を用いて再帰的にソートする．

例１： クイックソート

```
/* クラスメソッド sort と qsort 以外は例１と同じ */
   static void sort(Prefecture[] p) {
      qsort(p, 0, p.length-1);
   }
```

```
static void qsort(Prefecture[] p, int lower, int upper) {
   if (                ) {
      Prefecture pm =                    
      int  i = lower;
      int  j = upper;
      while (i <= j) {
         while (greaterThan(pm, p[i]))  i++;
         while (greaterThan(p[j], pm))  j--;
         if (i <= j) {
            Prefecture tmp = p[i];  p[i] = p[j];  p[j] = tmp;
            i++;  j--;
         }
      }
      qsort(p, lower, j);
      qsort(             );
   }
}
```

5.3.3 マージソート

2つのソート済みのデータがあるものとして，その2つのデータを1つにまとめる (マージする)．ここでは2つのクラスメソッド msort と merge によって実現する．

クラスメソッド msort**：** lower から upper までをソートする．

1. データを2つ (lower〜middle と middle+1〜upper) に分けて，各部分が2個以上の場合は，msort を用いて再帰的にソートする．
2. クラスメソッド merge を用いて，2つのデータを1つにマージする．本当の先頭より1つ前 (lower-1 番目) には適当なデータ DUMMY を入れる．

クラスメソッド merge**：** ソート済みの部分 llow から lup までと rlow から rup までをマージする．ただし，与えられたデータ head を pos に格納し，マージデータは pos+1 以降に格納する．

1. 未処理のデータがある (llow<=lup または rlow<=rup) 間は，(一方のデータがなくなっている場合には) 残っている方の先頭をあるいは (両方ともデータがある場合には) 双方を比べて小さい方を pos+1 に格納し，残りの部分を再帰的に merge でマージする．
2. 与えられたデータ head を pos に格納する．ただし，pos<llow(実際には pos==llow-1) の場合は，pos のデータは head と同じなので格納を省く (特に pos==lower-1 の場合には，DUMMY を書き込まないために省略が必要)．

例1： マージソート

```
/* クラスメソッド sort と msort, merge 以外は例1と同じ */
  static void sort(Prefecture[] p) {
    msort(p, 0, p.length-1);
  }
  static void msort(Prefecture[] p, int lower, int upper) {
    final Prefecture DUMMY = null;
    int middle = (lower+upper)/2;
    if (middle > lower)    msort(p, lower, middle);
    if (upper > middle+1)  msort(p, middle+1, upper);
    merge(p, DUMMY, lower-1, lower, middle, middle+1, upper);
  }
  static void merge(Prefecture[] p, Prefecture head, int pos,
                    int llow, int lup, int rlow, int rup) {
    if ((lup >= llow) || (rup >= rlow)) {
      if (lup < llow)
        merge(p, p[rlow], pos+1, llow, lup, rlow+1, rup);
      else if (rup < rlow)
        merge(p, p[llow], pos+1, llow+1, lup, rlow, rup);
      else if (greaterThan(p[llow], p[rlow]))
        merge(p, p[rlow], pos+1, llow, lup, rlow+1, rup);
      else
        merge(p, p[llow], pos+1, llow+1, lup, rlow, rup);
    }
    if (pos >= llow)
      p[pos] = head;
  }
}
```

このプログラムは再帰によるメソッド呼出が非常に沢山行なわれるために，計算の実行が困難になることがある (たとえば，この後のソートの比較などでデータ数を多くすると問題となる)．授業の UNIX 環境では 40231 個以上のデータをソートしようとすると，`Exception in thread "main" java.lang.StackOverflowError` というようなエラーメッセージとともにプログラムが異常終了する．これはエラーメッセージにあるように，多数のメソッド呼出によってスタック (領域) を使い果たしてしまい，実行不可能となっている．このような場合には，スタックサイズを大きくする必要がある．たとえば，スタックサイズを 14336 KByte に拡大したい場合には，下に示すように，まずシェル環境のスタックサイズを `ulimit` コマンドで拡大し，Java 実行時に Java 環境のスタックサイズを「`-Xss14336`」によって指定する．ただし，スタックサイズが 8192 KByte 以下の場合には，シェル環境のスタックサイズを変更する必要はない．

```
ca10101$ ulimit -s 14336 ¶
ca10101$ java -Xss14336 TimeMerge 45000 ¶
```

5.3.4 ソートの比較

ここではdouble型の乱数データのソートに要する時間を計測し，各ソートアルゴリズムの比較を試みる．
実際には，ソート対象のデータ個数をコマンド引数で入力し，Math.rand()メソッドを用いて配列にデータをセットする．TIMES=10回のソート時間を計測して，平均値を計算する．System.currentTimeMillis()メソッドは，1970年1月1日を基準とするmsec単位の時刻(1970年1月1日からのmsec数)を返す．

例1： 最小値選択法(5.2.1節の例3)の計測 - TimeSimple3.java
ソート対象となるデータがPrefecture型ではなく，double型であることに注意せよ．

```
public class TimeSimple3 {
  public static void main(String[] args) {
    final int TIMES = 10;                  // 計測回数
    long t = 0;                            // 計測時間
    double[] p = new double[Integer.parseInt(args[0])];
                                           // ソートデータ用配列
    for (int i = 0; i < TIMES; i++) {
      for (int j = 0; j < p.length; j++)  // ソートデータ作成
        p[j] = Math.random();
      System.out.print("Start sorting #" + i + " ... ");
      t -= System.currentTimeMillis();     // ソート開始時刻
      sort(p);
      t += System.currentTimeMillis();     // ソート終了時刻
      System.out.println("... finished.");
    }
    System.out.println("Average time for sorting " + p.length +
                       " elements: " + (t/TIMES) + " [msec]");
  }
  static boolean greaterThan(double p0, double p1) {
    return  p0 > p1;
  }
  static void sort(double[] p) {
    for (int i = 0; i < p.length-1; i++) {
      int min = i;
      for (int j = i+1; j < p.length; j++)
        if (greaterThan(p[min], p[j]))
          min = j;
      double tmp = p[i];
      p[i] = p[min];
      p[min] = tmp;
    }
```

```
    }
  }
```

付 録A　Javaクラスライブラリ

Java は標準で多くのクラスライブラリを持っており，それらを利用することでプログラム開発の効率を上げられる．標準クラスライブラリに関する情報は，次の場所で見つけられる．

インターネット　http://java.sun.com/j2se/1.4/ja/docs/ja/api/index.html

以下では「計算機プログラミングI」で利用するクラスライブラリについて，簡単に説明する．

A.1　入出力関連ライブラリ

データの入出力に関するクラスライブラリをまとめた．

A.1.1　Systemクラス

入出力とは直接関係ないが，オペレーティングシステムとのインタフェースなど，重要なクラス変数やメソッドを持っている．

パッケージ java.lang
　標準のパッケージのため import しなくてよい

err　　標準エラー出力ストリーム (クラス変数)

```
public static final PrintStream err
```

in　　標準入力ストリーム (クラス変数)

```
public static final InputStream in
```

out　　標準出力ストリーム (クラス変数)

```
public static final PrintStream out
<用例>
System.out.println(i);
```

currentTimeMillis　　現在時刻 (クラスメソッド)
　1970 年 1 月 1 日以来の時間を msec 単位で返す

```
public static long currentTimeMillis()
<用例>
long t = System.currentTimeMillis();
```

exit 実行終了 (クラスメソッド)
プログラムの実行を強制的に終了する

```
public static void exit(int status)
<用例>
System.exit(i);
```

A.1.2 InputStream クラス

入力用のバイトストリーム (文字以外のデータの読み込みにも利用可能).

パッケージ java.io

```
import java.io.*;
```

close ストリームの閉鎖 (インスタンスメソッド)

```
public void close() throws IOException
```

read 読み込み (インスタンスメソッド)
入力ストリームから 1Byte のデータを読み込み int 型で戻す

```
public abstract int read() throws IOException
```

A.1.3 InputStreamReader クラス

入力用の文字ストリーム (文字データの読み込みに利用). Reader のサブクラス (一種) で, 主として 1 文字単位の入力に用いられる.

パッケージ java.io

```
import java.io.*;
```

InputStreamReader コンストラクタ

```
public InputStreamReader(InputStream in)
```

close ストリームの閉鎖 (インスタンスメソッド)

```
public void close() throws IOException
```

read 読み込み (インスタンスメソッド)
入力ストリームから 1 文字分 (2Byte) の文字データを読み込み int 型で戻す

```
public int read() throws IOException
```

A.1.4 FileReader クラス

入力用の文字ストリーム (文字データの読み込みに利用). Reader, InputStreamReader のサブクラス (一種).

パッケージ java.io

```
import java.io.*;
```

FileReader　　コンストラクタ

```
public FileReader(String fileName) throws FileNotFoundException
```

A.1.5 BufferedReader クラス

入力用の文字ストリーム (文字データの読み込みに利用). Reader のサブクラス (一種) で, 主として文字列単位での入力に用いられる.

パッケージ java.io

```
import java.io.*;
```

BufferedReader　　コンストラクタ

```
public BufferedReader(Reader in)
<用例>
BufferedReader br = new BufferedReader(new FileReader("foo"));
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
```

close　　ストリームの閉鎖 (インスタンスメソッド)

```
public void close() throws IOException
<用例>
br.close();
```

read　　読み込み (インスタンスメソッド)

入力ストリームから 1 文字分 (2Byte) の文字データを読み込み int 型で戻す
ファイル終端の場合には -1 を戻す

```
public int read() throws IOException
<用例>
while ((i = br.read()) != -1)
```

readLine　　読み込み (インスタンスメソッド)

入力ストリームから 1 行 ('\n' まで) の文字列データを読み込み String 型で戻す
ファイル終端の場合には null を戻す

```
public String readLine() throws IOException
<用例>
while ((str = br.read()) != null)
```

A.1.6 PrintStream クラス

出力用のバイトストリーム (文字以外のデータの書き出しにも利用可能). OutputStream, FilterOutputStream のサブクラス (一種).

パッケージ java.io

```
import java.io.*;
```

close ストリームの閉鎖 (インスタンスメソッド)

```
public void close()
```

flush ストリームの一掃 (インスタンスメソッド)
出力ストリーム内の (内に溜った) 出力命令を実行する

```
public void flush()
```

write 書き出し (インスタンスメソッド)
出力ストリームへ int 型データの下位 1Byte を書き出す

```
public void write(int b)
```

print 書き出し (インスタンスメソッド)
出力ストリームにデータを文字列化して書き出す

```
public void print(基本データ型 p)
public void print(Object obj)
public void print(String str)
```

println 改行付き書き出し (インスタンスメソッド)
出力ストリームにデータを文字列化して書き出し, 最後に改行する

```
public void println()
public void println(基本データ型 p)
public void println(Object obj)
public void println(String str)
```

A.1.7 FileWriter クラス

出力用の文字ストリーム (文字データの書き出しに利用). Writer, OutputStreamWriter のサブクラス (一種).

パッケージ java.io

```
import java.io.*;
```

FileWriter コンストラクタ

```
public FileWriter(String fileName) throws IOException
public FileWriter(String fileName, boolean append) throws IOException
```

A.1.8 PrintWriter クラス

出力用の文字ストリーム (文字データの書き出しに利用). Writer のサブクラス (一種) で，主として文字列単位の出力に用いられる.

パッケージ java.io

```
import java.io.*;
```

PrintWriter　　コンストラクタ

```
public PrintWriter(Writer in)
<用例>
PrintWriter pw = new PrintWriter(new FileWriter("foo"));
```

close　　ストリームの閉鎖 (インスタンスメソッド)

```
public void close()
<用例>
pw.close();
```

flush　　ストリームの一掃 (インスタンスメソッド)

出力ストリーム内の (内に溜った) 出力命令を実行する

```
public void flush()
<用例>
pw.flush();
```

write　　書き出し (インスタンスメソッド)

出力ストリームへ int 型データの下位 2Byte を 1 文字分の文字データとして書き出す

```
public void write(int b)
<用例>
pw.write('x');
```

print　　書き出し (インスタンスメソッド)

出力ストリームにデータを文字列化して書き出す

```
public void print(基本データ型 p)
public void print(Object obj)
public void print(String str)
```

println　　改行付き書き出し (インスタンスメソッド)

出力ストリームにデータを文字列化して書き出し，最後に改行する

```
public void println()
public void println(基本データ型 p)
public void println(Object obj)
public void println(String str)
```

A.2 ラッパクラス (Wrapper Class)

基本データ型に対応するクラスで，基本データを文字列に変換したり，文字列を基本データ他データへの変換，比較演算などを揃えている．

基本データ型	byte	short	int	long	float	double	boolean	char
ラッパクラス	Byte	Short	Integer	Long	Float	Double	Boolean	Character

A.2.1 Integer クラス

基本データの int 型のラッパクラスで，Number のサブクラス．

パッケージ java.lang
　標準のパッケージのため import しなくてよい

MIN_VALUE　int 型の最小値 = -2147483648 (クラス変数)

```
public static final int MIN_VALUE
```

MAX_VALUE　int 型の最大値 = 2147483647 (クラス変数)

```
public static final int MAX_VALUE
```

Integer　コンストラクタ

```
public Integer(int value)
public Integer(String str) throws NumberFormatException
```

parseInt　文字列からの変換 (クラスメソッド)
　文字列から int 型の数値を生成する

```
public static int parseInt(String str) throws NumberFormatException
<用例>
int i = Integer.parseInt(str);
```

valueOf　文字列からの変換 (クラスメソッド)
　文字列から Integer 型のデータを生成する

```
public static Integer valueOf(String str) throws NumberFormatException
<用例>
Integer i = Integer.valueOf(str);
```

toString　文字列への変換 (クラスメソッド)
　int 型データ (数値) から String 型データ (文字列) を生成する

```
public static String toString(int i)
<用例>
String str = Integer.toString(i);
```

intValue　　int 型への変換 (インスタンスメソッド)
　当該オブジェクトに相当する int 型データ (数値) を生成する

```
public int intValue()
<用例>
int i = Integer.valueOf(str).intValue();
```

doubleValue　　double 型への変換 (インスタンスメソッド)
　当該オブジェクトに相当する double 型データ (数値) を生成する

```
public double doubleValue()
<用例>
double d = Integer.valueOf(str).doubleValue();
```

toString　　文字列への変換 (インスタンスメソッド)
　当該オブジェクトに相当する String 型データ (文字列) を生成する

```
public String toString()
```

A.2.2 Double クラス

基本データの double 型のラッパクラスで，Number のサブクラス.

パッケージ java.lang
　標準のパッケージのため import しなくてよい

MIN_VALUE　　double 型の正の最小値 (クラス変数)

```
public static final double MIN_VALUE
```

MAX_VALUE　　double 型の正の有限最大値 (クラス変数)

```
public static final double MAX_VALUE
```

POSITIVE_INFINITY　　double 型の正の無限大値 (クラス変数)

```
public static final double POSITIVE_INFINITY
```

NEGATIVE_INFINITY　　double 型の負の無限大値 (クラス変数)

```
public static final double NEGATIVE_INFINITY
```

NaN　　double 型であり得ない値 (クラス変数)

```
public static final double NaN
```

Double　　コンストラクタ

```
public Double(double value)
public Double(String str) throws NumberFormatException
```

parseDouble 文字列からの変換 (クラスメソッド)

文字列から double 型の数値を生成する (JDK1.2 以降でないと使えない)

```
public static double parseDouble(String str) throws NumberFormatException
<用例>
double d = Double.parseDouble(str);
```

valueOf 文字列からの変換 (クラスメソッド)

文字列から Double 型のデータを生成する

```
public static Double valueOf(String str) throws NumberFormatException
<用例>
Double d = Double.valueOf(str);
```

toString 文字列への変換 (クラスメソッド)

double 型データ (数値) から String 型データ (文字列) を生成する

```
public static String toString(double d)
<用例>
String str = Double.toString(i);
```

intValue int 型への変換 (インスタンスメソッド)

当該オブジェクトに相当する int 型データ (数値) を生成する

```
public int intValue()
<用例>
int i = Double.valueOf(str).intValue();
```

doubleValue double 型への変換 (インスタンスメソッド)

当該オブジェクトに相当する double 型データ (数値) を生成する

```
public double doubleValue()
<用例>
double d = Double.valueOf(str).doubleValue();
```

toString 文字列への変換 (インスタンスメソッド)

当該オブジェクトに相当する String 型データ (文字列) を生成する

```
public String toString()
```

A.2.3 Boolean クラス

基本データの boolean 型のラッパクラス.

パッケージ java.lang

標準のパッケージのため import しなくてよい

FALSE　　Boolean 型の false に対する値 (クラス変数)

```
public static final Boolean FALSE
```

TRUE　　Boolean 型の true に対する値 (クラス変数)

```
public static final Boolean TRUE
```

Boolean　　コンストラクタ

```
public Boolean(boolean value)
public Boolean(String str)
```

valueOf　　文字列からの変換 (クラスメソッド)
　文字列から Boolean 型のデータを生成する

```
public static Boolean valueOf(String str)
<用例>
Boolean b = Boolean.valueOf("True");
```

booleanValue　　boolean 型への変換 (インスタンスメソッド)
　当該オブジェクトに相当する boolean 型データを生成する

```
public boolean booleanValue()
<用例>
boolean b = Boolean.valueOf(str).booleanValue();
```

toString　　文字列への変換 (インスタンスメソッド)
　当該オブジェクトに相当する String 型データ (文字列) を生成する

```
public String toString()
```

A.3　数学ライブラリ

数学 (計算) に関するライブラリをまとめた.

A.3.1　Math クラス

パッケージ java.lang
　標準のパッケージのため import しなくてよい

E　　double 型の自然対数の底 e (クラス変数)

```
public static final double E
```

PI　　double 型の円周率 π (クラス変数)

```
public static final double PI
```

`abs` 絶対値 $|a|$ (クラスメソッド)

```
public static int abs(int a)
public static long abs(long a)
public static float abs(float a)
public static double abs(double a)
```

`min` 最小値 $\min(a, b)$ (クラスメソッド)

```
public static int min(int a, int b)
public static long min(long a, long b)
public static float min(float a, float b)
public static double min(double a, double b)
```

`max` 最大値 $\max(a, b)$ (クラスメソッド)

```
public static int max(int a, int b)
public static long max(long a, long b)
public static float max(float a, float b)
public static double max(double a, double b)
```

`round, rint` 四捨五入 (クラスメソッド)

```
public static int round(float a)
public static long round(double a)
public static double rint(double a)
```

`floor` 切捨て $\lfloor a \rfloor$ (クラスメソッド)

```
public static double floor(double a)
```

`ceil` 切上げ $\lceil a \rceil$ (クラスメソッド)

```
public static double ceil(double a)
```

`pow` ベキ乗 a^b (クラスメソッド)

```
public static double pow(double a, double b)
```

`sqrt` 平方根 $\sqrt{a}$ (クラスメソッド)

```
public static double sqrt(double a)
```

`exp` 指数関数 e^a (クラスメソッド)

```
public static double exp(double a)
```

`log` 対数関数 $\log_e a$ (クラスメソッド)

```
public static double log(double a)
```

`sin, cos, tan` 三角関数 $\sin a, \cos a, \tan a$ (クラスメソッド)

```
public static double sin(double a)
public static double cos(double a)
public static double tan(double a)
```

`asin,acos,atan,atan2` 逆三角関数 $\sin^{-1} a, \cos^{-1} a, \tan^{-1} a, \tan^{-1} \frac{b}{a}$ (クラスメソッド)

```
public static double asin(double a)
public static double acos(double a)
public static double atan(double a)
public static double atan2(double a, double b)
```

`toDegrees, toRadians` 角度変換 (クラスメソッド)

```
public static double toDegrees(double angleRadian)
public static double toRadians(double angleDegree)
```

`random` 乱数 $r \in [0.0, 1.0)$ (クラスメソッド)

```
public static double random()
```

A.4 文字列関連ライブラリ

文字列操作に関するライブラリをまとめた.

A.4.1 String クラス

パッケージ `java.lang`
標準のパッケージのため `import` しなくてよい

`String` コンストラクタ

```
public String(String str)
```

`valueOf` 文字列への変換 (クラスメソッド)
基本データ型を文字列に変換する

```
public static String valueOf(基本データ型 p)
```

`equals` 文字列の比較 (インスタンスメソッド)
当該文字列と別の文字列が等しいか否かを判定する

```
public boolean equals(Object o)
```

equalsIgnoreCase 文字列の比較 (インスタンスメソッド)

大文字小文字の区別をせずに，当該文字列と別の文字列が等しいか否かを判定する

```
public boolean equalsIgnoreCase(String str)
```

compareTo 文字列の大小比較 (インスタンスメソッド)

当該文字列と別の文字列の符号付き距離を計算する (当該文字列と等しければ 0，小さければ負，大きければ正を返す)

```
public int compareTo(String str)
```

compareToIgnoreCase 文字列の大小比較 (インスタンスメソッド)

大文字小文字の区別をせずに，当該文字列と別の文字列の符号付き距離を計算する (当該文字列と等しければ 0，小さければ負，大きければ正を返す)

```
public int compareToIgnoreCase(String str)
```

length 文字列中の文字数 (インスタンスメソッド)

当該文字列中の文字数を返す

```
public int length()
```

charAt 文字列中の文字 (インスタンスメソッド)

当該文字列から index 番の char 型データ (文字) を返す

```
public char charAt(int index)
```

substring 文字列中の部分文字列 (インスタンスメソッド)

当該文字列から beginIndex 番以降 (endIndex 番までの) 部分文字列データ (String 型) を返す

```
public String substring(int beginIndex)
public String substring(int beginIndex, int endIndex)
```

indexOf 文字列中の文字の位置 (インスタンスメソッド)

当該文字列での最初に文字または部分文字列が現れる位置 (int 型) を返す

```
public int indexOf(in ch)
public int indexOf(String str)
```

laxtIndexOf 文字列中の文字の位置 (インスタンスメソッド)

当該文字列での最後に文字または部分文字列が現れる位置 (int 型) を返す

```
public int lastIndexOf(in ch)
public int lastIndexOf(String str)
```

toLowerCase 文字列の小文字化 (インスタンスメソッド)

当該文字列の文字を全て小文字にした文字列を返す

```
public String toLowerCase()
```

toUpperCase 文字列の大文字化 (インスタンスメソッド)

当該文字列の文字を全て大文字にした文字列を返す

```
public String toUpperCase()
```

A.4.2 StringTokenizer クラス

StringTokenizer は，文字列データを単語単位の部分文字列に分解するためのクラス.

パッケージ java.util

```
import java.util.*;
```

StringTokenizer　　コンストラクタ

第2引数として String 型の delim が与えられると，delim 内の文字を単語の区切り文字として扱う
指定されない場合は空白文字 (スペース，タブ，改行) を単語の区切りとする

```
public StringTokenizer(String str)
public StringTokenizer(String str, String delim)
<用例>
StringTokenizer st = new StringTokenizer(str);
```

countTokens　　単語数 (インスタンスメソッド)

当該オブジェクトに含まれる単語数

```
public int countTokens()
```

nextToken　　次の単語 (インスタンスメソッド)

当該オブジェクト内の次の単語データ (文字列) を返す

```
public String nextToken()
<用例>
StringTokenizer st = new StringTokenizer(str);
for (int i = 0; i < st.countTokens(); i++)
  strArray[i] = st.nextToken();
```

hasMoreTokens　　次の単語の有無 (インスタンスメソッド)

当該オブジェクト内に次の単語があるかどうかを boolean 型で返す

```
public boolean hasMoreTokens()
<用例>
StringTokenizer st = new StringTokenizer(str);
while (st.hasMoreTokens())
  strArray[i++] = st.nextToken();
```

A.5 アプレット関連ライブラリ

アプレットに関するライブラリをまとめた.

A.5.1 Applet クラス

Applet は，アプレットを作成するためのクラスであり，すべてのアプレットは Applet クラスのサブクラスとなる．

パッケージ java.applet

```
import java.applet.*;
```

スーパークラス java.awt.Panel

次のようにスーパークラスが辿れる

java.applet.Applet → java.awt.Panel → java.awt.Container → java.awt.Component → java.lang.Object

init　アプレット初期化メソッド (インスタンスメソッド)

必要に応じてオーバライドするメソッドで，アプレットの初期化を定める．アプレットビューワないし WWW ブラウザによってアプレットがメモリ上に読み込まれた際に呼び出される

最初に start メソッドが呼び出される前に，必ず init メソッドが呼び出される

```
public void init()
```

start　アプレット起動メソッド (インスタンスメソッド)

必要に応じてオーバライドするメソッドで，アプレットが (再) 起動される際の動作を定める．アプレットビューワないし WWW ブラウザによってアプレットの実行が開始される際に呼び出される

WWW ブラウザの場合には，アプレットの含まれる WWW ページが開かれるごとに，アプレットの実行が開始される

```
public void start()
```

stop　アプレット停止メソッド (インスタンスメソッド)

必要に応じてオーバライドするメソッドで，アプレットが停止される際の動作を定める．アプレットビューワないし WWW ブラウザによってアプレットの実行が停止される際に呼び出される

WWW ブラウザの場合には，アプレットの含まれる WWW ページから他のページに移ると，アプレットの実行が停止される

```
public void stop()
```

destroy　アプレット破棄メソッド (インスタンスメソッド)

必要に応じてオーバライドするメソッドで，アプレットが破棄される際の動作を定める．アプレットビューワないし WWW ブラウザによってアプレットがメモリから捨てられる際に呼び出される

destory メソッドが呼び出される前に，必ず stop メソッドが呼び出される

```
public void destroy()
```

getParameter　　パラメタ文字列の獲得 (インスタンスメソッド)

WWW ページ内の<PARAM>タグによって与えられたパラメタ文字列を獲得する (アプリケーションのコマンド引数のようなもの)

```
public String getParameter(String name)
<用例>
WWW ページ内で
  <APPLET CODE="...." ....>
  <PARAM NAME="Message" VALUE="msg1">
  </APPLET>
と書かれていると getParamter("Message") は "msg1" を返す
```

showStatus　　アプレット状況表示 (インスタンスメソッド)

アプレットビューワや WWW ブラウザの最下部 (ステータス行) に文字列を表示する

```
public void showStatus(String msg)
<用例>
showStatus("Applet running");
```

getDocumentBase　　WWW ページの URL (インスタンスメソッド)

アプレットの含まれる WWW ページが存在する場所の URL を返す

```
public URL getDocumentBase()
```

getImage　　画像データの獲得 (インスタンスメソッド)

指定された URL から画像データを獲得する

```
public Image getImage(URL url, String name)
<用例>
Graphics g = getGraphics();
g.drawImage(getImage(getDocumentBase(), "image.jpg"), x, y, this);
```

A.5.2 Container クラス

Container は, Applet の 2 つ上のスーパークラスであり, アプレットを作る際には Container クラスのメソッドも利用される.

パッケージ java.awt

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

paint　　描画メソッド (インスタンスメソッド)

必要に応じてオーバライドするメソッドで, Container(またはその拡張のアプレット) の描画内容を定める

画面上で Container が隠された後で, 再度描画し直す必要があるときに, 自動的に呼び出される. 引数 g はアプレットの Graphics オブジェクトとなる

```
public void paint(Graphics g)
```

update 描画の更新 (インスタンスメソッド)

Container(またはその拡張のアプレット) の描画内容を更新する
デフォールトでは，背景色で塗りつぶした後で，paint メソッドを呼び出す．引数 g はアプレットの Graphics オブジェクトとなる

```
public void update(Graphics g)
```

repaint 描画更新の要求 (インスタンスメソッド)

Java の仮想マシンに対して update メソッドの起動を要求する

```
public void repaint()
```

A.5.3 Component クラス

Component は，Applet の 3 つ上のスーパークラスであり，アプレットを作る際には Component クラスのメソッドも利用される．

パッケージ java.awt

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

getGraphics Graphics オブジェクトの獲得 (インスタンスメソッド)

描画用のオブジェクト Graphics を獲得する

```
public Graphics getGraphics()
<用例>
Graphics g = getGraphics();
g.drawLine(x1, y1, x2, y2);
```

getSize サイズの獲得 (インスタンスメソッド)

Component の大きさ Dimension を調べる

```
public Dimension getSize()
<用法>
Dimension d = getSize();
int width = d.width;
int height = d.height;
```

getForeground 描画色の獲得 (インスタンスメソッド)

描画色 (paint メソッドでの描画に用いられる色) を調べる

```
public Color getForeground()
```

getBackground 背景色の獲得 (インスタンスメソッド)

背景色 (update メソッドでの塗りつぶしに用いられる色) を調べる

```
public Color getBackground()
```

setForeground　描画色の設定 (インスタンスメソッド)
描画色 (paint メソッドでの描画に用いられる色) を c に設定する

```
public void setForeground(Color c)
<用法>
setForeground(Color.blue);
```

setBackground　背景色の設定 (インスタンスメソッド)
背景色 (update メソッドでの塗りつぶしに用いられる色) を c に設定する

```
public void setBackground(Color c)
```

addMouseListener　マウスリスナの登録 (インスタンスメソッド)
マウスボタン関するマウスイベントを受けとるためのマウスリスナを登録する

```
public void addMouseListener(MouseListener l)
```

removeMouseListener　マウスリスナの解除 (インスタンスメソッド)
マウスボタン関するマウスイベントを受けとるためのマウスリスナの登録を解除する

```
public void removeMouseListener(MouseListener l)
```

addMouseMotionListener　マウスモーションリスナの登録 (インスタンスメソッド)
マウスの移動に関するマウスイベントを受けとるためのマウスモーションリスナを登録する

```
public void addMouseMotionListener(MouseMotionListener l)
```

removeMouseMotionListener　マウスモーションリスナの解除 (インスタンスメソッド)
マウスの移動に関するマウスイベントを受けとるためのマウスモーションリスナの登録を解除する

```
public void removeMouseMotionListener(MouseMotionListener l)
```

A.5.4 Graphics クラス

アプレットなどで描画を行なう際には Graphics クラスが利用される．Graphics オブジェクトでは，描画可能な領域，描画に用いる文字や色などの情報が管理される．描画領域の座標は，左上隅を原点 $(0, 0)$ とし，x 軸が右向き，y 軸が下向きの，整数値で表現される．

パッケージ java.awt

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

clearRect 矩形領域の消去 (インスタンスメソッド)

左上座標 (x, y)，幅 w，高さ h の矩形領域を背景色で塗りつぶす

```
public abstract void clearRect(int x, int y, int w, int h)
```

drawImage 画像の描画 (インスタンスメソッド)

左上座標 (x, y)，幅 w，高さ h の領域に画像 img を描画する．描画の進行状況を io (アプレットなど) に通知される

```
public abstract boolean drawImage(Image img, int x, int y, ImageObserver io)
public abstract boolean drawImage(Image img, int x, int y, int w, int h,
                                  ImageObserver io)
```

<用例>

```
Graphics g = getGraphics();
g.drawImage(getImage(getDocumentBase(), "image.jpg"), x, y, this);
```

drawLine 線分の描画 (インスタンスメソッド)

(x0, y0) と (x1, y1) を端点とする線分を描画する

```
public abstract void drawLine(int x0, int y0, int x1, int y1)
```

drawOval 楕円の描画 (インスタンスメソッド)

左上座標 (x, y)，幅 w，高さ h の楕円 (線のみ) を描画する

```
public abstract void drawOval(int x, int y, int w, int h)
```

drawPolygon 多角形の描画 (インスタンスメソッド)

n 個の頂点 (x[i], y[i]) からなる多角形 (境界のみ) を描画する

```
public abstract void drawPolygon(int[] x, int[] y, int n)
```

drawPolyline 折れ線の描画 (インスタンスメソッド)

n 個の頂点 (x[i], y[i]) からなる折れ線を描画する．多角形と異なり，折れ線の両端は閉じていない

```
public abstract void drawPolyline(int[] x, int[] y, int n)
```

drawRect 矩形の描画 (インスタンスメソッド)

左上座標 (x, y)，幅 w，高さ h の矩形 (枠のみ) を描画する

```
public abstract void drawRect(int x, int y, int w, int h)
```

drawString 文字列の描画 (インスタンスメソッド)

文字列 str を位置 (x, y) に描画する

```
public abstract void drawString(String str, int x, int y)
```

fillOval 楕円の塗りつぶし (インスタンスメソッド)

左上座標 (x, y)，幅 w，高さ h の楕円とその内部を塗りつぶす

```
public abstract void fillOval(int x, int y, int w, int h)
```

`fillPolygon` 多角形の塗りつぶし (インスタンスメソッド)

`n` 個の頂点 (`x[i]`, `y[i]`) からなる多角形とその内部を塗りつぶす

```
public abstract void fillPolygon(int[] x, int[] y, int n)
```

`fillRect` 矩形の塗りつぶし (インスタンスメソッド)

左上座標 (`x`, `y`)，幅 `w`，高さ `h` の矩形とその内部を塗りつぶす

```
public abstract void fillRect(int x, int y, int w, int h)
```

`getColor` 描画色の獲得 (インスタンスメソッド)

現在の描画色を調べる

```
public abstract Color getColor()
```

`setColor` 描画色の設定 (インスタンスメソッド)

描画色を `c` に設定する

```
public abstract void setColor(Color c)
```

A.5.5 Color クラス

色の指定には `Color` クラスが利用される．一般的に計算機では，色を赤 (Red)，緑 (Green)，青 (Blue) という光の三原色の混色として表現する．これは「RGB 表現」と呼ばれる．Java では，色相 (Hue)，彩度 (Saturation)，明度 (Brightness) からなる「HSB 表現」も利用できるが，ここでは省略する．

パッケージ `java.awt`

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

`Color` コンストラクタ

RGB の各要素によって色を合成する．`float` 型の場合は [0.0, 1.0]，`int` 型の場合には [0, 255] の範囲で与える．RGB のすべての要素が 0 であれば 黒，RGB のすべての要素が最大 (1.0 または 255) であれば白となる

```
public Color(float r, float b, float c)
public Color(int r, int b, int c)
```

`black` 黒 (クラス定数変数)

特定の色として, `black`，`blue`，`cyan`，`darkGray`，`gray`，`green`，`lightGray`，`magenta`，`orange`，`pink`，`red`，`white`，`yellow` が用意されている．

```
public static final Color black
<用法>
setForeground(Color.black);
g.setColor(Color.black);
```

equals 等色判定 (インスタンスメソッド)

2つの色が等しいか否かを判定するには，等号 (=) ではなく，equals メソッドを用いる．

```
public boolean equals(Object obj)
```

A.5.6 MouseAdapter クラス

マウスボタンなどに関係するイベント処理のクラスとしてMouseAdapterがある．MouseAdapterはMouseListener インタフェースを実装したクラスとして定義されている．

パッケージ java.awt.event

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

MouseAdapter コンストラクタ

```
public MouseAdapter()
```

mouseClicked マウスクリックメソッド (インスタンスメソッド)

必要に応じてオーバライドするメソッドで，マウスクリックのイベントが起きた際の動きを定める．アプレット内でマウスがクリックされると，mousePressed メソッドとmouseReleased メソッドに，引き続いて呼び出される

```
public void mouseClicked(MouseEvent me)
```

mouseEntered マウス進入メソッド (インスタンスメソッド)

必要に応じてオーバライドするメソッドで，マウスカーソルの進入イベントが起きた際の動きを定める．マウスカーソルがアプレット内に入ると呼び出される

```
public void mouseEntered(MouseEvent me)
```

mouseExited マウス退出メソッド (インスタンスメソッド)

必要に応じてオーバライドするメソッドで，マウスカーソルの退出イベントが起きた際の動きを定める．マウスカーソルがアプレット外に出ると呼び出される

```
public void mouseExited(MouseEvent me)
```

mousePressed マウスプレスメソッド (インスタンスメソッド)

必要に応じてオーバライドするメソッドで，マウスプレスのイベントが起きた際の動きを定める．アプレット内でマウスボタンがプレスされると呼び出される

```
public void mousePressed(MouseEvent me)
```

mouseReleased マウスリリースメソッド (インスタンスメソッド)

必要に応じてオーバライドするメソッドで，マウスリリースのイベントが起きた際の動きを定める．アプレット内でマウスボタンがリリースされると呼び出される

```
public void mouseReleased(MouseEvent me)
```

A.5.7 MouseMotionAdapter クラス

マウス移動などに関係するイベント処理のクラスとして MouseMotionAdapter がある．MouseMotionAdapter は MouseMotionListener インタフェースを実装したクラスとして定義されている．

パッケージ java.awt.event

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

MouseMotionAdapter　　コンストラクタ

```
public MouseMotionAdapter()
```

mouseDragged　　マウスドラッグメソッド (インスタンスメソッド)

必要に応じてオーバライドするメソッドで，マウスドラッグのイベントが起きた際の動きを定める．アプレット内でマウスがドラッグ (ボタンをプレスしたまま移動) されると呼び出される

```
public void mouseDragged(MouseEvent me)
```

mouseMoved　　マウス移動メソッド (インスタンスメソッド)

必要に応じてオーバライドするメソッドで，マウス移動のイベントが起きた際の動きを定める．アプレット内でマウスが移動されると呼び出される

```
public void mouseMoved(MouseEvent me)
```

A.5.8 MouseEvent クラス

マウス関連のイベントを表現するクラスとして MouseEvent がある．

パッケージ java.awt.event

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

スーパークラス java.awt.event.InputEvent

次のようにスーパークラスが辿れる

java.awt.event.MouseEvent → java.awt.event.InputEvent → java.awt.event.ComponentEvent → java.awt.AWTEvent → java.util.EventObject → java.lang.Object

MOUSE_CLICKED　　マウスクリック (クラス定数変数)

マウスイベントの種類 (int 型の数値) は，スーパークラス AWTEvent の getID メソッドを用いて獲得できる．マウスイベントの種類としては，MOUSE_CLICKED，MOUSE_DRAGGED，MOUSE_ENTERED，MOUSE_EXITED，MOUSE_MOVED，MOUSE_PRESSED，MOUSE_RELEASED が用意されている．

```
public static final int MOUSE_CLICKED
<用法>
if (me.getID() == me.MOUSE_CLICKED)
  System.out.println("Clicked");
// または
if (me.getID() == MouseEvent.MOUSE_CLICKED)
  System.out.println("Clicked");
```

getX　　カーソルの x 座標値の獲得 (インスタンスメソッド)

イベントが生じた際のマウスカーソル位置の x 座標値を獲得する

```
public int getX()
```

getY　　カーソルの y 座標値の獲得 (インスタンスメソッド)

イベントが生じた際のマウスカーソル位置の y 座標値を獲得する

```
public int getY()
```

A.5.9 InputEvent クラス

入力関連のイベントを表現するクラスとして InputEvent がある.

パッケージ java.awt.event

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

スーパークラス java.awt.event.ComponentEvent

次のようにスーパークラスが辿れる

java.awt.event.InputEvent → java.awt.event.ComponentEvent → java.awt.AWTEvent → java.util.EventObject → java.lang.Object

BUTTON1_MASK　　ボタン 1 マスク値 (クラス定数変数)

入力イベントに関する細かな情報は修飾子 (int 型の数値) として, getModifiers メソッドを用いて獲得できる. 修飾子は, BUTTON1_MASK, BUTTON2_MASK, BUTTON3_MASK, ALT_GRAPH_MASK, ALT_MASK, CTRL_MASK, META_MASK, SHIFT_MASK などのマスク値の組合せ (論理和) になっている.

```
public static final int BUTTON1_MASK
<用法>
if ((me.getModifiers() & me.BUTTON1_MASK) != 0)
  System.out.println("Button 1");
// または
if ((me.getModifiers() & InputEvent.BUTTON1_MASK) != 0)
  System.out.println("Button 1");
```

getModifiers　　修飾子の獲得 (インスタンスメソッド)

イベントに関する情報を修飾子 (int 型の数値) として獲得する. 修飾子は BUTTON1_MASK などのマスク値と論理積を取ることで判定に利用できる.

```
public int getModifiers()
```

A.5.10 AWTEvent クラス

AWT(Abstract Window Toolkit) 関連のイベントを表現するクラスとして AWTEvent がある.

パッケージ java.awt

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

スーパークラス java.util.EventObject
次のようにスーパークラスが辿れる
java.awt.AWTEvent → java.util.EventObject → java.lang.Object

getID　　イベントの種類の獲得 (インスタンスメソッド)
イベントの種類を int 型の数値として獲得する.

```
public int getID()
```

A.5.11 EventObject クラス

Java で扱われるイベントの最上位のクラスとして EventObject がある.

パッケージ java.util

```
import java.util.*;
```

getSource　　イベントソースの獲得 (インスタンスメソッド)
イベントソース (イベントを発生したオブジェクト) を獲得する.

```
public Object getSource()
```

A.5.12 MouseListener インタフェース

マウスボタンなどに関係するイベント処理の基本インタフェースとして MouseListener がある. MouseAdapter クラスは MouseListener を実装したクラスである.

パッケージ java.awt.event

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

mouseClicked　　マウスクリックメソッド (インスタンスメソッド)
マウスクリックのイベントが起きた際の動きを定める. アプレット内でマウスがクリックされると, mousePressed メソッドと mouseReleased メソッドに, 引き続いて呼び出される

```
public void mouseClicked(MouseEvent me)
```

mouseEntered　　マウス進入メソッド (インスタンスメソッド)
マウスカーソルの進入イベントが起きた際の動きを定める. マウスカーソルがアプレット内に入ると呼び出される

```
public void mouseEntered(MouseEvent me)
```

mouseExited　マウス退出メソッド (インスタンスメソッド)
マウスカーソルの退出イベントが起きた際の動きを定める．マウスカーソルがアプレット外に出ると呼び出される

```
public void mouseExited(MouseEvent me)
```

mousePressed　マウスプレスメソッド (インスタンスメソッド)
マウスプレスのイベントが起きた際の動きを定める．アプレット内でマウスボタンがプレスされると呼び出される

```
public void mousePressed(MouseEvent me)
```

mouseReleased　マウスリリースメソッド (インスタンスメソッド)
マウスリリースのイベントが起きた際の動きを定める．アプレット内でマウスボタンがリリースされると呼び出される

```
public void mouseReleased(MouseEvent me)
```

A.5.13 MouseMotionListener インタフェース

マウス移動などに関係するイベント処理の基本インタフェースとして MouseMotionListener がある．MouseMotionAdapter クラスは MouseMotionListener を実装したクラスである．

パッケージ java.awt.event

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

mouseDragged　マウスドラッグメソッド (インスタンスメソッド)
マウスドラッグのイベントが起きた際の動きを定める．アプレット内でマウスがドラッグ (ボタンをプレスしたまま移動) されると呼び出される

```
public void mouseDragged(MouseEvent me)
```

mouseMoved　マウス移動メソッド (インスタンスメソッド)
マウス移動のイベントが起きた際の動きを定める．アプレット内でマウスが移動されると呼び出される

```
public void mouseMoved(MouseEvent me)
```

付 録B　プログラミング課題

第1回 - Javaプログラムの導入

B.1　レポート作成の練習

HTML を用いてレポート作成する場合の練習をしなさい．最初に `report_sample.html` と `report_format.html` の2つのファイルをコピーして，前者のファイルを Safari によって表示してみよ．詳細については WWW ページの説明を読むこと．また後者のファイルを今後のレポート提出に用いること．この課題は，印刷 (および提出) する必要はない．
基本的に各課題は，プログラムリスト，実行結果，考察の3つから構成される．プリント中にプログラム全文が示されている (例題プログラムの実行) の場合には，プログラムリストは省略して，実行結果と考察だけを提出すること．ただし，初回の課題に関しては省略せずにプログラムリストを載せること．また，実行結果の出力が長いものは途中を適当に省略してよい．考察は評価のウェイトが最も高い部分なので，短くても構わないから必ず書くこと．

B.2　例題プログラムの実行

プリント中の以下のプログラムを実行して，実行結果について考察しなさい．

2.2節　例1, 2, 3, 4

課題や授業について (採点の対象外)

レポートの作成に要した時間，特に非常に時間がかかった場合には何に苦労したか，を書くこと．また，授業についての感想や希望があれば，自由に書いて良い (歓迎する)．

第2回 - 変数と文字列 ～ 制御構造

B.3 例題プログラムの実行

プリント中の以下のプログラムを実行して，実行結果について考察しなさい．プリント中にプログラム全文が示されている例題プログラムは，プログラム文は印刷せずに，実行結果と考察だけを提出すること．また，実行結果の出力が長いものは適当に省略してよい．

2.3節	例1, 4	2.4節	例1, 3	2.6節	例1	2.7節	例1
2.8節	例2, 4	2.10節	例1	2.11節	例1', 2	2.12節	例1

B.4 バックスラッシュ文字

バックスラッシュ文字 (\n，\t，\b，\f，\r，\\，\'，\'') を用いることで，どのようなことが可能となるだろうか？ たとえば，実際に以下のようなプログラム (部分) を書いて試してみなさい．ただし，これだけだと効果がよくわからない場合もあるので，その場合には多少書き換えること．

```
public class Backslash1 {
  public static void main(String[] args) {
    System.out.println("\\n:");
    System.out.println("[[" + "\n" +"]]");
    System.out.println("\\t:");
    System.out.println("[[" + "\t" +"]]");
    System.out.println("\\b:");
    System.out.println("[[" + "\b" +"]]");
    System.out.println("\\f:");
    System.out.println("[[" + "\f" +"]]");
    System.out.println("\\r:");
    System.out.println("[[" + "\r" +"]]");
    System.out.println("\\\\:");
    System.out.println("[[" + "\\" +"]]");
    System.out.println("\\\':");
    System.out.println("[[" + "\'" +"]]");
    System.out.println("\\\":");
    System.out.println("[[" + "\"" +"]]");
  }
}
```

第3回 - 制御構造 ～ 数値データ

B.5 例題プログラムの実行

プリント中の以下のプログラムを実行して，実行結果について考察しなさい．プリント中にプログラム全文が示されている例題プログラムは，プログラム文は印刷せずに，実行結果と考察だけを提出すること．また，実行結果の出力が長いものは適当に省略してよい．

2.13 節　例 1　2.14 節　例 1, 3　2.15 節　例 1　2.16 節　例 2
2.18 節　例 1　2.19 節　例 1, 3, 4

B.6 階乗のプログラム

2.17 節の例 1 や例 2 を参考にして，階乗 ($n! = n \times \cdots \times 1 = 1 \times 2 \times \cdots \times n$) を求めるプログラムを作ること．プログラムはコマンド入力で数値 n を受けとり (2.6 節参照)，階乗の計算結果 $n!$ を表示するものとする．ただし，変数が `int` 型と `double` 型の 2 種類のプログラムを作り，実行結果を比較しなさい*.

*n として 1～20 の範囲や 200 などを与えると，どのような結果が得られるか試してみること

第4回 - Javaの入出力 ～ メソッド，ゲームのプログラム

B.7 例題プログラムの実行

プリント中の以下のプログラムを実行して，実行結果について考察しなさい．プリント中にプログラム全文が示されている例題プログラムは，プログラム文は印刷せずに，実行結果と考察だけを提出すること．また，実行結果の出力が長いものは適当に省略してよい．なお，2.21 節の例 2 については，例 1 の実行結果で示されているリダイレクションも含めて試すこと．

2.20 節　例 1　2.21 節　例 1, 2, 3　2.22 節　例 1, 2, 3

B.8 数当てゲームのプログラム

以下のプログラムについて，次のような (少なくとも 5 種以上の) 改良を行なうこと†．なお，改良点が良く分かるように，プログラム中の改良部分にコメントをつけたり，考察を各改良点ごとにまとめて書いたりすること．

1. 当てる数の範囲を変える．
2. 試行の回数を変える．
3. 外れた時に正解を表示する．
4. 1 回目に当たった時は「大当たり!!」と表示する．
5. ゲームを繰返し実行する．
6. `do ～ while` ではなく，`while ～` を用いる．
7. 毎回の試行回数を表示する．

これらはあくまで改良点のヒントであり，もちろん自分なりの改良を加えて構わない．

```
import java.io.*;
public class Game1 {
  public static void main(String[] args) {
    try {
      final int  MAXTIMES = 3;
      final int  RANGE = 10;
      BufferedReader br =
        new BufferedReader(new InputStreamReader(System.in));
      System.out.println("0以上" + (RANGE-1) + "以下の数を" +
                         MAXTIMES + "回以内で当てよ");
      int answer = (int)(Math.random()*RANGE);
      int times = 0;
      int trial;
      do {
        System.out.print("数は? -> ");
        trial = Integer.parseInt(br.readLine());
```

†プログラムの 10 行目に現れる `Math.random()` は，0.0 以上 1.0 未満の乱数を返すメソッド (関数のようなもの) である．

```
        if (trial < answer)  System.out.println("小さすぎる");
        if (trial > answer)  System.out.println("大きすぎる");
        times++;
      } while ((trial != answer) && (times < MAXTIMES));
      if (trial == answer)
        System.out.println("やった!! " + times + "回で当たった");
      else
        System.out.println("残念!");
    }
    catch (Exception e) {
      System.out.println("Error: " + e);
      e.printStackTrace();
    }
  }
}
```

第5回 - メソッド 〜 配列，ハノイの塔

B.9 例題プログラムの実行

プリント中の以下のプログラムを実行して，実行結果について考察しなさい．プリント中にプログラム全文が示されている例題プログラムは，プログラム文は印刷せずに，実行結果と考察だけを提出すること．また，実行結果の出力が長いものは適当に省略してよい．

2.23 節　例 3, 4　2.24 節　例 2, 3　2.25 節　例 1, 2, 3
2.28 節　例 2, 3

B.10 配列のプログラム

2.25 節の例 4 で用いた県のデータファイル `KantoArea.dat` や `JapanArea.dat` を読み込み，以下の作業を実行するプログラム (1 つ) を書くこと．

1. 面積が最大の県の番号とその値 (最大値) を求める．
2. 面積が最小の県の番号とその値 (最小値) を求める．
3. 面積の順位が中間の県の番号とその値を求める．
4. 面積の平均を求める．
5. 各県の面積の最大値に対する割合 (比率) を求める．

B.11 再帰呼出 - ハノイの塔

3 本の棒 1, 2, 3 が立っており，大きさの異なる n 枚の穴の空いた円盤が大きさの順に (大きいものが下，小さいものが上) 棒に挿して積み重ねられている．円盤を一枚ずつ動かして，すべての円盤を棒 1 から棒 2 に移動する．このとき小さい円盤の上に大きい円盤を載せないようにする．以下の 2 点を解決することによって，1 回毎の操作と操作の全回数を表示するプログラムを完成せよ．

1. 1 回毎の操作の表示 - 解答例の参考プログラムの空欄を適当に埋める．
2. 操作の全回数の表示 - 操作の回数を `int` 型の値として返し，最終的に表示する (実行結果の例の下線部) ようにプログラムを修正する．

ヒント：

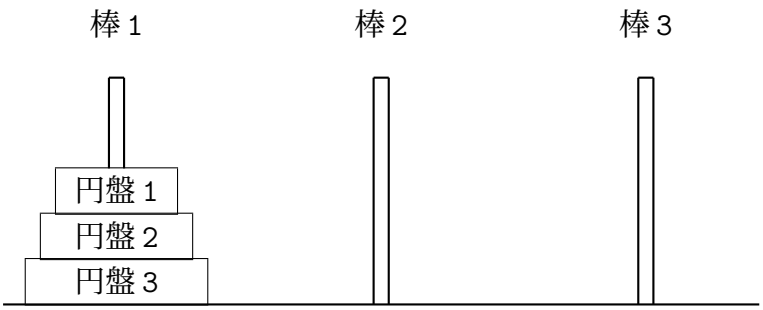
1. 移動する円盤がない場合には，何も動かさなくてよい．
2. 移動する円盤が n 枚の場合には，① まず上の $(n-1)$ 枚を横によけて，② 次に一番下の 1 枚を動かし，③ 最後に横によけた $(n-1)$ 枚を上に載せればよい．

再帰呼出のプログラムを考えるときには，具体的な値を入れてプログラムの実行を追いかけようとすると破綻しやすい．それよりは，メソッドを仕様 (引数の並びや戻し値) と機能 (働き) という観点で捉えて，1 つ小さい問題 (n の問題に対する $(n-1)$ の問題) に分解することが重要である．

```
public class Hanoi1 {
  public static void main(String[] args) {
    int  n = Integer.parseInt(args[0]);
    System.out.print("円盤の枚数 -> " + n);
    hanoi(1, 2, 3, n);
  }
  static void hanoi(int src, int dst, int buf, int n) {
    if (      ) {
      
      System.out.println("円盤" + n + "を棒" + src + "から棒" + dst + "に移動する");
      
    }
  }
}
```

実行結果の例 (ここでは円盤 3 枚の場合)

```
円盤の枚数 -> 3 ¶
円盤 1 を棒 1 から棒 2 に移動する
円盤 2 を棒 1 から棒 3 に移動する
円盤 1 を棒 2 から棒 3 に移動する
円盤 3 を棒 1 から棒 2 に移動する
円盤 1 を棒 3 から棒 1 に移動する
円盤 2 を棒 3 から棒 2 に移動する
円盤 1 を棒 1 から棒 2 に移動する
全移動回数 = 7
```



第6回 - クラスとインスタンス (1)

B.12 例題プログラムの実行

プリント中の以下のプログラムを実行して，実行結果について考察しなさい．プリント中にプログラム全文が示されている例題プログラムは，プログラム文は印刷せずに，実行結果と考察だけを提出すること．

2.29節 例1, 2 2.30節 例2, 4

B.13 プログラムの解読

以下のプログラム (WWWページからダウンロードできる) を解読して，その働きを説明せよ．また，余力のあるものは，プログラムを改良せよ．

```
import java.io.*;

public class CardStock {
  static BufferedReader br;
  int[] cards;

  CardStock(int n) {
    cards = new int[n];
    for (int i = 0; i < n; i++)
      cards[i] = i+1;
  }

  int discard(int i) {
    int card = -1;
    if ((i >= 0) && (i < cards.length)) {
      card = cards[i];
      int[] newcards = new int[cards.length-1];
      for (int j = 0; j < i; j++)
        newcards[j] = cards[j];
      for (int j = i+1; j < cards.length; j++)
        newcards[j-1] = cards[j];
      cards = newcards;
    }
    else
      System.out.println("Wrong index \'" + i +
                         "\',  # of cards: " + cards.length);
    return card;
  }
```

```
int automaticDiscard(int sum, int goal) {
  if (goal <= sum + cards[cards.length-1]) {
    for (int i = cards.length - 2; i >= 0; i--)
      if (cards[i] < (goal - sum))
        return  discard(i);
  }
  return  discard((int)(Math.random()*cards.length));
}

int userDiscard(int sum, int goal) {
  int i;
  System.out.println("You have the following cards:");
  for (i = 0; i < cards.length; i++)
    System.out.print("  " + cards[i]);
  System.out.println();
  do {
    System.out.print("Select a card -> ");
    try {
      int card = Integer.parseInt(br.readLine());
      for (i = 0; i < cards.length; i++)
        if (cards[i] == card)  break;
    }
    catch (Exception e) {
      i = cards.length;
    }
  } while (i >= cards.length);
  return  discard(i);
}

public static void main(String[] args) {
  if (args.length < 2) {
    System.out.println("Usage: java CardStock <# of players> <# of cards>");
  }
  else {
    System.out.println("You are player #0.");
    int nPlayers = Integer.parseInt(args[0]);
    int nCards = Integer.parseInt(args[1]);
    br = new BufferedReader(new InputStreamReader(System.in));
    int i = 0;
    for (; nPlayers > 1; nPlayers--) {
      int limit = nCards*(nCards-1)*nPlayers/3;
```

```
        System.out.println("NEW GAME with " + nPlayers + " players.  " +
                           "You may not exceed " + limit + ".");
        CardStock[] players = new CardStock[nPlayers];
        for (i = 0; i < nPlayers; i++)
          players[i] = new CardStock(nCards);
        int sum = 0;
        i = -1;
        do {
          i = (i+1) % nPlayers;
          int card = 0;
          if (i == 0)
            card = players[i].userDiscard(sum, limit);
          else
            card = players[i].automaticDiscard(sum, limit);
          sum += card;
          System.out.println("Player #" + i + " discards " + card +
                             ", the current sum is " + sum);
        } while (sum < limit);
        if (i != 0)
          System.out.println("Player #" + i + " lost!!\n");
        else
          break;
      }
      if (i != 0)
        System.out.println("You are the WINNER!!!");
      else
        System.out.println("You are a LOSER!!! ( " +
                           nPlayers + " / " + args[0] + " )");
    }
  }
}
```

第7回 - クラスとインスタンス (2)

B.14 例題プログラムの実行

プリント中の以下のプログラムを実行して，実行結果について考察しなさい．プリント中にプログラム全文が示されている例題プログラムは，プログラム文は印刷せずに，実行結果と考察だけを提出すること．

2.31 節 例 1,3 2.32 節 例 1, 2, 3

B.15 求解のプログラム

2.32 節の例 4 のクラス `Function` を拡張して，$x^2 = c,\ x^3 = c,\ x^4 = c,\ x^5 = c,\ \log x = c$ などの解を求めるプログラムを書け．log の計算には，`Math.log` メソッドを用いよ．
なお，方程式 $x^n = c$ の指数 n が偶数か奇数かで，解の探索範囲 `lower`，`upper` が異なることに着目して，指数が偶数の場合と奇数の場合の 2 つの中間的なクラスを作る方法も考えられる．余裕があるもの (上記の説明で納得できたもの) は，中間のクラスを作ってみること．

第8回 - サーチとソート ～ クラスとインスタンス (2)

B.16 例題プログラムの実行

プリント中の以下のプログラムを実行して，実行結果について考察しなさい．プリント中にプログラム全文が示されている例題プログラムは，プログラム文は印刷せずに，実行結果と考察だけを提出すること．

5.3.3 節　例 1　2.32 節　例 1, 2, 3

B.17 バイナリサーチ

5.1.2 節のバイナリサーチのプログラム (例１) は再帰呼出を利用している．しかし，このプログラムは末尾再帰であるから，単純な反復計算のプログラムにできる．再帰呼出は用いずに，反復 (`while` や `do`～`while`) を用いたバイナリサーチのプログラムを書いて，実行しなさい．

B.18 バブルソート

5.2.2 節のバブルソートのプログラム (例１) を完成して実行しなさい．

B.19 クイックソート

5.3.2 節のクイックソートのプログラム (例１) を完成して実行しなさい．

B.20 ソートの比較

5.3.4 節の例 1 のプログラム (最小値選択法のソート時間計測) に従って，バブルソート，クイックソート，マージソートの実行時間計測プログラムを書いて，実行しなさい．これら 4 つのソートプログラム (最小値選択法，バブルソート，クイックソート，マージソート) の実行時間を計測して，その結果について考察せよ．なお実行時間を計測する際には，各ソートプログラムごとにデータ量を変更して (5～10 通り程度)，データ量と実行速度の変化の関係を調べ，計算量と関連させて考察すること．

課題や授業について (採点の対象外)

レポートの作成に要した時間，特に非常に時間がかかった場合には何に苦労したか，を書くこと．また，授業についての感想や希望があれば，自由に書いて良い (歓迎する)．

提出方法 – 体裁の整っていないレポートは受理しないので注意せよ

1. report_format.html を用いて HTML 文書を作成する.
2. 複数枚に渡る場合には **左上** をホチキスやノリで止めること.
3. プログラム中の説明箇所には，コメントなど入れて分かり易くすること.
4. プログラムの実行結果を印刷すること.
5. プログラムの説明はできるだけ **箇条書** で書くこと.

第 9 回 - クラスとインスタンス (2) 〜 アプレットとグラフィクス

B.21 例題プログラムの実行

プリント中の以下のプログラムを実行して，実行結果について考察しなさい．プリント中にプログラム全文が示されている例題プログラムは，プログラム文は印刷せずに，実行結果と考察だけを提出すること．なお，実行結果画像の張り込みは，APPLET タグか IMAGE タグを利用する．それぞれ，report_applet.html と report_image.html というファイルを参考にすると良い．詳細については WWW ページの説明を読むこと．

2.32 節　例 4, 5　　2.33 節　例 2, 4, 6, 7

B.22 求解のプログラム

2.32 節の例 4 のクラス Function を拡張して，$x^2 = c,\ x^3 = c,\ x^4 = c,\ x^5 = c,\ \log x = c$ などの解を求めるプログラムを書け．log の計算には，Math.log メソッドを用いよ．
なお，方程式 $x^n = c$ の指数 n が偶数か奇数かで，解の探索範囲 lower，upper が異なることに着目して，指数が偶数の場合と奇数の場合の 2 つの中間的なクラスを作る方法も考えられる．余裕があるもの (上記の説明で納得できたもの) は，中間のクラスを作ってみること．

課題や授業について (採点の対象外)

レポートの作成に要した時間，特に非常に時間がかかった場合には何に苦労したか，を書くこと．また，授業についての感想や希望があれば，自由に書いて良い (歓迎する).

提出方法 – 体裁の整っていないレポートは受理しないので注意せよ

1. report_format.html を用いて HTML 文書を作成する．
2. 複数枚に渡る場合には **左上** をホチキスやノリで止めること．
3. プログラム中の説明箇所には，コメントなど入れて分かり易くすること．
4. プログラムの実行結果を印刷すること．
5. プログラムの説明はできるだけ **箇条書** で書くこと．

第10回 - アプレットとグラフィクス ～ イベント処理

B.23 例題プログラムの実行

プリント中の以下のプログラムを実行して，実行結果について考察しなさい．プリント中にプログラム全文が示されている例題プログラムは，プログラム文は印刷せずに，実行結果と考察だけを提出すること．なお，実行結果画像の張り込みは，APPLET タグか IMAGE タグを利用する．それぞれ，report_applet.html と report_image.html というファイルを参考にすると良い．詳細については WWW ページの説明を読むこと．

2.33 節　例 4, 8　2.34 節　例 1, 2, 3, 4, 5

B.24 正弦 (sin) 曲線の描画

図 B.1 のように関数曲線 $y = \sin x \quad (-\pi \leq x \leq \pi)$ を描画するプログラムを完成せよ．
ヒント： 座標系の原点が左上隅にあり，y 軸が下向きに正の方向であることに注意せよ．

```
import java.applet.*;
import java.awt.*;
public class SineCurve extends Applet {
  public void paint(Graphics g) {
    Dimension d = getSize();
    g.drawLine(0, d.height/2, d.width, d.height/2);
    g.drawLine(                                    );
    int oldY =                                      
    for (int x = 1; x <= d.width; x++) {
      double theta =                                
      int newY = (int)(d.height * (1.0 - Math.sin(theta)) / 2.0);
      g.drawLine(x-1, oldY, x, newY);
      oldY =      
    }
  }
}
```

B.25 ダイアモンドパターンの描画

図 B.2 のように正 n 角形の全対角線を描いた図形をダイヤモンドパターンと呼ぶ．ダイヤモンドパターンの表示するには，円周上に均等に置かれた n 点から全ての 2 点の組を選んで線分を描けばよい．
ヒント：円を描くプログラム (2.33 節 例 8) を参考に，円周上の点の座標値 x と y を一旦すべて配列に格納してから，線分を描画するとプログラムは簡単になる．

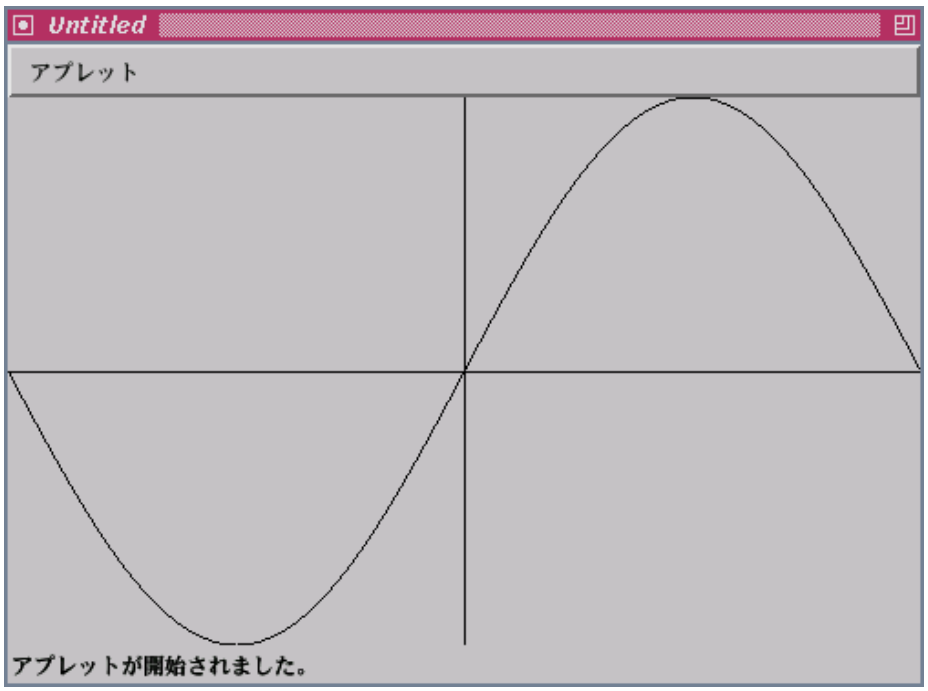


図 B.1: 正弦 (sin) 曲線

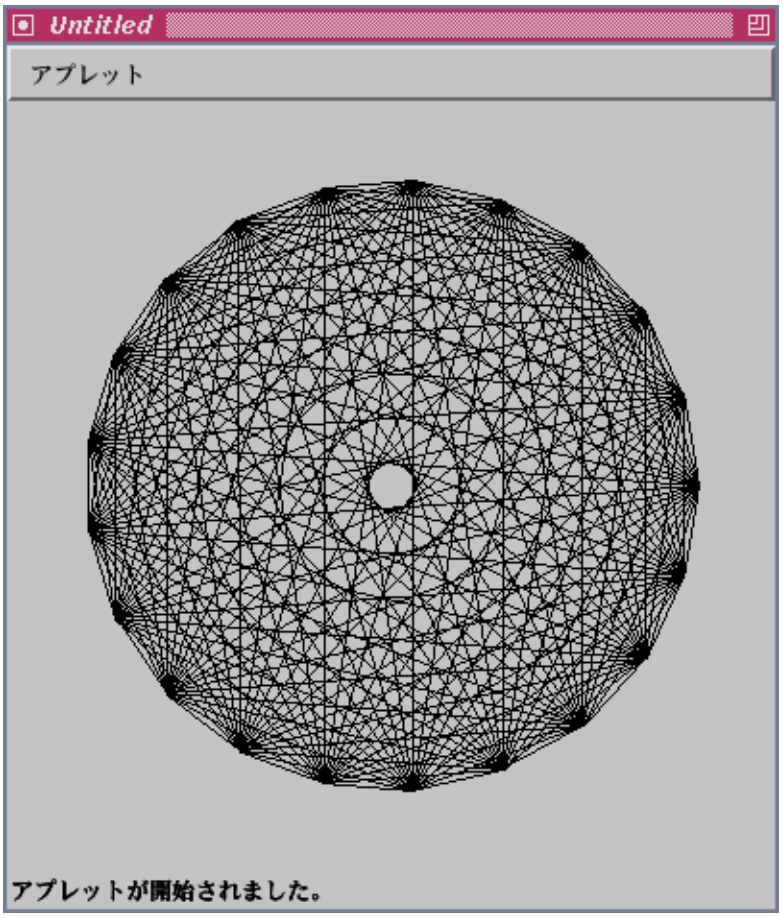


図 B.2: ダイアモンドパターン

課題や授業について (採点の対象外)

レポートの作成に要した時間，特に非常に時間がかかった場合には何に苦労したか，を書くこと．また，授業についての感想や希望があれば，自由に書いて良い (歓迎する).

提出方法 – 体裁の整っていないレポートは受理しないので注意せよ

1. `report_applet.html` ないし `report_image.html` などを用いてアプレット画像の入った HTML 文書を作成する.
2. 複数枚に渡る場合には **左上** をホチキスやノリで止めること.
3. プログラム中の説明箇所には，コメントなど入れて分かり易くすること.
4. プログラムの実行結果を印刷すること.
5. プログラムの説明はできるだけ **箇条書** で書くこと.

第11回 - イベント処理

B.26 マウスによる背景色の変更

2.34節の例1「マウスイベントによる背景色の変更」を参考に，次のような動きをするAppletプログラムを作成せよ．

1. 起動時の背景は白
2. マウスカーソルがApplet内に入ると背景は青
3. Applet内でボタンをプレスすると背景は赤
4. Applet内でボタンをリリースすると背景は緑
5. マウスカーソルがApplet外に出ると背景は白

B.27 マウスによる円の変更

2.34節の例4「マウスによる線分の描画」を参考に，次のような動きをするAppletプログラムを作成せよ．

1. マウスボタンをプレスしてドラッグすると円を描く
2. ボタンをプレスした位置が円の中心となる
3. マウスをドラッグすると，円はカーソル位置を通過するように拡大縮小する
4. 円の描画には，次のようにdrawOvalメソッドを用いるとよい
   `getGraphics().drawOval(`左上x座標，左上y座標，直径，直径`)`

B.28 プログラムの解読

以下のプログラム (WWWページからダウンロードできる) を解読して，その働きを説明せよ．また，余力のあるものは，プログラムを改良せよ．

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/*
  <applet code="ClickSpeed" width="300" height="300">
  </applet>
*/

public class ClickSpeed extends Applet {
  static final int TIMES = 10;
  static final int SIZE = 10;
  static long fastest = -1;
  int  count = 0;
  long time;
  int  x, y;
```

```
  public void init() {
    addMouseListener(new CSMouseAdapter(this));
  }

  public void mousePressed(MouseEvent me) {
    if (count > 0) {
      if ((me.getModifiers() & me.BUTTON1_MASK) != 0) {
        int xc = me.getX();
        int yc = me.getY();
        if ((x < xc) && (xc < (x+SIZE)) && (y < yc) && (yc < (y+SIZE))) {
          count--;
          setForeground(Color.black);
        }
        else {
          count++;
          setForeground(Color.red);
        }
        if (count > 0) {
          showStatus("" + count + " more!!");
          repaint();
        }
        else {
          time += System.currentTimeMillis();
          if ((fastest < 0) || (fastest > time))  fastest = time;
          showStatus("Finished in " + (time/1000.0) + " secs.  " +
                     "Fastest time: " + (fastest/1000.0) + " secs.");
          repaint();
        }
      }
    }
    else {
      if (fastest < 0)
        showStatus("Click right button to start.");
      else
        showStatus("Click right button to start. Fastest time: "
                   + (fastest/1000.0) + " secs.");
    }
  }

  public void mouseReleased(MouseEvent me) {
    if ((me.getModifiers() & me.BUTTON3_MASK) != 0) {
      count = TIMES;
```

```
      showStatus("Start clicking ... " + TIMES + " more!!");
      setForeground(Color.black);
      time = -System.currentTimeMillis();
      repaint();
    }
  }

  public void paint(Graphics g) {
    if (count > 0) {
      Dimension d = getSize();
      x = (int)((d.width - SIZE) * Math.random());
      y = (int)((d.height - SIZE) * Math.random());
      g.fillRect(x, y, SIZE, SIZE);
    }
  }
}

class CSMouseAdapter extends MouseAdapter {
  private ClickSpeed cs;
  public CSMouseAdapter(ClickSpeed cs) {
    this.cs = cs;
  }
  public void mousePressed(MouseEvent me) {
    cs.mousePressed(me);
  }
  public void mouseReleased(MouseEvent me) {
    cs.mouseReleased(me);
  }
}
```

B.29 自由課題 - オリジナルプログラム

これまで勉強してきたことを応用して，自分で考えたプログラムを書くこと．他人に「これは面白い！」と思わせるようなプログラムを目指そう．プログラムとして凝ったものである必要は特にない．同じことができるならば，より簡潔なプログラムの方が望ましい．この課題は自由課題であり，提出する必要はないが，成績評価にあたっての加点要素とする．

課題や授業について (採点の対象外)

レポートの作成に要した時間，特に非常に時間がかかった場合には何に苦労したか，を書くこと．また，授業についての感想や希望があれば，自由に書いて良い (歓迎する)．

提出方法 – 体裁の整っていないレポートは受理しないので注意せよ

1. report_applet.html ないし report_image.html などを用いてアプレット画像の入った HTML 文書を作成する.
2. 複数枚に渡る場合には **左上** をホチキスやノリで止めること.
3. プログラム中の説明箇所には，コメントなど入れて分かり易くすること.
4. プログラムの実行結果を印刷すること.
5. プログラムの説明はできるだけ **箇条書** で書くこと.

提出方法 (自由課題のみ - 採点対象外で遅れて提出する場合)

1. report_format.html などを用いて HTML 文書を作成する.
2. 作成した HTML 文書を java ファイルとともに yama@mail.ecc.u-tokyo.ac.jp 宛にメールする.

自主課題 - 数値計算と数値誤差

自然科学系 (特に工・理・農学系) に進学する者にとって，数値計算の初歩を学ぶことは重要だと思われる．駒場祭や冬休み期間中の自主勉強用の題材として数値計算の課題を示しておく．

- 必須課題ではないので提出の必要はない．
- 冬休み明け (1 月 8 日) の授業までに提出されれば，成績に加味される可能性がある．

B.30 二分探索法

4.1 節の二分探索法のプログラム (例１) を完成して実行しなさい．

B.31 ニュートン法

4.2 節のニュートン法のプログラム (例１) を完成して実行しなさい．さらに，二分探索法とニュートン法のプログラムの実行結果を比較して考察しなさい．この際に，二分探索法では $a = 0, b = 3$，ニュートン法では $x_c = 3$ を，それぞれの初期値として用いよ．

B.32 例題プログラムの実行

プリント中の以下のプログラムを実行して，実行結果について考察しなさい．プリント中にプログラム全文が示されている例題プログラムは，プログラム文は印刷せずに，実行結果と考察だけを提出すること．

4.4 節　例１　4.5 節　例１　4.6 節　例１　4.7 節　例１

B.33 台形公式

4.8 節の台形公式のプログラム (例１) を完成して実行しなさい．

B.34 シンプソン公式 (選択課題)

4.8 節のシンプソン公式のプログラムを台形公式のプログラムを参考に作成して実行しなさい．さらに，これら２つのプログラムの実行結果を比較して考察しなさい．ただし，比較にあたっては，台形公式と同じ Δx とするために，シンプソン公式のプログラムは $n = 5, 50, 500, 5000, 50000, 500000, 5000000, 50000000$ で計算しなさい．
ヒント：台形公式のプログラムを 5 行書き換えるだけである．

課題や授業について (採点の対象外)

レポートの作成に要した時間，特に非常に時間がかかった場合には何に苦労したか，を書くこと．また，授業についての感想や希望があれば，自由に書いて良い (歓迎する)．

提出方法 – 体裁の整っていないレポートは受理しないので注意せよ

1. `report_format.html` を用いて HTML 文書を作成する．
2. 複数枚に渡る場合には **左上** をホチキスやノリで止めること．
3. プログラム中の説明箇所には，コメントなど入れて分かり易くすること．
4. プログラムの実行結果を印刷すること．
5. プログラムの説明はできるだけ **箇条書** で書くこと．

索 引