1. 選択プログラミング技術で学ぶこと

（1） Java を学ぶ

プログラミングは、FORTRAN から始まり COBOL、LISP、BASIC、そして C 言語と、さまざまなプログラミング言語が開発され、さまざまな用途に合わせて利用されてきました。
もちろん、現在もさまざまなプログラミング言語が利用されていますが、その多くが「オブジェクト指向」という考え方が取り入れられ改良されてきています。たとえば、C 言語にオブジェクト指向の考え方を取り入れた「C++」や、Excel 等のマクロに利用されている「VBA」も BASIC にオブジェクト指向を取り入れたものです。
すなわち、現在のプログラミングの主流は「オブジェクト指向」といっても過言ではないのです。
そこで、この授業ではオブジェクト指向のプログラミング言語である「Java」について学んでいきます。
Java は、C++等のように旧来あったプログラミング言語にオブジェクト指向の機能を追加したものではなく、オブジェクト指向を利用するために新しく作られたプログラミング言語なので、純粋なオブジェクト指向プログラミング言語と言えます。また、C 言語に似ている部分も多いので、我々にとっては学びやすい言語です。

（2） オブジェクト指向って...

オブジェクト指向が利用されるようになった最も大きな理由は、プログラム開発の効率化です。オブジェクト指向の考え方は、「カプセル化」、「継承」、「多態性（多相性、ポリモーフィズム）」の３つにあります。それぞれの考え方をしっかり理解し、「いいプログラム」を作れるようになってほしいと思います。

（3） i アプリの制作

この授業を通して Java の文法について学んでいきますが、文法だけではおもしろくありません。せっかくなので最終的には実用的なプログラミングにチャレンジしたいと思います。
そこで、携帯電話で利用されている i アプリを作ってみたいと思います。i アプリも実は Java で作られているのです。この授業を通して、どんな i アプリがあったら便利か考えながら授業に参加してください。

2. java とは

（1） Sun が開発したプログラミング言語

Java は、1995 年頃に Sun Microsystems 社によって発表されたプログラミング言語です。

（2） Java の語源

Java の語源はコーヒーの Java（ジャワ）です。ただし、プログラミング言語の時は「ジャバ」と表記することが多いようです。英語の発音は「ジャヴァ」。何故、コーヒーの名前になったかには諸説あり、本当の理由は謎につつまれています。

（3） JavaScript とは別物

Java と JavaScript は名前も文法も似ていますが、別物です。Java で記述されたスクリプトが JavaScript ではありませんし、JavaScript の略称が Java でもありません。ルーツは同じであるとしても、英語とドイツ語くらい異なるものです。
JavaScript は、Web ブラウザなどでの利用に適したスクリプト言語(簡易プログラミング言語)。Sun 社の Java 言語に似た記法を用いることが名称の由来だが、直接の互換性は無い。従来は印刷物のような静的な表現しかできなかった Web ページに、動きや対話性を付加することを目的に開発され、主要な Web ブラウザのほとんどに搭載されている。ブラウザ以外のソフトウェアにも簡易な制御プログラムの記述用言語として移植されており、Microsoft 社の Windows や Web サーバソフト「IIS」、Macromedia 社の「Flash」などに、JavaScript あるいは類似の言語の処理系が内蔵されている。

（4） オブジェクト指向プログラミングが可能

Java は、「オブジェクト指向」的なプログラミングが可能な言語です。オブジェクト指向とは継承機能を持つクラスに基づいてインスタンスを生成することで記述性を高める・・・と言っても説明しきれないので、どこか別の場所で詳しく説明します。オブジェクト指向プログラミング言語には他に、Smalltalk、C++、C# などがあります。

（５） 中間コード（Java バイトコード）へのコンパイル言語

プログラミング言語は、プログラムを逐次解析しながら実行する インタープリタ型言語 と、あらかじめマシン語コードに変換しておく コンパイル型言語 に大別されます。Java は基本的にはコンパイル型言語ですが、CPU に依存したマシンコードではなく、CPU に依存しない 中間コード（Java バイトコード） にコンパイルするのが特徴です。

（６） 仮想マシン VM

CPU に依存したマシンコードは CPU が実行しますが、Java のような CPU に依存しない中間コードは、Java の実行環境に含まれるソフトウェアが読み込み、CPU 依存のマシンコードに変換しながら実行します。このソフトウェアを Java VM（Virtual Machine：仮想マシン）と呼びます。

（７） Write Once, Run Anywhere

「一度書いたら、どこでも動く」。Java では CPU や OS に依存しないプログラミングが可能で、一度書いたプログラムは、Windows や UNIX 系 OS や携帯電話など、様々なプラットフォームで動かすことが可能と言われています。ただし、Java VM のバージョンなどに強く左右され、まだまだ思うようには動かないのが現実のようです。

（８） Java のエディション

① J2EE（Java 2 Platform, Enterprise Edition）

サーバサイドで動作するアプリケーション向けの Java プラットフォームです。オンラインショップ、会員制サイトなど Web アプリケーションの開発でよく使用されます。主要 API には、サーブレット（Web アプリ開発）、JSP（スクリプトベース Web アプリ開発）、EJB（DB、トランザクション管理）などがあります。

② J2SE（Java 2 Platform, Standard Edition）

すべての Java において基本となるプラットフォームです。一般にデスクトップ環境向け Java プラットフォームと言われています。業務用ソフト、アプリ開発ソフトなどクライアント PC で動作するアプリケーションの開発でよく使用されます。主要 API には Swing（GUI アプリ開発）、Applet（Web ブラウザアプリ開発）、JDBC（DB 接続）などがあります。

③ J2ME（Java 2 Platform, Micro Edition）

モバイル環境向け Java プラットフォームです。処理速度、データ容量などリソースが乏しい環境でもスムーズに Java アプリが動作するように最適化されています。主要 API はカーナビ、セットトップボックスなど比較的大きめの端末向けの CDC と、PDA・携帯電話など比較的小さめの端末向けの CLDC にかれています。CLDC の元で動作する JVM は特に KVM と呼ばれ（K は K バイトのメモリで動作の意味）、モバイル端末のリソースに合わせ最適化されています。

④ Java Card

スマートカード（IC 搭載カード）環境向け Java プラットフォームです。Java Card 環境を実装したスマートカードであれば、スマートカード発行会社を問わず、同一の Java アプリを動作させることができます。

（９） Java の配布形態

J2SE は、下記の 2 つの形態で配布されています。Java のソフトウェアを動かすには JRE が、開発するには SDK が必要です。SDK は Java 1.1 までは JDK（Java Development Kit）と呼ばれていました。

① JRE(Java Runtime Environment) → Java の実行環境のみ。

② SDK(Software Development Kit) → Java の開発環境。JRE を含む。

Java はプログラムが動作する環境によりいくつかのプラットフォームに分類されます。

3. Java を利用するには

（1） JDK をインストールする

Java プログラミングを体験するには、Java の開発環境 JDK（Java Development Kit）が必要です。JDK は Sun のサイトから無償で入手することができます。2004 年 11 月時点での JDK の入手・インストール方法（Windows 用）を説明します。

① http://java.sun.com/ を開く。

② 左上のメニューから [Downloads] を開く。

③ J2SE の一覧から [J2SE 1.4.2 - Japanese Edition] を選択して [Go]。
④ [J2SE SDK のダウンロード] をクリック。

⑤ ライセンス規約を読んで [ACCEPT] ボタンを押して [continue]。

⑥ `[Download j2sdk-1_4_2_06-windows-i586-p.exe]` をクリック。

⑦ [保存] ボタンを押して `[j2sdk-1_4_2_06-Windows-i586-p.exe]` を保存。

⑧ `j2sdk-1_4_2_06-Windows-i586-p.exe` をダブルクリックしてインストールする。

⑨ JDK は通常、`C:\j2sdk1.4.2_06` といったフォルダにインストールされます。

（2） 環境を設定する

JDK を使用するには、環境変数 PATH に、javac.exe などへのパスを設定する必要があります。JDK が `C:\j2sdk1.4.2_06` にインストールされている場合の設定方法（Windows XP）を説明します。

① [スタート] →([設定])→ [コントロールパネル] →([パフォーマンスとメンテナンス])→ [システム] を起動。

② [詳細設定] → [環境変数] ボタンをクリック。

③ ユーザー環境変数に PATH が登録されている場合は PATH を選択して [編集] をクリックし、[変数値] の末尾に `;C:\j2sdk1.4.2_06\bin` を加えて `[OK]`。 ユーザー環境変数に `PATH` が登録されていない場合は `[新規]` をクリックして、`[変数名]` に `PATH` を、`[変数値]` に `C:\j2sdk1.4.2_06\bin` を入力して [OK]。

④ これにより、次回、コマンドプロンプト（MS-DOS プロンプト）起動時に、環境変数 PATH に JDK へのパスが設定されます。下記の手順で確認してみましょう。

⑤ [スタート] → [（すべての）プログラム] → [アクセサリ] → [コマンドプロンプト] を起動。

⑥ `set` コマンドを実行。
環境変数 `Path` に、`C:\j2sdk1.4.2_06\bin` が含まれていれば OK。

```
C:\>set ← 環境変数を表示
Path=C:\WINDOWS\system32;C:\WINDOWS;C:\j2sdk1.4.2_06\bin
```

（3） 作業環境を整える

① 作業フォルダの作成

ホームディレクトリ上に「java」というフォルダを作成しましょう。これは Java の学習を行うために、Java の作業用フォルダ（ディレクトリ）として、Java の学習に関するファイル等を保存しましょう。
また、これからコマンドプロンプト上でプログラムの開発を行っていきます。コマンドプロンプト上で作業用フォルダ（ディレクトリ）で行うには以下の作業を行います。なお、コマンドプロンプトとは Windows 上での CUI 環境です。

[スタート]→[プログラム]→[アクセサリ]→[コマンドプロンプト]

```
C>mkdir java    ← javaフォルダを作成
C>cd java       ← javaフォルダに移動
C>
```

② エディタを利用する

プログラムを製作するのに、エディタを利用します。テキストエディタなら何でもいいのですが、せっかくですからxyzzyを利用しましょう。

xyzzyは、Windowsで Emacsを利用するために開発されたエディタです。Emacs同様、エディタ内でシェル（コマンドプロンプト）が利用できるので大変便利です。

③ Eclipsの利用

Javaには、プログラム開発の統合環境として、Eclipsというソフトがあります。Eclips自身、Javaで作成され、フリーで利用できるソフトです。

プログラムの作成から実行まで、Eclips上で行うことが出来ます。

（4） Javaアプリケーションを作成する

① ソースプログラムの作成

以下の内容のファイルを作成し、ファイル名 test01.java で保存してください。ファイル名は大文字・小文字も正確に区別して指定してください。

```
class test01 {
    public static void main(String[] args) {
        System.out.println("Hello World!!");
    }
}
```

※ ここで、注意したいのがファイル名とクラス名です。今回作成したファイルはtest01.javaというファイル名と作成するクラスの名前を同じにしておく必要があります。

※ C言語は関数単位でプログラムを作成しましたが、Javaではクラス単位でプログラムを作成していきます。

※ クラスとは、C 言語の構造体とよく似ています。構造体の中に納められるのは「データ」だけでしたが、クラスは「データ」（フィールド）と「処理（関数）」（メソッド）を納められたものです。このようにフィールドとメソッドを一体化しフィールドデータを保護するように設計することを「カプセル化」といいます。

※ C言語同様1つの命令を書き終えたら、「;」をつけます。また{}で処理をまとめていくこともC言語によく似ています。

※ System.out.println は、指定した文字列や変数を画面上に表示する命令です。表示後に改行を自動的に挿入してくれます。同様にして改行を入れない System.out.print という命令もあります。

② コンパイル

作成したソースプログラムをコンパイル、中間コード（Javaバイトコード）を作成しましょう。

test01.java を、javac コマンドでコンパイルします。

```
C>javac test01.java  ← .javaファイルをコンパイル
```

中間コード（Javaバイトコード）が作成されていることをdirコマンドで確認してみます。dirコマンドは、UNIXでの lsコマンドに相当します。

```
C>dir

2004/11/07  02:07               427 test01.class
2004/11/07  02:06               110 test01.java
```

コンパイルによってできあがったファイル test01.class が中間コード（Java バイトコード）です。

③ 実行

test01.class ファイルを java コマンドで実行してみましょう。

```
C>java test01     ← .class ファイルを実行
Hello World!!
```

「Hello World!! 」と表示されれば成功です。

4. Java の変数

（1） 変数の基本型

Java には、以下のような型が存在します。

① 論理型

boolean	1 ビット真偽値	true または false		

② 文字型

char	16 ビットの Unicode	\u0000	～	\uFFFF

③ 整数型

byte	8 ビット符号付き整数	-128	～	+127
short	16 ビット符号付き整数	-32768	～	+32767
int	32 ビット符号付き整数	-2147483648	～	+2147483647
long	64 ビット符号付き整数	-9223372036854775808	～	+9223372036854775807

④ 浮動小数点型

float	32 ビット符号付き整数	単精度の浮動小数点数
double	64 ビット符号付き整数	倍精度の浮動小数点数

（2） 変数の宣言と初期化

変数には、フィールド（構造体のメンバ）として利用する「フィールド変数」とメソッド（関数）内の変数「メソッド変数」として利用します。
初期化は、C 言語同様、宣言する際に行います。

```
class test02 {
  int x=5,y=3;//フィールド変数
  public static void main(String[] args) {
    float a=1.1f;//メソッド変数
    System.out.println(a);
  }
}
```

（3） 参照型変数

① 文字列

C 言語では、文字列を扱う変数の型はありませんでしたが、Java では用意されています。
文字列を扱うには、String 型を用います。

```
class test03 {
  public static void main(String[] args) {
    String moji="Hello, World!!";
```

```
    System.out.println(moji);
    System.out.println ("文字数:" + moji.length());
  }
}
```

String 型は、実際にはクラスを用いて構成しています。そのため、「moji.length()」というようなクラス内のメソッドを呼び出すことが出来るのです。

※ System.out.println で複数の変数や文字列を表示するには「+」でつなげます。

② 配列

Java でも配列を利用することが出来ます。

```
class test04 {
  public static void main(String[] args) {
    int [] a =new int[5];
    a[0]=5;
    a[1]=4;
    a[2]=3;
    a[3]=2;
    a[4]=1;

    System.out.println("a[0]=" + a[0]);
    System.out.println("a[1]=" + a[1]);
    System.out.println("a[2]=" + a[2]);
    System.out.println("a[3]=" + a[3]);
    System.out.println("a[4]=" + a[4]);
  }
}
```

また、配列の初期化は以下のように行います。

```
class test05 {
  public static void main(String[] args) {
    int [] a =new int[] {5,4,3,2,1};

    System.out.println("a[0]=" + a[0]);
    System.out.println("a[1]=" + a[1]);
    System.out.println("a[2]=" + a[2]);
    System.out.println("a[3]=" + a[3]);
    System.out.println("a[4]=" + a[4]);
  }
}
```

5. 演算子

（1） 主な演算子

Java で利用可能な演算子を以下に示します。大半は、C、C++、JavaScript と同様の演算子です。

四則演算		
+	加算	a = b + c;
-	減算	a = b - c;
*	乗算	a = b * c;
/	除算	a = b / c;
%	剰余	a = b % c;
単項演算子		
++	インクリメント	a++; ++a;
--	デクリメント	a--; --a;
代入演算子		
=	代入演算子	a = b;

比較演算子

==	等しい	if (a == b)
!=	異なる	if (a != b)
<	より小さい	if (a < b)
>	より大きい	if (a > b)
<=	等しいかより小さい	if (a <= b)
>=	等しいかおり大きい	if (a >= b)

論理演算子

&&	かつ	if ((a == b) && (c == d))
||	または	if ((a == b) || (c == d))
!	ではない	if (!(a == b))

ビット演算子

&	論理積(AND)	a = b & c;
|	論理和(OR)	a = b | c;
!	論理否定(NOt)	a = ! b;
^	排他的論理和(EOR)	a = b ^ c;
~	ビット反転	a = ~ b;
<<	算術左シフト	a = b << 2;
>>	算術右シフト	a = b >> 2;
>>>	論理右シフト	a = b >>> 2;

算術代入演算子

+=	加算代入	a += b;
-=	減算代入	a -= b;
*=	乗算代入	a *= b;
/=	除算代入	a /= b;
%=	余り代入	a %= b;
&=	乗算代入	a &= b;
|=	除算代入	a |= b;
^=	剰余代入	a ^= b;
<<=	算術左シフト代入	a <<= b;
>>=	算術右シフト代入	a >>= b;
>>>=	論理右シフト代入	a >>>= b;

三項演算子

? :	三項演算子	a = (b == c) ? d : e;

Java で利用可能な演算子を以下に示します。大半は、C、C++、JavaScript と同様の演算子です。

```
class test06 {
  public static void main(String[] args) {
    int a =5,b=3,c;

    c=a+b;
    System.out.println(a + "+" + b + "=" + c);
  }
}
```

（2） その他の演算子

① 文字列連結演算子と文字列変換

演算子「+」は、数値の加算だけでなく文字列の連結にも利用できます。また、数値

```
class test07 {
  public static void main(String[] args) {
    int a =5,b=3,c;
    String s,moji="abc";
```

```
    s=moji+"def";
    System.out.println(s);
    s=a+moji;
    System.out.println(s);
    s=a+b+moji;
    System.out.println(s);
  }
}
```

② キャスト演算子

異なる変数の型同士の計算の場合エラーを起こす可能性があります。

```
class test08 {
  public static void main(String[] args) {
    int a =5;
    short b;

    b=a;
    System.out.println(b);
  }
}
```

int 型から short 型に変換するように 32bit の値を 16bit の値に変換しようとする場合、オーバーフローを起こしコンパイルエラーとなります。しかし、int 型から long 型のように 32bit から 64bit に変換するような場合は、コンパイルエラーは起きません。
このような場合、C 言語同様キャスト演算子を用います。

```
class test08 {
  public static void main(String[] args) {
    int a =5;
    short b;

    b=(short)a;
    System.out.println(b);
  }
}
```

異なる型の変数を計算する場合、エラーを起こす可能性があります。

③ オブジェクト生成演算子

クラスや配列などのオブジェクトを生成するとき用いるのが、「new」です。これも演算子です。

```
int a [] = new int[10];
MyCls b = new MyCls();
String ss = new String("abcde")
```

④ メンバアクセス演算子

Java では、C 言語での構造体に似たクラス単位でプログラムは作られていきます。C 言語での構造体のメンバを示すのには「.」で示しました。
クラスから生成されたオブジェクト obj のメンバ dt を示す場合も同様に「.」以下のように示される。

```
obj.dt
```

⑤ メソッド呼び出し演算子

メソッド（クラス内の処理）を呼び出すときに、() を使います。この () をメソッド呼び出し演算子といいます。

```
System.out.println(b);
```

⑥ 添え字演算子

配列の各要素を特定するために[]を使いました。この[]を添え字演算子といいます。

```
dt[3]=100;
```

⑦ 文字列の比較

比較演算子 == を文字列の比較の意味で String オブジェクトなどに使用することはできません。下記のような使用例は誤りです。

```
String s1 = "ABC";
String s2 = "DEF";
if (s1 == s2) {                    // 文字列の比較にはなっていない
    System.out.println("Match!");
}
```

文字列の比較には equals() を用います。

```
String s1 = "ABC";
String s2 = "DEF";
if (s1.equals(s2)) {
    System.out.println("Match!");
}
```

6. 制御文

（1） if 文

if 文は、「もし･･･なら」という制御を実現します。次の例では、もし a の値が 3 であれば、"a is 3." を表示します。

```
int a = 3;
if (a == 3) {
    System.out.println("a is 3.");
}
```

else は、「さもなくば」を意味します。次の例では、もし a の値が 3 であれば "a is 3." を、さもなくば "a is not 3." を表示します。

```
int a = 3;
if (a == 3) {
    System.out.println("a is 3.");
} else {
    System.out.println("a is not 3.");
}
```

else if を繰り返して使用することもできます。次の例では、もし a が 3 であれば "a is 3." を、a が 4 であれば "a is 4." を、さもなくば "a is unknown." を表示します。

```
int a = 3;
if (a == 3) {
    System.out.println("a is 3.");
} else if (a == 4) {
    System.out.println("a is 4.");
} else {
    System.out.println("a is unknown.");
}
```

（例）test09.java

```
class test09 {
  public static void main(String args[]) {
    String s1 = new String("abcde");
    String s2 = "abcdo";

    System.out.print(s1+"="+s2+":");

    if (s1.equals(s2)) {
      System.out.println("Match!");
    }
    else{
      System.out.println("No Match!");
    }

  }
}
```

（2） for 文

for 文は処理を繰り返して実行する場合に用います。下記の例は、println() を 10 回繰り返す例です。最初に i に 0 を代入し、i の値を ++ でインクリメントしながら、i の値が 10 よりも小さい間、System.out.println(i) の実行を繰り返します。

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

（例）test10.java

```
class test10 {
  public static void main(String args[]) {
    int []  a = {31,23,32,12,44,40};
    int i;

    for(i=0;i<a.length;i++)
      System.out.println("a["+i+"]="+a[i]);

  }
}
```

（3） while 文

while 文は、条件が真である間、処理を繰り返します。下記の例では、i の値が 10 よりも小さい間、println() の実行を繰り返します。

```
int i = 0;
while (i < 10) {
    System.out.println(i);
    i++;
}
```

（例）test11.java:以下のプログラムを while 文を用いて作りなさい。

```
class test11 {
  public static void main(String args[]) {
    String school="HAMAMATSU TECHNICAL HIGH SCHOOL";
    int i;
    for(i=0;i<school.length();i++)
      System.out.println(school.substring(i,i+1));
```

```
  }
}
```

※　1文字だけ表示するなら、「school.charAt(i)」という書き方もある。

（4） do～while 文

do while 文は、while 文と似ていますが、条件が偽でも最低 1 回は処理を実行する点がことなります。

```
int i = 0;
do {
    System.out.println(i);
    i++;
} while (i < 10);
```

（例）test12.java:配列 a の要素の中の最大値を求めるプログラムです。空白を埋めプログラムを作りなさい。

```
class test12 {
  public static void main(String args[]) {
    int []  a = {31,23,32,12,44,40};
    int max=0,i;

    for(i=0;i<a.length;i++)
      System.out.println("a["+i+"]="+a[i]);

    System.out.println("---------");

    i=1;
    do{
      if(a[_________]<a[i])_________=i;
      i_________;
    }while(i<a._________);

    System.out.println("Max=a["+max+"]="+a[max]);
  }
}
```

（5） switch 文

switch 文は、条件で指定した値に応じて処理を振り分けます。条件には数値のみを指定することができます。case 文の値には固定値を指定します。実行時に値の変わる変数を指定することはできません。default は、他のどの条件にもマッチしなかった場合に実行されます。

```
int n = 3;
switch (n) {
case 1:
    System.out.println("n is one.");
    break;
case 2:
    System.out.println("n is two.");
    break;
case 3:
    System.out.println("n is tree.");
    break;
default:
    System.out.println("unknwon");
    break;
}
```

（例）test13.java

```
class test13 {
  public static void main(String args[]) {
    switch (args[0].charAt(0)) {
      case '日':
        System.out.println("市場へ出かけ、糸と麻を買ってきた");
        break;
      case '月':
        System.out.println("お風呂を炊いて");
        break;
      case '火':
        System.out.println("お風呂に入り");
        break;
      case '水':
        System.out.println("友達が来て");
        break;
      case '木':
        System.out.println("送っていった");
        break;
      case '金':
        System.out.println("糸巻きもせず");
        break;
      case '土':
        System.out.println("おしゃべりばかり");
        break;
      default:
        System.out.println("そんな曜日はありません！");
        break;
    }
    System.out.print("ともだちよ、これが私の一週間の仕事です。");
  }
}
```

※ 以下のように実行する。なお、文字列 args[]は、コマンドライン上の引数（文字列）を示す。

```
C>java test13 火↵
お風呂に入り
ともだちよ、これが私の一週間の仕事です。
```

（６） break 文

break は、最も内側の for ループ、while ループ、do ループや、case 文の残りの処理を飛ばし、処理を抜けます。Perl 言語の last 文に相当します。下記の例では、i の値が 5 の時に for ループを抜け出し、finish!! を書き出します。

```
for (i = 0; i < 10; i++) {
    if (i == 5) {
        break;
    }o
    System.out.println(i);
}
System.out.println("finish!!");
```

ラベルを指定することにより、多重のループを一度に抜け出すこともできます。下記の例では、i == 1、j == 5 の時に、loop1 のラベルで指定した外側の for ループを抜けます。ラベルには任意の名前をつけることができます。

```
loop1:  for (i = 0; i < 10; i++) {
```

```
            for (j = 0; j < 10; j++) {
                if ((i == 1) && (j == 5)) {
                    break loop1;
                }
            }
        }
```

（7） continue 文

continue は、最も内側の for ループ、while ループ、do ループ文の残りの処理を飛ばし、次のループを開始します。Perl 言語の next 文に相当します。下記の例では、i の値が 5 の時に println() の実行をスキップし、i が 6 のループに移ります。

```
for (i = 0; i < 10; i++) {
    if (i == 5) {
        continue;
    }
    System.out.println(i);
}
```

break と同様に、ラベルで指定した多重ループの外側のループに作用させることも可能です。

7. クラス

（1） クラスの概念と定義

クラスとは、C 言語の構造体に処理を追加した物だと考えるとわかりやすいと思います。Java では以下のようにして、クラスを定義します。

```
class Book{
  int price;
  int num;                 } フィールド
  String title;

  int sum(){
    return price*num;      } メソッド
   }
}
```

クラスは構造体と違い、データを収めるフィールドだけでなく関数のように処理メソッドから成り立っています。そして、前述のように、Java ではクラス単位としてプログラムを作成していきます。

（2） オブジェクトを作る

クラスを定義しただけでは利用できません。クラスからオブジェクトを生成してはじめて利用することが出来ます。このようにオブジェクトを生成することを「**インスタンス化**」といいます。
以下のようにして、オブジェクトを生成します。

```
Book book1;
book1 = new Book();
```

または、

```
Book book1 =new Book();
```

（3） プログラム例

```
class Person {
  String myName;
  int myAge;
```

```
  public void SetName(String name) {
    myName = name;
    PrintSet();
  }
  public String GetName() {
    return myName;
  }
  public void SetAge(int age) {
    myAge = age;
    PrintSet();
  }
  public int GetAge() {
    return myAge;
  }
  public void PrintSet() {
    System.out.println("Set Value!!");
  }
}
class test14 {
  public static void main(String[] args) {
    Person tanaka = new Person(); // 田中さんオブジェクトを作る
    tanaka.SetName("Tanaka");     // 田中さんの名前を設定する
    tanaka.SetAge(26);            // 田中さんの年齢を設定する

    Person suzuki = new Person(); // 鈴木さんオブジェクトを作る
    suzuki.myName="Suzuki";     // 鈴木さんの名前を設定する
    suzuki.myAge=32;            // 鈴木さんの年齢を設定する

    System.out.println(tanaka.GetName());
    System.out.println(tanaka.GetAge());
    System.out.println(suzuki.GetName());
    System.out.println(suzuki.GetAge());
  }
}
```

（4） フィールドの利用

フィールドを利用するには、

```
suzuki.myName="Suzuki";        // 鈴木さんの名前を設定する
suzuki.myAge=32;               // 鈴木さんの年齢を設定する
```

のように、クラス名のあとに「.」を書いてフィールドを示します。
また、同じクラス内のフィールドなら、そのクラス名を書く必要はありません。

```
class Person {
  String myName;
  int myAge;
  public void SetName(String name) {
    myName = name;
  }
  public String GetName() {
    return myName;
  }
  public void SetAge(int age) {
    myAge = age;
  }
  public int GetAge() {
    return myAge;
```

（5） メソッドの利用

メソッドを利用するには、

```
tanaka.SetName("Tanaka");      // 田中さんの名前を設定する
tanaka.SetAge(26);             // 田中さんの年齢を設定する
```

のように、クラス名のあとに「.」を書いてメソッドを示します。
メソッドは、引数を持つ場合がありますので、引数も間違いないように指定してください。
また、同じクラス内のメソッドを利用する場合は、クラス名を書く必要はありません。

```
class Person {
  String myName;
  int myAge;
  public void SetName(String name) {
    myName = name;
    PrintSet();
  }
  public String GetName() {
    return myName;
  }
  public void SetAge(int age) {
    myAge = age;
    PrintSet();
  }
  public int GetAge() {
    return myAge;
  }
  public void PrintSet() {
    System.out.println("Set Value!!");
  }
}
```

メソッド名だけで結構です。もちろん引数がある場合は、引数を忘れずに。

（6） カプセル化 encapsulation

クラスでは、フィールドとメソッドを１つにまとめました。このようにデータと処理と１つにまとめることを「カプセル化」といいます。
この「カプセル化」とは、「継承」、「多態性（多相性、ポリモーフィズム）」とともに、オブジェクト指向プログラミングの大きな特徴です。
では、カプセル化することのメリットは何でしょう。
ここで例えば、商品在庫を表すクラス Zaiko を以下のように定義したとしましょう。

```
public class Zaiko {

    public int count = 0;

}
```

ここで、フィールド count は、商品の在庫数を保持するものとします。例えば、在庫数をセットするには以下のようなコードを書きます。

```
zaiko.count = 1;
```

しかし、これでは直接アクセスできてしまうためいくつか問題があります。
例えば、「count フィールドにはマイナスの値をセットしてはならない」という要件があったとします。
すると、このフィールドを使う部分にすべて以下のようなプログラムを書き加えなければなりません。

```
if (c >= 0) {
    zaiko.count = c;
}
```

もしこの要件が変更されたら、これらのプログラムはすべて書き直さなくてはなりません。つまり、柔軟性や保守性の低いアプリケーションとなってしまうのです。また、チーム開発を行っている場合は、ほかの開発者が count フィールドを“予想外”の使い方をしてしまう可能性もあります。
さらに、上のようなプログラムを付け忘れた場合、count フィールドにマイナスの値を入れてしまうかもしれないのでバグの温床となります。
そこで、上記の Zaiko クラスを以下のように書き換えてみます。

```
public class Zaiko {

    private int count = 0;

    public void setCount(int c) {
        if (c < 0) {
            // 例外を送出
        }
        count = c;
    }
}
```

まず、count フィールドのアクセス修飾子を public から private に変更し、ほかのクラスから読み書きできないようにします。(アクセス修飾子については、あとであらためて説明します。)
続いて、同フィールドに値をセットするための setCount メソッドを定義します。同メソッドでは、引数で渡された値をチェックし、ゼロ以上である場合にのみ count フィールドにセットするコードを記述します。
このように、クラスのフィールドを外部から隠し、必ずメソッドを通してアクセスさせるやり方を「カプセル化（encapsulation)」と呼び、カプセル化されたフィールドのことを「プロパティ」と呼びます。フィールドの代わりにプロパティを用いることで、まず「柔軟性・保守性が向上する」というメリットが生まれます。
Zaiko クラスの例では、在庫数チェックのビジネス・ロジックが同クラスの外側に分散せず、setCount メソッドに集約されます。ビジネス・ロジックの変更やデバッグは setCount メソッド 1 カ所の修正だけで済むため、仕様変更の要求にも積極的に応えられます。また、ほかの開発者によってフィールドが予想外の値にセットされる可能性がなくなり、バグの発生を少なくできます。

また在庫数チェック以外にも、例えば「プロパティの値が変更されたらログに記録したい」「プロパティの値が変更されたら別のプロパティの値も自動的に変更したい」といった多彩なロジックをプロパティに持たせることができます。
つまりプロパティとは、「賢くなったフィールド」といえます。こうした賢いフィールド＝プロパティを中心に Java クラスを記述することで、オブジェクト指向言語としての Java のメリットを引き出すことができます。
つまり、先ほど示したプログラム例で、以下のような記述は、オブジェクト指向では不適切な書き方となるのです。

```
suzuki.myName="Suzuki";         // 鈴木さんの名前を設定する
suzuki.myAge=32;                // 鈴木さんの年齢を設定する
```

8. コンストラクタとオーバーロード

（1） コンストラクタ

コンストラクタとは、クラス名と同じ名前のメソッドのことをいいます。
インスタンスが生成された時に自動的に呼ばれ、インスタンスの初期化処理を行います。

```
class Person {
  String myName;
  int myAge;
  public void SetName(String name) {
    myName = name;
    PrintSet();
  }
  public String GetName() {
    return myName;
  }
  public void SetAge(int age) {
    myAge = age;
    PrintSet();
  }
  public int GetAge() {
    return myAge;
  }
  public void PrintSet() {
    System.out.println("Set Value!!");
  }
  public Person(){
    myAge=0;
    myName="";
  }
}
class test15 {
  public static void main(String[] args) {
    Person tanaka = new Person();
    tanaka.SetName("Tanaka");
    tanaka.SetAge(26);

    System.out.println(tanaka.GetName());
    System.out.println(tanaka.GetAge());
  }
}
```

また、コンストラクタは引数を持つこともできます。

```
public Person(String name) {
  myName = name;
}
```

このように引数の定義がされていて Person クラスのインスタンス化する場合、以下のように記述します。

```
Person tanaka = new Person("tanaka");
```

（2） オーバーロード

オーバーロードとは同一クラス内でメソッド名が同一で引数の型、数、並び順が異なるメソッドを複数定義することを言います。
例えば、会員登録を行う機能で、名前と国名を登録させたいとします。国名を入力しなかった会員は国名：日本で登録します。同じ登録を行う機能であるのに、メソッドを toroku（String name，String country)、torokuJapan（String name）などと作成するするのはあまりきれいなプログラムではありません。この場合、toroku（String name)、toroku（String name，String country）というように同じ名前のメソッドを引数の数を変えて作成します。同じ機能を持つものは同じメソッド名とした方が、プログラムがきれいに見えます。
Java はメソッドの引数の型、数、並び順が異なる場合、それぞれを異なるメソッドとして扱います。これがオーバーロードの機能です。

```
class test16{
  //①引数を 2 つ持つ toroku メソッド
  void toroku(String name, String country) {
    System.out.println("名前は" + name);
    System.out.println("国は" + country);
  }
  //②引数を 1 つ持つ toroku メソッド
  void toroku(String name) {
    System.out.println("名前は" + name);
    System.out.println("国は" + "日本");
  }
  public static void main(String[] args) {
    test16 object = new test16();
    object.toroku("Java 太郎");//③
  }
}
```

① 数を name と country の 2 つ持つメソッド toroku を宣言します。
② 引数を name の一つしか持たず、country に当たる部分は日本を表示するメソッド toroku を宣言します。toroku メソッドを呼び出します。
③ 引数が 1 つしか指定されていないため、②のメソッドが呼び出されます

※ 自分自身のクラスをインスタンス化することも出来るんですねぇ。

オーバーロードを定義するの規定

（1） 異なるメソッドと認識される部分は、メソッドの次の個所です。「引数の型」、「引数の数」、「引数の並び順」。

（2） 次の個所が異なっていても異なるメソッドとは認識されません。コンパイルエラーとなります。「戻り型」、「アクセスレベル」、「引数名」、「throws 節」。

また、test15.java でコンストラクタ Person を定義しましたが、コンストラクタ自体メソッドであるので、このオーバーロードという機能が使えます。

```
class Person {
  String myName;
  int myAge;
  public void SetName(String name) {
    myName = name;
    PrintSet();
  }
  public String GetName() {
    return myName;
  }
  public void SetAge(int age) {
    myAge = age;
    PrintSet();
  }
  public int GetAge() {
    return myAge;
  }
  public void PrintSet() {
    System.out.println("Set Value!!");
```

```
  }
  public Person(){
    myAge=0;
    myName="";
  }
  public Person(int age){
    myAge=age;
    myName="";
  }
  public Person(String name){
    myAge=0;
    myName=name;
  }
  public Person(String name,int age){
    myAge=age;
    myName=name;
  }
}
class test17 {
  public static void main(String[] args) {
    Person tanaka = new Person(); // 田中さんオブジェクトを作る
    tanaka.SetName("Tanaka");     // 田中さんの名前を設定する
    tanaka.SetAge(26);            // 田中さんの年齢を設定する

    Person suzuki = new Person("Suzuki",32); // 鈴木さんオブジェクトを作る

    Person sato = new Person("Sato"); // 鈴木さんオブジェクトを作る

    System.out.println(tanaka.GetName());
    System.out.println(tanaka.GetAge());
    System.out.println(suzuki.GetName());
    System.out.println(suzuki.GetAge());
    System.out.println(sato.GetName());
    System.out.println(sato.GetAge());

  }
}
```

9. 内部クラス（インナークラス）

クラス内に宣言されたクラスを内部クラスといいます。ここでは、内部クラスの概要・使用例について解説します。
内部クラスはクラス間の関係を明確にするために使用します。あるクラスのメンバ変数やメソッドに強く依存しているクラスは、そのクラスの内部クラスとして宣言します。
内部クラスは、GUI のイベント処理ではイベント代理モデルの必要性からよく用いられます。

```
class test18 {
  private String exMsg = "test18 クラスのメンバ変数";
  public static void main(String[] args) {
    test18 ei = new test18();
    ei.exeClass();
  }
  //(1)
  private class InInner {
    private String inMsg = "InInner クラスのメンバ変数";
    //(2)
    void inMethod() {
      System.out.println(exMsg);
      System.out.println(inMsg);
```

```
    }
  }
  //(3)
  void exeClass() {
    InInner ii = new InInner();
    ii.inMethod();
  }
}
```

【解説】

(1)　内部クラス InInner を宣言します。

(2)　inMethod メソッドを宣言し、test18 クラスのメンバ変数と InInner クラスのメンバ変数を参照しています。

(3)　exeClass メソッドで内部クラス InInner のオブジェクトを生成し、inMethod メソッドを実行しています。

【特徴】

- 内部クラスはメンバ変数、メソッドなどと同じように、クラスの構成要素の一つです。そのため、private、protected、public といったアクセスレベルを付与することができます。

- 内部クラスから同じクラス内のメンバ変数、メソッドを参照することができます。メンバ変数、メソッドに private アクセスレベルが付与されている場合でも、同じクラス内のため参照することができます。

- メソッド内に内部クラスを宣言することもできます。その場合、その内部クラスはメソッド内でのみ機能します。同じクラスのメンバ変数、メソッドを参照することはできません。

- メソッド内に内部クラスを宣言した場合、内部クラスからメソッド内のローカル変数を参照することができます。ただし、参照できるローカル変数は final 修飾子が付与されている場合のみです。ラス内部に定義するクラスを インナークラス（内部クラス）と呼びます。

10. クラスの継承

継承 (extends) とは、あるクラスの実装を土台にして、別のクラスを実装するための方法です。元になるクラスを**スーパークラス**と呼び、それを継承する新しいクラスを**サブクラス**（継承クラス）と呼びます。一つのクラスは、任意の個数のサブクラスを持つことが出来ますが、逆に、スーパークラスは一つしか持てません。これを単一継承と呼び、クラスは継承されることで枝葉を広げていく木構造を成しています。そして、遡っていけば根っ子となる一つの**スーパークラス**（ルート・クラス）に行きつきます。

クラスの木構造の頂点、ルートとなるクラスが Object クラスです。これは java.lang パッケージに含まれ、スーパークラスを明示しないクラスの場合、自動的にこのクラスから派生されたことになります。スーパークラスを明示した場合でも、そのクラスも Object クラスのサブクラスか、または Object クラスを継承したクラスのサブクラスのいずれかなので、最終的にはあらゆるクラスが Object クラスをスーパークラスに持つことに変わりはありません。

（1） 継承の書式

あるクラスのメンバ変数やメソッドを 継承 した サブクラス（子クラス）を定義するには extends を用います。

```
class クラス名 extends 親クラス名 {
    :
}
```

下記の例では、上記で作成した Person クラスを継承する Member クラスを定義しています。Member クラスは Person クラスを継承しているので、myName、myAge などの属性や、GetName()、SetName() などのメソッドを引き継ぎ、加えて、myNumber や SetNumber() などの属性やメソッドを備えています。

```
class Person {
  String myName;
  int myAge;
  public void SetName(String name) {
    myName = name;
    PrintSet();
  }
  public String GetName() {
    return myName;
  }
  public void SetAge(int age) {
    myAge = age;
    PrintSet();
  }
  public int GetAge() {
    return myAge;
  }
  public void PrintSet() {
    System.out.println("Set Value!!");
  }
  public Person(){
    myAge=0;
    myName="";
  }
  public Person(int age){
    myAge=age;
    myName="";
  }
  public Person(String name){
    myAge=0;
    myName=name;
  }
  public Person(String name,int age){
    myAge=age;
    myName=name;
  }
}
class Member extends Person {
    int myNumber;
    public void SetNumber(int number) {
        myNumber = number;
    }
    public int GetNumber() {
        return myNumber;
    }
}
class test19 {
  public static void main(String[] args) {
    Member tanaka = new Member(); // 田中さんオブジェクトを作る
    tanaka.SetName("Tanaka");     // 田中さんの名前を設定する
    tanaka.SetAge(26);            // 田中さんの年齢を設定する
    tanaka.SetNumber(12);            // 田中さんの番号を設定する

    System.out.println(tanaka.GetName());
```

```
        System.out.println(tanaka.GetAge());
        System.out.println(tanaka.GetNumber());

    }
}
```

（2） サブクラスの性質

サブクラスは、「スーパークラスみたいなもの」、「スーパークラスの一種」だと言えます。「乗り物」が継承されて「自動車」が作られ、「自動車」を継承して「トラック」を作るようなもので、「トラック」は「自動車」の一種であり、更に「乗り物」の一種です。

また、サブクラスは、自身で実装したメソッドや変数に加えて、スーパークラスのメソッドや変数を全て継承して使えます。「トラック」は「自動車」の性質に独自の性質を加えたものであり、「自動車」は「乗り物」に独自の性質を加えたものです。

サブクラスは、スーパークラスの亜種／変種であり、スーパークラスの性質を全て備えています。

（3） スーパークラスのメソッド呼び出し

super は、クラスの継承における 親クラス（スーパークラス）を示します。java では、親クラスのコンストラクタは暗黙的には呼ばれないため、下記のように明示的に呼び出してやる必要があります。

```
class Member extends Person {
    int myNumber;
    Member() {
        super("tanaka");          // 親クラスのコンストラクタを呼び出す
        super.SetAge(26);         // 親クラスのメソッドを呼び出す
        return super.myName;      // 親クラスの属性を参照する
    }
    public void SetNumber(int number) {
        myNumber = number;
    }
    public int GetNumber() {
        return myNumber;
    }
}
class test20  (
  public static void main(String[] args) {
    Member tanaka = new Member(); // 田中さんオブジェクトを作る
    tanaka.SetNumber(12);             // 田中さんの年齢だけを設定する

    System.out.println(tanaka.GetName());
    System.out.println(tanaka.GetAge());
    System.out.println(tanaka.GetNumber());

  }
}
```

（4） クラスに属しているか調べる（instanceof）

instanceof 演算子は、オブジェクト（インスタンス）が、指定したクラスまたはその上位のクラスに属しているかどうかを調べます。

```
MyButton b = new MyButton();
if (b instanceof Button) {
```

```
        System.out.println("属してる");
    } else {
        System.out.println("属していない");
    }
```

（5） 継承の特徴

- 継承は一つのクラスからしか出来ません。スーパークラスは一つしかもてないと言うことです。
- サブクラスは幾つでも作れます。
- 継承の階層は問いません。何階層でも継承されます。
- メンバ変数、メソッドは継承されます。
- 但し、子孫の変数やメソッドは利用できません。つまり、インスタンス化したクラスが、スーパークラスから継承した機能しか利用できません。
- コンストラクタは継承されません。そのクラス固有のものです。
- 継承時にメンバ変数名、メソッド名が重なったら、インスタンス化されたクラスから最も階層が近いスーパークラスの定義が採用されます。つまり、継承時に既存のメンバ変数、メソッドを変更することが出来ます。

11. オーバーライド

オーバーライドは「上書き」のことです。継承時に、スーパークラスで定義されたメソッドと同じ名前、引数を持つメソッドを、サブクラスでもう一度定義することです。
たとえば、Person クラスで定義されたメソッド PrintSet を、継承した Member クラスで再定義することを「オーバーライド」といいます。

```
class Person {
        :
  public void PrintSet() {
    System.out.println("Set Value!!");
  }
        :
}
class Member extends Person {
        :
  public void PrintSet() {
    System.out.println("Set Member Value!!");
  }
        :
}
```

実際に動作を確認してみましょう。

```
class Person {
        :
}
class Member extends Person {
    int myNumber;
    Member() {
        super("tanaka");          // 親クラスのコンストラクタを呼び出す
        super.SetAge(26);         // 親クラスのメソッドを呼び出す
    }
    public void SetNumber(int number) {
        myNumber = number;
        PrintSet();;              // 追加を忘れないように!!
    }
    public int GetNumber() {
        return myNumber;
```

```
    }
    public void PrintSet() {
      System.out.println("Set Member Value!!");
    }
}
class test21 {
  public static void main(String[] args) {
    Member tanaka = new Member(); // 田中さんオブジェクトを作る
    tanaka.SetNumber(12);            // 田中さんの年齢だけを設定する

    System.out.println(tanaka.GetName());
    System.out.println(tanaka.GetAge());
    System.out.println(tanaka.GetNumber());

  }
}
```

結果には、「Set Member Value!!」と表示されるか、それとも「Set Value!!」と表示されるか、どちらでしょう。

12. 修飾子一覧

クラス、インタフェース、メソッド、コンストラクタ、変数には、下記の 修飾子 を指定することができます。

修飾子	クラス	インターフェース	メソッド	コンストラクタ	ブロック	変数	説明
public	○	○	○	○	×	○	アクセス修飾子
protected	○	○	○	○	×	○	アクセス修飾子
private	○	○	○	○	×	○	アクセス修飾子
static	○	○	○	×	×	○	スタティック修飾子
final	○	×	○	×	×	○	ファイナル修飾子
abstract	○	○	○	×	×	×	抽象修飾子
native	×	×	○	×	×	×	ネイティブ修飾子
synchronized	×	×	○	×	○	×	同期修飾子
transient	×	×	×	×	×	○	一時的修飾子
volatile	×	×	×	×	×	○	揮発性修飾子
strictfp	○	○	○	×	×	×	厳密浮動小数修飾子
const	×	×	×	×	×	×	定数

（1） アクセス修飾子（public,protected,private）

public、protected、private はアクセス修飾子と呼ばれ、指定した変数やクラスなどを、どの範囲から参照可能かのスコープを制御するのに用いられます。クラス、インタフェース、メソッド、コンストラクタ、変数の修飾子として利用できます。

	自ファイル			他ファイル	
アクセス修飾子	自クラス	サブクラス	他クラス	サブクラス	他クラス
public	○	○	○	○	○
protected	○	○	○	○	×
なし	○	○	○	×	×
private	○	×	×	×	×

public は「すべてのクラスからの参照を許す」、protected は「他ファイル・他クラスからのアクセスをプロテクトする」、private は「自クラスからのアクセスしか許さない」という意味になります。

- AccessTest1.java -

```
public class AccessTest1 {

    // public, protected, 無し, private な値の定義
    public int publicValue;
    protected int protectedValue;
    /* 無し */ int normalValue;
    private int privateValue;

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

    // 自ファイル・自クラスからのアクセステスト
    public AccessTest1() {
        this.publicValue = 1;
        this.protectedValue = 2;
        this.normalValue = 3;
        this.privateValue = 4;
    }
}

class AccessTest2 extends AccessTest1 {
    // 自ファイル・サブクラスからのアクセステスト
    void AccessTest2() {
        this.publicValue = 1;
        this.protectedValue = 2;
        this.normalValue = 3;
        // this.privateValue = 4;  ← アクセスできない
    }
}

class AccessTest3 {
    // 自ファイル・他クラスからのアクセステスト
    void AccessTest3() {
        AccessTest1 o = new AccessTest1();
        o.publicValue = 1;
        o.protectedValue = 2;
        o.normalValue = 3;
        // o.privateValue = 4;  ← アクセスできない
    }
}
```

- AccessTest4.java -

```
public class AccessTest4 extends AccessTest1 {

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

    // 他ファイル・サブクラスからのアクセステスト
    AccessTest4() {
        this.publicValue = 1;
        this.protectedValue = 2;
        // this.normalValue = 3;    ← アクセスできない
        // this.privateValue = 4;  ← アクセスできない
    }
}
```

```
class AccessTest5 {
    // 他ファイル・他クラスからのアクセステスト
    AccessTest5() {
        AccessTest1 o = new AccessTest1();
        o.publicValue = 1;
        // o.protectedValue = 2;  ← アクセスできない
        // o.normalValue = 3;     ← アクセスできない
        // o.privateValue = 4;    ← アクセスできない
    }
 }
```

（2） スタティック修飾子（static）

static は、クラスがインスタンス化されていなくても、そのクラス、メソッド、変数などを参照可能であることを示します。

```
class クラス名 {
    public static void main(String[] args) {
        :
    }

    public static final double PI = 3.14159265358979323846;

    public static double sin(double a) {
        :
    }

    public static class クラスB {
        :
    }
}
```

例えば、java.lang.Math クラスの sin() メソッドは static 宣言されているので、java.lang.Math.sin() として呼び出すことができますが、もし、static 宣言されていなければ、下記のように呼び出す必要があります。

```
// static 宣言されていれば
x = java.lang.Math.sin(y); // 直接呼び出せる

// static 宣言されていなければ
java.lang.Math m = new java.lang.Math();  // インスタンス化が必要
x = m.sin(y);
```

（3） ファイナル修飾子（final）

final は上書きされないことを意味します。クラスに用いた場合はサブクラスを定義できないことを、メソッドに用いた場合はサブクラスでメソッドをオーバーロードできないことを、変数に用いた場合は誰もその変数を変更できないことを意味します。

```
final class クラス名 {
    :
}
class クラス名 {
    final 型 メソッド名(引数) {
        :
    }
```

```
    public static final double PI = 3.14159265358979323846;
}
```

（4） 抽象修飾子（abstract）

abstract は、クラス、インタフェース、メソッドが抽象的なものであり、不完全なものであることを示します。メソッド名、引数の型などのみを定義し、サブクラスが必ず実装しなくてはならない機能を明確にしておくのが主な利用目的です。
抽象クラス はそのままではインスタンス化することができず、必ずサブクラスを定義して不完全なメソッドの実体を定義してから使用する必要があります。抽象クラスは通常、ひとつ以上の 抽象メソッド を含みます。抽象メソッドはメソッド名、引数の型のみが定義されていますので、どこかでその中身を定義する必要があります。

```
abstract class クラス名 {
        :
    abstract void testFunction(int a);
        :
}
```

インタフェースは常に抽象です。以前は下記のように abstract interface と記述していましたが、最近では書かないことが推奨されています。

```
abstract interface クラス名 {
        :
    abstract void testFunction(int a);
        :
}
```

抽象クラスとインタフェースは用途は似ていますが、抽象クラスは抽象メソッド以外の通常のメソッドも記述できる代わりに、ひとつのクラスが複数の抽象クラスを継承することができません。

（5） ネイティブ修飾子（native）

native は、メソッドが ネイティブメソッド であることを示します。ネイテブメソッドは型、メソッド名、引数の型といった作法のみを定義するもので、C 言語など他の言語を用いてその中身を実装する必要があります。ネイティブメソッドの詳しい作成方法の説明は省略します。

```
class クラス名 {
        :
    public native void testFunction(int a);
        :
}
```

（6） 同期修飾子（synchronized）

synchronized 修飾子付きのメソッドがマルチスレッド環境で実行される場合、メソッドのインスタンスを対象として排他制御が行われます。ひとつのインスタンスが複数のスレッドを持つ場合は排他制御が行われますが、スレッドが複数のインスタンスで実行される場合の排他制御は行われませんので注意してください。

```
class クラス名 {
    synchronized void メソッド() {
        :
    }
}
```

上記は、下記のコードと同等の意味を持ちます。

```
class クラス名 {
    void メソッド() {
        synchronized (this) {
            :
        }
    }
}
```

（7） 一時的修飾子（transient）

ディスクに保存したりネットワークで送信するために、オブジェクトの情報をバイトストリームに自動変換することをシリアライズといいますが、transient 修飾子付きの変数は一時的な変数であり、シリアライズの対象から除外されます。

```
class クラス名 {
    transient String tmp;
}
```

（8） 揮発性修飾子（volatile）

マルチスレッドの場合、それぞれのスレッドは性能向上のために変数のコピーを参照・変更し、その値を元の場所（メモリ）に書き戻さないことがあります。つまり、同じ変数でもスレッドによって値が異なるという現象が発生します。複数のスレッドから参照される可能性のある変数に volatile をつけることにより、この問題を回避することができます。

```
class クラス名 {
    volatile int nCount;
}
```

（9） 厳密浮動小数修飾子（strictfp）

strictfp を指定したクラスでは、float や double の浮動小数点演算がプラットフォームに依存しない、厳密な動作をすることになります。strictfp を指定しない場合の浮動小数点演算は、プラットフォームに依存し、高速だけれどもプラットフォームによって若干結果が異なります。

```
strictfp class クラスA {
    double x, y;
        :
}
```

（１０）定数修飾子（const）

const は、C/C++ 言語などで値を変更することができない変数を示しますが、現在の Java ではキーワードのみが定義されており、実際に使用されるケースはありません。

13. パッケージ

Java では、よく使うクラスをまとめて、パッケージとして管理します。
プログラマが必要に応じて、パッケージを作成することも出来ますし、便利で一般的な機能を持ったクラスを標準クラスライブラリとして実装しています。これらはカテゴリごとにパッケージ化されており、コア・パッケージと呼ばれます。

代表的なコア・パッケージ：

java.lang
Java プログラム言語の設計にあたり基本的なクラスを提供します。全てのクラスのルートとなる Object クラス、文字列を参照する String クラス、基本データ型のラップクラス（Integer, Character 等のクラス）などが含まれます。

java.io
データストリーム、直列化、ファイルシステムによるシステム入出力用に提供されています。

java.applet
アプレットの作成、およびアプレットとアプレットコンテキストとの通信に使用するクラスの作成に必要なクラスを提供します。

java.awt
ユーザインタフェースの作成と、グラフィックスとイメージのペイントのためのクラスがすべて含まれています。JDK1.2 (Java2) 以降から Swing に拡張された。

java.beans
JavaBeans の開発に関連するクラスが含まれています。

java.net
ネットワーク対応アプリケーションを実装するためのクラスを提供します。

java.util
このパッケージには、コレクションフレームワーク、従来のコレクションクラス、イベントモデル、日付と時刻の機構、国際化、およびさまざまなユーティリティクラス（文字列トークナイザ、乱数ジェネレータ、ビット配列） が含まれています。

javax.swing
GUI (Graphical User Interface) コンポーネントで、 JFC (Java Foundation Classes) の一部。すべてのプラットフォームで可能な限り同じように機能する、「軽量」(Java 共通言語) コンポーネントのセットを提供します。 AWT (Abstract Window Toolkit) の機能強化（継承）として実装。AWT が実行環境のウィンドウシステム特有のコードを利用しているのに対して、Swing は完全に Java だけで作られたコンポーネント。

org.omg.CORBA
OMG CORBA API の Java プログラミング言語 (ORB クラスを含む) へのマッピングを提供します。

大量のパッケージのそれぞれに大量のクラスが定義されているので、以下のサイトで API 仕様の確認をすると良いでしょう。

http://java.sun.com/javase/reference/api.jsp

（1） パッケージを作る

test21.java を改良して、クラス Person を myPackage というパッケージにして、test22.java から呼び出すよう作成します。

① パッケージが入るディレクトリを作る

今回は、myPackage というパッケージを作成しますので、同じ名前の myPackage という名前のディレクトリを作成します。

```
C>mkdir myPackage
```

② パッケージとなるクラスを作成

クラス Person を作成します。①で作成した myPackage の中に、Person.java と言う名前で作成します。なお、ファイル名は public で指定しているクラスの名前をファイル名として作成してください。プログラムの先頭に、パッケージの名前を package 文で定義します。

- myPackage\Person.java -

```
package myPackage;
public class Person {
  String myName;
  int myAge;
  public void SetName(String name) {
    myName = name;
    PrintSet();
  }
  public String GetName() {
    return myName;
  }
  public void SetAge(int age) {
    myAge = age;
    PrintSet();
  }
  public int GetAge() {
    return myAge;
  }
  public void PrintSet() {
    System.out.println("Set Value!!");
  }
  public Person(){
    myAge=0;
    myName="";
  }
  public Person(int age){
    myAge=age;
    myName="";
  }
  public Person(String name){
    myAge=0;
    myName=name;
  }
  public Person(String name,int age){
    myAge=age;
    myName=name;
  }
}
```

③ test22.java を作る

パッケージのクラスを利用するには、利用するクラスが記述されたファイルの冒頭で、 import 文を記述します。但し、 package 文も記述する場合は、 package 文の方が先になります。

```
import パッケージ名.クラス名;
```

指定したクラスの全てのクラスを利用する可能性があれば、クラス名の部分を "*" で記述します：

```
import パッケージ名.*;
```

複数のパッケージ（に含まれるクラス）をインポートしたければ、繰り返し import 文を書きます：

```
import パッケージ名.クラス名;
import パッケージ名.クラス名;
import パッケージ名.クラス名;
```

このソースをパッケージに含めたい場合は、インポート文の上にパッケージ宣言を書きます：

-test22.java-

```
package パッケージ名;
import パッケージ名.クラス名;

import myPackage.Person;
class Member extends Person {
    int myNumber;
    Member() {
        super("tanaka");          // 親クラスのコンストラクタを呼び出す
        super.SetAge(26);         // 親クラスのメソッドを呼び出す
    }
    public void SetNumber(int number) {
        myNumber = number;
        PrintSet();;              // 追加を忘れないように!!
    }
    public int GetNumber() {
        return myNumber;
    }
    public void PrintSet() {
      System.out.println("Set Member Value!!");
    }
}
class test22 {
  public static void main(String[] args) {
    Member tanaka = new Member(); // 田中さんオブジェクトを作る
    tanaka.SetNumber(12);         // 田中さんの年齢だけを設定する

    System.out.println(tanaka.GetName());
    System.out.println(tanaka.GetAge());
    System.out.println(tanaka.GetNumber());

  }
}
```

④ コンパイルと実行
まずは、パッケージに含まれるクラスをコンパイルします。

```
C> javac myPackage\Person.java
```

パス名の指定を間違わないようにしてください。
次に、test22.java をコンパイルします。

```
C>javac test22.java
```

エラーがなかったら実行します。

```
C>java test22
```

（2） コア・パッケージの利用

前述のように、一般的な機能を持ったクラスは標準クラスライブラリとしてパッケージ化されています。これらのパッケージを用いてプログラムを作成します。
まず、API 使用のホームページより、java.lang パッケージの中の String クラスについて調べてみましょう。

コンストラクタの概要

```
String(byte[] bytes)
```

プラットフォームのデフォルト文字セットを使用して、指定されたバイト配列を復号化することによって、新しい String を構築します。

メソッドの概要

int length()
この文字列の長さを返します。

String replace(char oldChar, char newChar)
この文字列内にあるすべての oldChar を newChar に置換した結果生成される、新しい文字列を返します。

以下のプログラムを作成しましょう。

- test23.java -

```
import java.lang.*;
class test23 {
  public static void main(String[] args) {
    String moji =new String("hamamatsu technical high school.");

    System.out.println(moji);
    System.out.println(moji.length());
    moji=moji.replace('a','A');
    System.out.println(moji);
  }
}
```

コンパイル、実行し、動作を確認しましょう。

```
C>javac test23.java

C>java test23
hamamatsu technical high school.
32
hAmAmAtsu technicAl high school.
```

（3） コア・パッケージを利用してファイルの入出力

コア・パッケージに収められているクラスを使って、ファイルの入出力をしてみましょう。
ファイルの読み書きなどのデータの流れのことを「ストリーム」といいます。とくに Java では、ファイルの読み書きをするには、ストリーム専用のクラスからオブジェクトを生成して行います。
また、ファイルの内容には文字として読める「テキストファイル」と写真等のように信号として保存されている「バイナリファイル」があり、Java ではファイルの種類によって、扱うデータの型が違います。

文字　　....16 ビット
バイナリ...8 ビット

これらのファイルの扱うクラスは、java.io パッケージに収められています。

ストリーム	入力ストリーム	出力ストリーム
文字ストリーム	Reader	Writer
バイトストリーム	InputStream	OutputStream

※ これらのクラスをもとに継承して実際に利用するクラスを生成しています。

① ファイルを開く

C 言語でも、ファイルを取り扱うには、最初に「開く(open)」という作業が必要でした。
Java では、FileReader クラスのオブジェクトを生成することで、ファイルを開くことが出来ます。

```
FileReader in = new FileReader("〇〇〇〇〇.〇〇〇");
```

※ 〇〇〇〇〇.〇〇〇はファイル名

② データを読む

データを読み込むには、read()メソッドを利用します。

```
int c;
c=in.read();
```

③ ファイルを閉じる

C 言語同様、ファイルの処理が終わったら、必ず閉じなければなりません。close メソッドを用いてファイルを閉じます。

```
in.close();
```

④ プログラム例

```
import java.io.*;
class test24 {
  public static void main(String[] args) {
    FileReader in = new FileReader(args[0]);
    int c;
    String moji = new String();

    while((c=in.read()) != -1)
      moji = moji + (char)c;
    System.out.println(moji);
    in.close();
  }
}
```

※ 実際にコンパイルするとエラーが出ます。

⑤ その他の入出力

テキストファイルの入力以外も同様に行います。

i)　テキストファイルに書き込むためには

クラス　　FileWriter
例）FileWriter out =new FileWriter("〇〇〇〇〇.〇〇〇");

※〇〇〇〇〇.〇〇〇はファイル名

書き出すメソッド　write
例）out.wite ("△△△△△△△△△△");

※△△△△△△△△△△は文字列
もちろん、変数でも良い

ii)　バイナリファイルを読み込むには

クラス　　FileInputStream
例）FileInputStream in =new FileInputStream("〇〇〇〇〇.〇〇〇");

※〇〇〇〇〇.〇〇〇はファイル名

読み込むメソッド　in
例）int c;
　　c=in.read ();

iii) バイナリファイルに書き込むためには

クラス　　FileOutputStream
例）FileOutputStream out =new FileOutputStream("〇〇〇〇.〇");

※〇〇〇〇.〇はファイル名

書き出すメソッド　write
例）out.wite (△△);

△△は int 型か byte 型の値を示します。

（4） 例外

Java では、プログラムを実行する最中に発生するエラーを 例外 (exception) として扱うことができます。例外には例えば、0 で割り算をした、メモリが枯渇した、存在しないファイルを開こうとしたなどがあります。

test24.java では、指定したファイルが存在しなかったりした場合、当然エラーが発生します。そのための処理をしていないので、コンパイル時にエラーが出てしまったのです。

そのため例外のための処理を追加しなくてはなりません。

（3）④のプログラム例をコンパイルをすると、

```
test24.java:4: 例外 java.io.FileNotFoundException は報告されません。スローするにはキャッチまたは、スロー宣言をしなければなりません。
    FileReader in = new FileReader(args[0]);
                    ^
test24.java:8: 例外 java.io.IOException は報告されません。スローするにはキャッチまたは、スロー宣言をしなければなりません。
    while((c=in.read()) != -1)
                    ^
test24.java:11: 例外 java.io.IOException は報告されません。スローするにはキャッチまたは、スロー宣言をしなければなりません。
    in.close();
               ^
エラー 3 個
```

というようなコンパイルエラーのメッセージが表示されました。

「test24.java:4: 例外 java.io.FileNotFoundException...」は「ファイルが見つからない」時の例外処理について記

述がないことを示し、「test24.java:8: 例外 java.io.IOException...」は「入出力時に例外が発生」の時の例外処理について記述がないことを示しています。
ここで示されている java.io.FileNotFoundException や java.io.IOException は「例外クラス」といい、例外が発生したときに生成されるクラスです。
そして、これらのクラスが発生した場合、以下の方法で例外処理を行います。

```
try {
        :
        例外が発生しそうな処理
        :
} catch (例外１クラス 変数) {
        :
        例外処理１
        :
} catch (例外２クラス 変数) {
        :
        例外処理２
        :
} finally {
        :
        共通処理
        :
}
```

catch 内では、適合する例外クラスを示し、変数は任意の名前を示せいます。このとき、変数には例外に関するメッセージが格納されます。
また、自分で作成したメソッドで例外を発生させる場合は、throws を定義します。

```
try{
    a(b);
    :
    int b=1;
}catch( Exception e){
    :
}
void a(int b) throws  Exception  {
    :
}
```

メソッド a(b)を実行したとき、例外が生じたら、Exception クラスを生成します。
それでは、test24.java に例外処理をつけてみましょう。

- test24.java -

```
import java.io.*;
class test24 {
  public static void main(String[] args) {
    try{
      FileReader in = new FileReader(args[0]);
      int c;
      String moji = new String();
      while((c=in.read())!=-1)
        moji = moji + (char)c;
      System.out.println(moji);
      in.close();
    }catch(IOException ie){
      System.out.println("ファイルがありません:"+ie);
    }catch(Exception e){
```

```
            System.out.println("ファイル指定がありません:"+e);
        }
    }
}
```

ここで挙げた例外クラス以外にも以下のようなものがあります。

Object		Java のすべてのクラスのルート
Throwable		Java の例外とエラーの階層のルート
Exception		Java の例外階層のルート
ClassNotFoundException		クラス定義が見つからない
CloneNotSupportedException		Cloneable インタフェース未実装
IllegalAccessException		クラスにアクセスできない
InstantiationException		抽象クラスまたはインタフェースをインスタンス化しようとした
InterruptedException		スレッドが中断された
NoSuchFieldException		指定されたフィールドがない
NoSuchMethodException		指定されたメソッドがない
IOException	*	入出力時に例外が発生
CharConversionException	*	文字変換で例外が発生
EOFException	*	ファイルの終わりに達した
FileNotFoundException	*	ファイルが見つからない
InterruptedｴOException	*	入出力処理が中断された
ObjectStreamException	*	ObjectStream 例外のスーパークラス
InvalidClassException	*	シリアライズ処理に関する問題がクラスにある
InvalidObjectException	*	デシリアライズされたオブジェクトで入力検証に失敗
NotActiveException	*	ストリーム環境がアクティブでないときにメソッド呼び出しをした
NotSerializableException	*	オブジェクトをシリアライズできない
OptionalDataException	*	オブジェクト読み込み時に期待外のデータに遭遇
StreamCorruptedException	*	読み取ったデータストリームが破損している
WriteAbortedException	*	書き込み中に例外発生したストリームを読み込んだ
SyncFailedException	*	システムバッファを同期させる FileDescriptor.sync() の呼び出しが失敗
UnsupportedEncodingException	*	指定された文字符号化形式をサポートしていない
UTFDataFormatException	*	不正な UTF-8 形式文字列に遭遇した
RuntimeException		ランタイム例外のスーパークラス
ArithmeticException		ゼロ除算などの算術例外が発生
ArrayStoreException		配列に不正な型のオブジェクトを格納しようとした
ClassCastException		不正なクラス型へのキャスト
EnumConstantNotPresentException		不正な定数に参照しようとした
IllegalArgumentException		メソッドへの引数が不正
IllegalThreadStateException		スレンドが要求された処理を行なうのに適した状態にない
NumberFormatException		不適切な文字列を数値に変換しようとした
IllegalMonitorStateException		モニタ状態が不正である
IllegalStateException		メソッドが要求された処理を行なうのに適した状態にない
IndexOutOfBoundsException		インデックスが範囲外
ArrayIndexOutOfBoundsException		範囲外の配列活字指定
StringIndexOutOfBoundsException		範囲外の String 添字指定
NegativeArraySizeException		負値で配列サイズを指定
NullPointerException		null オブジェクトにアクセスした
SecurityException		セキュリティ違反である
TypeNotPresentException		指定した型が見つからない
UnsupportedOperationException		サポートされていないメゾッドを呼び出した
Error		Java のエラー階層のルート
AssertionError		アサードテストが失敗した
LirikageError		クラスのリンク関係エラーのスーパークラス
ClassCircularityError		クラス初期化中に循環参照を検出
ClassFormatError		クラスファイルの形式に誤りがある
UnsupportedClassVersionError		JVM がサポートしていないバージョン番号をもつクラスファ

	イルを読もうとした
Exception ｴ n ｴ nitializerError	静的イニシャライザで例外が発生
IncompatibleClassChangeError	クラス定義変更により矛盾が生じた
AbstractMethodError	抽象メソッドを呼び出した
IllegalAccessError	アクセスできないメソッドやフィールドを使おうとした
InstantiationError	インタフェースまたは抽象クラスをインスタンス化しようとした
NoSuchFieldError	指定したフィールドが存在しない
NoSuchMethodKrror	指定したメソッドが存在しない
NoClassDefFoundError	クラス定義が見つからない
Unsatisfied ｴ liinkError	クラスに含まれるリンク情報を解決できない
VerifyError	クラスファイル内に不整合部分を検出した
ThreadDeath	スレッドを停止しなければならないことを意味するエラー
VirtualMachineError	JVM に問題が生じた
InternalError	内部エラーが生じた
OutOfMemoryError	メモリ不足でメモリ確保できない
StackOverflowError	スタックオーバーが生じた
UnknownError	未知の重大な例外が発生

注　：　*を付与したものは Java.io パッケーシ､他は Java.lang パッケージに属する。
注　：　JVM=Java 仮想マシン（Java Virtual Machine）
注　：　シリアライズとはオブジェクトをバイトストリームに変換して保存する機能、それを読み込んで元のオブジェクトを再現(デシリアライズ)できる

演習　キーボードから値を入力し、表示するプログラムを作成しなさい。(test25.java)

14. 抽象クラス

実際の処理を記述せず、その入出力だけ宣言した不完全なメソッド（抽象メソッド）を持つクラスを抽象クラスと呼びます。このようなクラスは、継承され、サブクラスでオーバーライドされることによって当該メソッドの実装を果たし、はじめてインスタンス化され得ます。

（1） 抽象メソッド

抽象メソッドは修飾子 abstract で宣言します：

```
abstract [戻り値型] <メソッド名>(シグネチャ);
```

通常のメソッドならば、実際の処理が {} の中に記述されているはずですが、抽象メソッドの場合はセミコロン ; になっています。
抽象メソッドでは、メソッド名とシグネチャ（引数とそのデータ型の組)、戻り値の型の宣言だけ行います。

（2） 抽象クラス

抽象メソッドを持つクラスは抽象クラスであり、修飾子で abstract 宣言しておかなければなりません：

```
abstract class <クラス名>{
        メンバ変数
        コンストラクタ
        抽象メソッド
        普通のメソッド
}
```

抽象クラスはインスタンス化できません。必ず継承されて、継承したサブクラスがインスタンス化されます。このとき、抽象クラスの抽象メソッドはオーバーライドされて、完全なものになっていることが必要です。

（3） プログラム例

```
// 抽象クラス
abstract class Oya {
	private int price = 1980;
	int getPrice() {
		return price;
	}
	// 抽象メソッド
	abstract int sales();
}
// 実装クラスその１
class Ko1 extends Oya {
	int sales() {
		double d = getPrice()*0.9;
		return (int)d;
	}
}
// 実装クラスその２
class Ko2 extends Oya {
	int sales() {
		double d = getPrice()*0.8;
		return (int)d;
	}
}

class test26 {
	public static void main(String[] args) {
		Ko1 koObj1 = new Ko1();
		Oya oyaObj = koObj1;
		System.out.println("Price: " + oyaObj.getPrice());
		System.out.println("90%: " + oyaObj.sales());
		Ko2 koObj2 = new Ko2();
		oyaObj = koObj2;
		System.out.println("80%: " + oyaObj.sales());
	}
}
```

15. インターフェイス

内容に抽象メソッドしか持たないクラスのようなものをインタフェースと呼びます。クラスと並んで、パッケージのメンバーとして存在します。インタフェースはクラスによって実装 （implements）され、実装クラスはインタフェースで宣言されている抽象メソッドを実装します。
インタフェースは、フィールドに定数を定義します。また、抽象メソッドのシグネチャを定義します。定数と抽象メソッドをリストしたものがインタフェースであり、クラスから実装されることによって使われます。クラスの場合は、単一のクラスしか継承（extends）できせんが、インタフェースの場合は、複数のインタフェースを実装（implements）することができます。

```
class InterfaceImpl implements Interface1, interface2, interface3 {
	...
}
```

（1） インタフェースの基本書式

インタフェースは、定数と抽象メソッドだけメンバーに持つことが出来ます。そもそも、オブジェクトの public インタフェースになるメソッドを宣言するためのものです。必要なのは、メソッド名、引数、戻り値の型だけです。

```
[修飾子] interface <インタフェース名> {
	データ型 変数名 = 値;
```

```
        修飾子 戻り値のデータ型 メソッド名(引数の型宣言);
}
```

① インタフェースの修飾子
interface の修飾子は public のみです。

② インタフェースのメンバ変数定義
インタフェースのメンバ変数は定数です。必ず値が代入されなければなりません。自動的に final public static 修飾子がつけられます。実装時に別の値を代入することは出来ません。他の修飾子は記述できません。

③ インタフェースのメソッド宣言
インタフェースのメソッドは抽象メソッドのみ記述可能です。自動的に abstract public 修飾子がつけられており、実装するクラス側で実装して完全なものにしておく必要があります。他の修飾子は記述できません。

④ インタフェースのサンプル

```
interface Interface1 {
        // フィールド
        int INT_VAL1 = 10;
        // 抽象メソッド
        String method(int a, int b);
}
interface Interface2 {
        // フィールド
        int INT_VAL2 = 100;
        // 抽象メソッド
        void method(String s);
}
```

（2） インタフェースの実装
クラスは任意の個数のインタフェースを実装できます。インタフェースを実装するクラスのことを実装クラスと呼びます。

```
[修飾子] class <クラス名>
        implements <インタフェース名リスト>{
        メンバ変数
        コンストラクタ
        抽象メソッドの実装
        普通のメソッド
        など
}
```

インタフェースを実装したら、そこで定義されている抽象メソッドを全て実装しておかないとコンパイルエラーになります。
インタフェースの抽象メソッドは、自動的に public 修飾子がついていますので、実装時にも宣言しておく必要があります。

```
// 二つのインタフェースを実装するクラス
class TestInterface implements Interface1, Interface2 {
        int x;
        // インタフェースの実装
        public String method(int a, int b) {
                this.x = a + b;
```

```
                return "----Interface1----";
        }
        // インタフェースの実装
        public void method(String s) {
                Syste.out.println(s);
        }
}
```

実装クラスは普通のクラスになるので、必要があれば、他のクラスを継承することも出来ます。

```
[修飾子] <クラス名>
        extends <スーパークラス名>
        implements <インターフェイ名スリスト>{
                メンバ変数
                コンストラクタ
                抽象メソッドの実装
                普通のメソッド
                など
}
```

クラスの継承は一つのクラスからしか出来ませんが、インタフェースの実装は複数のインタフェースから出来ます。

（3） インタフェース間の継承

インタフェースは他のインタフェースを継承することも出来ます。このとき、継承するインタフェースは複数あって構いません。クラスは単一継承ですが、インタフェースの場合は多重継承が許されます。

```
[修飾子] interface <インタフェース名>
        extends <スーパーインタフェース名のリスト>{
                内容
}
```

多重継承によって、インタフェースは網の目のような構造を構成しますが、その構造の中で変数は一意的でなければなりません。
多重継承でつながっているインタフェース内で、一つの変数名が複数箇所で定義されているとコンパイルエラーになります。

実装クラスでは、スーパーインタフェースも含めて全ての抽象メソッドを実装する必要があります。
インタフェースの多重継承の網の目構造の中で、メソッドが複数宣言されていても、実装クラスで必要なメソッドの実装は決まっているはずですから、その実装でオーバーライドするだけです。

（4） プログラム例

```
public class test27 {
  public static void main(String[] args) {
    TestClass cls = new TestClass();
    cls.superMethod1();
    cls.superMethod2();
    cls.method1();
    cls.method2();
  }
}

interface InterfaceSuperClass1 {
  int i = 10;
  void superMethod1();
}
```

```
interface InterfaceSuperClass2 {
  int k = 20;
  void superMethod2();

}

interface InterfaceClass1
        extends InterfaceSuperClass1, InterfaceSuperClass2 {
  void method1();
}

interface InterfaceClass2 {
  void method2();
}

class TestClass implements InterfaceClass1, InterfaceClass2 {
  public void superMethod1() {
    System.out.println("Message superMethod1 ; " + i);
  }

  public void superMethod2() {
    System.out.println("Message superMethod2 : " + k);
  }

  public void method1() {
    System.out.println("Message method1");
  }

  public void method2() {
    System.out.println("Message method2");
  }
}
```

（5） インタフェースの特徴

インタフェースの特徴をまとめておきます。大事なことは、定数と抽象メソッドしか記述できないということです。

* フィールドには自動的に修飾子 final public static が付けられる
* メソッドには自動的に修飾子 abstract public が付けられる
* インタフェースのメソッドは抽象メソッドでなければならない
* 実装クラスでは複数のインタフェースを実装 (implements) できる
* 別のインタフェースを多重継承 (extends) できる
* 実装クラスではインタフェースの全てのメソッド宣言を実装しなければならない
* 実装クラス型オブジェクトはインタフェース型変数に代入互換である
  （逆は明示的なキャストが必要）

（6） インタフェースの用途

① カプセル化

Java のようなオブジェクト指向言語では、データやメソッドの隠蔽によるカプセル化 (encapsulate) が重要です。データは全て private 修飾して外部から隠蔽し、public 修飾されたメソッドによってアクセスします。外部に公開する必要のない処理の実装も、private 修飾したメソッド内に記述することで、クラス外部からは隠蔽します。外部に対して、最低限度のメソッドだけを public 修飾して、公開するわけです。この、公開されたメソッドが、当該クラスをインスタンス化したオブジェクトに対して、別のオブジェクトがアクセス可能な全て、つまりインタフェースになるわけです。

② インタフェース

オブジェクトの、外部に対するインタフェースとなるメソッドを宣言するのが、パッケージのメンバーである Java インタフェースです。
ここでは、メソッド内部の処理は記述しません。つまり、抽象メソッドだけを内部に持つことができます。インタフェースで宣言されたメソッドは、抽象メソッドなので、クラスで実装することで利用します。インタフェースを実装するクラスは、インタフェースの型をデータ型とする変数に代入互換であり、そこで宣言されている全てのメソッドを実装する義務を負います。
使う側からすれば、対象のオブジェクトの型となるクラスが実装しているインタフェースが分かれば、どんなメソッド名に何の引数を与えれば、どんな型の戻り値が得られるのかが分かるわけです。

③ 抽象クラスとインタフェース

抽象クラスには、抽象メソッドだけではなく、実装を持つメソッドも記述できました。その場合は継承されることで利用されますが、Java は単一継承なので、別のクラスを継承している場合は使えません。一方、インタフェースは、継承 (extends) されるのではなく実装 (implements) され、複数のインタフェースをカンマ区切りリストで指定することができます。インタフェースは実装から型適合の仕組みを分離したものだといえます。
継承の多重継承が許されないのは、メソッド名だけでは、どのスーパークラスで実装されているものか区別できないからです。仮に、多重継承が許される場合に、複数のスーパークラスで、同じシグネチャのメソッドの実装が複数発見される場合、何れを選ぶか選択する仕組みが必要になります。単一継承ならば、継承階層の下から順番に探していき、最初に見つかった実装を採用するだけなので簡単です。
一方、インタフェースの多重実装が許されている理由は、重複するメソッド宣言の解決が不必要だからです。仮に複数のインタフェース間で同じメソッドが宣言されていても、それらは実装を持たないので、実装の競合は発生しません。そのメソッドの実装は、当該クラス内でなされているので、何れを選択するかが問題にならないのです。

④ 多態性（ポリモーフィズム）

実装クラスではインタフェースで宣言されているメソッドを全て実装する必要があります。実装していないメソッドが存在すれば、そのクラスは抽象クラスとして abstract 宣言しなければなりません。
インタフェースは、複数のクラスに実装されることを想定しています。このとき、インタフェース型の変数には、実装クラス型オブジェクトの参照を代入できます。同じインタフェース型の変数に対して、同じメソッドを呼び出しても、実行される実装は、そのオブジェクトがインスタンス化されたオリジナルのクラスでの実装によって、異なります。このような性質は、サブクラスとスーパークラスの場合にも成り立ちますが、インタフェースの場合は、多重実装が許されるので、この特徴が顕著だといえます。
インタフェースは、データ型として、クラスの仕様を定義するものであり、多態性のための言語仕様だといえます。

16. GUI プログラミング

1984 年 Macintosh の登場により、GUI 環境でのコンピュータの操作が普及した。それ以前は、キーボードからのコマンド入力によってコンピュータの操作をしていたが、マウスの操作により、だれでも簡単にコンピュータを扱うことができるようになった。
しかし、コンピュータの操作の容易さとは裏腹に、プログラミングはウインドウやマウスの取り扱い、メニューの作成など面倒になっていった。
この章では、Java での GUI プログラミングにいて学んでいく。

（1） GUI パッケージ

Java では、GUI 環境での面倒なウインドウ、メニューの取り扱いは、パッケージ化されています。代表的なパッケージには、「AWT」、「Swing」といったものがあります。

（2） AWT

AWT（Abstract Windowing Tools）は、Java で GUI アプリケーションを作成するためのクラスライブラリです。現在では Swing もよく利用されていますが、Swing よりも軽い GUI アプリケーションを作成することが可能です。

（3） Swing

Swing（スィング）は、JDK1.2 でサポートされた AWT よりも新しい GUI コンポーネントです。AWT に対して以下のような特徴があります。

- OS が変わっても見栄えが変わらない。
- AWT よりも重い。
- AWT での部品名に J をつけたものが多い。

この授業では、シンプルな AWT を用いて進めていきます。

17. 部品の配置

（1） フレーム（Frame）

フレーム（Frame）はウインドウやダイアログの基本となるものです。

```
import java.awt.*; // java.awt.* をインポートします。

public class test28 extends Frame {//アプリケーションを Frame クラスのサブクラスとして実装します。
    public static void main(String [] args) {
        new test28();
    }
    test28() {
        super("FrameTest");//super()で親クラスのコンストラクタを呼び出します。
        setSize(200, 100); //setSize()でフレームのサイズを指定します。
        setVisible(true); //setVisible(true);でフレームを表示します。
    }
}
```

200×100 の大きさのウィンドウ（フレーム）が表示されれば成功です。
ウィンドウを閉じる機能が実装されていないので、コマンドプロンプトで Ctrl+C を入力して終了させてください。（ただし、xyzzy のシェル上ではうまくいかないと思います。）

```
C>java test28↵
Ctrl+C  ← Ctrl キーを押しながら C キーを押す
```

（2） ボタン（Button）

ボタン（Button）を配置します。

```
import java.awt.*;

public class test29 extends Frame {
    public static void main(String [] args) {
        new test29();
    }
    test29() {
        super("ButtonTest");
        setSize(200, 100);
        setLayout(new FlowLayout());
        Button b1 = new Button("OK");
        add(b1);
        setVisible(true);
    }
}
```

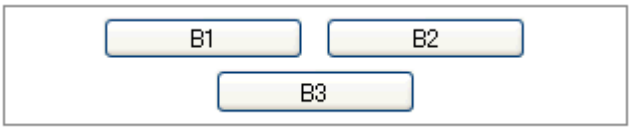
（3） 部品のレイアウト

画面上にボタンなどの部品を配置する際に使用するレイアウトアルゴリズムには下記のものなどがあります。

① FlowLayout - 流し込みレイアウト

FlowLayout は、左上から右下に向けて、部品を流し込むようにレイアウトします。HTML で文字や画像を流し込むのに似ています。

```
setLayout(new FlowLayout());
  add(b1);
   add(b2);
   add(b3);
```



② GridLayout - グリッドレイアウト

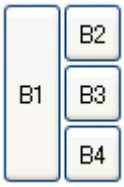
GridLayout は、画面を n 列×m 行 の格子に分割し、それぞれの升目に部品をレイアウトします。

```
setLayout(new GridLayout(2, 3));
add(b1);
add(b2);
add(b3);
add(b4);
add(b5);
add(b6);
```

③ GridBagLayout - グリッドバッグレイアウト

GridBagLayout を用いることで、画面を n 列×m 行 の升目に分割し、(0, 0) の座標から幅 1、高さ 3 の部品を配置することができます。

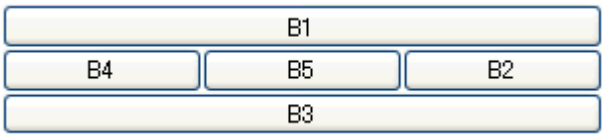
```
GridBagLayout gbl = new GridBagLayout();

void addButton(Button b, int x, int y, int w, int h) {
    GridBagConstraints gbc = new GridBagConstraints();
    gbc.fill = GridBagConstraints.BOTH;
    gbc.gridx = x;
    gbc.gridy = y;
    gbc.gridwidth = w;
    gbc.gridheight = h;
    gbl.setConstraints(b, gbc);
    add(b);
}

Test() {
        :
    setLayout(gbl);
    addButton(b1, 0, 0, 1, 3);   // (0, 0) 幅=1, 高さ=3
    addButton(b2, 1, 0, 1, 1);   // (1, 0) 幅=1, 高さ=1
    addButton(b3, 1, 1, 1, 1);   // (1, 1) 幅=1, 高さ=1
    addButton(b4, 1, 2, 1, 1);   // (1, 2) 幅=1, 高さ=1
        :
}
```



④ BorderLayout - ボーダーレイアウト

BorderLayout は、画面を上下左右中の 5 つのブロックに分け、それぞれに部品を配置します。5 つのブロックはそれぞれ、東西南北中央（East, West, South, North, Center）で示されます。

```
setLayout(new BorderLayout());
add("North", b1);
add("East", b2);
add("South", b3);
add("West", b4);
add("Center", b5);
```

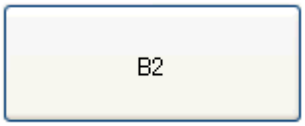


※　BorderBagLayout（ボーダーバッグレイアウト）もあるかな？

⑤ CardLayout - カードレイアウト

CardLayout は、複数の部品をカードのスタックとして保持し、一番最初のカードだけを画面に表示します。カードを入れ替えるには next()、previous()、first()、last()、setVisible(true);; などのメソッドを用います。

```
CardLayout cl = new CardLayout();
setLayout(cl);
add("b1", b1);
add("b2", b2);
add("b3", b3);
cl.show(this, "b2");
```



⑥ BoxLayout - ボックスレイアウト

BoxLayout は Swing で追加されたレイアウトで、部品を単純に横方向または縦方向にならべていきます。横方向の場合、画面の横幅が狭くなっても自動改行されない点が FlowLayout と異なります。

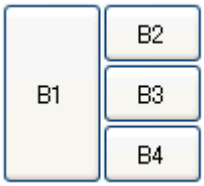
```
setLayout(new BoxLayout(this, BoxLayout.X_AXIS);
add(b1);
add(b2);
add(b3);
```

※　Swing でのレイアウトなので、表示例は省略

⑦ パネルによるレイアウト（Panel）

パネル（Panel）は、ボタンなどの部品や他のパネルを乗せることができる無地の板です。パネルを用いることで、柔軟なレイアウトが可能になります。下記の例では、フレームを左右に分割してそれぞれにパネル p1、p2 を貼り付けています。さらに、p1 には b1 を、p2 には b2、b3、b4 を貼り付けています。

```
import java.awt.*;

public class test30 extends Frame {
    Panel p1 = new Panel();
    Panel p2 = new Panel();
    Button b1 = new Button("B1");
    Button b2 = new Button("B2");
    Button b3 = new Button("B3");
    Button b4 = new Button("B4");
    public test30() {
        super("Test");
        setLayout(new GridLayout(1, 2));
        add(p1);
        add(p2);
        p1.setLayout(new GridLayout(1, 1));
        p2.setLayout(new GridLayout(3, 1));
        p1.add(b1);
        p2.add(b2);
        p2.add(b3);
        p2.add(b4);
        setSize(300, 200);
        setVisible(true);
    }
    public static void main (String args []) {
        new test30();
    }
}
```



（4） ラベル（Label）

ラベル（Label）を配置します。

```
public class test31 extends Frame {
    public static void main(String [] args) {
        new test31();
    }
    test31() {
        super("LabelTset");
        setSize(200, 100);
        setLayout(new FlowLayout());
        Label l1 = new Label("HelloWorld!!");
        add(l1);
        setVisible(true);;
    }
}
```

（5） テキストフィールド（TextField）

テキストフィールド（TextField）を配置します。

```
import java.awt.*;

public class test32 extends Frame {
    public static void main(String [] args) {
```

```
            new test32();
        }
        test32() {
            super("TestTextField");
            setSize(200, 100);
            setLayout(new FlowLayout());
            TextField t1 = new TextField("Hello World!!");
            add(t1);
            setVisible(true);
        }
    }
```

※ テキストの内容を参照する → textField.getText()
※ テキストの内容を設定する → textField.setText()

（6） チェックボックス（Checkbox）

チェックボックス（Checkbox）を配置します。

```
    import java.awt.*;

    public class test33 extends Frame {
        public static void main(String [] args) {
            new test33();
        }
        test33() {
            super("CheckboxTest");
            setSize(200, 100);
            setLayout(new FlowLayout());
            Checkbox c1 = new Checkbox("OK?");
            add(c1);
            setVisible(true);
        }
    }
```

※ チェックボックスのオンオフを監視する → ItemListener、ActionListener
※ 現在の状態を調べる → checkbox.getState()
※ オン・オフを設定する → checkbox.setState(true)

（7） チョイス（Choice）

チョイス（Choice）を配置します。

```
    import java.awt.*;

    public class test34 extends Frame {
        public static void main(String [] args) {
            new test34();
        }
        test34() {
            super("ChoiceTest");
            setSize(200, 100);
            setLayout(new FlowLayout());
            Choice c1 = new Choice();
            c1.add("ChoiceA");
            c1.add("ChoiceB");
            c1.add("ChoiceC");
            add(c1);
            setVisible(true);
        }
    }
```

※ 変更を監視する → ItemListener の延長で itemevent.getItem()
※ 選択中の項目名を得る → choice.getSelectedItem()
※ 選択中の項目インデックスを得る → choice.getSelectedIndex()
※ n 番目の項目名を得る → choice.getItem()
※ 項目数を得る → choice.getItemCount()
※ 項目を追加する → choice.add()
※ 項目を削除する → choice.remove()

（8） テキストエリア（TextArea）

テキストエリア（TextArea）を配置します。

```
import java.awt.*;

public class test35 extends Frame {
    public static void main(String [] args) {
        new test35();
    }
    test35() {
        super("TextAreaTest");
        setSize(200, 100);
        setLayout(new FlowLayout());
        TextArea b1 = new TextArea("Hello World!!", 3, 20);
        add(b1);
        setVisible(true);
    }
}
```

（9） リスト（List）

リスト（List）を配置します。

```
    import java.awt.*;

    public class test36 extends Frame {
        public static void main(String [] args) {
            new test36();
        }
        test36() {
            super("ListTest");
            setSize(200, 100);
            setLayout(new FlowLayout());
            List list1 = new List();
            list1.add("ListA");
            list1.add("ListB");
            add(list1);
            setVisible(true);
        }
    }
```

※ 項目の選択状態を監視する → ItemListener で list.getSelectedItem()
※ 項目のダブルクリックを監視する → ActionListener で list.getSelectedItem()
※ 選択中の項目名を得る → list.getSelectedItem()
※ 選択中の項目インデックスを得る → list.getSelectedIndex()
※ n 番目の項目名を得る → list.getItem()
※ 項目数を得る → list.getItemCount()
※ 項目を追加する → list.add()
※ 項目を削除する → list.remove()

（１０）スクロールバー（Scrollbar）

スクロールバー（Scrollbar）は、スクロールバーを表示します。スクロールバーの動きに従って部品を自動的にスクロールさせるには、ScrollPane を利用します。

```
import java.awt.*;

public class test37 extends Frame {
    public static void main(String [] args) {
        new test37();
    }
    test37() {
        super("ScrollbarTest");
        setSize(200, 100);
        setLayout(new BorderLayout());
        Scrollbar sb1 = new Scrollbar(Scrollbar.HORIZONTAL);
        add(sb1, BorderLayout.SOUTH);
        setVisible(true);
    }
}
```

※ スクロールを監視する → AdjustmentListener
※ 現在のスクロール位置を得る → scrollbar.getValue()

（１１）キャンバス（Canvas）

キャンバス（Canvas）は、線、丸、矩形などの図形を描画する際に用いられる部品です。通常、Canvas クラスのサブクラスを定義して使用します。paint() は、setVisible(true);; や repaint() が呼ばれたときや、隠れていたウィンドウが表に表示されるときなどに自動的に呼び出されるメソッドです。

```
import java.awt.*;

class test38 extends Frame {
    public static void main(String[] args) {
        new test38();
    }
    test38() {
        super("CanvasTest");
        setSize(200, 100);
        setLayout(new BorderLayout());
        MyCanvas mc1 = new MyCanvas();
        add(mc1, BorderLayout.CENTER);
        setVisible(true);
    }
}

class MyCanvas extends Canvas {
    public void paint(Graphics g) {
        g.drawLine(10, 10, 120, 40);
    }
}
```

※ マウスの動きを監視する → MouseListener、MouseMotionListener
※ 描画する → paint()
※ 再描画を指示する → repaint()

（１２）メニュー（Menu）

メニューは、主に下記の 4 種類のオブジェクトから構成されます。

クラス	説明
MenuBar	メニューバー。
Menu	メニュー。子アイテムを持つメニュー項目。
MenuItem	メニューアイテム。子アイテムを持たないメニュー項目。
CheckboxMenuItem	チェックタイプのメニューアイテム。

実行例では、下記の階層構造をもつメニューバーを実装しています。

■ MenuBar
├□ File
│ ├○ Open Ctrl+O
│ └○ Exit
└□ View
  ├◎ Status Bar
  └□ Size
    ├○ Large
    └○ Small

記号の意味

■ ：MenuBar クラス
□ ：Menu クラス
○ ：MenuItem クラス
◎ ：CheckboxMenuItem クラス

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

class test39 extends Frame implements ActionListener, ItemListener {
    public static void main(String[] args) {
        new test39();
    }
    test39() {
        setTitle("MenuTest");
        setSize(200, 120);
        setLayout(new FlowLayout());
        MenuBar menuBar = new MenuBar();
        setMenuBar(menuBar);
        // [File]
        Menu menuFile = new Menu("File");
        menuFile.addActionListener(this);
        menuBar.add(menuFile);
        // [File]-[Open]
        MenuItem menuOpen = new MenuItem("Open...", new MenuShortcut('O'));
        menuFile.add(menuOpen);
        // [File]-[----]
        menuFile.addSeparator();
        // [File]-[Exit]
        MenuItem menuExit = new MenuItem("Exit");
        menuFile.add(menuExit);
        // [View]
        Menu menuView = new Menu("View");
        menuView.addActionListener(this);
        menuBar.add(menuView);
        // [View]-[Status Bar]
        CheckboxMenuItem menuStatusBar = new CheckboxMenuItem("Status Bar");
        menuStatusBar.addItemListener(this);
        menuView.add(menuStatusBar);
        // [View]-[Size]
        Menu menuSize = new Menu("Size");
        menuSize.addActionListener(this);
        menuView.add(menuSize);
        // [View]-[Size]-[Large]
        MenuItem menuSizeLarge = new MenuItem("Large");
        menuSize.add(menuSizeLarge);
        // [View]-[Size]-[Small]
        MenuItem menuSizeSmall = new MenuItem("Small");
```

```
            menuSize.add(menuSizeSmall);
            setVisible(true);
        }
        public void actionPerformed(ActionEvent e) {
            System.out.println(e.getActionCommand());
        }
        public void itemStateChanged(ItemEvent e) {
            CheckboxMenuItem menu = (CheckboxMenuItem)e.getSource();
            if (menu.getState()) {
                System.out.println(menu.getLabel() + " SELECTED");
            } else {
                System.out.println(menu.getLabel() + " DESELECTED");
            }
        }
    }
```

※ メニューバー → MenuBar
※ 子メニューを持つメニュー → Menu
※ 子要素を持たないメニュー → MenuItem または CheckboxMenuItem
※ MenuItem の実行を監視する → 親の Menu オブジェクトに ActionListener
※ CheckboxMenuItem の実行を監視する → ItemListener
※ セパレータを追加する → menu.addSeparator()
※ MenuItem を識別する → menuitem.getActionCommand()
※ CheckboxMenuItem を識別する → checkboxmenuitem.getLabel()

（１３）ダイアログ（Dialog）

ダイアログ を表示するには、Dialog クラスのサブクラスを作成してダイアログの名前や機能を定義し、これを setVisible(true);; メソッドで表示します。ダイアログの作り方や、作法、ダイアログ上への部品の配置方法はフレームと同様です。

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

    class test40 extends Frame {
        public static void main(String[] args) {
            new test40();
        }
        test40() {
            super("DialogTest");
            setSize(200, 100);
            MyDialog dlg = new MyDialog(this);
            dlg.setVisible(true);
            setVisible(true);
        }
    }

    class MyDialog extends Dialog implements ActionListener {
        MyDialog(Frame owner) {
            super(owner);
            setLayout(new FlowLayout());
            Button b1 = new Button("OK");
            b1.addActionListener(this);
            add(b1);
            setTitle("MyDialog");
            setSize(80, 80);
        }
        public void actionPerformed(ActionEvent e) {
            hide();
        }
```

```
    }
```

18. イベント処理

ボタンをクリックした、メニューを実行した、ウィンドウをリサイズした、マウスを動かした、キーを押したなどの操作のことをイベントといいます。
このイベントを監視し、イベント発生時に対応するアクションを実行するオブジェクトことを「リスナー」といいます。
主なリスナーには次のようなものがあります。

リスナー	イベント	主な監視対象
ActionListener	actionPerformed	MenuItem, List, TextField, Button, AbstructButton, JTextField, ButtonModel, JComboBox, Timer, DefaultButtonModel, ComboBoxEditor, JFileChooser, BasicComboBoxEditor
WindowListener	windowOpened windowClosing windowClosed windowIconified windowDeiconified windowActivated windowDeactivated	Window, JWindow, Frame, Dialog
MouseListener	mouseClicked mousePressed mouseReleased mouseEntered mouseExited	Component, Button, Canvas, Checkbox, Choice, Container, Label, List, Scrollbar, TextComponent, TextField, TextArea
MouseMotionListener	mouseDragged mouseMoved	Component, Button, Canvas, Checkbox, Choice, Container, Label, List, Scrollbar, TextComponent, TextField, TextArea
KeyListener	keyPressed keyReleased keyTyped	Component, Button, Canvas, Checkbox, Choice, Container, Label, List, Scrollbar, TextComponent, TextField, TextArea
TextListener	textValueChanged	TextComponent, TextField, TextArea
ItemListener	itemStateChanged	Checkbox, List, Choice, ItemSelectable, CheckboxMenuItem, AbstractButton, ButtonModel, JComboBox, DefaultButtonModel

※ 監視対象で、J ではじまるクラスは Swing のクラス。

（1） アダプターを使わない

リスナーは、「インターフェイス」として提供されます。インターフェイスは定数と抽象メソッドで構成されています。抽象メソッドは、名前だけを定義した中身のないメソッドです。
そのためインターフェイスを実装（implement）した場合、抽象メソッドの中身を書かなくてはなりません。
WindowListener を実装（implement）する場合、７つの抽象メソッド（windowOpened、windowClosing、windowClosed、windowIconified、windowDeiconified、windowActivated、windowDeactivated）について記述しなければなりません。

① アクションリスナー（ActionListener）

ボタンが押されたなどのイベントを監視して、イベント発生時にアクションを実行するサンプルです。

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

public class test41 extends Frame implements ActionListener {
    test41() {
        super("ActionListenerTest");
        Button b1 = new Button("BUTTON1");
```

```
            b1.addActionListener(this);//イベントの登録
            add(b1);
            setSize(200, 100);
            show();
        }
        public void actionPerformed(ActionEvent e) {
            System.exit(0);
        }
        public static void main(String [] args) {
            new test41();
        }
    }
```

ボタン（b1）が押されたことの監視は アクションリスナオブジェクト が行います。アクションリスナオブジェクトは、ActionListener インタフェースを実装（implement）したオブジェクトです。リスナーは実装すべメソッド（抽象メソッド）が決まっています。ActionListener の場合は actionPerformed() というメソッドを実装しなくてはなりません。サンプルでは、自分自身（this）がアクションリスナオブジェクトとなっています。イベント発生時に actionPerformed() メソッドでこれを捕獲し、System.exit(0); でプログラムを終了しています。

下記のように、インナークラスのインスタンスをアクションリスナオブジェクトとすることもできます。

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

    public class test42 extends Frame {
        test42() {
            super("ActionListenerTest2");
            Button b1 = new Button("BUTTON1");
            b1.addActionListener(new MyActionListener());
            add(b1);
            setSize(200, 100);
            setVisible(true);
        }
        class MyActionListener implements ActionListener {
            public void actionPerformed(ActionEvent e) {
                System.exit(0);
            }
        }
        public static void main(String [] args) {
            new test42();
        }
    }
```

② ウィンドウリスナー（WindowListener）

ウィンドウが、開かれた、閉じようとしている、閉じた、アイコン化された、アイコン化解除された、アクティブになった、非アクティブになったというイベントを監視します。

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

    public class test43 extends Frame implements WindowListener {
        test43() {
            super("WindowListenerTest");
            this.addWindowListener(this);
            setSize(200, 100);
            setVisible(true);
        }
```

```
    public void windowOpened(WindowEvent e) {       // 開かれた
        System.out.println("windowOpened");
    }
    public void windowClosing(WindowEvent e) {      // 閉じられている
        System.out.println("windowClosing");
    }
    public void windowClosed(WindowEvent e) {       // 閉じた
        System.out.println("windowClosed");
    }
    public void windowIconified(WindowEvent e) {    // アイコン化された
        System.out.println("windowIconified");
    }
    public void windowDeiconified(WindowEvent e) {  // 非アイコン化された
        System.out.println("windowDeiconified");
    }
    public void windowActivated(WindowEvent e) {    // アクティブになった
        System.out.println("windowActivated");
    }
    public void windowDeactivated(WindowEvent e) {  // 非アクティブになった
        System.out.println("windowDeactivated");
    }
    public static void main(String [] args) {
        new test43();
    }
}
```

③ マウスリスナー（MouseListener）

マウスのボタンに関するイベントを監視します。

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

public class test44 extends Frame implements MouseListener {
    test44() {
        super("MouseListenerTest");
        this.addMouseListener(this);
        setSize(200, 100);
        setVisible(true);
    }
    public void mouseClicked(MouseEvent e) {
        System.out.println("mouseClicked");
    }
    public void mousePressed(MouseEvent e) {
        System.out.println("mousePressed");
    }
    public void mouseReleased(MouseEvent e) {
        System.out.println("mouseReleased");
    }
    public void mouseEntered(MouseEvent e) {
        System.out.println("mouseEntered");
    }
    public void mouseExited(MouseEvent e) {
        System.out.println("mouseExited");
    }
    public static void main(String [] args) {
        new test44();
    }
}
```

④ マウスモーションリスナー（MouseMotionListener）
マウスの動きに関するイベントを監視します。

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

public class test45 extends Frame implements MouseMotionListener {
    test45 () {
        super("MouseMotionListenerTest");
        addMouseMotionListener(this);
        setSize(200, 100);
        setVisible(true);;
    }
    public void mouseDragged(MouseEvent e) {
        System.out.println("D: " + e.getX() + ", " + e.getY());
    }
    public void mouseMoved(MouseEvent e) {
        System.out.println("M: " + e.getX() + ", " + e.getY());
    }
    public static void main(String [] args) {
        new test45();
    }
}
```

⑤ キーリスナー（KeyListener）
キー入力に関するイベントを監視します。

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

public class test46 extends Frame implements KeyListener {
    test46() {
        super("KeyListenerTest");
        TextField tf1 = new TextField();
        tf1.addKeyListener(this);
        add(tf1);
        setSize(200, 100);
        setVisible(true);
    }
    public void keyPressed(KeyEvent e) {
        System.out.println("Press: " + e.getKeyText(e.getKeyCode()));
    }
    public void keyReleased(KeyEvent e) {
        System.out.println("Release: " + e.getKeyText(e.getKeyCode()));
    }
    public void keyTyped(KeyEvent e) {
        System.out.println("Type: " + e.getKeyChar());
    }
    public static void main(String [] args) {
        new test46();
    }
}
```

⑥ テキストリスナー（TextListener）
テキスト入力に関するイベントを監視します。Swing では DocumentListener を使用するのが一般的です。

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

```
public class test47 extends Frame implements TextListener {
    test47 () {
        super("TextListenerTest");
        TextField tf1 = new TextField();
        tf1.addTextListener(this);
        add(tf1);
        setSize(200, 100);
        setVisible(true);
    }
    public void textValueChanged(TextEvent e) {
        TextField tf = (TextField)e.getSource();
        System.out.println(tf.getText());
    }
    public static void main(String [] args) {
        new test47();
    }
}
```

⑦ アイテムリスナー（ItemListener）

チェックボックスなどのアイテムの状態を監視するリスナーです。

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

public class test48 extends Frame implements ItemListener {
    test48() {
        super("ItemListenerTest");
        Checkbox cb1 = new Checkbox("CB1");
        cb1.addItemListener(this);
        add(cb1);
        setSize(200, 100);
        setVisible(true);
    }
    public void itemStateChanged(ItemEvent e) {
        if (e.getStateChange() == ItemEvent.SELECTED) {
            System.out.println("SELECTED");
        } else if (e.getStateChange() == ItemEvent.DESELECTED) {
            System.out.println("DESELECTED");
        }
    }
    public static void main(String [] args) {
        new test48();
    }
}
```

（2） アダプターを使う

ウィンドウリスナーを実装するには、必要の有無に関わらず windowOpened など 7 個のインタフェースを実装する必要があります。この面倒を簡略するために、インタフェースを複数持つものについては アダプター が用意されています。下記のように、ウィンドウリスナーを WindowAdapter のサブクラスとして定義することにより、必要なインタフェースのみを実装することが可能になります。

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

public class test49 extends Frame {
    test49() {
        super("WindowAdapterTest");
        this.addWindowListener(new MyWindowListener());
        setSize(200, 100);
        setVisible(true);
    }
```

```
    public static void main(String [] args) {
        new test49();
    }
}

class MyWindowListener extends WindowAdapter {
    public void windowActivated(WindowEvent e) {
        System.out.println("windowActivated");
    }
}
```

19. GUI プログラムを作る

以下のプログラムを改良して、1 つのウインドウを開き、マウスをドラッグするとその中心からマウスの動きに応じて線が書かれるプログラムを作りなさい。

```
 1 : // applet を使うため
 2 : import java.applet.*;
 3 :    // AWT を使うため,
 4 : import java.awt.*;
 5 :    // イベント駆動関係のクラスを用いるため
 6 : import java.awt.event.*;
 7 :    // Vector クラスを用いるため
 8 : import java.util.*;
 9 :
10 :    // 線分のクラスを定義する.
11 : class Line{
12 :    // 始点, 終点の X 座標, Y 座標を int で保持する.
13 :    public int start_x,start_y,end_x,end_y;
14 :    // Line のコンストラクタ
15 :    public Line(int x1,int x2,int x3,int x4){
16 :     start_x=x1;
17 :     start_y=x2;
18 :     end_x=x3;
19 :     end_y=x4;
20 :    }
21 : }
22 : // 外側の Frame に直接 paint するのではなく, お絵書き領域を作る.
23 : class MyCanvas extends Canvas implements KeyListener, MouseListener, MouseMotionListener{
24 :     // Line の配列を保持する Vector クラスの変数 lineArray の宣言
25 :    Vector< Line> lineArray;
26 :     // マウスをドラッグ中かどうかを示す boolean 型(真偽値)の変数 dragging の宣言
27 :    boolean dragging;
28 :     // 表示する色を保持する変数
29 :    Color lineColor;
30 :     // コンストラクタの宣言
31 :    public MyCanvas(){
32 :       super();
33 :       lineArray=new Vector<Line>();
34 :        // ドラッグ中ではない
35 :       dragging=false;
36 :        // 線の色は黒に
37 :       lineColor=Color.black;
38 :          // GUI 部品と, Event Listener を関連づける
39 :       setSize(600,400);
40 :       addKeyListener(this);
41 :       addMouseListener(this);
42 :       addMouseMotionListener(this);
43 :    }
44 :    Image offScreenImage;
45 :    Graphics offScreenGraphics;
46 :    public void update(Graphics g){
47 :       if(offScreenImage==null){
48 :          offScreenImage=createImage(600,400); // オフスクリーンイメージを 600x400 のサイズで作成
49 :          offScreenGraphics=offScreenImage.getGraphics(); // オフスクリーンイメージに描画するための Graphics オブジェクト
50 :       }
51 :       paint(offScreenGraphics); // 次の画面のイメージを作る.
52 :       g.drawImage(offScreenImage,0,0,this); // イメージを本物のスクリーンに書き込む
```

```
 53 :    }
 54 :     // offScreenImage の書き直しをする際に呼ばれる
 55 :    public void paint(Graphics g){
 56 :      int i;
 57 :        // 白で(0,0)-(600,400)を塗り潰す
 58 :      g.setColor(Color.white);
 59 :      g.fillRect(0,0,600,400);
 60 :        // 色を設定
 61 :      g.setColor(lineColor);
 62 :      int size=lineArray.size();
 63 :      if(dragging) size--;
 64 :      for(i=0;i< size;i++){
 65 :        Line l=(Line)lineArray.elementAt(i);
 66 :        g.drawLine(l.start_x,l.start_y,l.end_x,l.end_y);
 67 :      }
 68 :       // マウスをドラッグ中の時は
 69 :      if(dragging){
 70 :          // 赤い色で
 71 :        g.setColor(Color.red);
 72 :         // lines[lineCount] を描画する.
 73 :        Line l=(Line)lineArray.elementAt(i);
 74 :        g.drawLine(l.start_x,l.start_y,l.end_x,l.end_y);
 75 :      }
 76 :    }
 77 :     // KeyListener を実装するためのメソッド
 78 :    public void keyPressed(KeyEvent e){
 79 :        // イベントからキーのコードを取り出す
 80 :      int key=e.getKeyChar();
 81 :       // デバッグ用の表示
 82 :      System.out.println("keyPressed("+e+","+key+")");
 83 :       // 入力が 'q'の時は終了する
 84 :      if(key=='q') System.exit(0);
 85 :    }
 86 :     // 要らないイベントに対応するメソッドも中身は空で書いておく必要がある.
 87 :    public void keyReleased(KeyEvent e){}
 88 :    public void keyTyped(KeyEvent e){}
 89 :     // MouseListener を実装するためのメソッド
 90 :    public void mousePressed(MouseEvent e){
 91 :        // 押された時のマウスカーソルの位置を得る
 92 :      int mx=e.getX(),my=e.getY();
 93 :       // デバッグ用の表示
 94 :      System.out.println("mousePressed("+e+","+mx+","+my+")");
 95 :       // 配列 lines の lineCount 番目に線分を登録
 96 :      lineArray.addElement(new Line(mx,my,mx,my));
 97 :       // ドラッグ中であることを示す
 98 :      dragging=true;
 99 :       // 再表示をおこなう
100 :      repaint();
101 :    }
102 :     // マウスのボタンが離された時のイベント
103 :    public void mouseReleased(MouseEvent e){
104 :        // マウスカーソルの位置を得る
105 :      int mx=e.getX(),my=e.getY();
106 :       // デバッグ用の表示
107 :      System.out.println("mouseUp("+e+","+mx+","+my+")");
108 :       // 配列 lines の lineCount 番目の始点を変更
109 :      Line l=(Line)lineArray.elementAt(lineArray.size()-1);
110 :      l.end_x=mx;
111 :      l.end_y=my;
112 :      dragging=false;
113 :       // 再表示をおこなう
114 :      repaint();
115 :    }
116 :    public void mouseClicked(MouseEvent e){}
117 :    public void mouseEntered(MouseEvent e){}
118 :    public void mouseExited(MouseEvent e){}
119 :     // MouseMotionListener を実装するためのメソッド
120 :    public void mouseDragged(MouseEvent e){
121 :        // マウスカーソルの位置を得る
122 :      int mx=e.getX(),my=e.getY();
123 :       // デバッグ用の表示
124 :      System.out.println("mouseDrag("+e+","+mx+","+my+")");
125 :       // 配列 lines の lineCount 番目の始点を変更
126 :      Line l=(Line)lineArray.elementAt(lineArray.size()-1);
127 :      l.end_x=mx;
```

```
128 :       l.end_y=my;
129 :        // 再表示をおこなう
130 :       repaint();
131 :    }
132 :    public void mouseMoved(MouseEvent e){}
133 :
134 :    // Delete ボタンが押された時の処理
135 :    public void deleteLine(){
136 :       int size;
137 :       if((size=lineArray.size())>0){
138 :          lineArray.removeElementAt(size-1);
139 :          repaint();
140 :       }
141 :    }
142 :    // Clear ボタンが押された時の処理
143 :    public void clearLine(){
144 :       lineArray.removeAllElements();
145 :       repaint();
146 :    }
147 : }
148 :
149 : class CanvasTest extends Frame implements ActionListener{
150 :    MenuBar menuBar;
151 :    Menu fileMenu,editMenu;
152 :    MyCanvas myCanvas;
153 :    Button deleteButton,clearButton;
154 :    CanvasTest(){
155 :       super("CanvasTest");
156 :       // メニューバーを作成する
157 :       menuBar=new MenuBar();
158 :       // File というメニュー
159 :       fileMenu=new Menu("File");
160 :       fileMenu.add("Exit");
161 :       // メニューが選択された時に, CanvasTest がイベントを受け取る
162 :       fileMenu.addActionListener(this);
163 :       // メニューバーに fileMenu を加える.
164 :       menuBar.add(fileMenu);
165 :       editMenu=new Menu("Edit");
166 :       editMenu.add("Clear");
167 :       editMenu.add("Delete");
168 :       editMenu.addActionListener(this);
169 :       menuBar.add(editMenu);
170 :       // Frame にメニューバーを付け加える.
171 :       setMenuBar(menuBar);
172 :       // パネル(ボタンなどを配置するための入れ物)部分を作成する.
173 :       Panel panel=new Panel();
174 :       panel.setLayout(new FlowLayout());
175 :       panel.add(deleteButton=new Button("Delete"));
176 :       deleteButton.addActionListener(this);
177 :       panel.add(clearButton=new Button("Clear"));
178 :       clearButton.addActionListener(this);
179 :       // BorderLayout を用いる.
180 :       setLayout(new BorderLayout());
181 :       // 上部に Panel
182 :       add(panel,"North");
183 :       // 下部に MyCanbas を配置する.
184 :       add(myCanvas=new MyCanvas(),"South");
185 :       // キー入力が button に横取りされないようにするため
186 :       deleteButton.addKeyListener(myCanvas);
187 :       clearButton.addKeyListener(myCanvas);
188 :       // 部品をおさめるのに適当と思われるサイズにする.
189 :       setSize(getPreferredSize());
190 :       setVisible(true);
191 :    }
192 :    public void actionPerformed(ActionEvent e){
193 :       System.out.println(e);
194 :       Object source=e.getSource();
195 :       // Delete ボタンが押された時
196 :       if(source.equals(deleteButton)){
197 :          myCanvas.deleteLine();
198 :       }
199 :       // Clear ボタンが押された時
200 :       else if(source.equals(clearButton)){
201 :          myCanvas.clearLine();
202 :       }
203 :       // [Edit]メニューのどれかが選択された時
204 :       else if(source.equals(editMenu)){
```

```
205 :          String command=e.getActionCommand();
206 :          // Delete メニューが選択された時
207 :          if(command.equals("Delete")){
208 :              myCanvas.deleteLine();
209 :          }
210 :          // Clear メニューが選択された時
211 :          else if(command.equals("Clear")){
212 :              myCanvas.clearLine();
213 :          }
214 :       }
215 :       // [File]メニューのどれかが選択された時
216 :       else if(source.equals(fileMenu)){
217 :          String command=e.getActionCommand();
218 :          // Exit メニューが選択された時
219 :          if(command.equals("Exit")){
220 :              System.exit(0);
221 :          }
222 :       }
223 :    }
224 :    public static void main(String args[]) {
225 :        // CanvasTest のインスタンスを作成 frame に代入
226 :       CanvasTest frame=new CanvasTest();
227 :    }
228 : }
```

出来だけ短いプログラムなるように作成せよ。
プログラムの終了の方法は、各自で工夫せよ。
また、出来上がったプログラムは全員の前で発表するとともに、以下の内容でレポートを作成し手逸出する。

① GUI プログラムの基本構文
② プログラムリスト（行番号付き）
③ プログラム概要説明
④ プログラム終了方法
⑤ 考察（工夫した点など..）
⑥ 感想

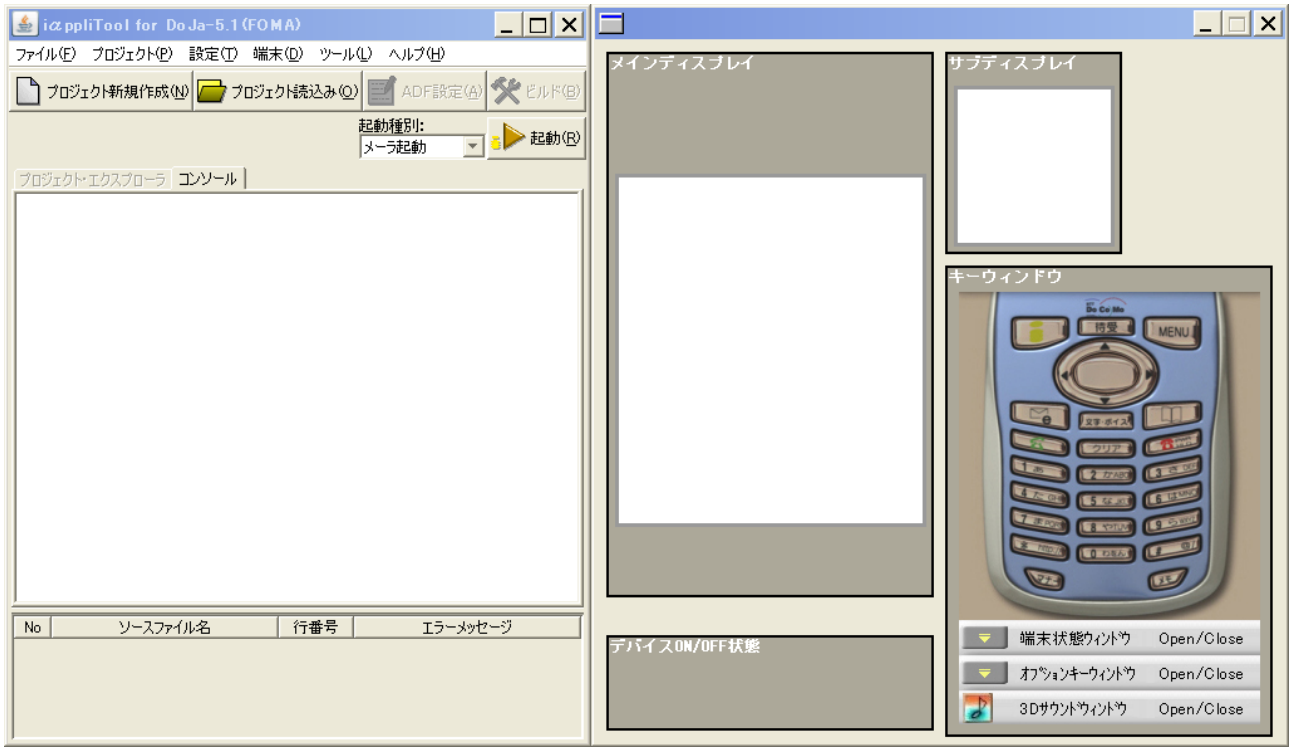
20. i アプリの開発手順

（1） 開発環境

i アプリは、2001 年、NTT ドコモの携帯電話 503i シリーズから搭載された機能で、Java で作ったプログラムが携帯電話上で実行できる機能である。
プログラムを開発するために、NTT ドコモから提供された開発環境が Doja である。新しい携帯電話のシリーズの登場とともにバージョンアップが行われ、現在のバージョンは 5.1 である。

（2） Doja の立ち上げ

スタートメニューから Doja を立ち上げると、以下のような画面が出る。
左側が開発をいていくための操作をするウインドウで、右側がプログラムを動作をシミュレーションするウインドウである。



（3） プロジェクトの新規作成

i アプリは、「プロジェクト」という単位で作成する。実質的にプロジェクトは 1 つのフォルダで、その中に必要なプログラムやファイルを管理する。

① プロジェクト新規作成

前述のようにプロジェクトは 1 つのフォルダで、「どこに」「どういう名前で」作成するかを指定する。このとき、「引数のソースファイルを作成する」にチェックマークを入れておくと、プロジェクトを作成するのと同時に、ソースファイルを自動的に生成してくれる。

② プロジェクトの詳細設定

プロジェクトを生成すると、次にプロジェクトの詳細を設定する。

i） テンプレートを利用する

この項目にチェックを入れると「作成」ボタンをクリックしたあと、自動的に基本構文の書かれたソースプログラムがエディタにより立ち上がる。

ii） アプリケーションの種別

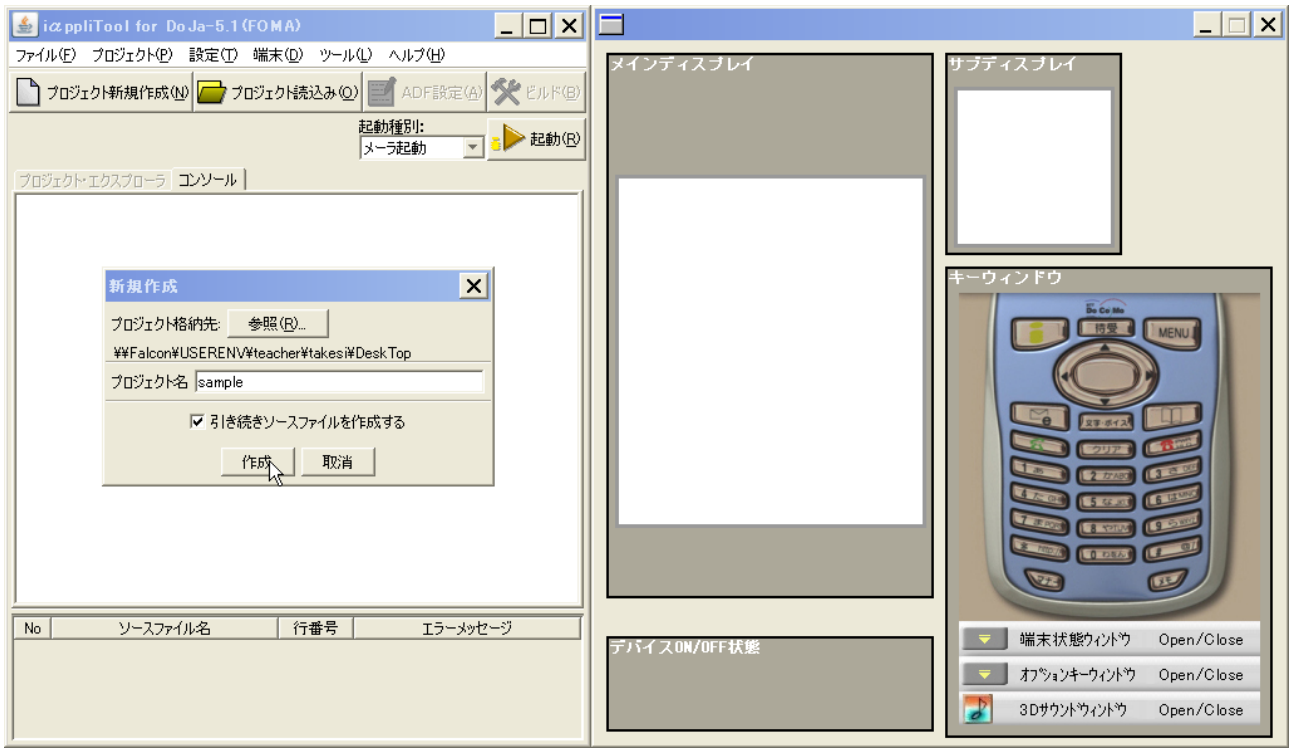
「通常のアプリケーション」「待ち受けアプリケーション」「なし」の中から、自分がこれから作成するプログラムにを選択する。これにあわせて基本構文が作成される。

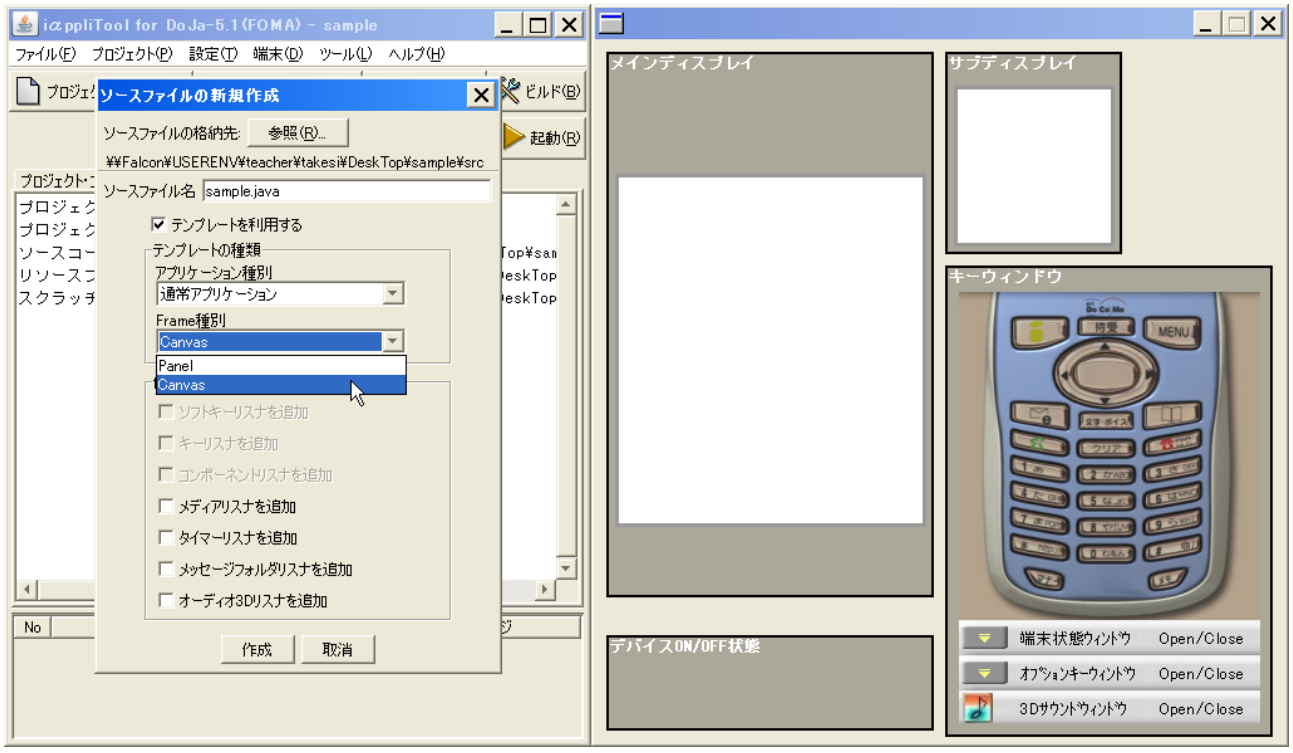
iii） Frame の種別

「Panel」「Canvas」からどちらか選択する。

Panel は、ボタンやフィールドなどの UI（ユーザーインターフェイス）部品を用いてアプリケーションを作成するプログラムで、Canvas はそういった部品を使わず自分で絵を描くようにしてアプリケーションを作成する場合に設定する。一般的に、Canvas の方を低レベルのアプリケーションと呼ばれている。

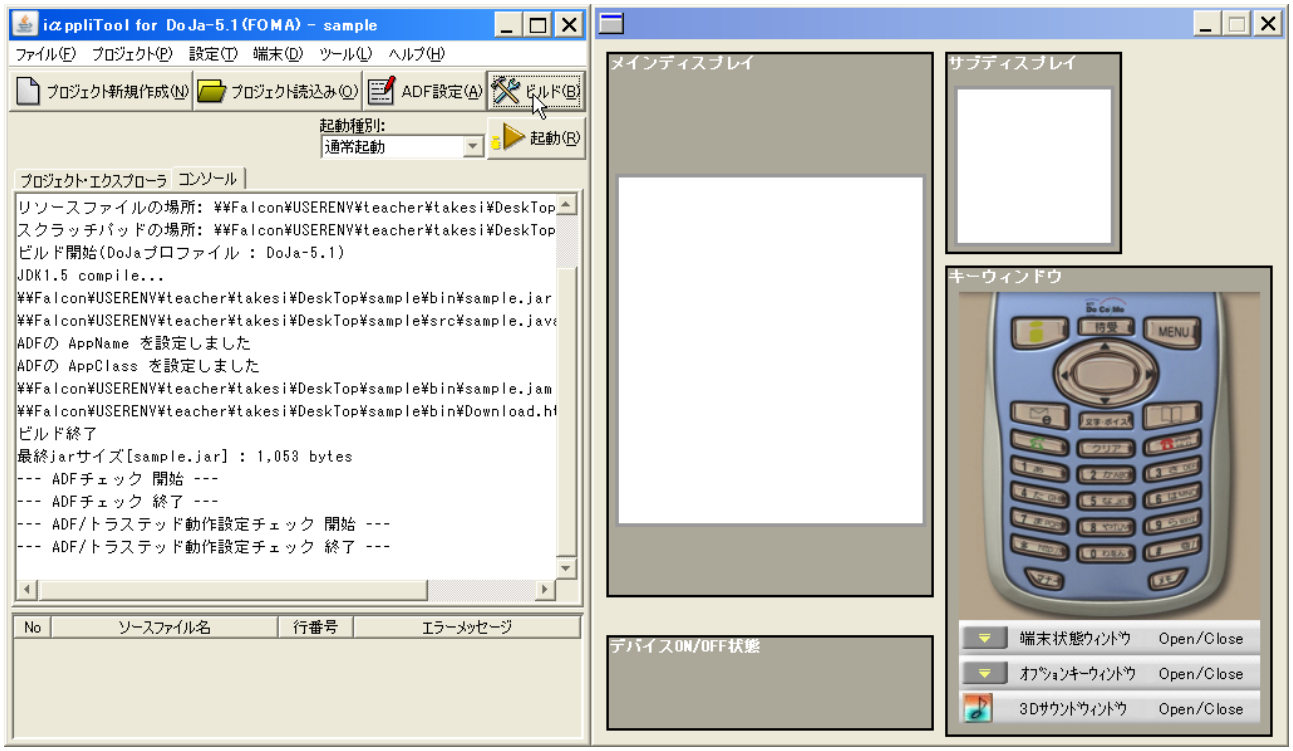
③ エディタが立ち上がりプログラムを作成する





（4） ビルド

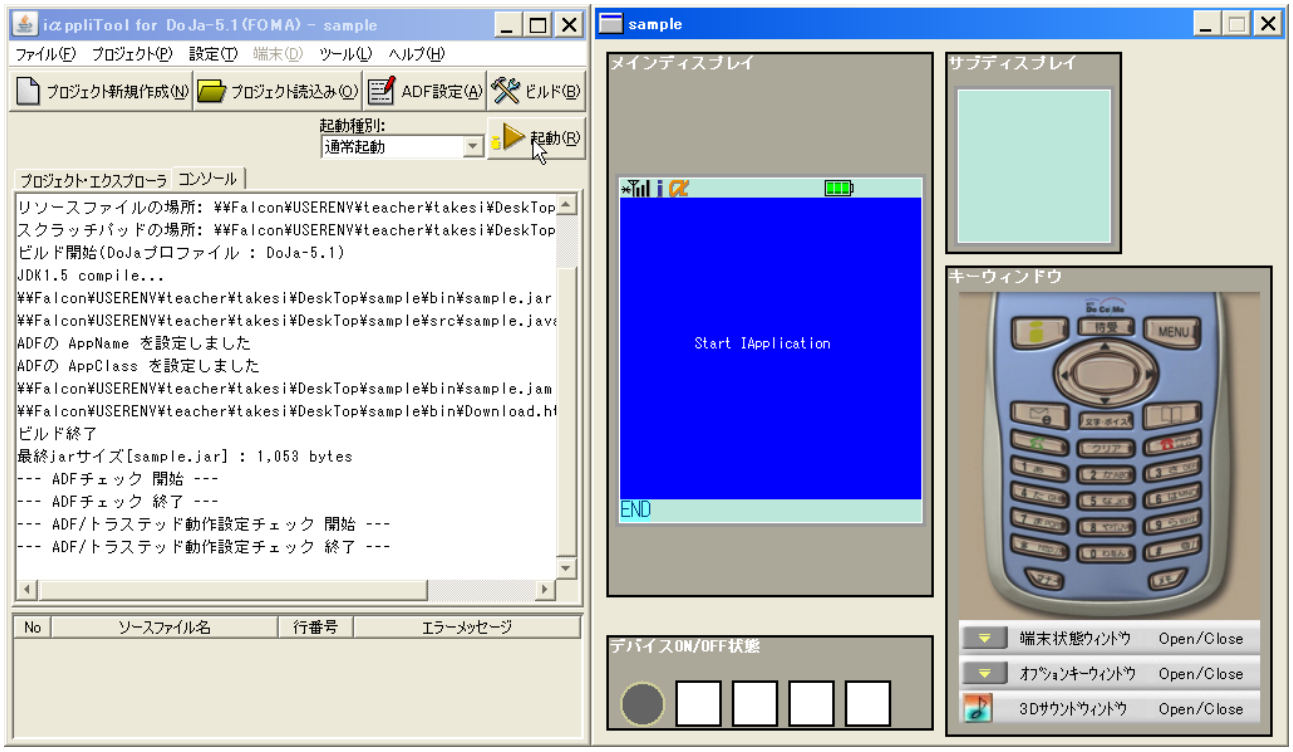
できあがったプログラムは、コンパイルしなくてはなりません。Doja では「ビルド」ボタンをクリックすることで、プロジェクトで用いられるプログラムのコンパイルをしてくれます。



エラーがなければ、上記のようなメッセージが出ます。
エラーがある場合は必要なソースファイルを修正し保存後、再びビルドしてください。

（5） 起動

エラーがなくできあがれば、「起動」ボタンをクリックすると、コンピュータ上でシミュレーションし、動作を確認します。



（6） プロジェクトの構成
プロジェクトは以下のようなフォルダやファイルで構成されています。

① binフォルダ
コンパイルされできあがった中間コードとダウンロードするための HTML ファイル。
- sample.jam
- sample.jar
- Download.html

② resフォルダ
プログラムで用いる絵のファイルなど「資源」を保管しておくフォルダ。

③ spフォルダ
スクラッチパッド。ゲームなどのハイスコアを保管しておく場所。ハイスコアはアプリケーションを終了しても、データを保管してある。

④ srcフォルダ
ソースプログラムを保管しておくフォルダ。

⑤ tempフォルダ

⑥ sampleファイル

⑦ sample.propaertiesファイル

（7） 携帯電話への実装
できあがったプログラムを携帯電話に実装するには、インターネットを介して行います。
インターネット上の Web サーバに bin フォルダの内容をアップロードして、携帯電話から Download.html にアクセスする。

```
http:// …… /Download.html
```

21. 基本構文
i アプリは以下のような基本構文によって構成されている。プログラムが最初に実行されるメソッドが、main ではなく i アプリでは start であるので注意。

```
public class sample extends IApplication {
  public void start() {
    Display.setCurrent((Frame)(new MainCanvas()));
  }
}
class MainCanvas extends Canvas {
  MainCanvas() {
    初期化
  }
  public void paint(Graphics g) {
    画面の描画する内容（再描画も）
  }
  public void processEvent(int type, int param) {
    イベント処理
  }
}
```

（1） 解説

以下は、テンプレートとして自動的に生成されたiアプリのプログラムである。

```
 1 :     /*
 2 :      * sample.java
 3 :      *
 4 :      * DATE : 2008/09/17 12:55
 5 :      */
 6 :     import com.nttdocomo.ui.IApplication;
 7 :     import com.nttdocomo.ui.Canvas;
 8 :     import com.nttdocomo.ui.Graphics;
 9 :     import com.nttdocomo.ui.Frame;
10 :     import com.nttdocomo.ui.Display;
```

6行目から10行目

このプログラムで利用されるクラスをインポート。

```
import com.nttdocomo.ui.*;
```

とまとめて書くことの方が多い。

```
11 :
12 :
13 :     /**
14 :      * sample
15 :      *
16 :      * @author NAME
17 :      */
18 :     public class sample extends IApplication {
19 :
20 :         public void start() {
21 :             Display.setCurrent((Frame)(new MainCanvas()));
22 :         }
23 :
24 :     }
25 :
```

18行目

このプログラムのクラス名の指定。iアプリで実行されるクラスは、Iapplicationクラスを継承して作成する。

20行目

iアプリで実行されるメソッドはmainではなく、「start」となる。

21行目

iアプリはCanvasクラスを継承したクラスをセットすることで実行される。

```
26 :     /**
27 :      * MainCanvas
28 :      *
29 :      */
30 :     class MainCanvas extends Canvas {
31 :
```

31 行目
実行されるクラス MainCanvas を Canvas クラスから継承する。

```
 32 :         MainCanvas() {
 33 :             setSoftLabel(SOFT_KEY_1, "END");
 34 :             setBackground(Graphics.getColorOfName(Graphics.BLUE));
 35 :
 36 :         }
 37 :
```

32 行目
コンストラクタ（クラス名=メソッド名）で、クラスの初期化。

33 行目
画面上に「END」のボタン表示をさせる。
左側　SOFT_KEY_1
右側　SOFT_KEY_2

34 行目
背景色の指定。
Graphics.getColorOfName(色名)…色名指定

色名		
	Graphics.BLACK	黒
	Graphics.BLUE	青
	Graphics.LIME	緑
	Graphics.AQUA	水色
	Graphics.RED	赤
	Graphics.FUCHSIA	紫
	Graphics.YELLOW	黄
	Graphics.WHITE	白
	Graphics.GRAY	灰色
	Graphics.NAVY	暗い青
	Graphics.GREEN	暗い緑
	Graphics.TEAL	暗い水色
	Graphics.MAROON	暗い赤
	Graphics.PURPLE	暗い紫
	Graphics.OLIVE	暗い黄
	Graphics.SILVER	銀

Graphics.getColorOfRGB(赤,緑,青)…光の３原色を数値(0～255)で指定

```
 38 :         public void paint(Graphics g) {
 39 :             g.lock();
 40 :             g.clearRect(0, 0, Display.getWidth(), Display.getHeight());
 41 :             g.setColor(Graphics.getColorOfName(Graphics.WHITE));
 42 :             g.drawString ("Start IApplication", Display.getWidth() / 4,
                                 Display.getHeight() / 2);
 43 :             g.unlock(true);
 44 :         }
 45 :
```

38 行目
画面を描画するメソッド paint をオーバーライド。

39 行目
ダブルバッファリング開始。

画面のちらつきを防止するために、何か描く命令を実行しても画面上に反映させないようにする。

40 行目
画面のクリア。
0,0 は始点の座標を示し、Display.getWidth(), Display.getHeight()は、クリアするサイズを示す。Display クラスからディスプレイ情報を得る。

```
Display.getWidth()    画面の幅（ピクセル）
Display.getHeight()   画面の高さ（ピクセル）
```

41 行目
（次に実行する文字列表示の）色の指定。

42 行目
文字列の表示。

```
"Start IApplication"      表示する文字列（””で囲んで）
       Display.getWidth() / 4    表示する座標（x 軸）
       Display.getHeight() / 2   表示する座標（y 軸）
```

43 行目
ダブルバッファリング解除。

Graphics クラスは、画面に文字や絵などを描画するためのメソッドを持つクラスです。
画面の描画ルーチンは初期化部分（コンストラクタ）で書かれることもあります。

```
 46 :         public void processEvent(int type, int param) {
 47 :             if (type == Display.KEY_RELEASED_EVENT) {
 48 :                 if (param == Display.KEY_SOFT1) {
 49 :                     (IApplication.getCurrentApp()).terminate();
 50 :                 }
 51 :             }
 52 :         }
 53 :
```

---

46 行目
イベント処理をするメソッド processEvent をオーバーライド。

47 行目
キーイベントの種類（タイプ）を判断。

```
Display.KEY_PRESSED_EVENT        キーを押した時
Display.KEY_RELEASED_EVENT       キーを離した時

TIMER_EXPIRED_EVENT              タイマーイベントの発生した時
```

※高レベル API の場合は、イベントリスナーを使用します。(後述)

48 行目
押されたたキーを判断。

```
Display.KEY_SOFT1       ソフトキー１（左）
Display.KEY_SOFT2       ソフトキー２（右）

Display.KEY_0           0 キー
Display.KEY_1           1 キー
Display.KEY_2           2 キー
Display.KEY_3           3 キー
```

```
Display.KEY_4          4 キー
Display.KEY_5          5 キー
Display.KEY_6          6 キー
Display.KEY_7          7 キー
Display.KEY_8          8 キー
Display.KEY_9          9 キー
Display.KEY_ASTERISK   *キー
Display.KEY_POUND      #キー
Display.KEY_SELECT     決定キー
Display.KEY_UP         上キー
Display.KEY_DOWN       下キー
Display.KEY_LEFT       左キー
Display.KEY_RIGHT      右キー
```

49 行目
プログラムの終了。

```
54 :     }
```

（2） 文字列表示関する補足事項

drawString で文字列を表示しましたが、Font クラスを用いると表示する文字列のサイズを変更することもできます。
paint メソッドを以下のように変更します。

```
public void paint(Graphics g) {

    Font font;

    g.lock();
    g.clearRect(0, 0, Display.getWidth(), Display.getHeight());
    g.setColor(Graphics.getColorOfName(Graphics.WHITE));

    font=Font.getFont(Font.SIZE_LARGE);
    g.setFont(font);

    g.drawString ("Start IApplication", Display.getWidth() / 4,
                                Display.getHeight() / 2);
    g.unlock(true);
  }
```

Font クラスの getFont メソッドを用いてフォント情報を font に入れ、その情報を Graphics クラスの setFont メソッドで設定してフォントのサイズを指定します。

```
Font.SIZE_LARGE    大    28 または 30 ドット（ピクセル）
Font.SIZE_MEDIUM   中    24 ドット（ピクセル）
Font.SIZE_SMALL    小    16 または 20 ドット（ピクセル）
Font.SIZE_TNIY     極小  12 ドット（ピクセル）
```

また、Font クラスでは文字列の幅（stringWidth("文字列")や文字列の高さ（getHeight("文字列")）を示すメソッドも用意されています。

（使用例）

```
g.drawString ("Start IApplication",
   (Display.getWidth()-font.stringWidth("Start IApplication "))/ 2,
   (Display.getHeight()-font.getHeight("Start IApplication ")) / 2
 );
```

22. 図を描く

（1） ピクセルの描画

void setPixel(x,y)
座標(x,y)に setColor()で設定した色のピクセルを描画する。

x	int 型	
y	int 型	

void setPixel(x,y,color)

座標(x,y)に指定した色のピクセルを描画する。

x	int 型	
y	int 型	
color	getColorOfRGB()メソッドなどで返される色を指定:int 型	

（2） ラインの描画

void drawLine(x0,y0,x1,y1)
始点(x0,y0)から終点(x1,y1)を結ぶ setColor()で設定した色の直線を描く。

x0	int 型
y0	int 型
x1	int 型
y1	int 型

（3） 矩形の描画

void drawRect(x,y,width,height)
setColor()で設定した色で、座標(x,y)を左上に、幅 width+1、高さ height+1 の矩形の輪廓を描く。

x	int 型
y	int 型
width	int 型
height	int 型

void fillRect(x,y,width,height)
setColor()で設定した色で、座標(x,y)を左上に、幅 width+1、高さ height+1 の矩形を描く。

x	int 型
y	int 型
width	int 型
height	int 型

（4） ポリラインの描画

void drawPolyline(xPoints,yPoints,nPoints)
setColor()で設定した色で、引数で指定された頂点を直線で結んで描画する。

xPoints	int[]型
yPoints	int[]型
nPoints	int 型

void fillPolygon(xPoints,yPoints,nPoints)
setColor()で設定した色で、引数で指定された頂点を結んで塗りつぶす。

xPoints	int[]型
yPoints	int[]型
nPoints	int 型

（5） 円弧の描画

void drawArc(x,y,width,height,startAngle,arcAngle)

setColor()で設定した色で、座標(x,y)を左上に、幅width+1、高さheight+1の矩形内に納まる円を、startAngle度からarcAngle度の円弧を描く。ただし、startAngleと arcAngleは省略できる。

x	int[]型
y	int[]型
width	int型
height	int型
startAngle	int型
endAngle	int型

void fillArc(x,y,width,height,startAngle,arcAngle)
setColor()で設定した色で、座標(x,y)を左上に、幅width+1、高さheight+1の矩形内に納まる円を、startAngle 度から arcAngle 度の円弧を塗りつぶして描く。ただし、startAngle と arcAngleは省略できる。

x	int[]型
y	int[]型
width	int型
height	int型
startAngle	int型
endAngle	int型

― プロジェクト：sample001 ―

```
/*
 * sample001.java
 *
 * DATE : 2008/09/29 14:27
 */
import com.nttdocomo.ui.IApplication;
import com.nttdocomo.ui.Canvas;
import com.nttdocomo.ui.Graphics;
import com.nttdocomo.ui.Frame;
import com.nttdocomo.ui.Display;


/**
 * sample001
 *
 * @author NAME
 */
public class sample001 extends IApplication {

  public void start() {
    Display.setCurrent((Frame)(new MainCanvas()));
  }

}

/**
 * MainCanvas
 *
 */
class MainCanvas extends Canvas {

  MainCanvas() {
    setSoftLabel(SOFT_KEY_1, "END");
    setBackground(Graphics.getColorOfName(Graphics.BLUE));
```

```
    }

    public void paint(Graphics g) {
      g.lock();

      //ピクセルの描画
      g.setColor(g.getColorOfRGB(255,0,0));
      g.setPixel(130,30);
      //ラインの描画
      g.setColor(g.getColorOfRGB(255,100,100));
      g.drawLine(180,30,220,30);
      //矩形の描画
      g.setColor(g.getColorOfRGB(0,255,0));
      g.drawRect(130,60,40,40);
      //矩形の塗り潰し
      g.setColor(g.getColorOfRGB(100,255,100));
      g.fillRect(180,60,40,40);
      //ポリラインの描画
      g.setColor(g.getColorOfRGB(0,0,255));
      int[] dx0={130,130,150,170,170};
      int[] dy0={150,120,110,120,150};
      g.drawPolyline(dx0,dy0,dx0.length);
      //ポリゴンの描画
      g.setColor(g.getColorOfRGB(100,100,255));
      int[] dx1={180,180,200,220,220};
      int[] dy1={150,120,110,120,150};
      g.fillPolygon(dx1,dy1,5);
      //円弧の描画
      g.setColor(g.getColorOfRGB(255,0,255));
      g.drawArc(130,160,40,40,0,270);
      //円の塗り潰し
      g.setColor(g.getColorOfRGB(255,100,255));
      g.fillArc(180,160,40,40,0,360);

      g.unlock(true);
    }

    public void processEvent(int type, int param) {
      if (type == Display.KEY_RELEASED_EVENT) {
        if (param == Display.KEY_SOFT1) {
          (IApplication.getCurrentApp()).terminate();
        }
      }
    }
  }
```

（6） イメージの描画

用意された画像のファイルを表示するには以下のような構文を用いて行う。
ただし、表示させたい絵は、プロジェクトフォルダ内の res フォルダに入れておくこと。

```
Image image=null;//Image 型の変数 image の定義

//イメージが空の時
if (image==null) {
  //画像ファイルの読み込み
  try {
    MediaImage m;
```

```
      m=MediaManager.getImage("resource:///〇〇〇〇.gif");
      m.use();
      image=m.getImage();
    } catch (Exception e) {
      System.out.println(e.getClass().getName());
    }
  }
```

```
  //イメージの描画 座標(0,0)を左上として絵のファイルを表示する。g は Graphics クラス。
  g.drawImage(image,0,0);
```

void drawImage(image,x,y)
座標(x,y)を左上にしてイメージを描画する。

Image	イメージオブジェクト:Image 型
x	int 型
y	int 型

void drawScaledImage(image,dx,dy,width,height,sx,sy,swidth,sheight)
座標(x,y)を左上にしてイメージを拡大縮小して描画する。

Image	イメージオブジェクト:Image 型
dx	描画先の矩形の X 座標：int 型
dy	描画先の矩形の Y 座標：int 型
width	描画先の矩形の幅：int 型
height	描画先の矩形の高さ：int 型
sx	描画元の矩形の X 座標：int 型
sy	描画元の矩形の Y 座標：int 型
swidth	描画元の矩形の幅：int 型
sheight	描画元の矩形の高さ：int 型

48×48 の画像を拡大縮小するには...

1/2 倍拡大

```
g.drawScaledImage(image,60,0,48/2,48/2,0,0,48,48);
```

2 倍拡大

```
g.drawScaledImage(image,120,0,48*2,48*2,0,0,48,48);
```

void setFlipMode(flipmode)
イメージ描画時に反転または回転して、座標(x,y)を左上にしてイメージを描画する。flipmode には、以下の定数を入れることができる。

Graphics,FLIP_NONE	反転なし
Graphics,FLIP_HORIZONTAL	左右反転
Graphics,FLIP_VERTICAL	上下反転
Graphics,FLIP_ROTATE_RIGHT	時計回りに 90 度回転
Graphics,FLIP_ROTATE	時計回りに 180 度回転
Graphics,FLIP_ROTATE_LEFT	反時計回りに 90 度回転

― プロジェクト：sample002 ―

```
/*
 * sample002.java
 *
 * DATE : 2008/09/29 14:53
 */
import com.nttdocomo.ui.*;
```

```
/**
 * sample002
 *
 * @author NAME
 */
public class sample002 extends IApplication {

  public void start() {
    Display.setCurrent((Frame)(new MainCanvas()));
  }

}

/**
 * MainCanvas
 *
 */
class MainCanvas extends Canvas {
  Image image=null;//Image 型の変数 image の定義

  MainCanvas() {
    setSoftLabel(SOFT_KEY_1, "END");
    setBackground(Graphics.getColorOfName(Graphics.BLUE));

  }

  public void paint(Graphics g) {
    g.lock();
    if (image==null) {
      //画像ファイルの読み込み
      try {
        MediaImage m;
        m=MediaManager.getImage("resource:///g.gif");
        m.use();
        image=m.getImage();
      } catch (Exception e) {
        System.out.println(e.getClass().getName());
      }
    }

    //イメージの描画
    g.drawImage(image,0,0);

    //1/2 倍拡大
    g.drawScaledImage(image,60,0,48/2,48/2,0,0,48,48);

    //2 倍拡大
    g.drawScaledImage(image,120,0,48*2,48*2,0,0,48,48);

    //左右反転
    g.setFlipMode(Graphics.FLIP_HORIZONTAL);
    g.drawImage(image,0,120);

    //上下反転
    g.setFlipMode(Graphics.FLIP_VERTICAL);
    g.drawImage(image,60,120);
```

```
    //反転なし
    g.setFlipMode(Graphics.FLIP_NONE);
    g.drawImage(image,0,180);

    //時計回りに 90 度回転
    g.setFlipMode(Graphics.FLIP_ROTATE_RIGHT);
    g.drawImage(image,60,180);

    //時計回りに 180 度回転
    g.setFlipMode(Graphics.FLIP_ROTATE);
    g.drawImage(image,120,180);

    //時計回りに 270 度回転
    g.setFlipMode(Graphics.FLIP_ROTATE_LEFT);
    g.drawImage(image,180,180);

    g.unlock(true);
  }

  public void processEvent(int type, int param) {
    if (type == Display.KEY_RELEASED_EVENT) {
      if (param == Display.KEY_SOFT1) {
        (IApplication.getCurrentApp()).terminate();
      }
    }
  }
}
```

23. イベントリスナーを用いたイベント処理

新規にプロジェクトを作成したとき、時間などのイベントはイベントリスナーとしてテンプレート上に書かれる。このテンプレートを利用して、簡単なタイマープログラムを作成する。

― プロジェクト：sample003 ―

```
/*
 * sample003.java
 *
 * DATE : 2008/09/30 09:57
 */
import com.nttdocomo.ui.IApplication;
import com.nttdocomo.ui.Canvas;
import com.nttdocomo.ui.Graphics;
import com.nttdocomo.ui.Frame;
import com.nttdocomo.ui.Display;

import com.nttdocomo.util.Timer;
import com.nttdocomo.util.TimerListener;

/**
 * sample003
 *
 * @author NAME
 */
public class sample003 extends IApplication {

  public void start() {
    Display.setCurrent((Frame)(new MainCanvas()));
  }
```

```
}

/**
 * MainCanvas
 *
 */
class MainCanvas extends Canvas implements TimerListener{

  Timer tm = new Timer();// (1)
  int count=0;// (2)

  MainCanvas() {
    setSoftLabel(SOFT_KEY_1, "END");
    setBackground(Graphics.getColorOfName(Graphics.BLUE));

    tm.setTime(100);// (3)
    tm.setRepeat(true);
    tm.setListener((TimerListener)this);

    tm.start();// (4)
    }

  public void paint(Graphics g) {
    String s="" +(count/10)+ "." +(count%10)+ "秒経過";// (5)

    g.lock();
    g.clearRect(0, 0, Display.getWidth(), Display.getHeight());
    g.setColor(Graphics.getColorOfName(Graphics.WHITE));
    g.drawString(s, Display.getWidth() / 4, Display.getHeight() / 2);// (6)
    g.unlock(true);
  }

  public void processEvent(int type, int param) {
    if (type == Display.KEY_RELEASED_EVENT) {
      if (param == Display.KEY_SOFT1) {
        (IApplication.getCurrentApp()).terminate();
      }
    }
  }

  public void timerExpired(Timer source) {
    count++;// (7)
    repaint();// (8)
  }

}
```

(1) Timer クラスの tm をコンストラクタ外で利用するので、宣言を外に出す。
(2) イベント発生ごとにカウントアップする変数。
(3) イベント発生時間を 0.1 秒とする。
(4) タイマーイベントスタート
(5) 経過時間を表示する文字列
(6) 経過時間を表示
(7) イベント発生ごとにカウントアップ
(8) 再描画

24. デジタル時計を作る

タイマーイベント利用し、デジタル時計を作成する。
日付や時刻を扱うのクラス Calendar と 現在の時間を扱うクラス Date をインポートしておかなければならない。

```
import java.util.Calendar;
import java.util.Date;
```

以下のような手順で、現在時間を取得する。
まず、日付や時間のデータ（例：my_calendar）は Calendar クラスで扱うので以下のようにインスタンス化したいのですが、

```
Calendar my_calendar= new Calendar();
```

抽象クラスである Calendar クラスはインスタンス化はできません。そこで次のように宣言したまま利用します。

```
Calendar my_calendar;
```

まず、宣言された my_calendar のタイムゾーンを、getInstance()メソッドで設定します。

```
my_calendar=Calendar.getInstance();
```

このとき getInstance()よって、指定されたタイムゾーンまたはデフォルト(引数の指定なし)のロケールのタイムゾーンに基づいた時刻を表していきます。
タイムゾーンが設定したあとで、my_calendar に時間の設定をします。時間の設定には Date クラス(オブジェクト)を利用します。

Date( )	現在の日時を保持した Date オブジェクトを生成します。
Date(long の値)	引数に 1970 年 1 月 1 日午前 0 時からの経過時間をミリ秒で指定し、指定したミリ秒が表す日時を保持した Date オブジェクトを生成します。

my_calendar に現在の時間の設定するには、以下のように行います。

```
my_calendar.setTime(new Date());
```

これで、時間のデータが my_calendar に入りました。あとは my_calendar 内の get()メソッドを使って、日付や時間のデータを取り出します。

```
my_calendar.get(Calendar.○○○)
```

このとき、○○○は Calendar クラス内の定数を意味し、以下に示すような定数が用意されている。

定数

AM_PM	フィールド値で、HOUR が正午より前であるか後であるかを示します。
DATE	フィールド値で、月の日を示します。
DAY_OF_MONTH	フィールド値で、月の日を示します。
DAY_OF_WEEK	フィールド値で、曜日を示します。
DAY_OF_WEEK_IN_MONTH	フィールド値で、現在の月の何度目の曜日かを示します。
DAY_OF_YEAR	フィールド値で、現在の年の何日目かを示します。
DST_OFFSET	フィールド値で、夏時間のオフセットをミリ秒単位で示します。
ERA	フィールド値で、ユリウス暦の AD または BC などの年代を示します。
FIELD_COUNT	get および set によって識別される重複しないフィールドの数を示します。
HOUR	フィールド値で、午前または午後の何時かを示します。
HOUR_OF_DAY	フィールド値で、時刻を示します。

MILLISECOND	フィールド値で、ミリ秒を示します。
MINUTE	フィールド値で、分を示します。
MONTH	月を示すフィールド値です。
SECOND	フィールド値で、秒を示します。
WEEK_OF_MONTH	フィールド値で、現在の月の何週目かを示します。
WEEK_OF_YEAR	フィールド値で、現在の年の何週目かを示します。
YEAR	年を示すフィールド値です。
ZONE_OFFSET	フィールド値で、GMT から直接計算したオフセットをミリ秒単位で示します。

たとえば、現在の年を取り出したいのなら、

```
my_calendar.get(Calendar.YEAR)
```

― プロジェクト：sample004 ―

```
/*
 * sample004.java
 *
 * DATE : 2008/09/30 09:57
 */
import com.nttdocomo.ui.IApplication;
import com.nttdocomo.ui.Canvas;
import com.nttdocomo.ui.Graphics;
import com.nttdocomo.ui.Frame;
import com.nttdocomo.ui.Display;

import com.nttdocomo.util.Timer;
import com.nttdocomo.util.TimerListener;

import java.util.Calendar;
import java.util.Date;


/**
 * sample004
 *
 * @author NAME
 */
public class sample004 extends IApplication {

  public void start() {
    Display.setCurrent((Frame)(new MainCanvas()));
  }
}

/**
 * MainCanvas
 *
 */
class MainCanvas extends Canvas implements TimerListener{

  Timer tm = new Timer();

  MainCanvas() {
    setSoftLabel(SOFT_KEY_1, "END");
    setBackground(Graphics.getColorOfName(Graphics.WHITE));
```

```
    tm.setTime(100);
    tm.setRepeat(true);
    tm.setListener((TimerListener)this);

    tm.start();
    }

  public void paint(Graphics g) {
    Calendar my_calendar;
    String my_date;
    int date_value;

    g.lock();

    g.clearRect(0, 0, Display.getWidth(), Display.getHeight());
    g.setColor(Graphics.getColorOfName(Graphics.BLACK));

    my_calendar=Calendar.getInstance();
    my_calendar.setTime(new Date());
    my_date= my_calendar.get(Calendar.YEAR) +"/";
    date_value = my_calendar.get(Calendar.MONTH) + 1;
    my_date=my_date + date_value + "/";
    my_date=my_date + my_calendar.get(Calendar.DATE)+" ";
    my_date=my_date + my_calendar.get(Calendar.HOUR_OF_DAY)+":";
    my_date=my_date + my_calendar.get(Calendar.MINUTE)+":";
    my_date=my_date + my_calendar.get(Calendar.SECOND);

    g.drawString(my_date, Display.getWidth() / 4, Display.getHeight() / 2);

    g.unlock(true);
  }

  public void processEvent(int type, int param) {
    if (type == Display.KEY_RELEASED_EVENT) {
      if (param == Display.KEY_SOFT1) {
        (IApplication.getCurrentApp()).terminate();
      }
    }
  }

  public void timerExpired(Timer source) {
    repaint();
  }
}
```

25. 音を出す

音楽などの音を出すプログラムを作成します。
新規プロジェクトでプロジェクトを作成する際に、「メディアリスナー」にチェックマークを入れておきましょう。ただし、音を出すだけなら、イベント処理の必要はないのでメディアリスナーは必要ないのですが、音を出すためのプログラムがテンプレートにあるので、これを利用します。

```
/*
 * sample005.java
 *
 * DATE : 2008/10/06 11:42
 */
import com.nttdocomo.ui.IApplication;
import com.nttdocomo.ui.Canvas;
import com.nttdocomo.ui.Graphics;
import com.nttdocomo.ui.Frame;
import com.nttdocomo.ui.Display;

import com.nttdocomo.ui.AudioPresenter;
import com.nttdocomo.ui.MediaListener;
import com.nttdocomo.ui.MediaPresenter;
import com.nttdocomo.ui.MediaSound;// (1)
import com.nttdocomo.ui.MediaManager;// (2)
/**
 * sample005
 *
 * @author NAME
 */

public class sample005 extends IApplication {

    public void start() {
        Display.setCurrent((Frame)(new MainCanvas()));
    }
}
/**
 * MainCanvas
 *
 */
class MainCanvas extends Canvas implements MediaListener{
    AudioPresenter ap = AudioPresenter.getAudioPresenter();// (3)

    MainCanvas() {
        setSoftLabel(SOFT_KEY_1, "END");
        setBackground(Graphics.getColorOfName(Graphics.BLUE));

        ap.setMediaListener((MediaListener)this);

        /* The example of a resource set */  // (4)
        MediaSound ms = MediaManager.getSound("resource:///sound.mld");
        try {
          ms.use();
        } catch (Exception ce) {  // (5)
        System.out.println("Exception:" + ce.toString());
        IApplication.getCurrentApp().terminate();
        }
        ap.setSound(ms);
        ap.play();
    }
```

```
    public void paint(Graphics g) {
        g.lock();
        g.clearRect(0, 0, Display.getWidth(), Display.getHeight());
        g.setColor(Graphics.getColorOfName(Graphics.WHITE));
        g.drawString("Start IApplication",
                             Display.getWidth() / 4, Display.getHeight() / 2);
        g.unlock(true);
    }

    public void processEvent(int type, int param) {
        if (type == Display.KEY_RELEASED_EVENT) {
            if (param == Display.KEY_SOFT1) {
                (IApplication.getCurrentApp()).terminate();
            }
        }
    }

    public void mediaAction(MediaPresenter source, int type, int param) {
        switch (type) {
            case AudioPresenter.AUDIO_PLAYING:
                break;
            case AudioPresenter.AUDIO_COMPLETE:
                ap.play();//(6)
                break;
            case AudioPresenter.AUDIO_PAUSED:
                break;
            case AudioPresenter.AUDIO_STOPPED:
                break;
            case AudioPresenter.AUDIO_RESTARTED:
                break;
            default:
                break;
        }
    }

}
```

（1）新規に利用するクラスの追加。
（2）新規に利用するクラスの追加。
（3）コンストラクタ（MainCanvas()）内にあるクラスの宣言を、コンストラクタ外でも利用したいので外に出す。
（4）テンプレートではコメントアウトになっているが、この部分が音を再生するための部分なので「//」を削除する。
（5）例外処理でエラーの内容が ConnectionException だとうまくいかないので、エラーの範囲を広げて Exception とする。
（6）イベント処理として、再生が終了したら再び再生させる。

このプログラムで利用している sound.mld というファイルは、音のデータが納められた MLD 形式（i メロディ形式）のファイルです。このファイルは、着メロコンバータという種類のソフトで、MIDI 形式や WAVE 形式のサウンドファイルを変換して利用します。MIDI ファイルなどはインターネット上にたくさんあるので自分の好きな曲をダウンロードするとよいでしょう。