学生証番号：＿＿＿＿＿＿＿＿＿＿＿＿　名前：＿＿＿＿＿＿＿＿＿＿＿＿

2016 年度情報コミュニケーション学実験 II「マルチメディア」テキスト

立命館大学 情報理工学部 情報コミュニケーション学科

2016 年度後期

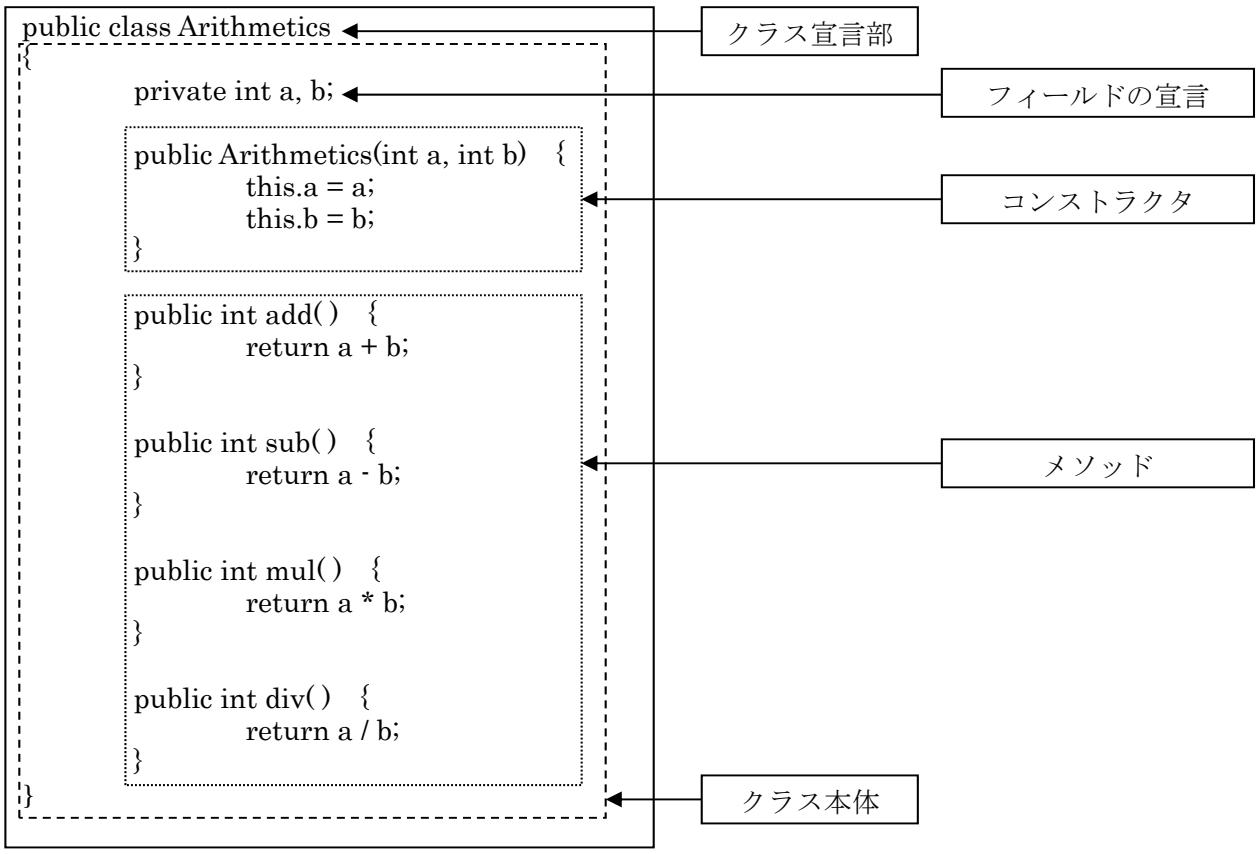
1. 実験の目的

本実験では、プログラミング言語 Java を用いて、マルチメディア・アプリケーションを開発する。（本テーマのホームページは http://www.aislab.org/ にあり、例などはホームページからダウンロードできる。）本実験の主な目的は Java のマルチメディア関連 API を体験することと複数のメディアを用いるマルチメディア・アプリケーションを自らの力で開発することである。実験を行うにあたり必要な API の情報は以下のドキュメントを参考する。

Java Platform Standard Edition 6.0, 7.0, 8.0 の AP 仕様：

http://docs.oracle.com/javase/jp/6/api/
http://docs.oracle.com/javase/jp/7/api/
https://docs.oracle.com/javase/jp/8/docs/api/

2. Java の復習



クラス(class)とはデータ(属性)とそれに対する処理をひとまとめにして定義した抽象データ型である。Java プログラムは一つ以上のクラスで構成され、属性をフィールド(field)、処理をメソッド(method)と呼ぶ。実行時には、クラスからオブジェクトを生成し、生成したオブジェクトが動作することで処理を行う。生成したオブジェクトのことを、クラスに対してインスタンス(instance)と呼ぶ。つまり、クラスとはインスタンスを生成する雛形のようなものである。

2.1 クラスライブラリ

Java でプログラミングを行う際には次のような手順でクラスを使ってプログラムを作成する。

① クラスを設計するコードを書く
　→ クラスを宣言する
② クラスを利用するコードを書く
　→ オブジェクトを作成してインスタンス変数・インスタンスメソッドを使う
　　あるいは
　→ クラス変数・クラスメソッドを使う

しかし、いつもプログラムを作成するため①、②の手順を経て自分が使うコードをすべて自分で書くのは大変無駄が多い作業になる。たとえば、だれかが「車」に関するクラスを設計していたとするならば、車を管理するプログラムを作成する人は①を省いて、②の手順から作成することが可能となる。

Java の標準的な開発環境である JDK には、よく使われる機能をまとめたクラスライブラリ(class library)と呼ばれるクラスの集まりが添付されている。よく使われる多くのクラスが用意されているため、①の手順を省いて②の手順からコードを記述することができる。クラスライブラリは Java では API(Application Program Interface)とも呼ばれている。

2.2 継承

クラスやインスタンスを拡張する方法として継承がある。継承(inheritance)とは、既存クラスのインスタンス変数やメソッドを引き継いで新しいクラスを定義することである。継承を用いると、すでに存在するクラスに機能を追加したり、一部の機能を書き換えたりするだけで新しい機能を持つクラスを作成することが出来る。このように、新しいクラスを作成することを、クラスを拡張(extend)するという。

構文　サブクラスの宣言

```
class サブクラス名 extends スーパークラス名
{
        サブクラスに追加するメンバ…
        サブクラスのコンストラクタ(引数リスト)
                …
}
```

```
/*  例 2.2 */

class Car
{
        int num;
        double gas;

        public Car( )
        {
                num = 0;
                gas = 0.0;
                System.out.println("車を作成した。");
        }
        public void setCar(int n, double g)
        {
                num = n;
                gas = g;
                System.out.println("ナンバーを" + num + "にガソリン量を" + gas
                        + "にした");
        }
        public void show( )
        {
                System.out.println("車のナンバーは" + num + "である。");
                System.out.println("ガソリン量は" + gas + "である。");
        }
}


class RacingCar extends Car
{
        private int course;

        public RacingCar( )
        {
                course = 0;
                System.out.println("レーシングカーを作成した。");
        }
        public void setCourse(int c)
        {
                course = c;
                System.out.println("コース番号を" + course + "にした。");
        }
}


class Sample
{
        public static void main(String arg[ ])
        {
                RacingCar rccar1;
                rccar1 = new RacingCar( );

                rccar1.setCar(1234, 20.5);
                rccar1.setCourse(5);
        }
}
```

- 例 2.2 で Car をスーパークラス、RacingCar をサブクラスと呼ぶ。例 2.2 を実行、その結果を確認し、継承の意味を理解せよ。

2.3　オーバーライド

サブクラスでメソッドの上書きをすることをオーバーライド(Override)したと呼ぶ。サブクラスでスーパークラスと同じメソッド名、引数、戻り値のメソッドを定義することで、スーパークラスのメソッドではなく、サブクラスのメソッドが呼ばれるようにする機能である。

Java では、クラスのインスタンスを作成したときに「自分が持っているメソッド」を覚えており、その際に「スーパークラスにのみ存在するメソッド」はそのまま覚え、スーパークラスと同じメソッド（＝オーバーライドしたメソッド）がある場合にはそのメソッドを覚える。

```
/*  例 2.3 */

class RacingCar extends Car
{
        private int course;

        public RacingCar( )
        {
                course = 0;
                System.out.println("レーシングカーを作成した。");
        }
        public void setCourse(int c)
        {
                course = c;
                System.out.println("コース番号を" + course + "にした。");
        }
        public void show( )
        {
                System.out.println("レーシングカーのナンバーは" + num + "である。");
                System.out.println("ガソリン量は" + gas + "である。");
                System.out.println("コース番号は" + course + "である。");
        }
}


class Sample
{
        public static void main(String arg[ ])
        {
                RacingCar rccar1;
                rccar1 = new RacingCar( );

                rccar1.setCar(1234, 20.5);
                rccar1.setCourse(5);

                rccar1.show( );
        }
}
```

- 例 2.2 の後半部分を例 2.3 に換えて実行し、オーバーライドの仕組みを理解せよ。

2.4　抽象クラス

抽象クラスは(abstract class)は、オブジェクトを作成することができないクラスを意味する。通常、このクラスは一つまたは複数のサブクラスによって実装される機能を宣言するために使われる。抽象クラスのメリットは、クラスの機能さえ定義すれば、その機能をどのように実現するかまでは定義しなくて済むということである。

構文　抽象クラスの宣言

```
abstract class  クラス名
{
        フィールドの宣言;
        abstract  戻り値の型  メソッド名(引数リスト)
}
```

```
/*  例 2.4 */

abstract class Shape
{
        abstract void display( );
}


class Circle extends Shape
{
        void display( )
        {
                System.out.println("円");
        }
}


class Rectangle extends Shape
{
        void display( )
        {
                System.out.println("四角");
        }
}


class Triangle extends Shape
{
        void display( )
        {
```

```
                System.out.println("三角");
        }
}


class AbstractClassDemo
{
        public static void main(String args[ ])
        {
                Shape s = new Circle( );
                s.display( );
                s = new Rectangle( );
                s.display( );
                s = new Triangle( );
                s.display( );
        }
}
```

- 例 2.4 を実行し抽象クラスの使い方を習得せよ。

2.5　インタフェースと実装

Java には抽象クラスとよく似た目的で用いられる、カテゴリーを作るためだけに使われる特殊なクラスであるインタフェース(interface)がある。インタフェースはフィールドとメソッドをもつことができるが、コンストラクタはもたない。インタフェースのフィールドは定数で、メソッドは抽象メソッドである。インタフェースではフィールドを変更したり、メソッドの処理内容を定義したりすることはできない。また、インタフェースはクラスに良く似ているがオブジェクトを作成することはできない。

インタフェースはクラスと組み合わせて使うことになっている。インタフェースをクラスと組み合わせることを、インタフェースを実装(implementation)するという。継承と似ているが、インタフェースの場合は継承と呼ばない。インタフェースを implementation したクラスは、元となったインタフェースで定義されているメソッドをもつこととなる。しかし、それはその名前のメソッドをもつよう強制されたから定義したということであり、自動的に受け継いだわけではない。インタフェースを implementation するということは、すなわちインタフェースで定義されているインタフェースの実装を義務付けるということである。また、あるインタフェースを実装したクラスは、そのインタフェースに定義されているすべての抽象メソッドを実装しなくてはならない。

構文　インタフェースの宣言とインタフェースの実装

```
interface インタフェース名
{
        型名　フィールド名　= 式;
        戻り値の型　メソッド名( );
}
```

```
class クラス名 implements インタフェース名
{
        …
}
```

```
/* 例 2.5 */

interface employee
{
   void startWorking();
   void stopWorking();
}


interface Secretary extends employee
{
   void doSecretaryJob();
}


interface Programmer extends employee
{
   void doProgrammingJob();
}


class Tanaka implements Secretary
{
   public void startWorking()
   {
      System.out.println("田中が業務開始します");
   }
   public void stopWorking()
   {
      System.out.println("田中が業務終了します");
   }
   public void doSecretaryJob()
   {
      System.out.println("田中が秘書の仕事をします");
   }
}


class Nakamura implements Programmer
{
   public void startWorking()
   {
      System.out.println("中村が業務開始します");
   }

   public void stopWorking()
   {
      System.out.println("中村が業務終了します");
   }
```

```
    public void doProgrammingJob()
    {
        System.out.println("中村がプログラムを作ります");
    }
}


class Test
{
    public static void main(String[ ] args)
    {
        employee[ ] employee = {new Tanaka(), new Nakamura() };

        for (int i = 0; i < employee.length; i++)
        {
            employee[i].startWorking();
            if (employee[i] instanceof Secretary)
            {
                ((Secretary)employee[i]).doSecretaryJob();
            }
            else if (employee[i] instanceof Programmer)
            {
                ((Programmer)employee[i]).doProgrammingJob();
            }
            employee[i].stopWorking();
        }
    }
}
```

- 例 2.5 を実行し、結果を確認してインタフェースとその実装について理解せよ。

2.6　パッケージとインポート

大規模なプログラムを開発する際には、複数の人間で効率的にクラスを作成したい。また、Java ではクラスごとにクラスファイルを作成するため、複数のクラスを同じファイルに記述するよりもクラスごとに Java ソースファイルを作成する方が、不必要なコンパイルを削減できる。また、大規模なプログラムを作成する場合は、他の人が作成したクラスを再利用する場合が多い。様々な人が作成したクラスを混在させて利用する場合に名前の衝突を避けるため、Java にはパッケージという仕組みが組み込まれている。ソースファイルの先頭に package 文が存在すると、そのファイルで定義したクラスはそのパッケージに属することになる。

ソースファイルに package 文が存在しない場合は、そのファイルで定義されているクラスは無名パッケージに属することになる。同じパッケージに属するクラス同士は、特にパッケージ名を指定しなくてもお互い利用可能である。異なるパッケージに属するクラスを利用する場合は、パッケージ名を含む完全限定名（fully qualified name, FQN）で指定する。

たとえば次の例でわかるように、ritsumei に含まれるクラス File を指定する場合は、ritsumei.File とすればよい。また、import 文を用いると、完全限定名を指定しなくてもクラス名だけでクラスを指定可能となる。

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

特定のパッケージの直下に属するクラス群をまとめて指定する場合は、次のように＊をもちいて import 文を記述することができる。

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

このように記述することで、パッケージ java.io に属するクラスをクラス名だけで指定できる。この import 文は主にクラスライブラリのクラス群を使う場合によく用いられる。

（java.lang パッケージについては、特に import 文を記述しなくても自動的に取り込まれる。java.lang にはどんなクラスがあるか、API 仕様を調べよ。）

2.7　例外処理

想定していない入力を受け取ったり，プログラム自体のバグにより正しくない処理を行ったりした時は，通常の制御とは違った処理をする必要がある。このような処理を例外処理(exception handling)と言う。Java では、プログラム実行時のエラーを適切に処理するため、例外(exception)という仕組みを備えている。

構文　例外処理

```
try
{
        例外の発生を調べる文;
          …
}
catch (例外のクラス　変数名)
{
        例外が起きたときに処理する文;
          …
}
finally
{
        必ず最後に処理する文;
          …
}
```

```
/*  例 2.7 */

class Divider
{
   public static void main(String args[ ])
   {
      try
      {
         System.out.println("Before Division");
         int i = Integer.parseInt(args[0]);
         int j = Integer.parseInt(args[1]);
         System.out.println(i / j);
         System.out.println("After Division");
      }
      catch (ArithmeticException e)
      {
         System.out.println("ArithmeticException");
      }
      catch (ArrayIndexOutOfBoundsException e)
      {
         System.out.println("ArrayIndex" + "OutOfBoundsException");
      }
      catch (NumberFormatException e)
      {
         System.out.println("NumberFormatException");
      }
      finally
      {
         System.out.println("Finally block");
      }
   }
}
```

- 例 2.7 を実行、結果を確認し、try、catch、finally の使い方を習得せよ。(実行方法は java Divider 3 9 のように引数を後ろに書く。)

2.8　スレッド

Java 言語ではスレッド(thread)という概念を用いて並列処理を実現している。論理的に並列というのは、本当に並列に動くのではなく、一つの計算機上で時分割(time-sharing)に動くことを意味する。

これまでの皆さんが開発して来たプログラムは一つの流れで処理が行われるものだったが、プログラムの複数の箇所の処理が同時に行われるようにすることができる。処理の流れの一つ一つをスレッドと呼び、プログラム上でスレッドを増やすことを「スレッドを起動する」という。複数のスレッドを用いることで音楽を鳴らしながら、画像データを変化させるなど並列処理が必要なプログラムの作成が可能なわけである。

スレッドの主なメソッド(API java.lang.Tread 参考)

start

public void **start**()

このスレッドの実行を開始します。Java 仮想マシンは、このスレッドの run メソッドを呼び出します。

その結果、(start メソッドへの呼び出しから復帰する) 現在のスレッドと (その run メソッドを実行する) 別のスレッドという 2 つのスレッドが並列に実行されます。

Run

public void **run**()

このスレッドが別個の Runnable 実行オブジェクトを使用して作成された場合、その Runnable オブジェクトの run メソッドが呼び出されます。そうでない場合、このメソッドは何も行わずに復帰します。

Thread のサブクラスは、このメソッドをオーバーライドしなければなりません。

Sleep

public static void **sleep**(long millis)

現在実行中のスレッドを、指定されたミリ秒数の間、スリープ (一時的に実行を停止) させます。スレッドはモニターの所有権を失いません。

Join

public final void **join**()

このスレッドが終了するのを待機します。

```
/* 例 2.8 */
class Threadtest extends Thread
{
        private String name;

        public Threadtest(String nm)
        {
                name=nm;
        }

        public void run()
        {
                int a;

                for (a = 1; a <= 30; a++)
                        System.out.println(name + a);
        }
}


class test
{
        public static void main(String args[ ])
        {
                Threadtest testthr_A= new Threadtest("testA");
```

```
                Threadtest testthr_B= new Threadtest("        testB");

                int a;

                testthr_A.start();
                testthr_B.start();
                for (a = 1; a <= 30; a++)
                        System.out.println("              main" + a);
        }
}
```

- 例 2.8 の結果を確認し、マルチスレッドの概念を理解せよ。また、スレッドの使い方を習得せよ。

2.9 アプレット

アプレット(applet)とは Web ページの HTML から参照されるプログラムのことである。Web サーバーからブラウザに動的にダウンロードされ、ブラウザの環境で実行される。それ以外にも、アプレットビューアなどのツールでアプレットを実行することが可能である。

Java アプリケーションの実行はクラスの main( )メソッドから始まったのに対し、アプレットは web ブラウザまたはアプレットビューアなどのツールに用意された環境で実行される。したがって、main()メソッドは存在しない。その代わり、アプレットのライフサイクルにしたがって init()、start()、stop()、destroy()の 4 つのメソッドが呼び出される。これらのメソッドは java.applet.Applet クラスに定義されているので、すべてのアプレットに継承される。

• **init()** init()メソッドはアプレット起動されるときにのみ呼び出される
• **start()** start()メソッドはアプレットの実行を再開するとき(ブラウザまたは appletviewer が元に戻る)に呼び出される。
• **stop()** stop()メソッドはアプレットの実行が中断される時(ブラウザや appletviewer が最小化)に呼び出される。
• **destroy()** destroy()メソッドはアプレットが終了されるときに呼び出される。
• **paint(), update(), repaint()** paint()メソッドはアプレットが表示されるときに呼び出される。repaint()メソッドはプログラム側でアプレットを再描画したい場合に使います。update メソッドは repaint メソッドをプログラム側で呼び出した際に paint メソッドが呼ばれる前に呼び出されるメソッドです。update メソッドをうまく使うことで画面のちらつきをなくせます。

```
/* 例 2.9 */
import java.applet.Applet;
import java.awt.Graphics;

public class AppletLifecycle extends Applet
{
   String str = "";

   public void init()
{
      str += "init; ";
   }

   public void start()
{
      str += "start; ";
   }

   public void stop()
{
      str += "stop; ";
   }

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

   public void paint(Graphics g)
{
      g.drawString(str, 10, 25);
   }
}
```

```
<html>
<head>
   <title>Applet</title>
</head>
<body>
   <applet code="AppletLifecycle" width=300 height=50>
   </applet>
</body>
</html>
```

例 2.9 をコンパイルし、同じフォルダに上記の html を「タイトル.html」のファイル名で保存する。その後、

appletviewer タイトル.html ↲

で実行する。paint( )は init( )または start( )によって自動で呼び出され実行されるメソッドで、アプレット領域に描画する。

- 例 2.9 の結果を確認し、アプレットを構成するメソッドの実行時期とアプレットの製作する方法を身に付けよう。

3. レポート作成・提出・注意事項

3.1 レポートの作成

レポートは原則として MS ワードで作成する。ホームページからレポートのテンプレートをダウンロードして利用する。レポートは誠意をもって作成し、感想などは簡単すぎないよう十分な量を書く。参考にしたホームページ、資料等があった場合は具体的にどの部分をどのように参考にしたか、明確に書く。また、友人に手伝ってもらった場合も、手伝ってもらった友人の学籍番号、名前、どの部分をどのように手伝ってもらったか明記する。

3.2 注意事項

① 実験室には飲食物を持ち込まない。
② 実験中は私語を慎む。
③ 実験中には携帯電話の電源を切り、かばんの中にしまっておく。
④ 他人が作成中のプログラムや出力は見ない。
⑤ 実験中にはメールの送受信をしない。
⑥ 実験中には実験と関係ないサイトは見ない。
⑦ 休みの時間以外は実験室から出ない。(やむを得ず出る場合は教員に許可を得る。)
⑧ 教員から説明があるときは、作業を中断し、説明に集中する。
⑨ 実験終了後は必ずログオフしてから退室する。
⑩ レポートは期限を遵守する
⑪ 授業開始 15 分までの入室を遅刻、それ以降は欠席とする。
⑫ ２回以上欠席時には評価対象外とする。
⑬ 教員の判断により、レポートのできばえが悪い場合はそのレポートは受理しない。
⑭ レポートのコピーまたは一部改正が発覚された場合は試験のカンニングとして扱い、元と写しを問わず両方処分するので十分注意する。一緒に考えたり、手伝ってもらったりした場合は 3. 1 に書いてあるとおり、その事実をレポートに明記する。

- 参考文献

[1] 高橋麻奈、やさしい Java 第 5 版、ソフトバンク、2013.
[2] ジョゼブ・オニール、独習 Java 第 4 版、翔泳社、2008.

メモ