

インターネットプログラミング  
2401教室  
第10回  
2014/11/26

岩井将行

[http://www.soe.dendai.ac.jp/kyomu/portal/2014\\_schedule\\_t.pdf](http://www.soe.dendai.ac.jp/kyomu/portal/2014_schedule_t.pdf)

- 2015年1月21日期末テスト?+優秀課題発表会(参加)
- 2015年1月14日期末テスト?課題提出日
- 2015年1月7日あり
- 2014年12月17日休講
- 2014年12月10日企画案チェック
- 2014年12月2日企画案提出+(通信+GUI)

# 授業資料

- <http://www.cps.im.dendai.ac.jp>

# 講師紹介

- <http://cps.im.dendai.ac.jp/index.php?Members%2Fiwai>
- 岩井研究室
- <http://cps.im.dendai.ac.jp>
- 岩井研究室の研究分野
- <http://cps.im.dendai.ac.jp/index.php?Research%2FTopics>
- 連絡先 1号館11F 11107b
- iwaiあっと im.dendai

# TA・SA・副手

- 岩井研の精鋭

tjm

- kondo
- iwahori

# 成績

- 毎回取り組み姿勢(出席)
  - 毎回課題(演習のみ)
  - 中間試験(座学)
  - 最終試験(座学)
  - 最終課題(演習のみ)
- 
- ★演習は演習最終発表会を加味

# 講義内容

[第1回](#) Java理解度チェック

[第2回](#) Javaプログラミング基礎2

[第3回](#)

TCP/IPの復習 TCPサーバ

[第4回](#)

TCPクライアント/サーバ通信 チャットプログラム

[第5回](#)

UDP通信

[第6回](#)

中間学力考査（持ち込み不可 紙提出）

[第7回](#)

スレッド基礎 サーバのスレッド化 マルチスレッド

[第8回](#)

デザインパターン

ファクトリメソッド シングルトン

[第9回](#)

ノンブロッキングI/O Javaプログラミング応用

[第10回](#)

マルチスレッド スレッドプール

[第11回](#)

Webクライアント

[第12回](#)

WEBサーバ,プロジェクト設計

[第13回](#)

プロジェクト実装1

[第14回](#)

2014/11/26  
プロジェクト実装2

[第15回](#)

学力考査（持ち込み可 プログラミング提出）

# 授業予定日日程

- [http://www.soe.dendai.ac.jp/kyomu/portal/2014\\_schedule\\_t.pdf](http://www.soe.dendai.ac.jp/kyomu/portal/2014_schedule_t.pdf)
- スケジュール 十
- (1)9/17 第1回 Java理解度チェック
- (2)9/24 第2回 Javaプログラミング基礎1
- (3)10/1 第3回 Javaプログラミング基礎2 TCP/IPの復習  
TCPサーバ
- (4)10/8 第4回 TCPクライアント/サーバ通信 チャットプログラム
- (5)10/15 第5回 UDP通信
- (6)10/22 第6回 中間学力考査（持ち込み不可 紙提出）



# 概要

- クライアント／サーバモデル、TCP/IPネットワークのアプリケーションプログラミングインタフェースの基本および、ネットワークアプリケーションを効率的に動作させるためのマルチスレッドプログラミングを講義する。この基本の後、チャット等の対話型アプリケーション、Twitter4J等のアプリケーション開発の実例を講義する。

# ゴール

- 通信ネットワークを利用したアプリケーションソフトウェアを、TCP/IP を意識したレベルで作成できる力を養成することを目標とする。

# Java プログラムの基本

クラス

```
public class Hello {
```

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

```
        // この中に、処理内容を書きます
```

```
    }
```

```
}
```

メソッド

注意1: Hello の部分がプログラム名

注意2: String は、文字列。String[] 文字列の配列

注意3: args は、コマンド引数の配列

args[0], args[1]

# 繰り返し (for文)

```
int i;  
for(i=1; i<=4; i=i+1) {  
    内容  
}
```



```
変数の宣言;  
for(初期化式; 条件式; 増加式) {  
    内容  
}
```

# 繰り返し (for文)

- 変数の宣言
  - 繰り返しの回数をメモしておく変数を用意する
- 初期化式
  - 変数の最初の数字は何か
- 条件式
  - 変数がどうなっている間、続けるか
- 増加式
  - 一回繰り返すごとに、変数をどうするか

```
変数の宣言;  
for(初期化式; 条件式; 増加式) {  
    内容  
}
```

# 繰り返し (for文)

- 変数の宣言
  - 整数型の名前がiという変数を用意
- 初期化式
  - 変数iの最初の数字は1
- 条件式
  - 変数iが4以下の間、続ける
- 増加式
  - 一回繰り返すごとに、変数iに1を足したものを、変数iに入れる

```
int i;  
for(i=1; i<=4; i=i+1) {  
    内容  
}
```

# 繰り返し (while文)

```
int i;  
i=1;  
while(i<=4) {  
    内容  
    i=i+1;  
}
```



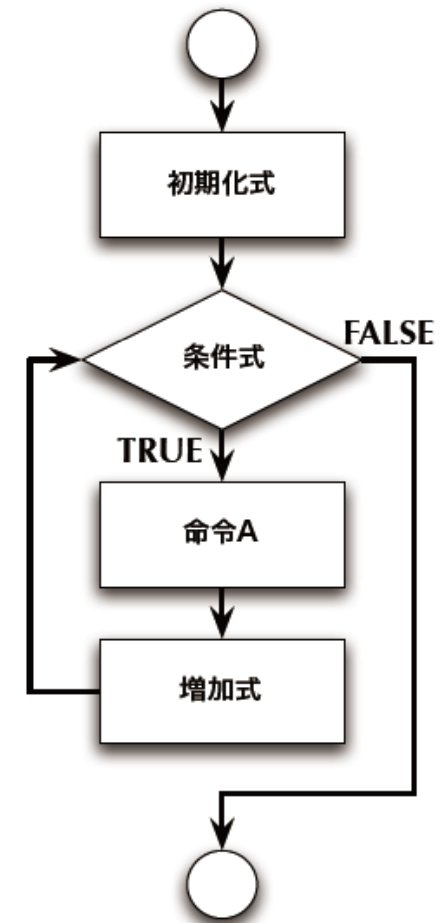
```
変数の宣言;  
初期化式;  
while(条件式) {  
    内容  
    増加式;  
}
```

# フローチャート (繰り返し)

```
int i;  
for(i=1; i<=4; i=i+1) {  
    命令A  
}
```



```
変数の宣言;  
for(初期化式; 条件式; 増加式) {  
    命令A  
}
```





# 省略演算

- 足し算

- $i=i+1;$

- $i+=1;$

- $i++;$

- 引き算

- $i=i-1;$

- $i-=1;$

- $i--;$

- 掛け算

- $i=i*2;$

- $i*=2;$

- 割り算(切捨て)

- $i=i/10;$

- あまりの計算

- $p=i\%2;$

- もし*i*が偶数なら $p==0;$

- 奇数なら $p==1$

# for文

```
for ( 初期化; 条件式; 次の一歩 ) {  
    繰り返す処理  
}
```

```
for (int i = 0; i < 3; i++) {  
    System.out.println(i);  
}
```

により、

0

1

2

が表示される。

# 変数の有効範囲(スコープ)

```
for (int i = 0; i < 3; i++) {  
    System.out.println(i);  
}
```

System.out.println("i = " + i): ← iは有効範囲外なので  
コンパイルエラー。

## 解決策

```
int i;  
for (i = 0; i < 3; i++) {  
    System.out.println(i);  
}  
System.out.println("i = " + i):
```

# while 文

```
while (条件式) {  
    繰り返す処理  
}
```

## null (ナル、ヌル)の意味

```
while (line != null) {
```

null 入力の終わりに達したときの特殊なオブジェクト

# 繰り返し文の練習問題

1 から 100 までの整数を足し合わせる

(1) その1: for文を使う

(2) その2: while文を使う

CountTest.java

WhileTest.java

次週以降

CountTenRunnable.java

CountTesterTweThreads.java

# Class

クラスは、物の設計図。  
中に変数やメソッドが定義される。

オブジェクトは、クラス定義に基づく実際の物。プログラム上は、変数。

例： Automobile というクラスを定義する。  
Automobileクラスで、volkesWagen,  
audi, volvo というオブジェクトを作る。

# method

車というクラスを定義する。メソッドとして、

乗る、止める  
を用意する。

ポルシェというオブジェクトを車という  
クラスで生成すると、

ポルシェ. 乗る、  
ポルシェ. 止める  
というメソッドが使える。

# 定義する場所

## クラス

```
戻り値 メソッド1 {  
  XXXXXXXXXXXX  
}
```

```
戻り値 メソッド2 {  
  XXXXXXXXXXXX  
}
```

```
戻り値 メソッド3 {  
  XXXXXXXXXXXX  
}
```

```
戻り値 メソッド4 {  
  XXXXXXXXXXXX  
}
```

いくつでも  
作ってよい。



# public ?

## クラス

```
public 返回值 メソッド1 {  
    XXXXXXXXXXXX  
}
```

```
public 返回值 メソッド2 {  
    XXXXXXXXXXXX  
}
```

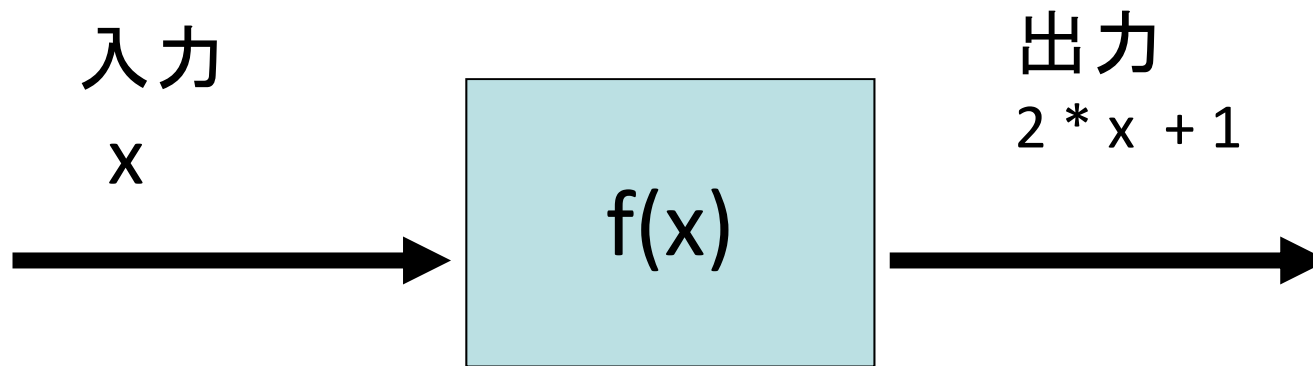
```
返回值 メソッド3 {  
    XXXXXXXXXXXX  
}
```

```
返回值 メソッド4 {  
    XXXXXXXXXXXX  
}
```

# 関数

- 関数

$$f(x) = 2 * x + 1$$

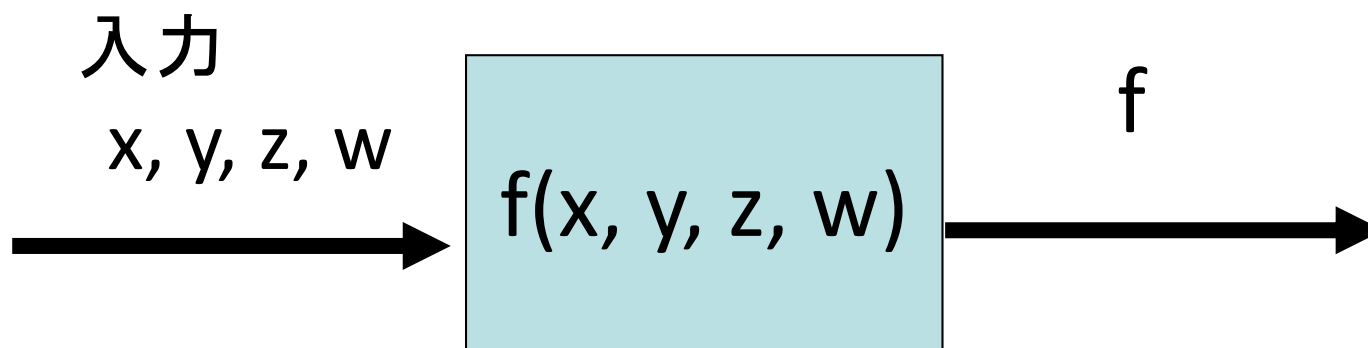


関数の定義部分

しかし、関数の入力はいくらでもあってよい。

- 関数

$$f(x, y, z, w) = 2 * x + y - z + \lfloor w \rfloor$$



関数の定義部分

# メソッドで引数がたくさんあるとき

```
int  calcComplex(int x, int y, int z,  
                float w) {  
    if ( x > y ) {  
        return z;  
    } else {  
        return (int)w;  
    }  
}
```

# メソッド分け

- 合成関数

$$f(x) = 2 * x + 1$$

$$h(x) = 3 * (2 * x + 1) + 5$$

のとき、 $h(x) = (g \circ f)(x)$

```
int h(int x) {  
    return 3 * (2 * x + 1) + 5;  
}
```



```
int h(int x) {  
    return 3 * g(x) + 5;  
}  
  
int g(int x) {  
    return 2 * x + 1;  
}
```

Javaプログラミングも同じ。メソッドとして独立させた方がよいかどうか、よく考える。

# メソッドの形式

公開するか  
否か

クラス  
メソッドとす  
る

戻り値の型

```
public static int メソッド名(引数宣言) {
```

## メソッドの中身

```
    return (戻り値);  
}
```

# void

関数によっては、戻り値がいらぬものもある。そのときには、戻り値なし (void) を指定する。

前回作成した、drawBar に戻り値は必要なかった。

引数がない場合もある。

# 型

int 整数

float 浮動小数点数 (実数)

char 文字型

等



# メソッドの引数

戻り値   メソッド名(型 変数名1,  
                  型 変数名2,  
                  型 変数名3,  
                  型 変数名4  
                  .....) {

メソッドの本体

}



# メソッド呼び出し

本来は、

```
g.drawString(XXXXXXXXXXXXXXXXXX);
```

のように、

```
オブジェクト.メソッド名(引数...);
```

と書く。

## メソッド呼び出し(2)

しかし、自分で定義したクラスの中のメソッドを呼び出すときは、  
オブジェクト.

なしに、  
メソッド名(引数...);  
でよい。

例:

```
drawBar(XXXXXXXXXXXXX);
```

# methodとクラス

- Heikin.java と Kamoku.java
- Heikin と Kamoku クラスを作る
  - public class Heikin
  - class Kamoku
- Heikin クラス
  - Kamokuクラスのインスタンスとして、englishとmath を作る
  - english の name に "英語" を設定する
  - english の score に 80 を設定する
  - math も english と同様に (name→数学, score→70)
  - 英語と数学のscoreを読み出して、平均値を表示する
- Kamoku クラス
  - String name
  - setScore というメソッドを定義する。score に値を設定する。
  - getScore というメソッドを定義する。scoreを返す。

# 定数の宣言

C++/C では、#define 文を使用した。

(例)

```
#define WIDTH 80
```

Javaでは、final static で修飾する。

(例)

```
public final static int WIDTH = 80;  
public final static String school = "dendai";
```

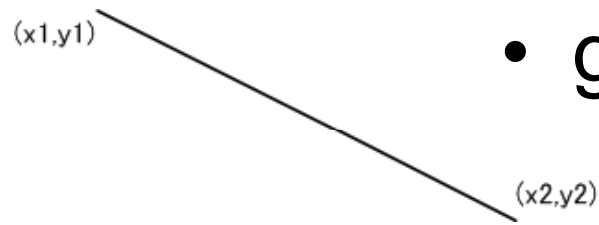
# Graphics

Graphics というクラスには、  
drawString, drawCircle 等の  
メソッドが定義されている。

Graphics クラスである g という  
オブジェクトに対して、  
g.drawString、  
g.drawCircle  
という形でメソッドを呼び出せる。

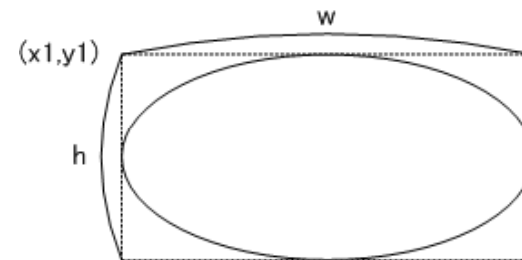
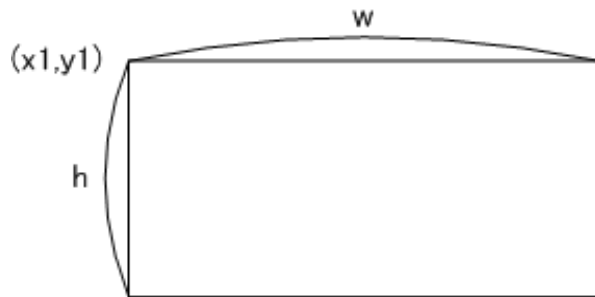
# Graphicsのmethod

- `g.drawLine(x1,y1,x2,y2);`



- `g.drawOval(x,y,w,h);`

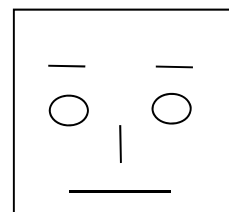
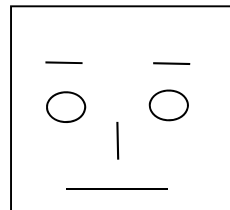
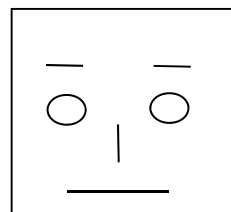
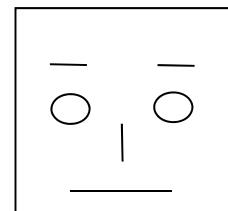
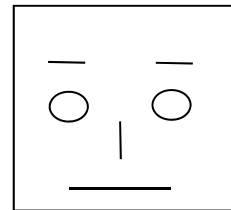
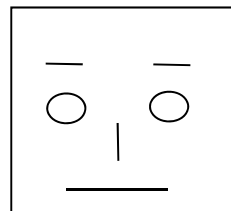
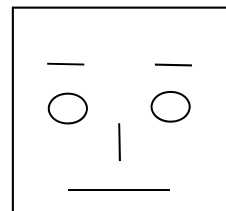
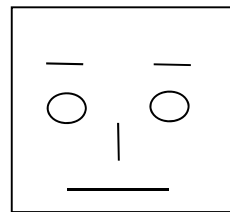
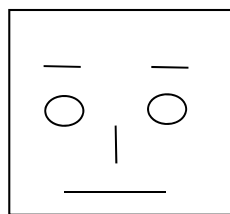
- `g.drawRect(x,y,w,h);`





課題faceを沢山つくってみよう  
Face.javaを改造してFaces.java  
を提出してください。

課題faceを沢山つくってみようFace.javaを改造して  
Faces.javaを提出してください。二重の繰り返し文を  
用いること。例



# Face ヒント1

```
void drawFace(Graphics g,  
               int xStart,  
               int yStart) {
```

左隅の座標を (xStart, yStart) と  
して、一つの顔を描くメソッドを記述する。

```
}
```

# Faceヒント2

paint メソッドの中には、

```
for(int i = 0; i < 3; i++) {  
    for(int j=0; j < 3; j++) {  
        drawFace(g, 20 + 80 *i,  
                20 + 80 *j);  
    }  
}
```

80 という数字は仮。顔の大きさを  
考えて計算する。

# しかし、数字決め打ちは避けたい

```
for(int i = 0; i < 3; i++) {  
    for(int j=0; j < 3; j++) {  
        drawFace(g, 20 + step *i,  
                20 + step *j);  
    }  
}
```

# 眉毛や鼻の形を自由に変えたい

```
void drawFace(Graphics g, int xStart, int yStart) {  
    drawFrame(g, xStart, yStart);  
    drawEyeBrow(g, xStart, yStart);  
    drawEye(g, xStart, yStart);  
    drawNose(g, xStart, yStart);  
    drawMouth(g, xStart, yStart);  
}
```

さらに内部で  
メソッドに分ける。

```
void drawFrame(Graphics g, int xStart, int yStart) {  
    記述  
}  
void drawEyeBrow(Graphics g, int xStart, int yStart) {  
    記述  
}  
void drawEye(Graphics g, int xStart, int yStart) {  
    記述  
}  
void drawNose(Graphics g, int xStart, int yStart) {  
    記述  
}  
void drawMouth(Graphics g, int xStart, int yStart) {  
    記述  
}
```

# クラスとインスタンス

- クラス



## インスタンス



# Kamoku.java

```
class Kamoku {
    String name;
    int score;

    Kamoku(int s) { // これがコンストラクタ。特殊なメソッド。クラス名と同じ。
        score = s;
    }

    // setScore というメソッドを定義する。
    // score に値を設定する。
    public void setScore(int num){
        score = num;
    }

    // getScore というメソッドを定義する。
    // scoreを返す。
    public int getScore() {
        return score;
    }
}

// メソッド 関数のこと
// public 戻り値(戻り値return value) 関数名() {
//     中に具体的な処理を書く
// }
```

```
public class HeikinA {  
    public static void main(String[] args){  
  
    // Kamokuクラスのインスタンスとして、englishを作る  
        Kamoku english = new Kamoku(80);  
    // 同様に、mathインスタンスをつくる。  
        Kamoku math = new Kamoku(70);  
  
    // english の name に "英語" を設定する  
        english.name = "英語";  
        int a = english.getScore();  
        System.out.println("英語の点は" + a + "ですね");  
        a = math.getScore();  
        System.out.println("数学の点は" + a + "ですね");  
    }  
}
```



# 【難】HeikinB objectの配列

```
public class HeikinB {
    public static final int N=100;
    Kamoku[] english = new Kamoku[N];
    String kamokuname="不明";

    HeikinB(String s){
        this.kamokuname=s;
    }
    void initalizeScores(){

        for (int i = 0; i < N; i++) {
            int score = 50+(int)
(Math.random() * 50);
            english[i] = new Kamoku(score);
            english[i].name = kamokuname +
i;
        }
    }
}
```

```
void printAverage(){
    double sum=0;
    for (int i = 0; i < N; i++) {
        int score =
english[i].getScore();

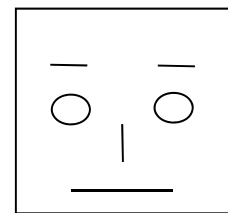
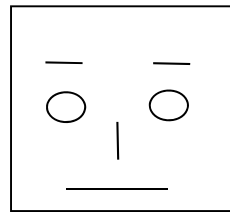
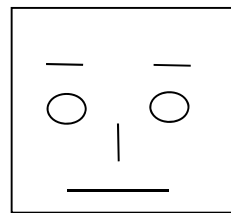
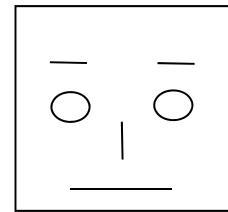
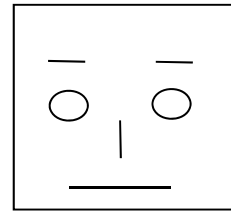
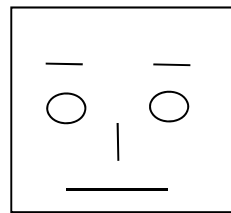
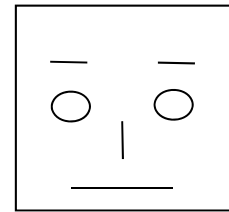
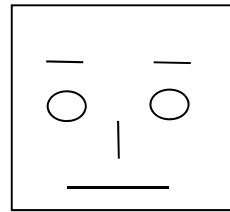
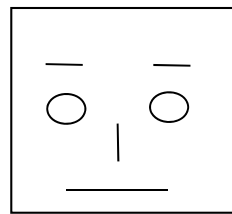
        System.out.println(english[i].name + "の点
は" + score+"点");
        sum+=score;
    }

    double average=sum/N;
    System.out.println(this.kamokuname+"の"
+ "平均の点は"
+average+"点");
}

public static void main(String[] args) {

    HeikinB heikinB=new HeikinB("英語");
    heikinB.initalizeScores();
    heikinB.printAverage();
}
}
```

課題faceを沢山つくってみよう  
FacesMain.javaを改造してobject指向で  
FacesMain.javaを提出してください。  
なるべくなら二重の繰り返し文を用いること。例



# Face ヒント1

```
void drawFace(Graphics g,  
               int xStart,  
               int yStart) {
```

左隅の座標を (xStart, yStart) と  
して、一つの顔を描くメソッドを記述する。

```
}
```

# Objectの配列

# Objectの配列: FaceObjKadaiAns.java

- `FaceObjAns[] fobjns = new FaceObjAns[9];`

```
for (int j = 0; j < 3; j++) {//行  
    yStart = j * 220 + 50;  
    for (int i = 0; i < 3; i++) {//列  
        xStart = i * 220 + 40;  
        fobjns[i + 3 * j] = new FaceObjAns(xStart, yStart);  
    }  
}
```

# 描画

```
// 9個数の顔を書く  
for (int i = 0; i < fobjs.length; i++) {  
    fobjs[i].makeFace(g);  
}
```

# FaceObjectのコンストラクタ

```
class FaceObjAns {  
    // コンストラクタ  
    int h;  
    int w;  
    int xStart = 0;  
    int yStart = 0;  
    public FaceObjAns(int x, int y) {  
        h = 200;  
        w = 200;  
        this.xStart = x;  
        this.yStart = y;  
    }  
    // 個々にメソッドを追加  
    public void makeFace(Graphics g) {  
        makeRim(g);  
        makeEyes(g, 20);  
        makeNose(g, 40);  
        makeMouth(g, 80);  
    }  
}
```

# Interface



- 内容に抽象メソッドしか持たない**クラスのようなもの(バールのようなもの)**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て**抽象メソッドを実装**します。





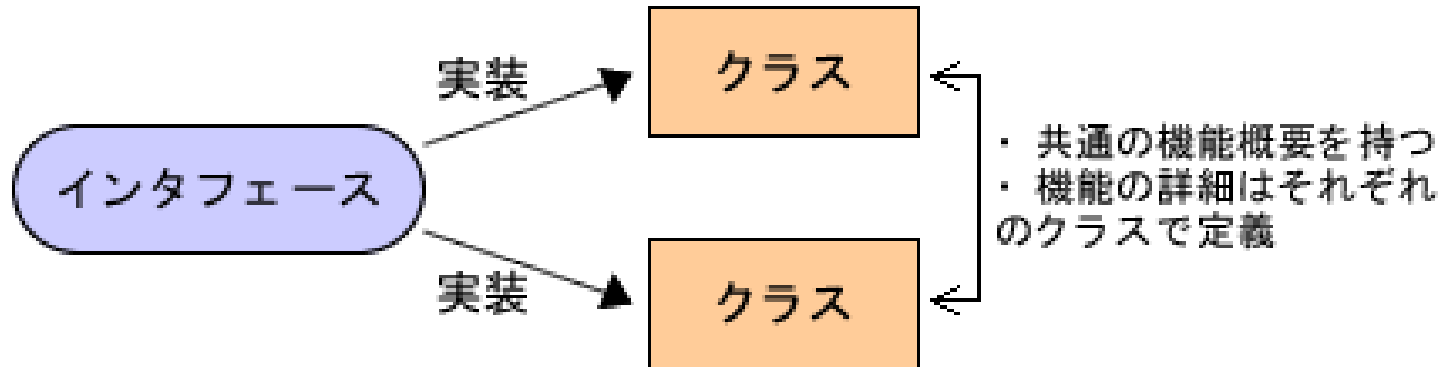
# インタフェースの複数実装

- クラスの場合は、単一のクラスしか継承 (extends) できませんが、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。

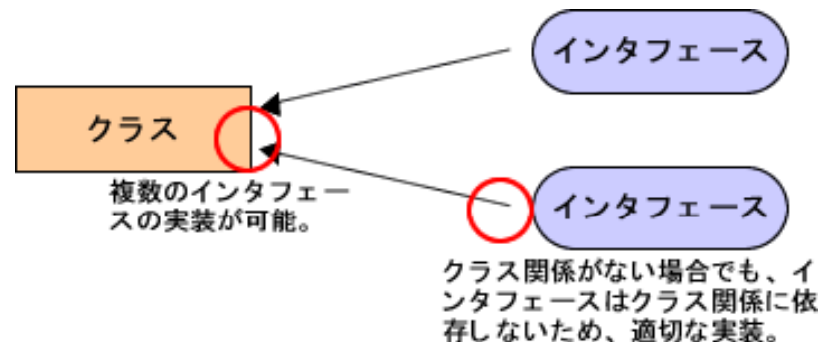
```
class Interfacelmpl implements Interface1, interface2, interface  
... }
```

interface の修飾子は public のみ。

# インタフェースのメリット



- Javaは複数のスーパークラスを継承することはできません。Javaでは複数のスーパークラスの継承(多重継承)が認められていないため。



# Plugin hybrid車は災害時にバッテリーとして利用可能



# ICar.javaインタフェース

- 車輪in getNumOfTiers()がある。
- スピートを設定する
  - void setSpeed (int sp)
  - int getSpeed()
  - void printCarName()

# IElectricCharge.javaインタフェース

- void chargeBattery(int b)
- int getAllBattery()
- int consumeBattery(int b)

# HybridCarImpl.java

- を実装してください。
- Yourには自分の名前を入れてください。
- 例 MasaHybridCarImpl.java

# 呼び出しのMainCall.javaを実装しよう。

- Hint
- `MasaHybridCarImpl masaCar= new MasaHybridCarImpl();`
- `ICar car=(ICar) masaCar;`
- `car.setSpeed(); car.printCarName();`
- `IElectricCharge charger =(IElectricCharge) masaCar`
- `charger.chargeBattery(100);`

# Thread

**Thread,Runnable**  
**MovingBall**



# ThreadSleep 停止

- 300ミリ秒処理を停止する。

```
try{
```

```
    Thread.sleep(3000);
```

```
    //3000ミリ秒Sleepする
```

```
}catch(InterruptedException e){}
```

# Threadの2種類の作りかた

- **1) implements Runnable**

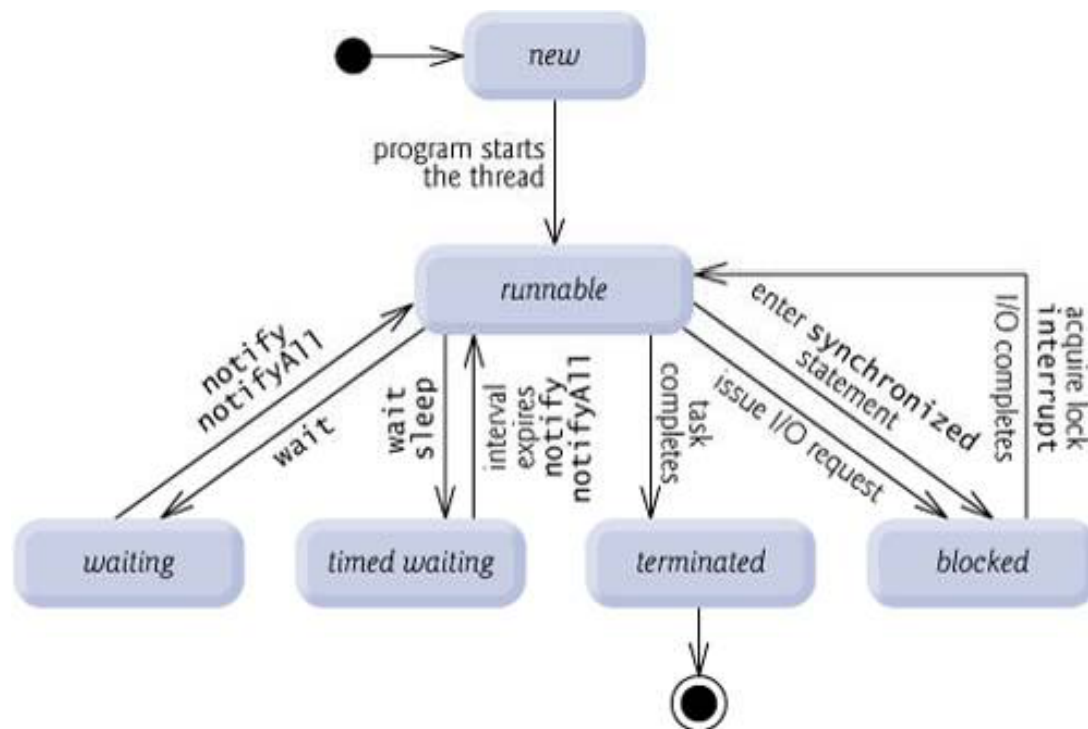
Runnableを実装したクラスをThreadクラスのコンストラクタとして渡す。start()で開始。

```
CountTenRunnable ct = new CountTenRunnable();  
Thread th = new Thread(ct);  
th.start();
```

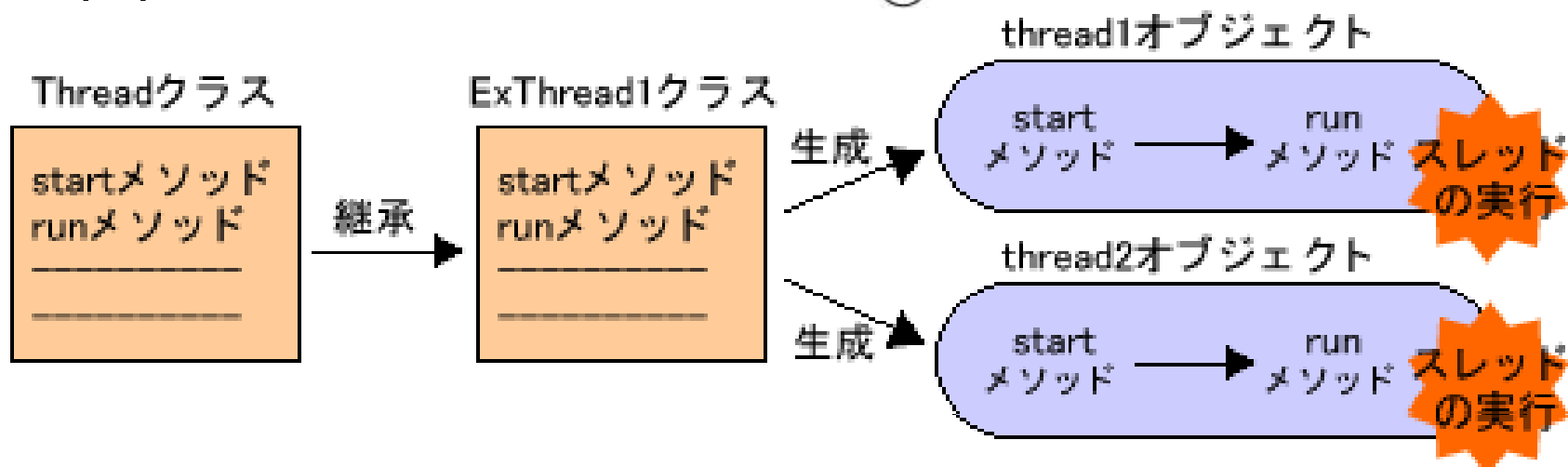
- **2) extends Thread**

Threadを拡張したクラスをnewしてstart()メソッドを呼び出す。

• (1)



• (2)



```
class MainThread{
    public static void main(String args[]){
        /* 別スレッドとして動作させるオブジェクトを作成 */
        SubThread sub = new SubThread(); /* 別のスレッドを作成し、スレッドを開始する */
        Thread thread = new Thread(sub) thread.start();
    }
}
class SubThread implements Runnable{
    public void run(){ }
}
```

# 動かしてみよう

- `CountTest.java`
- `CountTenRunnable.java`
- `CountTesterTwoThreads.java`

# MovingBall

```
MovingInnerFFrame f = new MovingInnerFFrame();  
Thread th = new Thread(f);  
th.start();
```

```
class MovingInnerFFrame extends Frame  
    implements Runnable {  
  
    public void run() {}  
}
```

# 第5回演習課題

- MovingBallを改造し3色以上の複数のボールを表示せよ。
- 壁の跳ね返りを画面のサイズに修正せよ。
- コンソールに残り秒数カウントダウンしてゲームが終了するようにThread.sleep()メソッドを用いよ。
- なお終了時間は20秒とせよ。
- ヒント
  - Thread.sleep(1\*1000);

MovingBallを配列にしよう。



# Singleton

# Singleton

- たった一つのインスタンスしか作らせないようにするパターン。  
普通はインスタンスを沢山作る
- 場合によってはインスタンスを一つしか作らせたくない
- プログラマ任せにすると、間違っnewを複数回呼び出してしまう。  
Singletonパターンを適用すると、指定したクラスインスタンスが1つしか存在しないことを保証する。

# Singleton

Singleton
- singleton
- Singleton() + getInstance()

```
public class MyFirstSingleton {  
    /* 唯一のインスタンス。*/  
    private static final MyFirstSingleton instance = new MyFirstSingleton();  
    /** * コンストラクタ。*/  
    private MyFirstSingleton() { }  
        /** * このクラスの唯一のインスタンスを返す。*/  
        public static MyFirstSingleton getInstance() {  
            return instance;  
        }  
}
```

# Singleton

- private な static 変数を定義して初期化  
インスタンスは、このクラスのロード時に一度だけ生成処理。  
コンストラクタはpublicでなく外部に公開せず  
private。外部からインスタンス生成されることを防ぐ。  
。privateにししないと外部から new とされてしまい  
インスタンスが自由に作れてしまう  
getInstance() を作成し外部に公開します。  
作られた唯一のインスタンスを返却することがこのメソッドの役目。  
public メソッドにしてどこからでも呼び出せるように。  
生成のタイミングは、初めて getInstance() をが呼ばれた時です。

# MyFirstSingletonCall

```
void Test() {  
    /* new できません。コンパイルエラーとなります。 */  
    // MyFirstSingleton mys = new MyFirstSingleton ();  
    // この時点で インスタンス生成 & 返却  
    MyFirstSingleton mys2 = MyFirstSingleton.getInstance();  
    // 同じインスタンス返却  
    MyFirstSingleton mys2 = MyFirstSingleton.getInstance();  
    if (mys1 == mys2) {  
        System.out.println("同じインスタンスです。");  
    }  
}
```

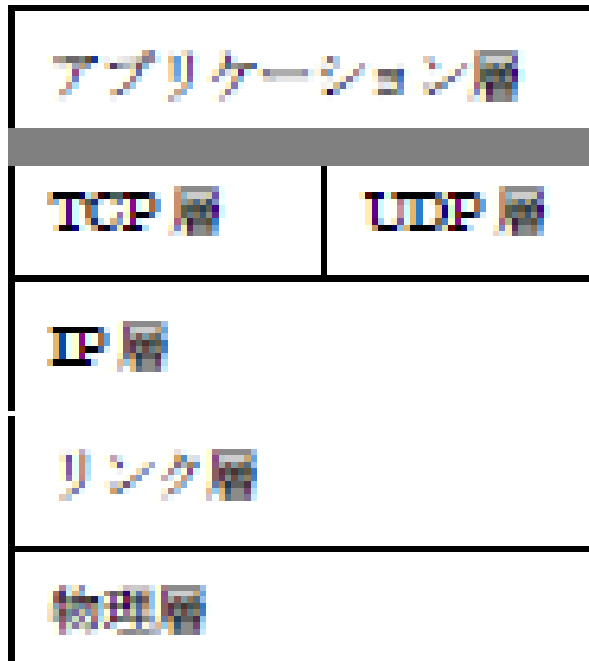
# 次週以降Swing

- `SwingAnimationBasic.java`
- `SwingAnimationFaceObj.java`

# 動かしてみよう。

- SwingAnimationBasic
- SwingAnimationFaceObj
- Swing より美しいグラフィックスを提供  
java2D
- ちらつき防止
- [http://www.java2s.com/Tutorial/Java/0240\\_Swing/Catalog0240\\_Swing.htm](http://www.java2s.com/Tutorial/Java/0240_Swing/Catalog0240_Swing.htm)

# Socket interface



ソケットインターフェイス



# UDP通信

- UDP通信では接続という概念はない。
- ユーザは毎回データを送信し、また毎回自分のソケット、及び相手のソケットとアドレスを指定することになる。TCPでは最初に相手のソケットとの間の接続を行い、双方のソケット間の接続が確立されたら、互いの通信が可能になる。
- TCPではそのような制限はない。
- TCPでは接続が確立されたら2つのソケットはストリームのように振る舞う。TCPでは受信したデータの信頼性と順序が保障される。

# DatagramPacket

- SocketやServerSocketでは、データのやりとりは入出カストリームに抽象化されていたため、パケットという概念が見えてこなかった。
- DatagramPacketとDatagramSocketの場合はUDPパケットはUDPパケットとして抽象化されており、パケットにデータも送信先アドレスも含める必要がある。
- SocketやServerSocketとは別物

# UDPServer

```
int serverPort = 5000;
```

```
DatagramSocket socket = new  
    DatagramSocket(serverPort);
```

```
DatagramPacket receivePacket = new  
    DatagramPacket(new byte[DMAX], DMAX);
```

```
socket.receive(receivePacket);
```

```
socket.close();
```

# Udp Client

```
String servhostname="localhost";
```

```
InetAddress serverAddress =
```

```
    InetAddress.getByName(servhostname);
```

```
String message="hello UDP from yourname";
```

```
byte[] bytesToSend = message.getBytes();
```

```
int serverPort = 5000;
```

```
DatagramSocket socket = new DatagramSocket();
```

```
DatagramPacket sendPacket = new
```

```
    DatagramPacket(bytesToSend, bytesToSend.length,  
    serverAddress, serverPort);
```

```
socket.send(sendPacket);
```

```
socket.close();
```

# ソケット通信

# プログラム同士の通信は

- ソケットを使ってデータの送受信
- ソケットを使った通信

ソケット通信

# ソケット

意味：「接続の端点」

コンピュータとTCP/IPを  
つなぐ出入り口

ソケット



# ソケット通信

- ソケットを使って通信を行うには  
2つのプログラムが必要

## クライアントプログラム

ソケットを用意して  
サーバに接続要求を行う

## サーバプログラム

ソケットを用意して接続要求を待つ



# ソケット通信の全体の流れ

クライアント

ソケット生成(socket)

サーバを探す  
(gethostbyname)

接続要求(connect)

データ送受信(send/rcv)

ソケットを閉じる(close)

サーバ

ソケット生成(socket)

接続の準備(bind)

接続待機(listen)

接続受信(accept)

データ送受信(send/rcv)

ソケットを閉じる(close)

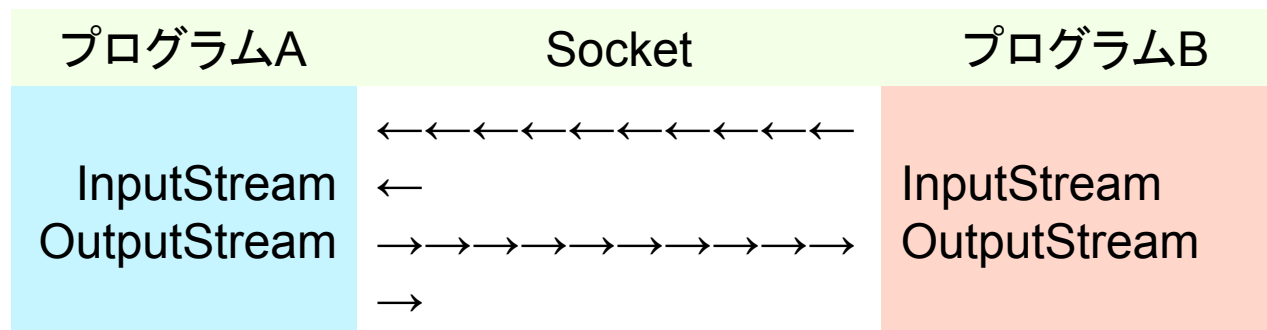
識別情報

# Socket通信

- 双方がデータのやりとりを行う場合、双方を結んで情報を運ぶための「線」が必要となります。Javaにおいてはこの「線」のことを「**Socket** (ソケット)」という概念で表します。「Socket」は逆方向の情報の流れ (Stream) をもつ二本の通信線を一本に束ねたものだと考えられます。

# InputStream, OutputStream

- 仮にAとBという二つのプログラムが通信を行うとすれば、一方の通信線がAにとっての「InputStream」でありかつBにとっての「OutputStream」、もう一方の通信線はAにとっての「OutputStream」かつBにとっての「InputStream」となる。
- この2本の通信線を束ねる「Socket」を介して情報のやりとりが行われる。

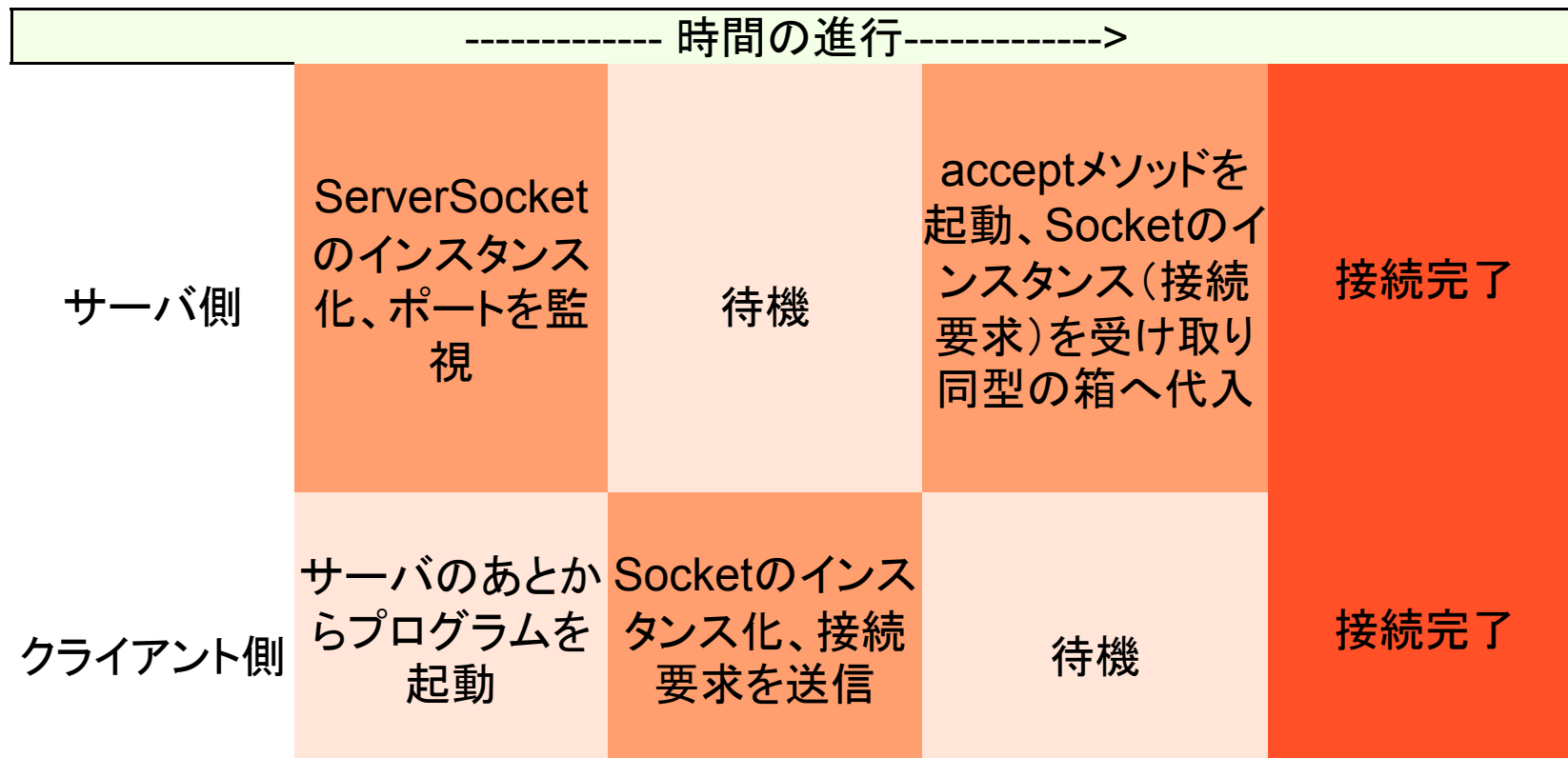


# Socketとport番号

- クライアント側からサーバ側に向かって「Socket」が送り込まれて、初めて通信が開通
- クライアントが「Socket」を送り込む際には、必ず「どのサーバコンピュータ」の「何番の通信口」に向かって送り込むのかを明らかにしなければなりません。
- その際の「どのサーバコンピュータ」のことを一般に「サーバ名」、「何番の通信口」のことを「ポート番号」、と呼びます。サーバには複数の「通信口＝ポート」があり、番号で管理しています。サーバの側は、プログラムによって決まった番号のポートを監視しています。
- クライアントはサーバが監視するポート番号に「Socket」を送らなければなりません。

# ServerSocketクラス、Socketクラス

- **ServerSocketクラス** サーバの側に用いられます。インスタンス化の際に引数として「ポート番号(整数型)」を要求します。インスタンス化が済んだ時点でポートの監視が始まります。そのとき、もしもクライアントから接続要求(「Socketクラス(後ほど説明)」のインスタンスが送られてくる)があれば、「acceptメソッド」で受け取ります。この時点で通信の準備が完了します。
- **Socketクラス** クライアントの側に用いられます。インスタンス化の際に引数として「サーバ名(文字列型)、ポート番号(整数型)」の順に二つの引数を要求します。インスタンス化が行われた時点で接続要求がサーバに送られます。



# Objectのやりとり

- OutputStreamへのデータの書き込み → データの送信
- InputStreamからのデータの読み込み → データの受信
- **OutputStream/InputStreamの取得** OutputStream/InputStreamの取得を行うためには、Socketクラスのメソッドである「**getOutputStream/getInputStream**」メソッドを利用します。これらのメソッドが起動されると、返値として、取得したOutputStream/InputStreamのインスタンスが返されます。
- このインスタンスを引数にして、Streamを用いてデータの通信を行うクラスをインスタンス化することで、OutputStream/InputStreamが取得されデータ送受信の準備が完了します。
- 通信の際にやりとりするデータの型が「任意のクラスのインスタンス」の場合、データ通信は「**ObjectOutputStream/ObjectInputStream**」クラスを用いて行います。従ってgetOutputStream/getInputStreamメソッドの返値であるOutputStream/InputStreamインスタンスを引数として、これらのクラスのインスタンス化を行うことで、OutputStream/InputStreamが取得されデータ送受信の準備が完了します。

# ObjectOutputStream、 ObjectInputStream

- データ送信の準備 (但しObjectOutputStreamのインスタンス名をoos、Socketのインスタンス名をsocketとする)
- ObjectOutputStream oos =  
new  
ObjectOutputStream(socket.getOutputStream());
- データ受信の準備 (但しObjectInputStreamのインスタンス名をois、Socketのインスタンス名をsocketとする)
- ObjectInputStream ois = new  
ObjectInputStream(socket.getInputStream());



# OutputStreamへのデータの書き込み(送信)

- OutputStreamへのデータの書き込みは、「ObjectOutputStream」の「writeObject」メソッドを利用して行います
- データの書き込み(送信)
- `oos.writeObject(送信したいインスタンス名);`
- `oos.flush();`

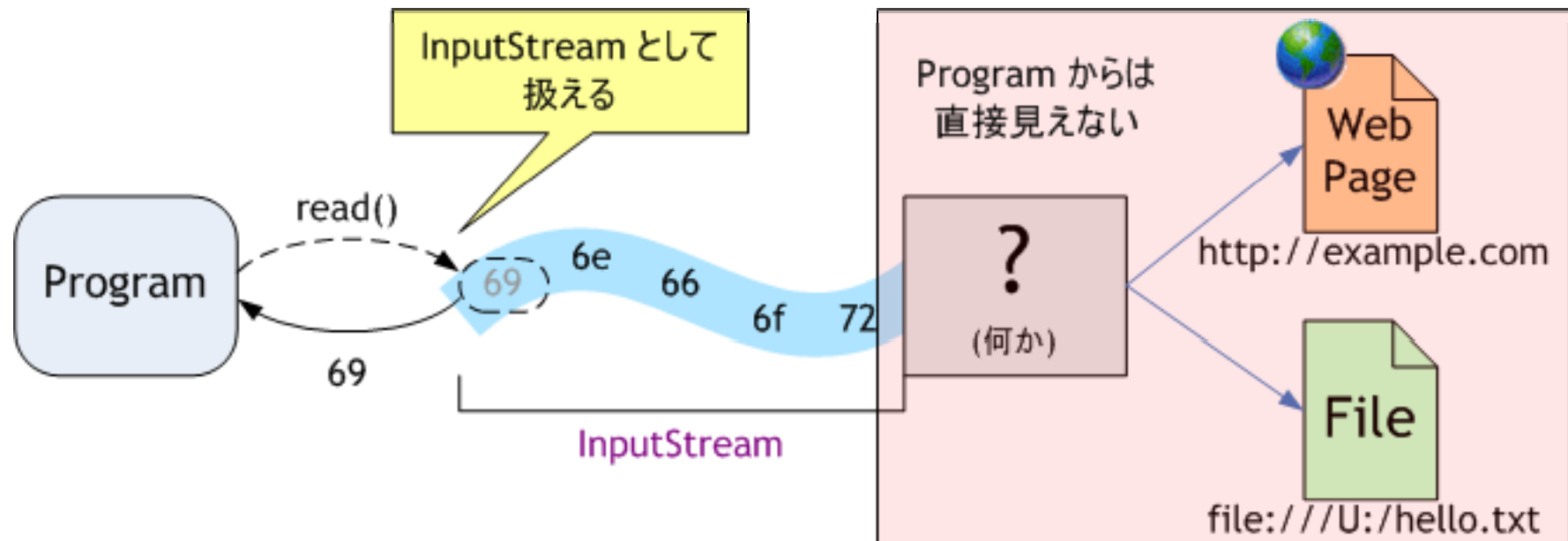
# InputStreamからのデータの読み込み(受信)

- InputStreamからのデータの読み込みも、ファイルからの読み込みと同様「ObjectInputStream」の「readObject」メソッドを利用します。
- データの読み込み(受信)(但し、ObjectInputStreamのインスタンス名をois、読み込むデータをVector型のインスタンスとする)
- `Vector vec = (Vector)ois.readObject();`

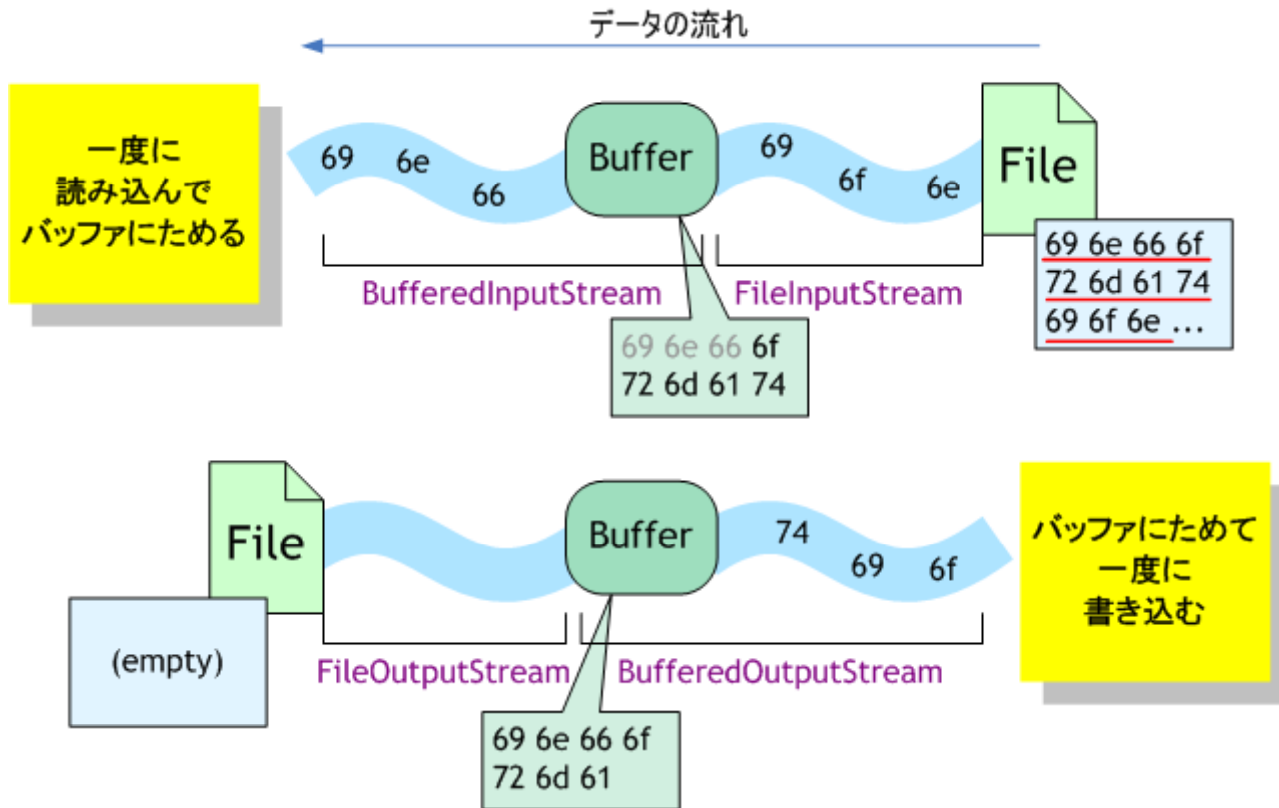
# 通信終了の合図

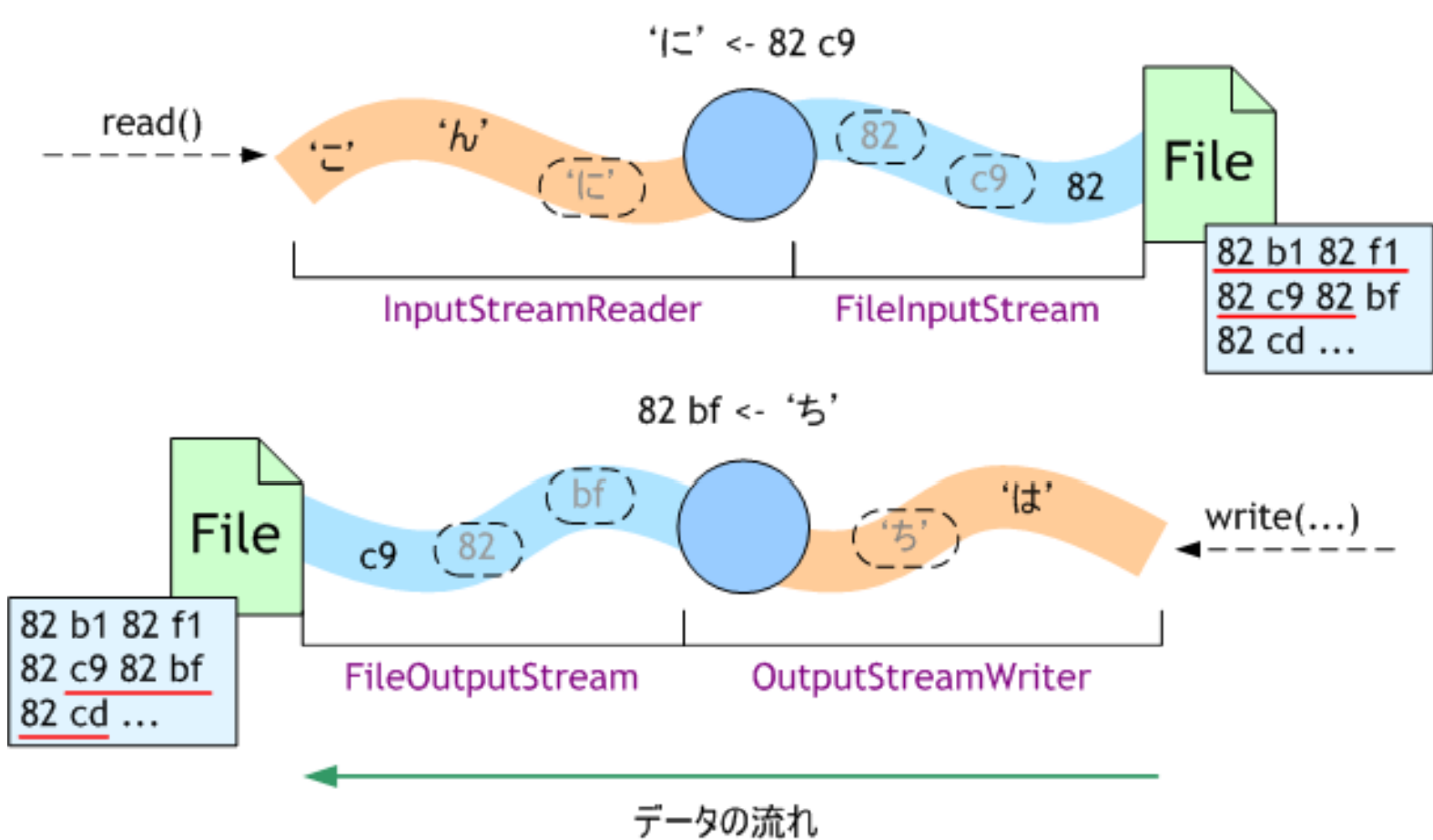
- 通信終了の合図(但し、ObjectOutputStreamのインスタンス名をoosとする)
- `oos.close();`
- `socket.close();`

# IO Stream

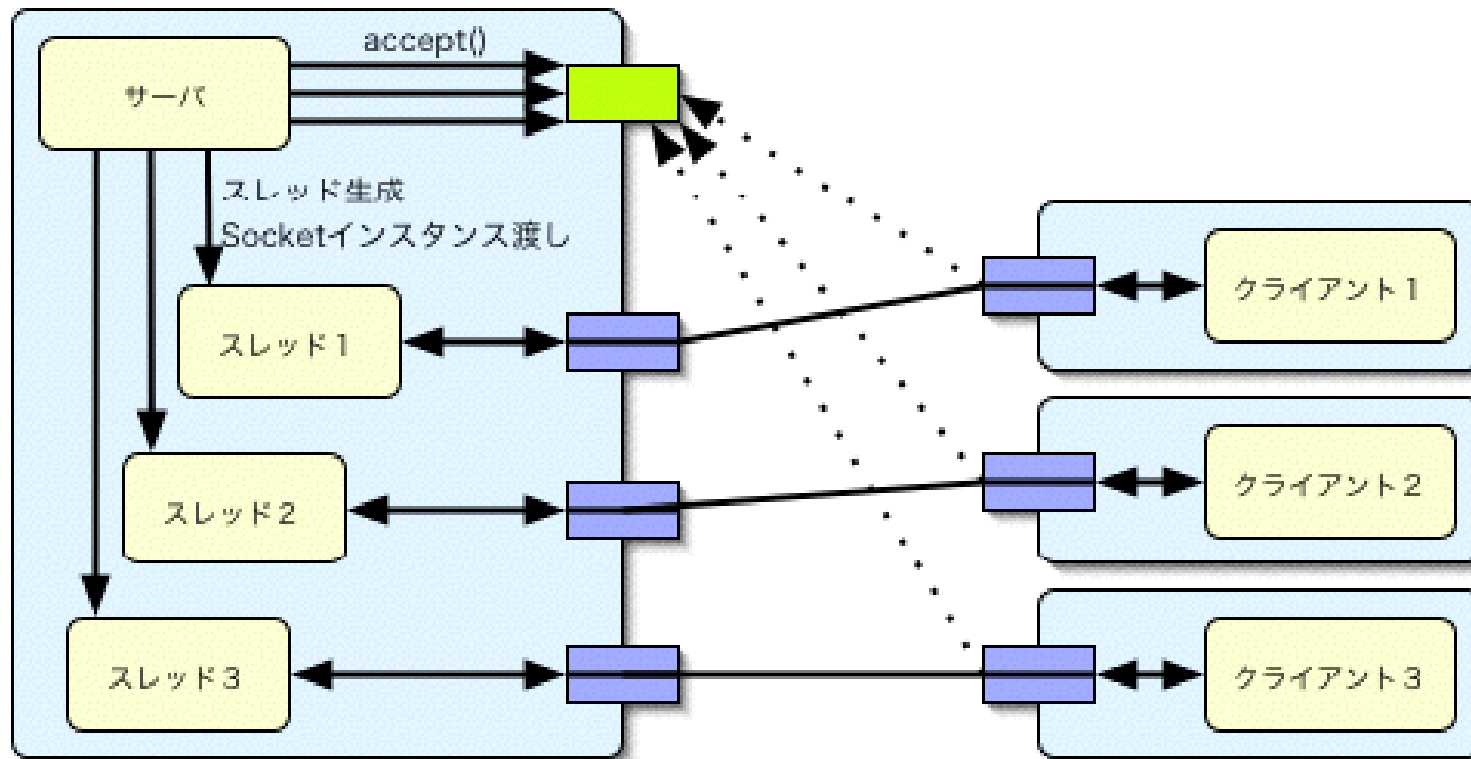


# BufferdI/OStream





# MultiThread



# 動かしてみよう

- CountTenRunnable



# try-catch

```
try {  
    例外のスロー  
} catch (Exception e ) {  
    例外のキャッチ  
}
```

# 例外を投げる

```
throw new Exception();
```

```
Public void method1 (int x) throws Exception  
{  
    throw new Exception();  
}
```

# 例外を投げる。

```
// NumberFormatException を捕捉するための try-catch
try {
    n = Integer.parseInt(input);
}
catch (NumberFormatException e) {
    // 数値の形式が不正である場合は、入力自体が不正
    throw new IllegalArgumentException("不正な入力 " + input);
}

if (n < 0) {
    // 負の値が入力された場合は、不正な入力
    throw new IllegalArgumentException("不正な入力 " + input);
}

System.out.println("入力された正の値は" + n);
```

# 例外クラス

```
public class IllegalArgumentException extends  
    Exception {  
    public IllegalArgumentException(String  
message) {  
        // 親クラスのコンストラクタにメッセージを  
渡す  
        super(message);  
    }  
}
```

# スタックトレース

- `e.printStackTrace();` を使ってみる

うごかしてみよう。

- MultiServerSample.java
- MultiClientSample.java