

# 上級プログラミング1(第7回)

---

工学部 情報工学科  
杉本 徹

# 復習：プログラムの構成(1)

プログラムは(1つ以上の)クラスとして構成される

```
class SampleClass01{
```

```
    static int i=10;  
    public static void main(String[] args){  
        add10(); System.out.println("i="+i);  
    }  
    public static void add10(){  
        i+=10;  
    }
```

```
}
```

# 復習:プログラムの構成(2)

クラス≡構造体メンバ(フィールド)  
+メンバを操作する関数(メソッド)

```
class SampleClass01{  
    static int i=10;  
    public static void main(String[] args){  
        add10(); System.out.println(i);  
    }  
    public static void add10(){  
        i+=10;  
    }  
}
```

フィールド

メソッド

# 復習:プログラムの構成(3:アプリケーションの場合)

クラスのmainメソッドが実行される  
(C言語のmain関数のようなもの)

```
class SampleClass01{  
    static int  i=10;  
    public static void main(String[] args){  
        add10(); System.out.println("i="+i);  
    }  
    public static void add10(){  
        i+=10;  
    }  
}
```

# 復習：クラスとオブジェクト

---

- プログラム上ではクラス(型、雛型)を定義
- プログラム実行時に、クラスの内容の「実体」(インスタンス)であるオブジェクトを生成して、それを操作することで計算を進めていく

クラス

オブジェクト

Person kimura=new Person();


... Personクラスのオブジェクトkimuraを生成

# 復習: コンストラクタ

---

- オブジェクトの初期化の処理を定義する
- クラス名と同じ名前を持つ
- デフォルトは引数なし
  - クラスにコンストラクタを定義しなければ引数なしのコンストラクタがデフォルトで定義される
  - 定義すれば引数によって処理を変えることも可能

コンストラクタ



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

# 参考：Javaにおける慣習

---

## □ クラス名は大文字で始める

■ 例：InputStreamReader クラス

（ラクダ記法 (camel case) : 頭文字だけ大文字にした複数の英単語をつなげる形の命名法）

## □ フィールド名, メソッド名, パッケージ名 (後述) などは小文字で始める

■ 例：toUpperCase() メソッド

## □ その他

■ 一般に, クラス名は名詞形, メソッド名は動詞形 等

---

# 今日のテーマ

---

- 継承とクラス
    - クラスの親子関係
    - コンストラクタ
  - インタフェース
    - 抽象クラスとインタフェース
  - パッケージ
  - スコープと修飾子(public,private,protectedなど)
-



# クラスの継承

---

オブジェクト指向プログラミングにおける  
重要な概念

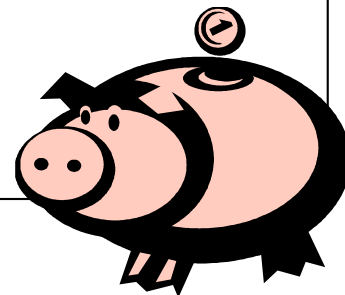
# まずプログラム（銀行預金と貯金箱）

---

```
class BankAccount {  
    int okane=0;  
    String bank = "TokyoBank";  
    void addOkane(int i){  
        okane += i;  
    }  
    void print(){  
        System.out.println(  
            bank+okane+"yen");  
    }  
}
```



```
class ChokinBako{  
    int okane=0;  
    boolean available=true;  
    void addOkane(int i){  
        okane += i;  
    }  
    void print(){  
        System.out.println(  
            okane+"yen "+available);  
    }  
    boolean getAvailability(){  
        return available;  
    }  
}
```



# プログラムを効率よく再利用したい

---

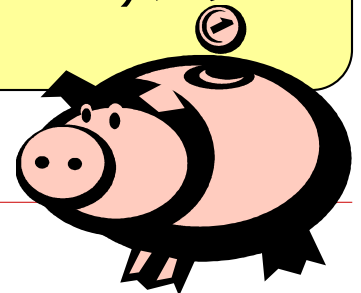
貯金クラス

雛型として利用する

銀行預金(BankAccount)クラス



貯金箱(ChokinBako)クラス



# 貯金クラス

---

```
class Chokin{  
    int okane=0;  
  
    void addOkane(int i){  
        okane += i;  
    }  
    void print(){  
        System.out.println(  
            okane+"yen");  
    }  
}
```

■ 共通のフィールドやメソッド  
を定義しておく



# 銀行預金クラスと貯金箱クラスを貯金クラスをつかって書くと(1)

Chokinクラスを  
拡張しているよ

```
class BankAccount extends Chokin{
```

```
    String bank = "TokyoBank";
```

```
    void print() {  
        System.out.println(  
            bank+okane+"yen");
```

```
    }
```

```
}
```

Chokinクラスにない  
フィールドを定義する

Chokinクラスと処理を変える  
場合は定義しなす

定義しなしていない  
Chokinクラスの機能は  
そのまま使える



# 銀行預金クラスと貯金箱クラスを貯金クラスをつかって書くと(2)

Chokinクラスを  
拡張しているよ

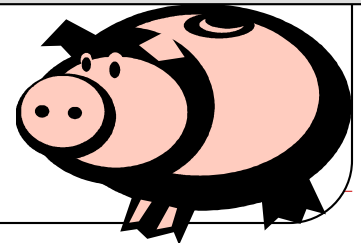
```
class ChokinBako extends Chokin{  
    boolean available=true;  
    void print(){  
        System.out.println(  
            okane+"yen "+available  
        );  
    }  
    boolean getAvailability(){  
        return available;  
    }  
}
```

Chokinクラスにない  
フィールドを定義する

Chokinクラスと処理を変える  
場合は定義しなおす

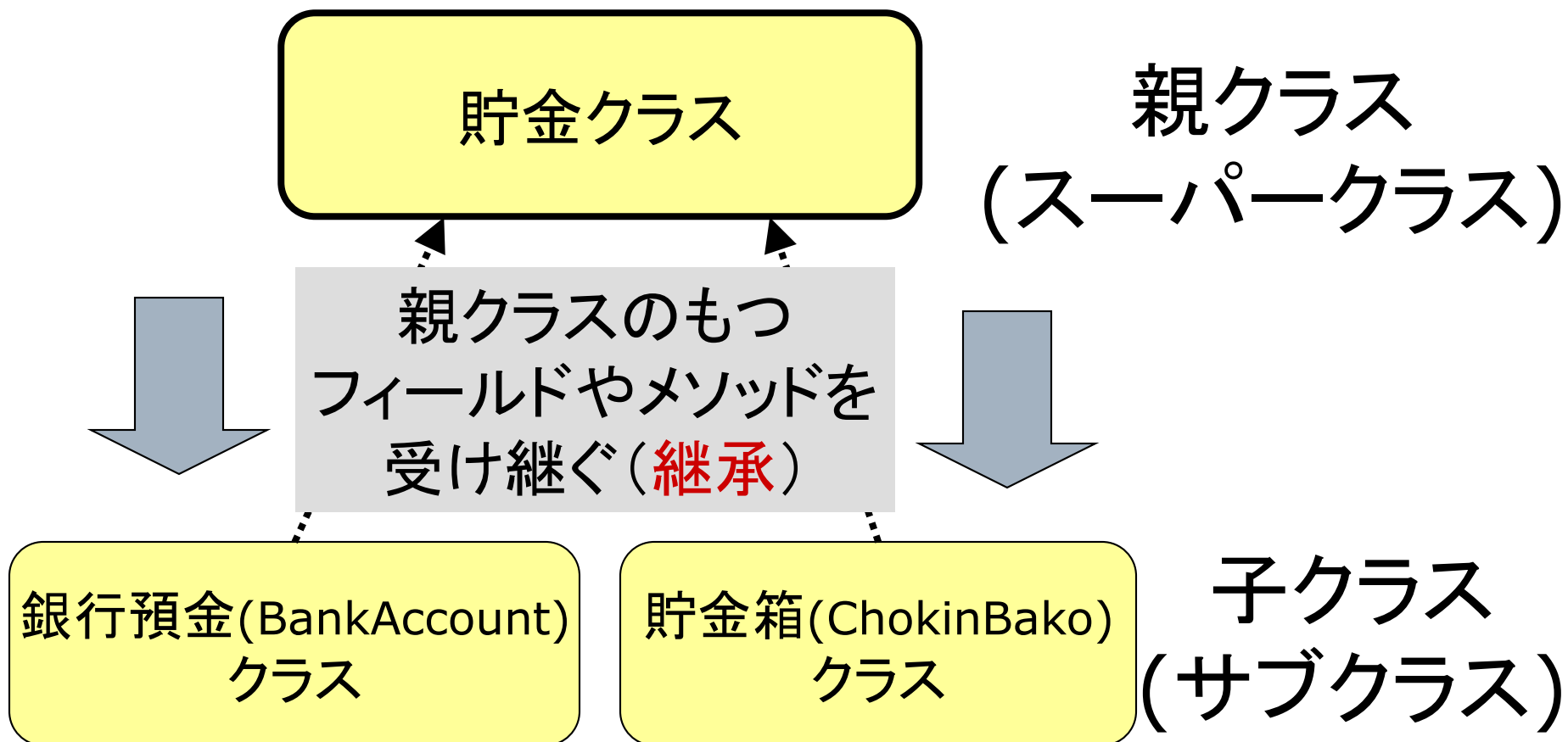
Chokinクラスにない処理  
を定義する

定義しなおしていない  
Chokinクラスの機能は  
そのまま使える



見方を変えると・・・(貯金クラスからみて)

---



# 子クラス(サブクラス)でできること

---

- 親クラスのフィールドやメソッドを受け継ぐ(継承)
  - 親クラスにないフィールドの定義を追加する
  - 親クラスにないメソッドの定義を追加する
  - 親クラスのメソッドを定義しなおす  
(メソッドの上書き, **オーバーライド**)  
オーバーロードと名前が似ていて間違えやすいので注意
-



# this変数

---

- this変数は、オブジェクト自分自身を表し、そのフィールドを明示したいときに使う

```
class Chokin{  
    int okane=0;  
    void addOkane(int i){  
        this.okane += i;  
    }  
}
```

okaneがこのクラスのフィールドであることを明示(この場合は this は省略可能)

---

# this変数が役に立つ場合

---

```
class Chokin{  
    int okane=0;  
    void addOkane(int okane){  
        this.okane += okane;  
    }  
}
```

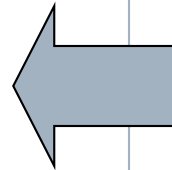
この例のように異なるものに同じ名前を付けるのはそもそも紛らわしく避けるべきであるが..

---

# コンストラクタ

## □ オブジェクトの初期化をするための、クラスと同じ名前を持つ処理のこと(前出)

```
class Chokin{  
    int okane=0;  
    Chokin(int i){  
        okane = i;  
    }  
    Chokin(){  
        this(10);  
    }  
    ....  
}
```



- クラスと同じ名前
- 戻り値はなし
- 引数の数や型が異なる複数のコンストラクタを定義してもよい(オーバーロード)
- 自分のクラスの他のコンストラクタを呼び出すときはthis()を使う。ただしthis()の前に他の処理を書かない。

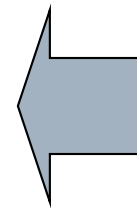
※前出のChokinクラスに変更を加えた

# 参考：デフォルトコンストラクタ

---

- コンストラクタが定義されていないときにはデフォルトコンストラクタとして引数なしのコンストラクタが暗に定義される

```
class Chokin{  
    int okane=0;  
    void addOkane(int i){  
        this.okane += i;  
    }  
    ....  
}
```



```
Chokin(){  
    //何もしない  
}
```

というコンストラクタ  
を定義しているのと同  
じ

---

※元のChokinクラスはこうだった

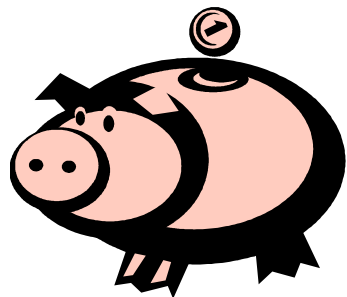
# コンストラクタ(継承している場合)-(1)

```
class ChokinBako extends Chokin{
```

```
    boolean available=true;
```

```
    boolean getAvailability()  
        return available;  
    }
```

```
}
```

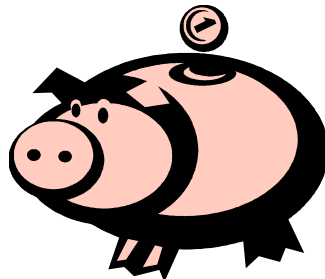


コンストラクタが定義されていない場合、  
親クラスのコンストラクタが自動的に呼び出される

```
ChokinBako buta=new Chokinbako()  
    となるとokaneに10がセットされる  
    (2ページ前をみよ)
```

# コンストラクタ(継承している場合)-(2)

```
class ChokinBako extends Chokin{  
    boolean available;  
    ChokinBako(int i){  
        super(i);  
        available=true;  
    }  
    boolean getAvailability()  
        return available;  
    }  
}
```



- 親クラスのコンストラクタを明示的に呼び出す場合はsuper()を使う。
- super()の前には処理を入れられない
- super()がないときは親クラスのデフォルトコンストラクタがコンストラクタの最初に自動的に実行される

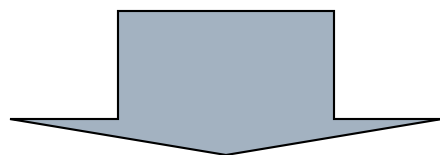
# インタフェース

---

# 抽象クラス

---

- 今まで雛型として親クラスを作ってきた
  - ただし、そうは言ってもそれなりの機能は持っていた
  - こんなメソッドを実装しなさいという約束事を定義するだけで機能そのものを実装しない雛型があってもよいのでは？（実際の機能は子クラスの実装に任せる）



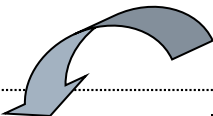
抽象クラス

---

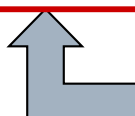


# 抽象クラス

abstractメソッドがある場合は抽象クラス



```
abstract class Chokin{  
    int okane = 0;  
    void addOkane(int i){  
        okane += i;  
    }  
    abstract void print();  
}
```



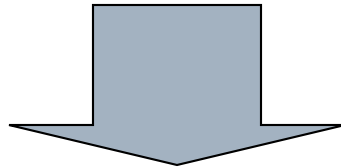
abstractメソッドは宣言だけで、処理内容の定義を行わない。  
子クラスでは必ず処理を定義する。

例えば ChokinBakoクラスで、printメソッドの処理内容を定義する

# インタフェース

---

- ……さてよ、Javaは単一継承。ということは、抽象クラスによる雛型は一つしか扱えない。不便だ！！



別の仕組み=インタフェースを使おう!

---

# インタフェース

```
interface Rect{  
    double getArea();  
}
```

```
interface Painted{  
    boolean isPainted();  
}
```

```
class ColoredRect implements Rect, Painted{  
    double a=3.0, b=5.5;  
    public double getArea()  
        { return a*b;}  
    public boolean isPainted(){ return false;}  
}
```

実装したいメソッドを定義

複数の”雛型”を指定可能

インタフェースに指定されたメソッドを実装

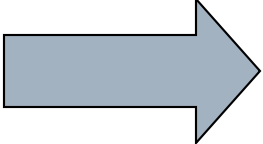
# インタフェースの用途

---

□ 単に雛型というだけでなく、

最低限このメソッドは実装しておけよ!

ということを保証するために利用される

 ある機能を実現するために  
前提となるメソッドが定義されている  
かどうかのチェックが可能  
(例えば、今後勉強する「スレッド」など)

---

# パッケージ

---

# パッケージ

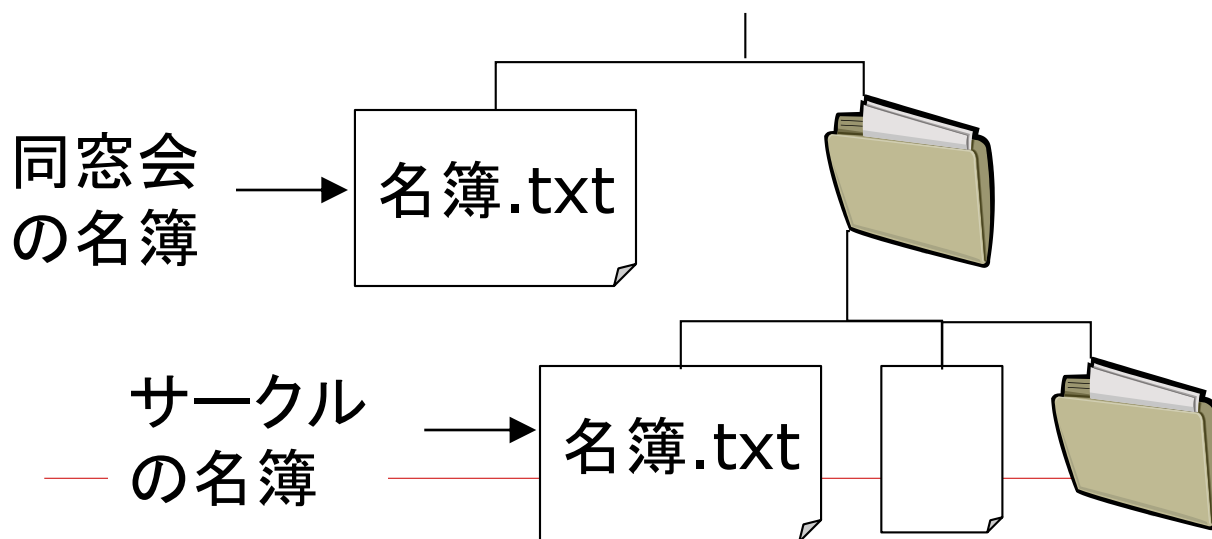
---

- 似た機能を持つクラスは同じ名前をつけたいくなる。
    - 車を現すクラスならCar、ファイルを扱うクラスならFileなど
  - だが、同じ名前のクラスを世の中で一つしか作れなかったら不便
    - ある人がFileクラスを作ってしまったら、自分でFileという名前のクラスを絶対作ってはいけないとしたら不便。  
・・・そこで
-

# パッケージ

---

- OSのファイル名の衝突と似ていることに注意
  - 同じ名前をもつファイルが一つのシステムで一つしか使えない場合、不便。
  - 別のディレクトリにおいておけばOK!



# パッケージ

com

shibaura

tools

system

Car.class

File.class

実際にこのようなディレクトリ構成でクラスファイルを保持するか、JARファイルを作成するかして、  
一つにまとめておく  
＝パッケージ

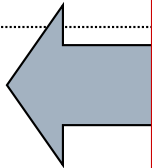


# パッケージ文

---

```
package com.shibaura.tools;
```

パッケージの  
どこに入るか宣言



```
class Car{  
    double speed=0.0;  
    void setSpeed(double ds){  
        speed=ds;  
    }  
    double getSpeed(){  
        return speed;  
    }  
}
```

# パッケージの利用(第1段階)

---

## □ クラスパスの設定

- JavaのVMにパッケージがどこにあるかを教えてあげないといけない
- パッケージの場所がわかれば、パッケージの中のパスをたどってクラスに到達可能
- Unix系OS(Linux、FreeBSD、Solaris、AIXなど)
  - sh系のshellでは  
`export CLASSPATH=/home/sugimoto/java/ : .`
  - csh系では  
`setenv CLASSPATH /home/sugimoto/java/ : .`

---

※パスはパッケージのあるディレクトリ  
“.” はカレントディレクトリ

# パッケージの利用(第2段階)

---

- import文を使って、パッケージ内のクラスのある場所を指定する(\*はそのディレクトリにあるクラス全てを指定)。

```
import com.shibaura.tools.*;

class MyClassA{
    Car myCar = new Car();
    myCar.setSpeed(50.0);
}
```

---

# アクセス修飾子

---

- メソッドやフィールドの宣言の前に、誰にまでその機能を公開するかを指定することができる
    - private : そのクラスのみ利用可
    - (指定なし) : 同じパッケージのクラスのみ利用可
    - protected : 同じパッケージのクラスおよびサブクラスのみ利用可
    - public : どのクラスからも利用可能
-

# アクセス修飾子の例

```
class PP{  
    private void showPrivate(){  
        System.out.println("Private");  
    }  
    public void showPublic(){  
        System.out.println("Public");  
    }  
}  
class MyClass04 {  
    public static void main(String[] args){  
        PP myPP = new PP();  
        myPP.showPrivate();  
        myPP.showPublic();  
    }  
}
```

他のクラスからは  
見えない

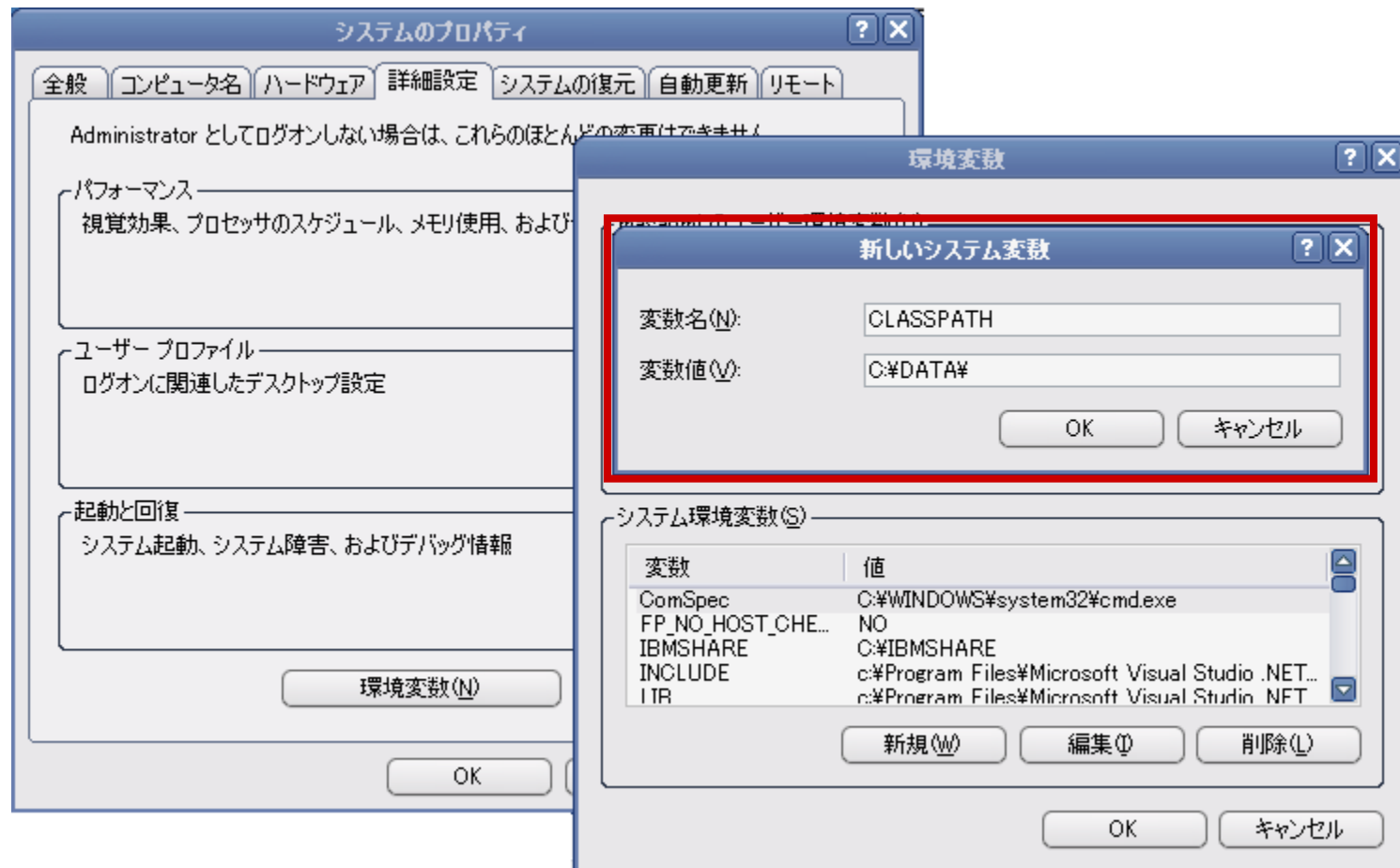
どのクラスからも  
見える

エラーが発生  
コメントアウトすると正しく  
動作

# 參考資料

---

# クラスパスの設定(Windowsの場合)



その他、javaコマンドのオプションで指定する方法あり