

上級プログラミング1(第7回)

工学部 情報工学科
木村昌臣

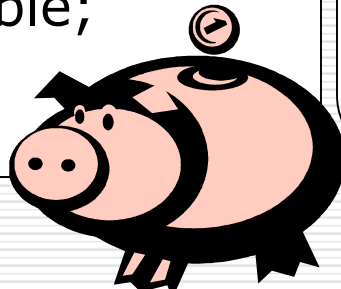
今日のテーマ

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

クラスの継承

まずプログラム(貯金箱と銀行預金)

```
class ChokinBako{  
    int okane=0;  
    boolean available=true;  
    void setOkane(int i){  
        okane+=i;  
    }  
    int getOkane(){  
        available=false;  
        return okane;  
    }  
    boolean getAvailability(){  
        return available;  
    }  
}
```



```
class Bank{  
    int okane=0;  
    void setOkane(int i){  
        okane+=i;  
    }  
    int getOkane(){  
        return okane;  
    }  
    int furikomi(int i){  
        okane=okane-i;  
        return i;  
    }  
}
```



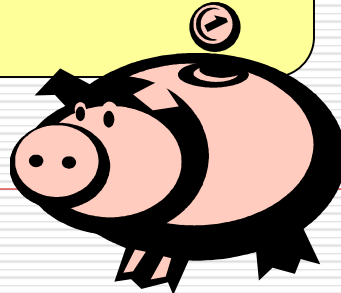
プログラムを効率よく再利用したい

貯金クラス

雛形として利用する

貯金箱(ChokinBako)クラス

銀行(Bank)クラス



貯金クラス

```
class Chokin{  
  
    int okane=0;  
  
    void setOkane(int i){  
        okane+=i;  
    }  
    int getOkane(){  
        return okane;  
    }  
}
```

■ 共通のフィールドやメソッドを定義しておく



貯金箱クラスと銀行クラスを貯金クラスをつかって書くと(1)

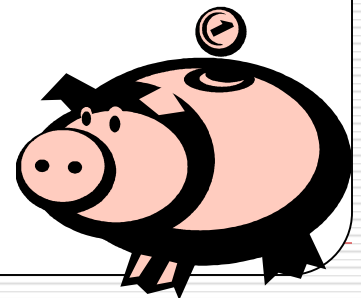
Chokinクラスを
拡張しているよ

```
class ChokinBako extends Chokin{  
    boolean available=true;  
    int getOkane(){  
        available=false;  
        return okane;  
    }  
    boolean getAvailability(){  
        return available;  
    }  
}
```

Chokinクラスにない
フィールドは定義する

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



貯金箱クラスと銀行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 Bank extends Chokin{
```

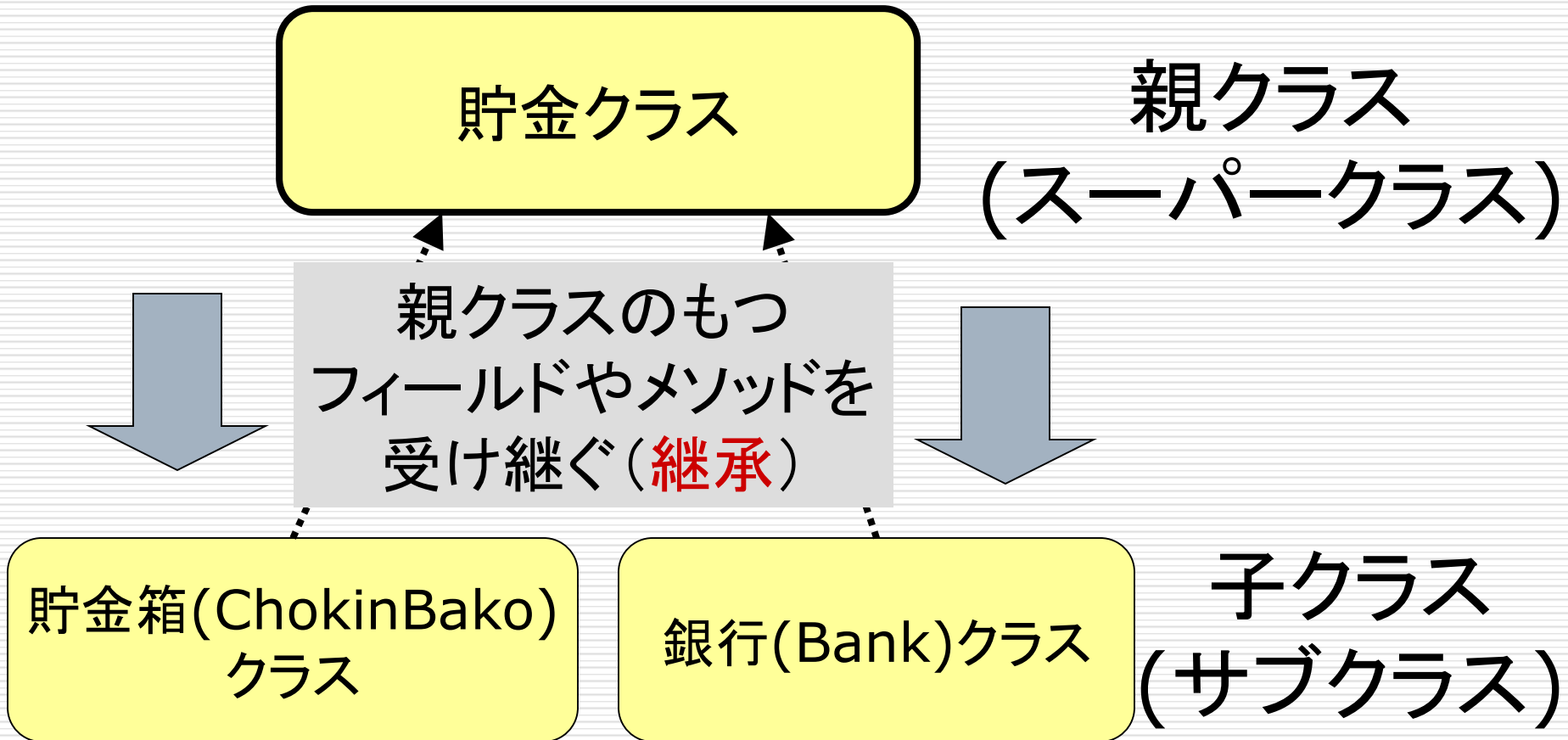
```
    int furikomi(int i){  
        okane=okane-i;  
        return i;  
    }  
}
```

Chokinクラスにない処理
は定義する

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)



オーバーライド

- 親クラスで定義されているメソッドを定義しなおすことができる。
 - ChokinクラスのgetOkaneメソッドでは残高を返すだけだった
 - それを継承したChokinBakoクラスのgetOkaneは残高を返すだけでなく使えなくなる (available=false)ように定義しなおしている
 - オーバーロードと名前が似ていて間違えやすい。注意。
-

this変数

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

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

メソッド内の変数でなくクラスのフィールドにアクセス

super変数

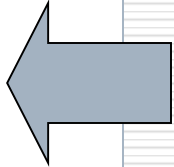
- super変数は、親クラス(のオブジェクト)のフィールドやメソッドにアクセスしたい場合に使う(親クラスのものであることを明示する)

```
class Bank extends Chokin{  
    int furikomi(int i){  
        super.okane=super.okane-i;  
        return i;  
    }  
}
```

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



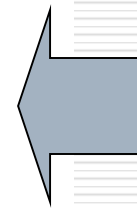
- クラスと同じ名前
- 戻り値はなし
- メンバ変数(フィールド)にアクセスするときには this 変数を使う
- 自分のクラスの他のコンストラクタを呼び出すときは this() を使う。ただし this() の前に他の処理を入れない。

※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 setOkane(int i){  
        this.okane+=i;  
    }  
    int getOkane(){  
        return this.okane;  
    }  
}
```



```
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;
```

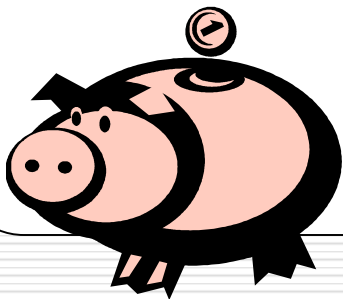
```
    int getOkane(){  
        available=false;  
        return okane;
```

```
    }
```

```
}
```

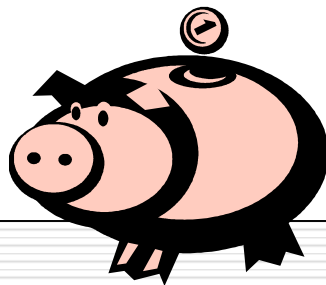
コンストラクタが定義されていない場合、
親クラスのコンストラクタが自動的に呼び出される

ChokinBako buta=new Chokinbako()
とすると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;  
    }  
    int getOkane(){  
        available=False;  
        return okane;  
    }  
}
```

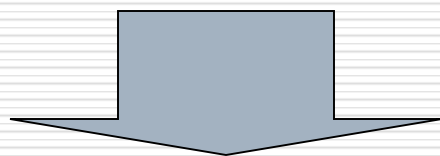


- 親クラスのコンストラクタを明示的に呼び出す場合はsuper()を使う。
- super()の前には処理を入れられない
- super()がないときは親クラスのデフォルトコンストラクタがコンストラクタの最初に自動的に実行される

インターフェイス

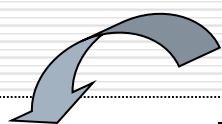
抽象クラス

- 今まで雛形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形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）



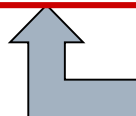
抽象クラス

抽象クラス



abstractメソッドがある場合は抽象クラス

```
abstract class Busho{  
    String name;  
    abstract void defName();  
}
```

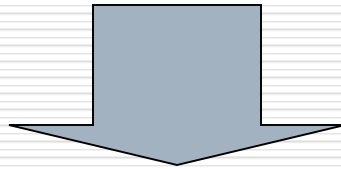


子クラスが指定したメソッドを必ず
実装するように定義

```
Class SengokuBusho extends Busho{  
    int val=100;  
    public void show(){ System.out.println(name);}  
    public void defName(){name=“織田信長”;}  
}
```

インターフェイス

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



別の仕組み=インターフェイスを使おう!

インターフェイス

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

実装したいメソッドを定義

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

複数の”雛形”を指定可能

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

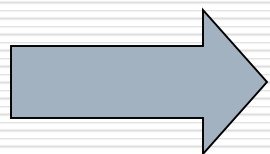
インターフェイスに指定されたメソッドを実装

インターフェースの用途

□ 単に雛形というだけでなく、

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)

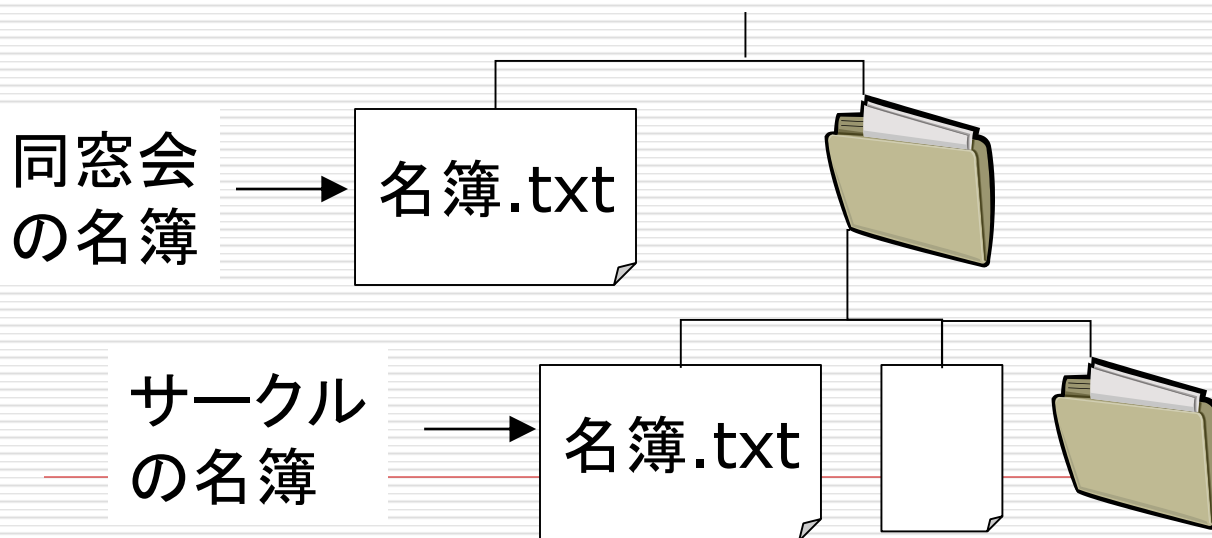
パッケージ

パッケージ

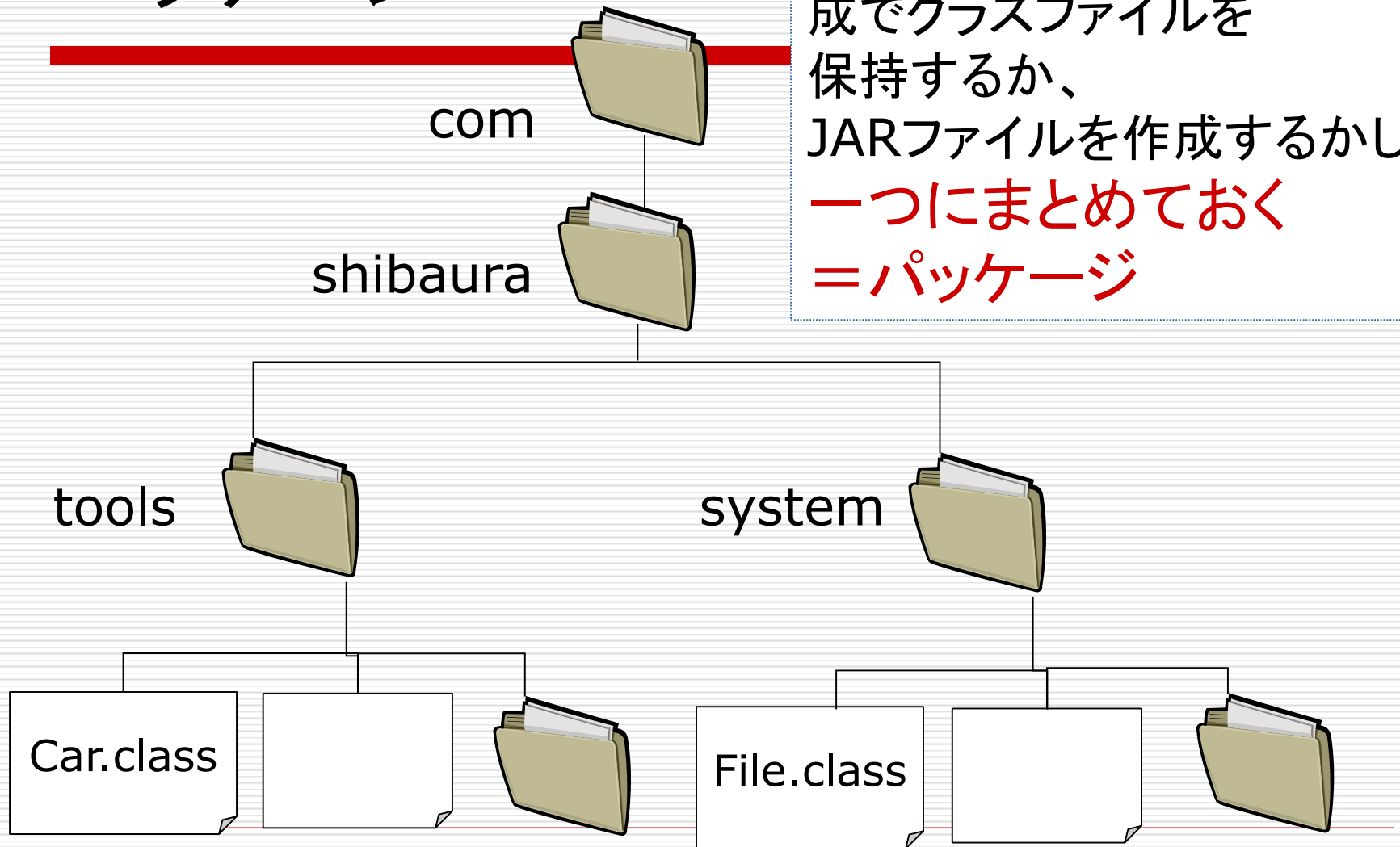
- 似た機能を持つクラスは同じ名前をつけたいくなる。
 - 車を現すクラスならCar、ファイルを扱うクラスならFileなど
 - だが、同じ名前のクラスを世の中で一つしか作れなかったら不便
 - ある人がFileクラスを作ってしまったら、自分でFileという名前のクラスを絶対作ってはいけないとしたら不便。
...そこで
-

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

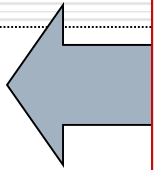


実際にこのようなディレクトリ構成でクラスファイルを保持するか、JARファイルを作成するかして、
一つにまとめておく
= パッケージ

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/kimura/java/ : .`
 - csh系では
`setenv CLASSPATH /home/kimura/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 : どのクラスからの利用可能
-

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();  
    }  
}
```

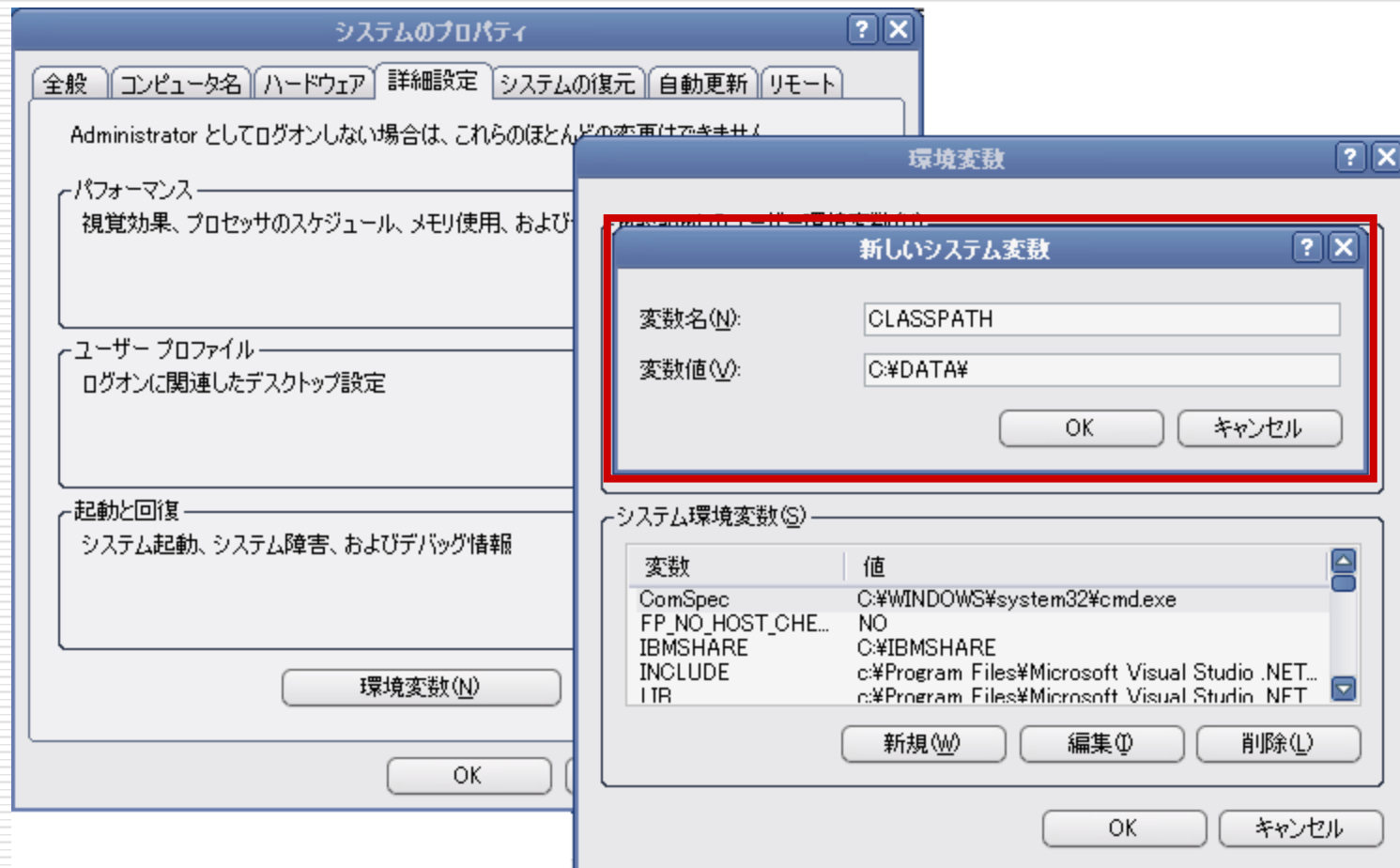
他のクラスからは
見えない

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