

# Javaプログラミング(4)

## クラスとインスタンス(2)

成蹊大学理工学部  
情報科学科

# クラスの宣言と構造(復習)

## クラスの宣言と構造

```
class クラス名 {  
    フィールドの宣言;  
    コンストラクタの宣言; //省略する場合もある  
    メソッドの宣言;  
}
```

- クラス名は大文字で始めること！
- フィールドとメソッドがクラスの性質を決める(鑄型の仕様書)
  - フィールド: クラスの属性情報, データ
  - メソッド: クラスの処理・操作
- コンストラクタ(インスタンスを作る鑄型)
  - インスタンスの初期化の方法を決めるもの.
  - インスタンスを作成する鑄型

# 科目クラスの全体像(復習)

- (例) 科目名とその点数をプリントするメソッドscorePrintを加えると, Kamokuクラスは以下のようなになる.

```
class Kamoku {  
    String subject;  
    int score;  
    Kamoku (String kamoku, int ten) {  
        this.subject=kamoku;  
        this.score=ten;  
    }  
    void scorePrint () {  
        System.out.println(this.subject+"の点数は"+this.score+"です");  
    }  
}
```

フィールド

コンストラクタ

メソッド

# Javaのオブジェクト指向

- Javaでのオブジェクトとはインスタンスのこと
- オブジェクトとは何か？
  - オブジェクト＝フィールド＋メソッド
  - オブジェクト＝情報＋操作
  - オブジェクト＝データ＋手続き(処理)
  - オブジェクト＝それは何か＋それをどうするか
- 「オブジェクト」「指向」＝オブジェクトを重視した考え方で設計されている言語

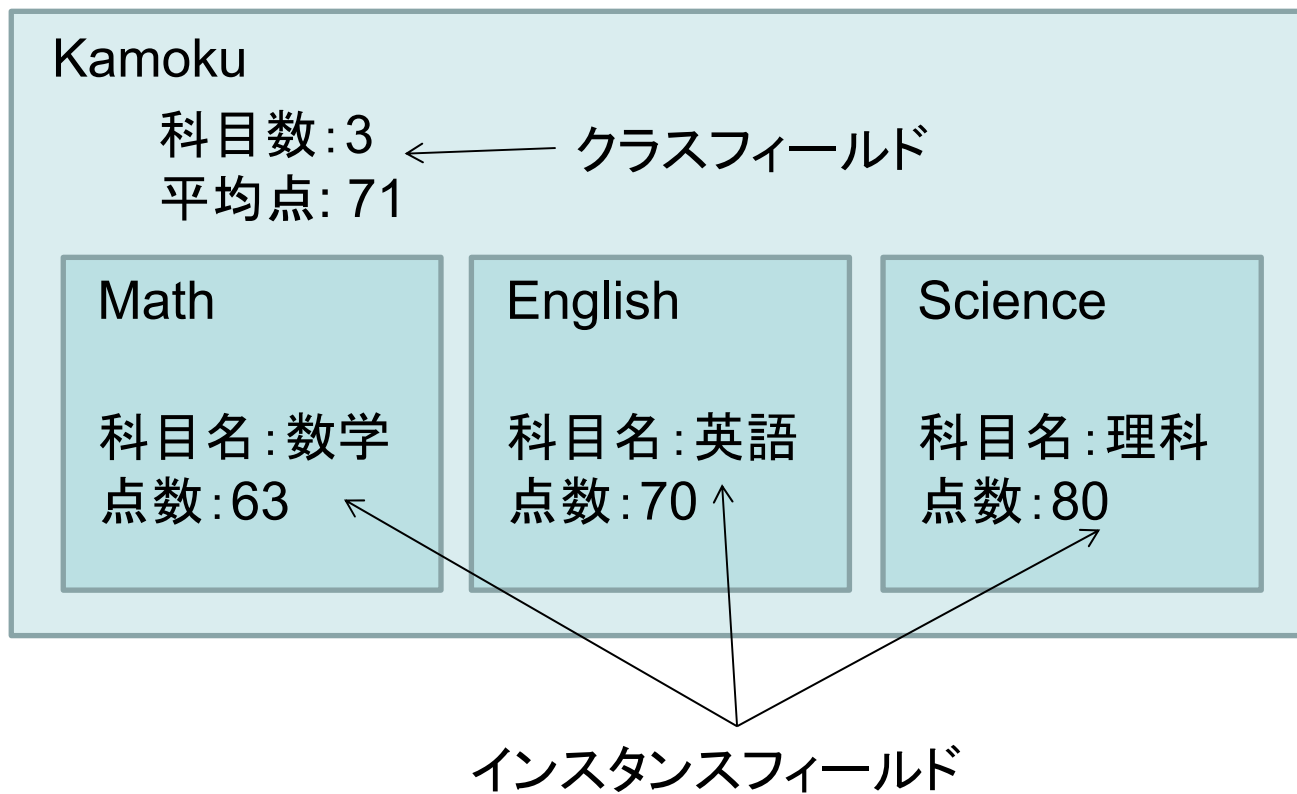
# クラスフィールド

- クラス内の共通の情報を保持しておく場所として、クラスフィールド（または、クラス変数、スタティックフィールド）を宣言することができる。

クラスフィールドの宣言
static 型 変数名; (例) static int kamokusuu

- クラスフィールド（クラス変数）
  - クラスの情報を保持
  - クラスに1つ
- インスタンスフィールド（インスタンス変数）
  - インスタンスの情報を保持
  - インスタンスごとに1つ

# クラスフィールド (Cont.)



# 科目クラスv2

- (例)これまでに作られたインスタンスの数を保持しておくクラス変数kamokusuuを定義する. kamokusuuはコンストラクタが呼ばれるたびに更新される.

```
class Kamoku {  
    static int kamokusuu=0;  
    String subject;  
    int score;  
    Kamoku (String kamoku, int ten) {  
        kamokusuu++;  
        this.subject=kamoku;  
        this.score=ten;  
    }  
    void scorePrint () {  
        System.out.println(this.subject+"の点数は"+this.score+"です");  
    }  
}
```

# クラスメソッド

- クラス変数に対して計算や処理を行うメソッドを「クラスメソッド」あるいは、「staticメソッド」ともいう. という.
- またインスタンス変数に対して処理を行うメソッドを「インスタンスメソッド」, あるいは「staticではないメソッド」という.
- mainメソッドもstaticメソッドである. これは, mainメソッドを動かすときにインスタンスを必要としないからである.



# 科目クラスv3

```
class Kamoku {  
    static int kamokusuu=0;  
    String subject;  
    int score;  
    Kamoku (String kamoku, int ten) {  
        kamokusuu++;  
        this.subject=kamoku;  
        this.score=ten;  
    }  
    void scorePrint () {  
        System.out.println(this.subject+"の点数は"+this.score+"です");  
    }  
  
    static int getKamokusuu() {  
        return kamokusuu;  
    }  
}
```

# 科目クラスv3(Cont.)

```
public class Heikin3 {  
    public static void main(String[] args) {  
        System.out.println("科目数:"+Kamoku.getKamokusuu());  
        Kamoku english = new Kamoku("英語", 63);  
        Kamoku mathematics = new Kamoku("数学", 92);  
        Kamoku science = new Kamoku("理科", 75);  
        english.scorePrint();  
        mathematics.scorePrint();  
        science.scorePrint();  
        System.out.println("科目数:"+Kamoku.getKamokusuu());  
    }  
}
```

実行結果

```
科目数:0  
英語の点数は63です  
数学の点数は92です  
理科の点数は75です  
科目数:3
```

# 既存クラスの利用

- JDKの中ですでに用意されているクラスのメソッドを利用することができる
- 既存クラスのstaticメソッドの例
  - クラス名.クラスメソッド  
(例) Integer.parseInt(String s)
- 既存クラスのstaticでないメソッドの例
  - オブジェクト名.メソッド  
(例) 文字列名.charAt(int i): 文字列のi番目の文字を返す  
(例) 文字列名.length(): 文字列の長さを返す
- ドット(. )の意味が違うことに注意！
- (復習)フィールド参照とメソッド適用を混同しないように！
  - 配列名.length lengthフィールドへのアクセス
  - 文字列名.length() length()メソッドの適用

# 修飾子

- クラス, フィールド, メソッド, コンストラクタには修飾子をつけることができる(後日再度正確なまとめを行う)
- アクセス修飾子
  - public:他のクラスからアクセス可能
  - private:他のクラスからアクセス不可
  - 修飾子なし:同じパッケージ(同じディレクトリ)内であればアクセス可能)
- static:クラス変数／メソッドであることを示す
- 改めてmainメソッドの意味  
`public static void main (String[] args) {}`

# 内部クラス（クラス内のクラス）

- クラス内部に定義されたクラスを内部クラス、またはインナークラスと呼ぶ

2つのクラスを並列して定義

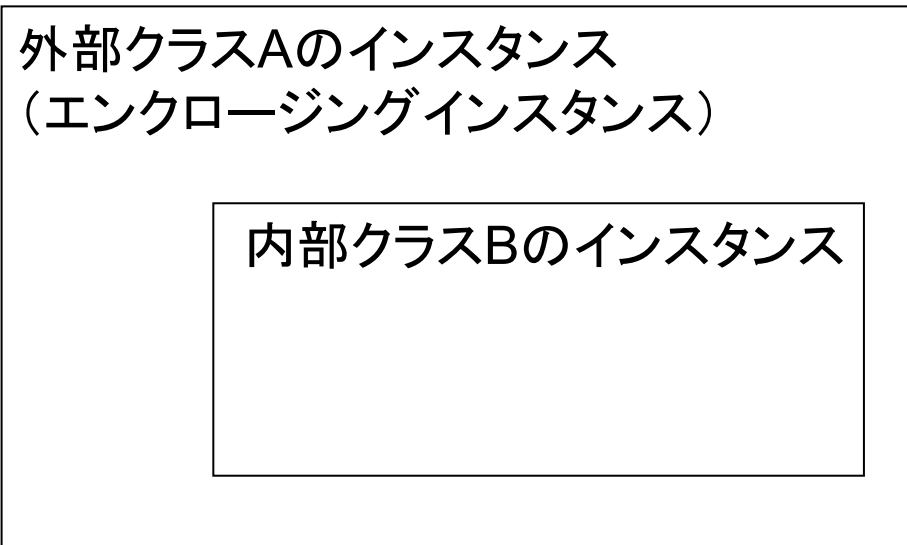
```
public class A {  
    ...  
}  
class B {  
    ...  
}
```

クラスBをインナークラスとして定義

```
public class A {  
    ...  
    class B {  
        ...  
    }  
}
```

# 内部と外部の関係

- 内部クラスのインスタンスは，外部クラスのインスタンス経由で作られる
- つまり，内部クラスのインスタンスは，外部クラスのインスタンスの中に作られる



# 内部クラスの特徴(1)

- 外側のクラス(外部クラス)から内部クラスを自由に使うことができる.
  - 内部クラス型変数の宣言, インスタンスの作成, private のフィールド／メソッドへのアクセスが可

2つのクラスを並列して定義

```
public class A {
```

```
...
```

```
}
```

```
class B {
```

```
    private フィールド1;
```

```
    private メソッド1 {}
```

```
}
```



クラスBを内部クラスとして定義

```
public class A {
```

```
...
```

```
    class B {
```

```
        private フィールド1;
```

```
        private メソッド1 {}
```

```
    }
```

```
}
```



# 内部クラスの例(1)

```
public class RectangleSample2 {  
    public static void main(String[] args) {  
        RectangleSample2 rs2= new RectangleSample2();  
        rs2.makeNewRectangles();  
    }  
    void makeNewRectangles() {  
        //内部クラスのインスタンス生成  
        MyRectangle mr1 = new MyRectangle();  
        //privateフィールドを参照  
        System.out.println("高さ:"+mr1.height+" 幅:"+mr1.width);  
        //privateフィールドを変更  
        mr1.height=30;  
        mr1.width=50;  
        System.out.println("高さ:"+mr1.height+" 幅:"+mr1.width);  
        System.out.println("mr1の面積  
        ="+mr1.height+"*"+mr1.width+"="+mr1.height*mr1.width);  
    }  
    ... つづく
```



# 内部クラスの例(1) (Cont.)

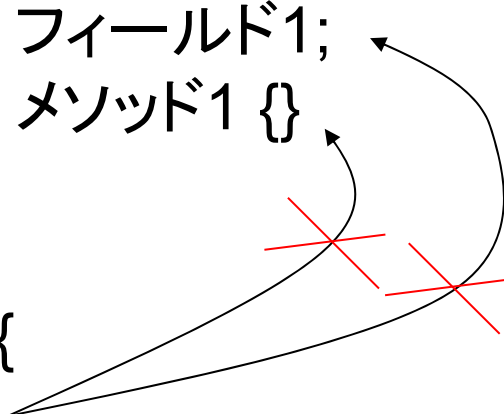
```
... つづき
//      内部クラスの宣言
      class MyRectangle {
          //フィールドの宣言
          private int width;
          private int height;
          //引数なしコンストラクタ
          MyRectangle () {
              width=10;
              height=20;
          }
      } //内部クラス定義ここまで
} //RectangleSample2の終わり
```

# 内部クラスの特徴(2)

- 同様に内部クラスは, 外側を自由に使うことができる.

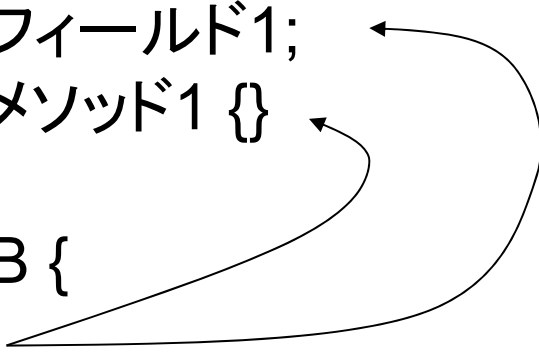
2つのクラスを並列して定義

```
public class A {  
    private フィールド1;  
    private メソッド1 {}  
    ...  
}  
class B {  
    ...  
}
```



クラスBを内部クラスとして定義

```
public class A {  
    private フィールド1;  
    private メソッド1 {}  
    ...  
    class B {  
        ...  
    }  
}
```



# 内部クラスの例(2)

```
class RectangleSample2 {  
    private int counter=0;  
  
    public static void main(String[] args) {  
        RectangleSample2 rs2= new RectangleSample2();  
        rs2.makeNewRectangles();  
    }  
  
    void makeNewRectangles() {  
        System.out.println("個数:"+counter);  
        MyRectangle mr1 = new MyRectangle();  
        System.out.println("高さ:"+mr1.height+" 幅:"+mr1.width);  
        System.out.println("個数:"+counter);  
    }  
}
```

# 内部クラスの例(2) (Cont.)

```
.... つづき
//      内部クラスの宣言
class MyRectangle {
    //フィールドの宣言
    private int width;
    private int height;
    //引数なしコンストラクタ
    MyRectangle () {
        counter++; //外部クラスのフィールドを変更
        width=10;
        height=20;
    }
} //内部クラス定義ここまで
}
```

実行結果

個数:0  
高さ:20 幅:10  
個数:1

# 内部クラスの使用

- あるクラスが特定のクラスに強く依存している場合、内部クラスにする.
- 内部クラスを使うことにより、プログラムを短くすることができる.
- しかし、クラス間の関係を十分吟味することが重要.