知識プログラミング方法論

第1～3回：クラス#1～3

ライフスタイルデザイン研究センター
金井 秀明

オブジェクト指向プログラミング

従来手法

- 「機能」に着目し，実現する機能を定義し，細かい機能に分割してソフトウェアを開発していた.
- 構造分析・設計手法

手続き型プログラミング

- コンピュータでやりたいことを，「データ」と「作業」という切り口で分析.
- 「データ」に対して，目的に合わせて「作業」（＝「処理」）を適用してプログラムを組み立てていく.

○ オブジェクト指向

- 「オブジェクト＝モノ」単位で考えるソフトウェア開発手法
- 機能を一枚岩と捉えず，データと手続き（振る舞い）をもった「オブジェクト」の集まりとして捉える．

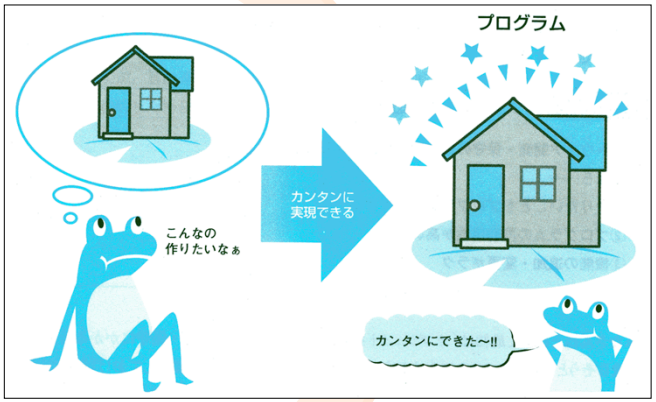
オブジェクト指向：Object Oriented
オブジェクト指向プログラミング：Object Oriented Programming(OOP)

○ オブジェクト指向プログラミング

- コンピュータでやりたいことを，「現実にあるモノ」という切り口で分析．
- 「現実にあるモノ」という切り口で，「データ」と「振る舞い」（＝「処理」）を括ってプログラムを組み立てていく．
- 「オブジェクト」とは「現実にあるモノ」という切り口・単位でまとめた「データ」と「処理」のかたまり．

○ OOPのメリット#1

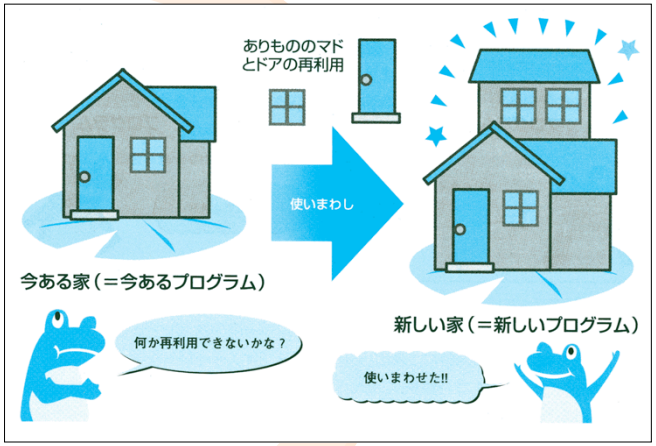
- プログラム化しやすい



出典：立山秀利「Javaのオブジェクト指向がゼッタイにわかる本」秀和システム

○ OOPのメリット#2

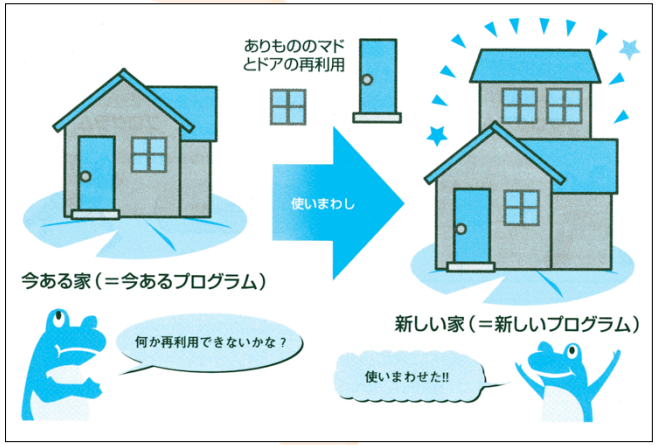
- 再利用性の向上



出典：立山秀利「Javaのオブジェクト指向がゼッタイにわかる本」秀和システム

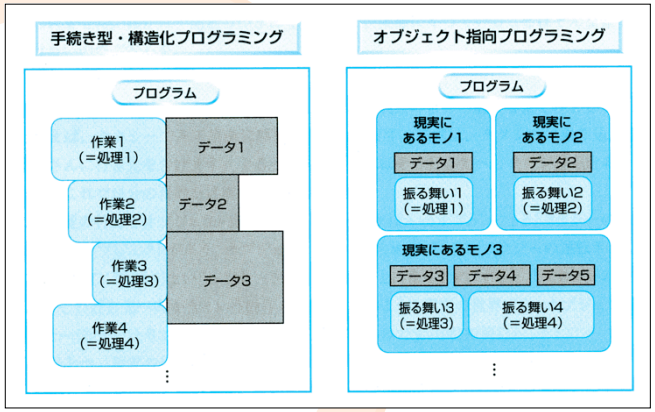
○ OOPのメリット#3

- 追加・変更がラク



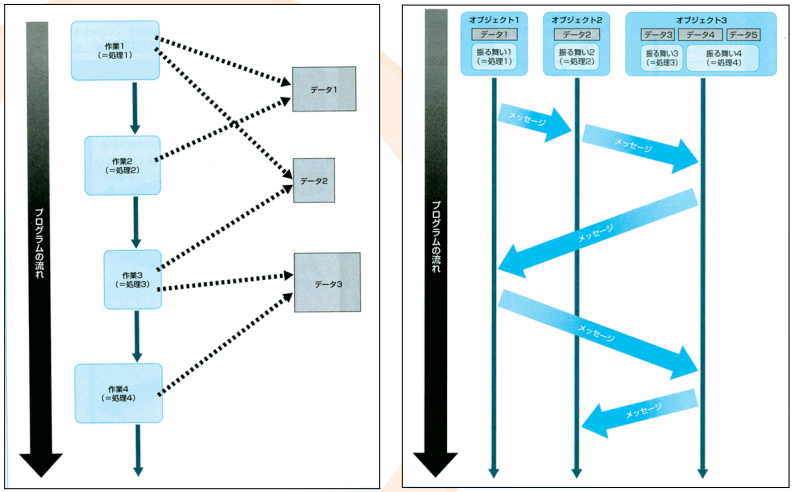
出典：立山秀利「Javaのオブジェクト指向がゼッタイにわかる本」秀和システム

○ 手続き型 v.s. OOP #1



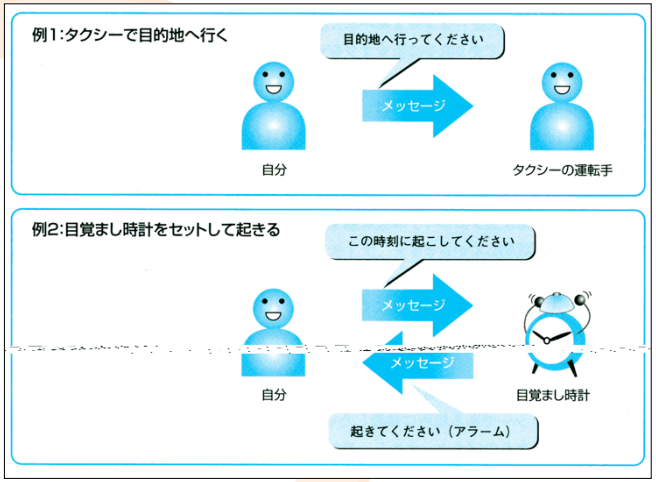
出典：立山秀利「Javaのオブジェクト指向がゼッタイにわかる本」秀和システム

○ 手続き型 v.s. OOP #2



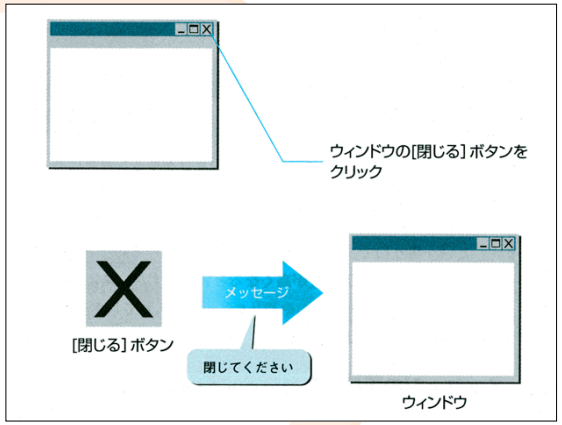
出典：立山秀利「Javaのオブジェクト指向がゼッタイにわかる本」秀和システム

○ 実世界のモノ間のメッセージ



出典：立山秀利「Javaのオブジェクト指向がゼッタイにわかる本」秀和システム

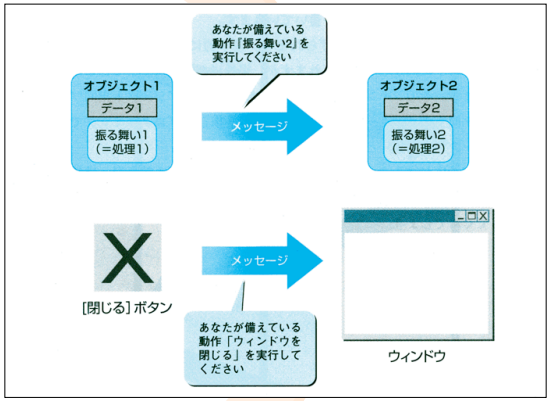
○ オブジェクト間のメッセージ



出典：立山秀利「Javaのオブジェクト指向がゼッタイにわかる本」秀和システム

○ メッセージのやりとり

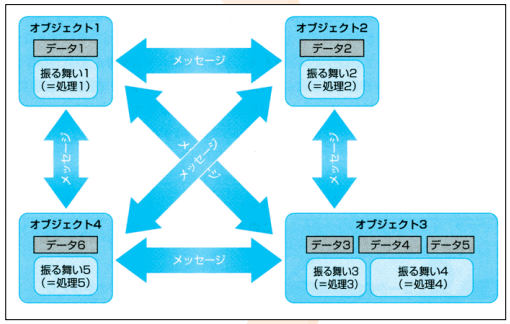
- あるオブジェクトが他のオブジェクトに「この動作をやってください」と"お願い"すること.



出典：立山秀利「Javaのオブジェクト指向がゼッタイにわかる本」秀和システム

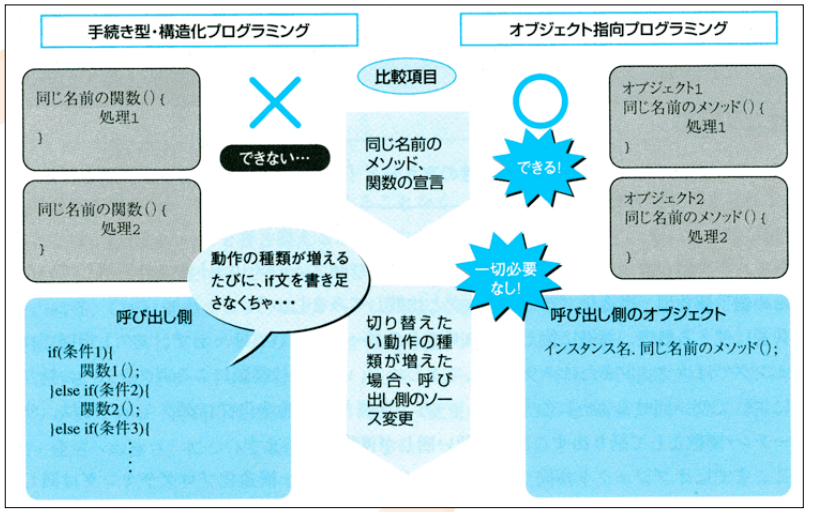
○ メッセージのやりとり

- 各オブジェクト同士がお互いに「メッセージ」をやり取りし，プログラムが実行される.



出典：立山秀利「Javaのオブジェクト指向がゼッタイにわかる本」秀和システム

○ 手続きとの比較



出典：立山秀利「Javaのオブジェクト指向がゼッタイにわかる本」秀和システム

○ OOPの特徴

- クラスとインスタンス：第4,5回
- カプセル化 ：第6回
- 多相性（ポリモフィズム）：第6回
- 継承（インヘリタンス）：第6回

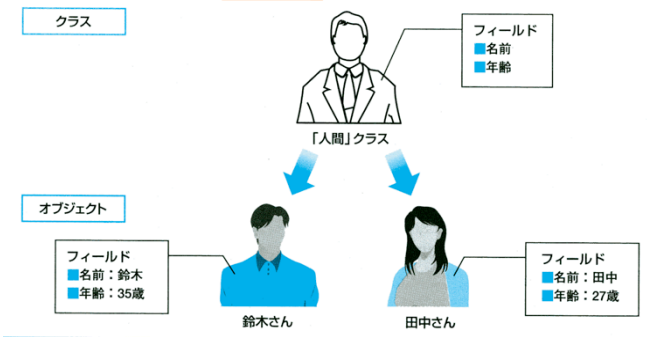
○ クラスとオブジェクト

- クラス：同じ特性をもつモノの集合に名前を付けたもの．オブジェクトの特性を抽象化したもの．
- オブジェクト（インスタンス）：クラスの実体，例

○ フィールドとメソッド

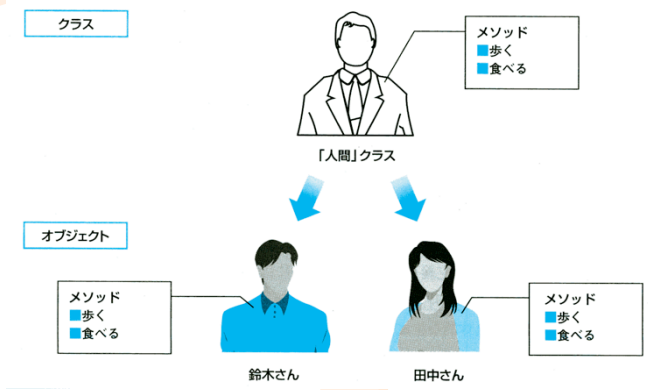
- フィールド：クラスの「性質」（属性）（それは何か？）
- メソッド：クラスの「機能」，操作，手続き．（それをどうするのか？）

○ 例：「人間」クラス



出典：立山秀利「Javaのオブジェクト指向がゼッタイにわかる本」秀和システム

○ 例：「人間」クラス



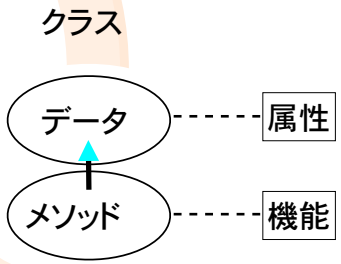
出典：立山秀利「Javaのオブジェクト指向がゼッタイにわかる本」秀和システム

○ カプセル化#0

○ カプセル化#0

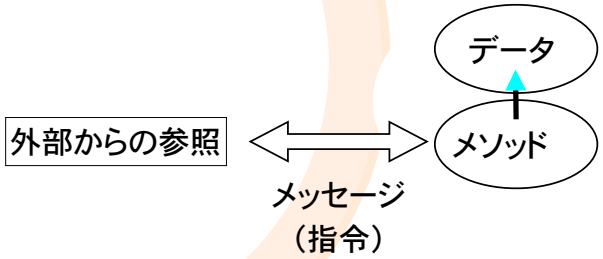
○ カプセル化#1

- 属性（データ，フィールド）と機能（その操作：メソッド）を一塊として，外部からのアクセスによる誤りがないように保護すること



カプセル化#2

- クラス宣言では，極力データを外部から直接参照できないようにし，メソッドを通してデータへのアクセスをするように設計する．



カプセル化#3

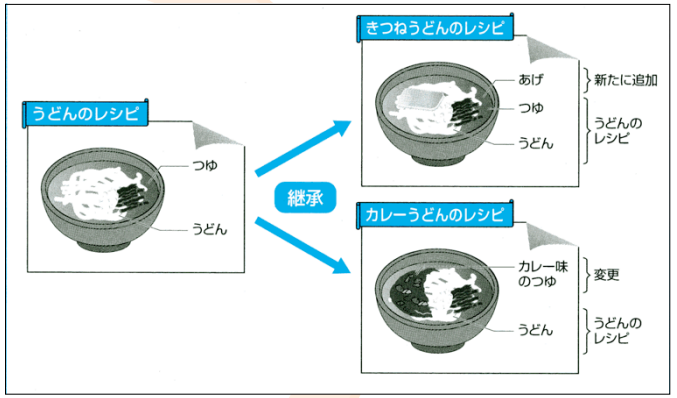
- 他のオブジェクトに「使っていいよ」と公開するフィールドとメソッドを最小限に抑える．
- 公開するなら，操作できる範囲を最小限に抑える．
- 他のオブジェクトはメソッドの中身をしらなくとも実行できる．

継承(inheritance)#1

- あるクラスの属性や機能を，他のクラスが引き継ぐこと．
- 継承したクラスでは，継承した属性や機能を利用することも，変更して異なった振る舞いをするように加工することもできる．
- 「似てるけど少しだけ違う」というプログラムをより簡単に作るためのしくみ．

継承(Inheritance) #2

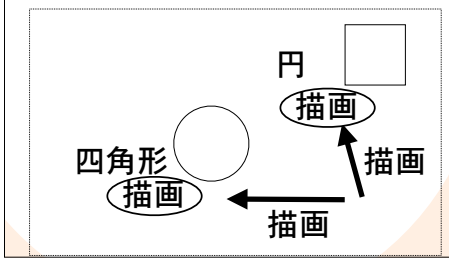
- 例

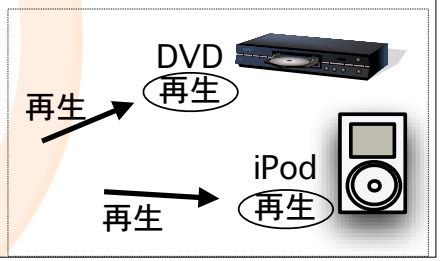


出典：立山秀利「Javaのオブジェクト指向がゼッタイにわかる本」秀和システム

○ 多相性(Polymorphism)#1

- オブジェクトへのメッセージ（指令）を統一すること.
- 同一メッセージであっても，オブジェクトに応じた振る舞いをするようにすること.





○ 多相性(Polymorphism)#2

- 同じ名前のメソッドに対して，それぞれ異なる振る舞いを持たせることができる.
- **呼び出す側のソースを全く変更することなく**，プログラムの動作を切り替えることができる.

○ 多相性の実現

- オーバーロード（多重定義）
  - 同一クラス内で同一名のメソッドを複数定義できる.
- オーバーライド
  - スーパークラスで定義されたメソッドと「同じメソッド名・引数の数・型」をもつメソッドをサブクラスで定義できる.

○ クラスとインスタンス

○ クラスの定義

- クラス：**フィールド**と**メソッド**を持つ
- フィールド：クラスに属する「変数」
- メソッド：クラスに属する「処理」

```
class クラス名 {
    フィールド

    メソッド
}
```


○ フィールドの宣言

- フィールドは変数なので．フィールドの宣言は，変数宣言と同じ．

```
型名 フィールド名;
型名 フィールド名1, フィールド名2, …, …;
```


○ メソッドの書式

```
修飾子 戻り値型 メソッド名(引数1, 引数2, ...) {
    メソッド内の処理
}
```


○ 例：「人間」クラス


```
class Person {
        String  name;   //名前
        int age;  //年齢
}
```

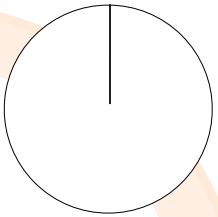
出典：立山秀利「Javaのオブジェクト指向がゼッタイにわかる本」秀和システム

クラス「Car」の例

属性
・車種
・ナンバープレート

```
class Car
{
        String type;  // 車種
        String license_plate;  //ナンバプレート
}
```


クラス「円」の例



属性
・中心座標(x,y)
・半径

```
class Circle
{
        double x;   // 中心のX座標
        double y;  // 中心のY座標
        double r;  //  半径
}
```

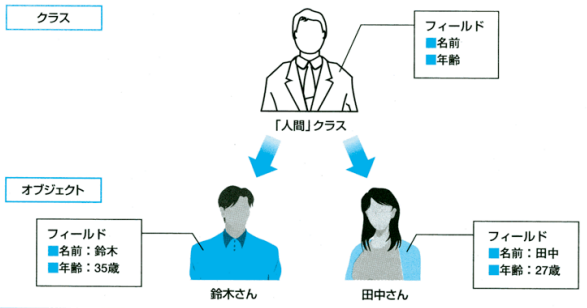

クラス「書籍情報」

属性
・タイトル
・ISBN コード

```
class  BookInfo
{
        String title;   // タイトル
        String isbn;  // ISBNコード
}
```


オブジェクト

- クラスから生成された実体
- インスタンスともいう



出典：立山秀利「Javaのオブジェクト指向がゼッタイにわかる本」秀和システム

オブジェクトの作成

- オブジェクトを扱う変数を宣言する.
- オブジェクトを作成し，その変数を扱えるようにする.

クラス名 変数名
変数名= new クラス名

クラス名 変数名= new クラス名

「人間」の場合

クラス宣言

```
class Person {
        String  name;  //名前
        int age;  //年齢
}
```

オブジェクトの作成

```
Person p1;
p1 = new Person();
```

```
Person p1 = new Person();
```


フィールドへの操作

- オブジェクトをさす変数名．フィールド名

```
Person p1 = new Person();
p1.name = ”田中”;
p1.age = 27;
System.out.println(p1.name+” ”+p1.age+”才 ” );
```


クラス「Car」

属性
・車種
・ナンバプレート

「Car」のオブジェクト

Car1

属性
・車種: 乗用車
・ナンバプレート:金沢33い1000

Car2

属性
・車種:特殊作業車
・ナンバプレート:金沢88あ8888

「Car」の場合

クラス宣言

```
class Car
{
        String type;  // 車種
        String license_plate; //ナンバプレート
}
```

オブジェクトの作成

```
Car car1;
car1 = new Car();
```

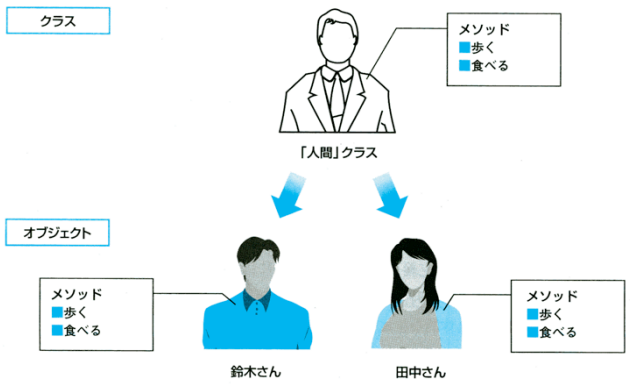
```
Car car1 = new Car();
```


フィールドへの操作

- オブジェクトを示す変数名．フィールド名

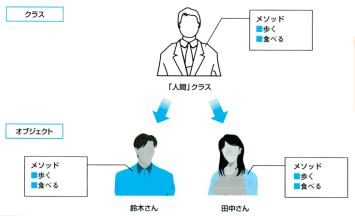
```
Car car2 = new Car();
car2.type = ”特殊作業車”;
car2.license_plate = ”金沢88あ8888”;
System.out.println(car2.type+  +car2.license_plate);
```


例：「人間」クラス



出典：立山秀利「Javaのオブジェクト指向がゼッタイにわかる本」秀和システム

クラス「人間」の宣言




```
class Person {
        String  name;   //名前
        int age;  //年齢

        public void walk( ) {...}
        public void  eat() {...}
}
```

クラス「Car」

メソッド
・車種とナンバプレートを表示する

「Car」のオブジェクト

Car1

メソッド
・車種とナンバプレートを表示する

Car2

メソッド
・車種とナンバプレートを表示する

メソッドの例

- 「車クラスのフィールド値を出力する」

```
class Car
{
  String type;  // 車種
  String license_plate; //  ナンバプレート

  void showInfo() {
      System.out.println(”車種は ”+type);
      System.out.println(”ナンバプレートは ”+license_plate);
}
```


引数なし

```
class Car {
   String type; String license_plate;
   void showInfo() {
      System.out.println(”車種は”+type);
      System.out.println(”ナンバプレートは”+license_plate);
   }
}
```

```
Car car1 = new Car();
car1.type = ”乗用車”;
car1.license_plate = ”金沢33い1000”;
car1.showInfo();
```


練習1

- 長方形を表すクラス「**Rectangle**」を作成せよ
  - フィールド：**int width** // 幅
  - フィールド： **int height** //高さ
  - メソッド：長方形の面積を戻り値とする
    getArea()
- 作成したクラスで，長方形の面積を求めるコードを書き，実行を確認せよ

練習2

- 次のように，整数値の座標を表すMyPointクラスを作成せよ．
  - フィールド：X （X座標），Y（Y座標）
  - メソッド：void setX(int px) （X座標を設定する）
  - メソッド：void setY(int py) （Y座標を設定する）
  - メソッド：int getX() （X座標を得る）
  - メソッド：int getY() （Y座標を得る）

コンストラクタ

クラスの定義

- フィールド：クラスに属する「変数」
- メソッド：クラスに属する「処理」
- コンストラクタ：インスタンスの「初期化処理」

```
class クラス名 {
    フィールド
    コンストラクタ
    メソッド
}
```


コンストラクタ#1

- インスタンスが作成（new）されたときに，最初に呼ばれる特殊なメソッド．インスタンスの「初期化処理」
  - コンストラクタを省略しても，引数のないデフォルトコンストラクタが自動的によばれる．

```
<クラス名>(<引数>) {
    コンストラクタで行う処理
}
```

コンストラクタ#2

- メソッドとの相違点
  - コンストラクタには**戻り型がない**.
  - コンストラクタの名前はクラス名と同じ.

例：「人間」クラス


```
class Person2 {
  String  name;   //名前
  int age;  //年齢

  void showInfo() {
    System.out.println(name+" "+age+"才 ");
}
```

出典：立山秀利「Javaのオブジェクト指向がゼッタイにわかる本」秀和システム

コンストラクタの例1

```
class Person2 {
  String  name;   //名前
  int age;  //年齢

  Person2() {
    System.out.println("コンストラクタが呼ばれました ");
    name = "名無しさん ";
    age = 10;
  }
  void showInfo() {
    System.out.println(name+" "+age+"才 ");
}
```


コンストラクタの例2

```
class Person3 {
  String  name;   //名前
  int age;  //年齢

  Person3(String iname, int iage) {
    System.out.println("コンストラクタが呼ばれました ");
    name = iname;
    age = iage;
  }
  void showInfo() {
    System.out.println(name+" "+age+"才 ");
}
```

練習3

- 練習１のクラスRectangleのコンストラクタを作成せよ.
- 作成したコンストラクタを実行せよ.

練習4

- 練習２のクラスMyPointのコンストラクタを作成せよ.
- 作成したコンストラクタを実行せよ.

this

自クラスの変数へアクセスする方法

```
this.変数名
```

自クラスのメソッドにアクセスする方法

```
this.メソッド名
```

自クラスのコンストラクタにアクセスする方法

```
this(引数リスト)
```


多重定義（オーバロード）

オーバロード#1

- 「同じクラスに，同じ名前で引数の型が異なるメソッドを定義すること」をメソッドのオーバロード（多重定義）という

例:Calcのオーバロード

```
class Calc{
  int add(int a, int b) {
    System.out.println(”intのaddが呼ばれました ”);
    return a+b;
  }
  double add(double a, double b) {
    System.out.println(”doubleのaddが呼ばれました ”);
    return a+b;
  }
```


例:Calcのオーバロード

```
class CalcMain{
  public static void main(String[] args) {
    Calc calc = new Calc();
    int num1 = calc.add(1, 2);
    System.out.println(”num1=”+num1);

    double num2 = calc.add(1.2, 2.3);
    System.out.println(”num2=”+num2);
  }
```


メソッドのシグネチャ

- Javaでは，オーバロードされたメソッドから，必要なメソッドを見分ける方法として，メソッドのシグネチャを利用する.
  - メソッドのシグネチャとは，メソッドの「名前」，「引数の並び（型，個数）」のこと

```
void set(double  tx, double ty) {...};
void set(double tx, double ty, double r){...};
```

```
int add(int a, int b) {...};
double add(double a, double b){...};
```

例

- 間違った例：同じシグネチャなので

```
void add(int a, int b){...}
void add(int a){...}
```

- 正しい例：違うシグネチャなので

```
void add(int a, int b){...}
void add(double num1, double num2){...}
```


例

- 正しい例：違うシグネチャなので

```
void add(int a){...}
void add(int a, int b){...}
```

- 誤った例：同じシグネチャなので

```
double add(int a, int b){return (double)(a+b);}
int add(int num1, int num2){return (a+b);}
```

注）：「戻り値の違い」は，「シグネチャの違い」とは関係がない

例：max#1

```
static int max(int a, int b, int c) {
  int max = a;
  if (b > max) max = b;
  if (c > max) max = c;
  return max;
}

static int max(int[] a) {
  int max = a[0];
  for (int i=1; i<a.length; i++) {
    if (a[i] > max) max = a[i];
  }
  return max;
}
```

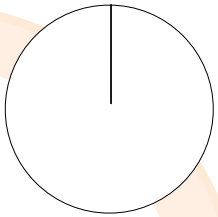

例：max#2

```
static int max(int a, int b, int c) {
  int max = a;
  if (b > max) max = b;
  if (c > max) max = c;
  return max;
}

static double max(double a, double b, double c)
{
  double max = a;
  if (b > max) max = b;
  if (c > max) max = c;
  return max;
}

...
```

例：「円」#1



属性
・中心座標(x,y)
・半径

```
class Circle
{
        double x;  // 中心のX座標
        double y;  // 中心のY座標
        double r;  //  半径
}
```


例：「円」#2

- 中心座標値 x， y および半径 r を入力するメソッドの多重定義の例

```
class Circle {
  double x; // 中心のx座標値
  double y; // 中心のy座標値
  double radius; //  半径

  void set(double tx, double ty){
    x = tx;
    y = ty;
  }

  void set(double tx, double ty, double r) {
    x = tx;
    y = ty;
    radius = r;
  }
}
```


練習5

- 練習１で作成した長方形クラス「Rectangle」のフィールド値を設定するメソッドsetを作成せよ．以下の２つのメソッドを多重定義せよ．
  - height だけを設定するメソッド
  - widthとheightを設定するメソッド

オーバロード＃２

- デフォルトコンストラクタ
  - 引数なしのコンストラクタのこと．例）Person(){};
- メソッド同様オーバーロードが可能
  - コンストラクタ（引数あり）を定義していない場合，**デフルトコンストラクタ定義なく，デフォルトコンストラクタを呼び出すことができる．**
  - コンストラクタ（引数あり）を定義した場合，デフォルトコンストラクタを呼び出すには，**明示的にデフォルトコンストラクタ定義する必要がある．**

○ 例

```
class Person5 {
  String  name;  //名前
  int age;  //年齢

  Person5(String iname, int iage) {
    name = iname; age = iage;
  }
  Person5(String iname) {
    name = iname; age=99;
  }
  ...
```


○ 例

```
class Person6 {
  String  name;  //名前
  int age;  //年齢

  Person6();
  Person6(String iname, int iage) {
    name = iname; age = iage;
  }
  ...
```


○ 例：「円」

```
class Circle
{
        double x;  // 中心のX座標
        double y;  // 中心のY座標
        double radius;  //  半径
}
```

```
Circle() {}
Circle(double tr) { radius = tr;}
Circle(double tx, double ty, double tr) {
  x = tx; y = ty; radius = tr;
}
```


○ 練習6

- 練習１のクラスRectangleのオーバーロドしたコンストラクタを作成せよ.
  - 例えば，（幅），（幅，高さ）を引数にもったもの
  - 作成したコンストラクタを実行する.

練習7

- 練習2のクラスMyPointのオーバーロードしたコンストラクタを作成せよ.
  - 例えば，（x），（x，y）を引数にもったもの
  - 作成したコンストラクタを実行する.

クラス変数

インスタンス{変数，メソッド}

- オブジェクトに属するもの
- オブジェクトが生成（new）されて，はじめて利用できるもの.
  - インスタンス変数：c1.x
  - インスタンスメソッド：c1.set(...)

例：

```
class Circle {
  double x; // 中心のx座標値
  double y; // 中心のy座標値
  double radius; //  半径
  void set(double tx, double ty, double r) {
    x = tx;
    y = ty;
    radius = r;
  }
}


//
Circle c1 = new Circle(); c1.set(10.0, 10.0, 5.0);
Circle c2 = new Circle(); c2.set(20.0, 20.0, 3.0);
System.out.println(c1.x);
System.out.println(c2.raduius);
```

インスタンス変数

インスタンスメソッド

クラス{変数,メソッド}

- クラスに属するもの
- 宣言方法

  static 型名 クラス変数名;

  static 戻り値の型 クラスメソッド名(引数リスト)

- オブジェクトが存在しなくても，利用できるもの.
  - クラス変数：<クラス名>.<フィールド名>で呼べる.
  - クラスメソッド：<クラス名>.<メソッド名>で呼べる.

例：「円」

```
class Circle {
  static int count = 0; //個数
  double x; // 中心のx座標値
  double y; // 中心のy座標値
  double radius; //  半径
  void set (double tx, double ty, double r) {
    count=count+1;
    x = tx;
    y = ty;
    radius = r;
  }
}

//
Circle c1 = new Circle(); c1.set(10.0, 10.0,5.0);
Circle c2 = new Circle(); c2.set(10.0, 1.0,5.0);
System.out.println(Circle.count);
```

クラス変数

例：「円」

```
class Circle {
  static int count = 0; //個数
  double x; // 中心のx座標値
  double y; // 中心のy座標値
  double radius; //  半径
  public static int getCount() {
    return count;
  }
}

//
Circle c1 = new Circle(); c1.set(10.0, 10.0,5.0);
Circle c2 = new Circle(); c2.set(10.0, 10.0,5.0);
System.out.println(Circle.getCount());
```

クラスメソッド

インスタンス変数とクラス変数

インスタンス変数 （インスタンスフィールド）	オブジェクト（インスタンス）ごとに，情報を保持
クラス変数 （クラスフィールド）	クラス全体でその情報を保持

練習8

- 名前（String型），身長（double型），体重（double型）をフィールドに持つ「人間クラス(Human)を作成せよ.

練習9

- 練習8のHumanクラスのコンストラクタを定義せよ

練習10

- 練習8のHumanクラスに以下のメソッドを定義せよ
  - getName：名前を返すメソッド
  - getHeight：身長を返すメソッド
  - getWeight：体重を返すメソッド
  - gainWeight(w)：引数w分体重を増やすメソッド
  - reduceWeight(w)： 引数w分体重を減らすメソッド

標準ライブラリ

クラスライブラリ

- Javaには，多くの人が利用するプログラムの部品を標準クラスライブラリとして，あらかじめ用意されています.
- 例えば
  - Java言語の中心機能：java.lang
  - 入出力機能：java.io
  - XML関連の機能：javax.xml

パッケージとimport

- Javaの標準ライブラリはクラスの機能ごとに，パッケージと呼ばれる単位で分類される.
- パッケージに属するクラスを利用数には，javaファイルの先頭でimport宣言を行う.

```
import <パケージ名>.<クラス名>;
```


利用例

```
import java.util.Random;
...
Random random = new Random();
```

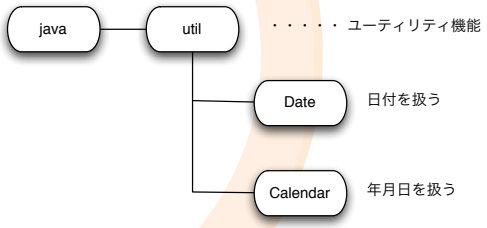
```
java.util.Random random
            = new java.util.Random();
```


パッケージ構造

- java.lang
  - 言語の中心機能．デフォルトでimportされので，importは不要（例：System, String...）
- java.io
  - 入出力機能（例：InputStream, Write...）
- java.awt
  - GUI用のクラス群（例：Panel, Label...）

パッケージの構造

- Java標準ライブラリに含まれるクラスのパッケージは，java or javaxから始まる．階層構造になってる．
- 例



APIリファレンス

- Java標準ライブラリには，含まれるクラスやクラスの使い方の記述したリファレンス
  - Java SE 6 APIリファレンス

    http://java.sun.com/javase/ja/6/docs/ja/api/

いろいろクラス

- Systemクラス
- Dateクラス
- Calendarクラス
- SimpleDateFormatクラス
- 文字を扱うクラス