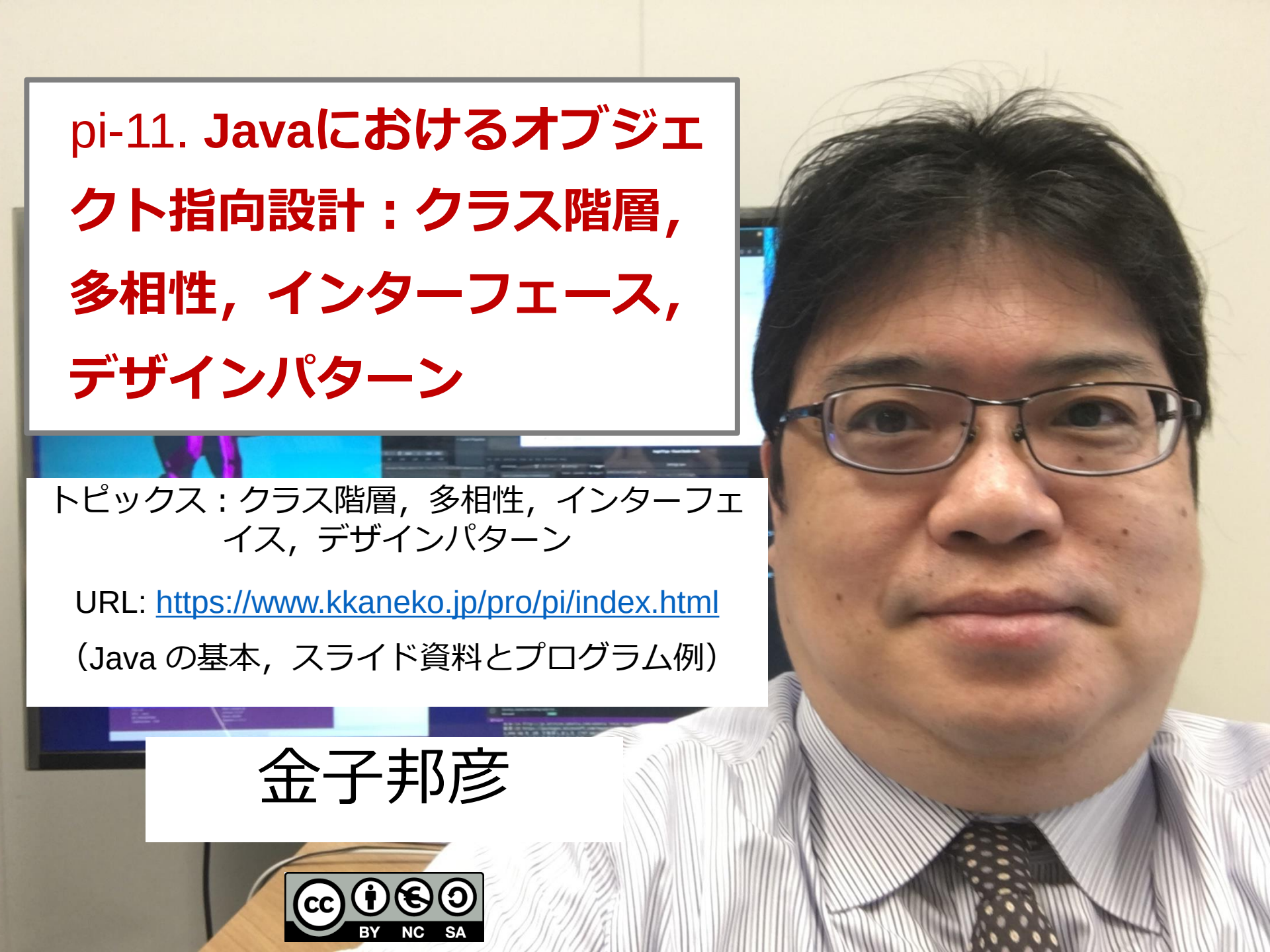
# pi-11. Javaにおけるオブジェクト指向設計：クラス階層，多相性，インターフェース，デザインパターン



トピックス：クラス階層，多相性，インターフェイス，デザインパターン

URL: https://www.kkaneko.jp/pro/pi/index.html

（Java の基本，スライド資料とプログラム例）

金子邦彦

# アウトライン

| 番号 | 項目 |
| --- | --- |
| | 復習 |
| 11-1 | クラス階層と多相性 |
| 11-2 | インターフェイス |
| 11-3 | デザインパターン |

**各自、資料を読み返したり、課題に取り組んだりも行う**

この授業では、**Java** を用いて基礎を学び、マスターする

# Java Tutor の起動

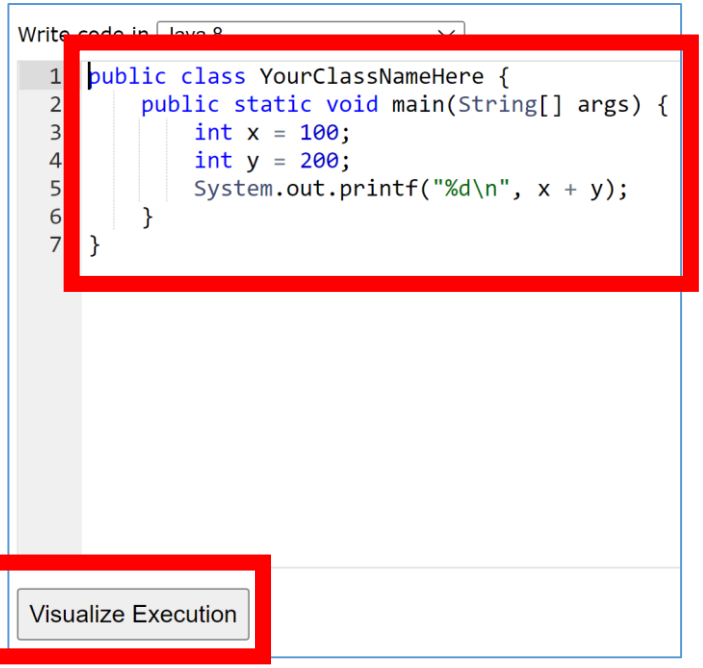
① **ウェブブラウザ**を起動する

② **Java Tutor** を使いたいので，次の URL を開く

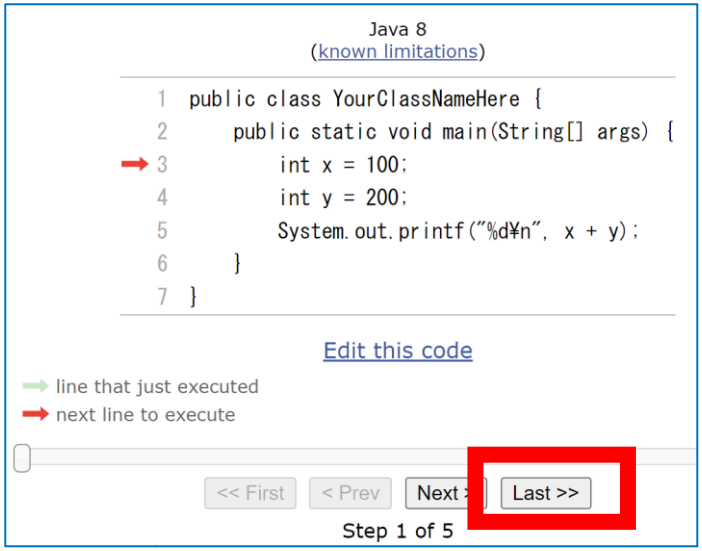
**http://www.pythontutor.com/**

③ 「**Java**」をクリック　⇒ **編集画面**が開く

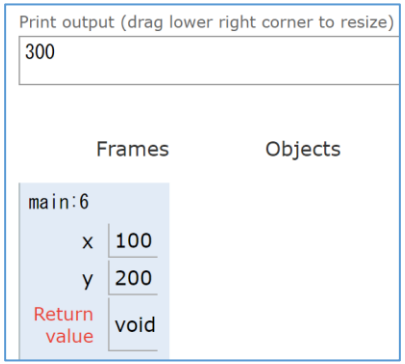
# Java Tutor でのプログラム実行手順



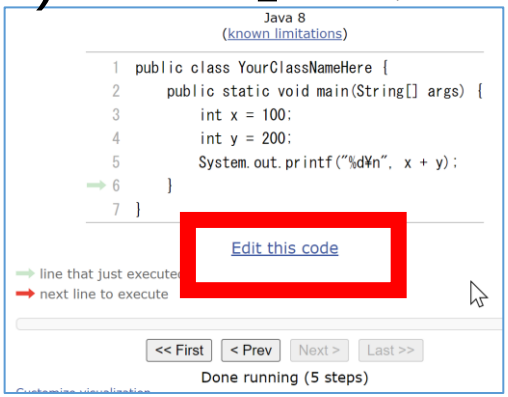

(1)「**Visualize Execution**」をクリックして**実行画面**に切り替える




(2)「**Last**」をクリック.




(3) **実行結果を確認**する.




(4)「**Edit this code**」をクリックして**編集画面**に戻る

# Java Tutor 使用上の注意点①

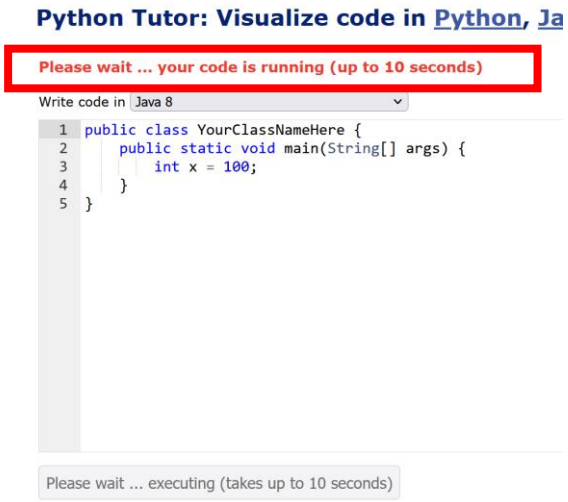
- **実行画面で，次のような赤の表示が出ることがある → 無視してよい**

  過去の文法ミスに関する確認表示

  邪魔なときは「**Close**」

# Java Tutor 使用上の注意点②

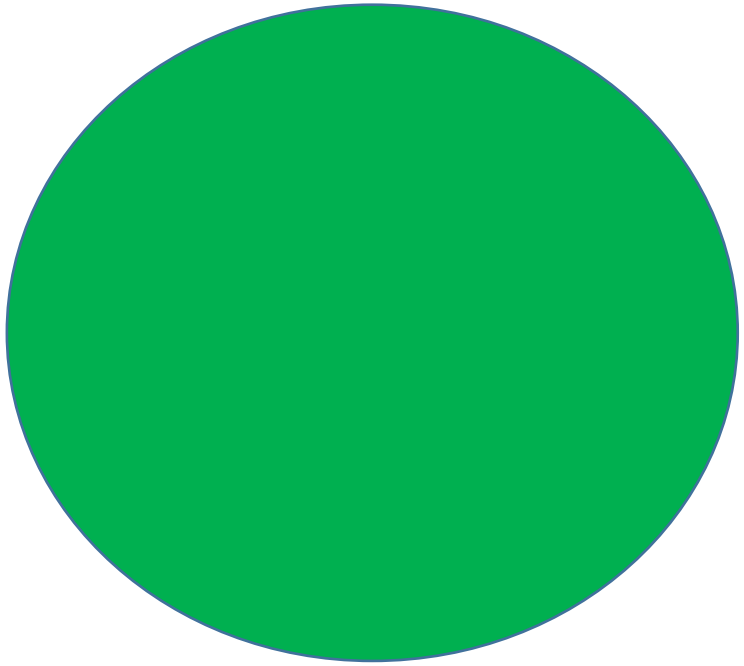
「**please wait ... executing**」のとき，**１０秒ほど待つ**.




→ **混雑しているとき**は，**「Server Busy・・・」というメッセージが出る**ことがある.

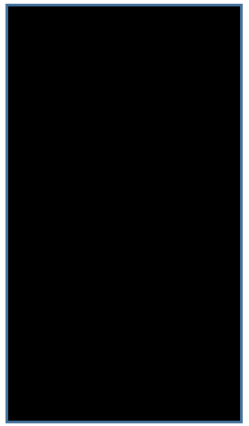
混雑している．**少し（数秒から数十秒）待つ**と自動で表示が変わる（変わらない場合には，操作をもう一度行ってみる）


# 11-1. クラス階層と多相性



半径 3，場所（2, 4）
色 green

**円（Circle）**

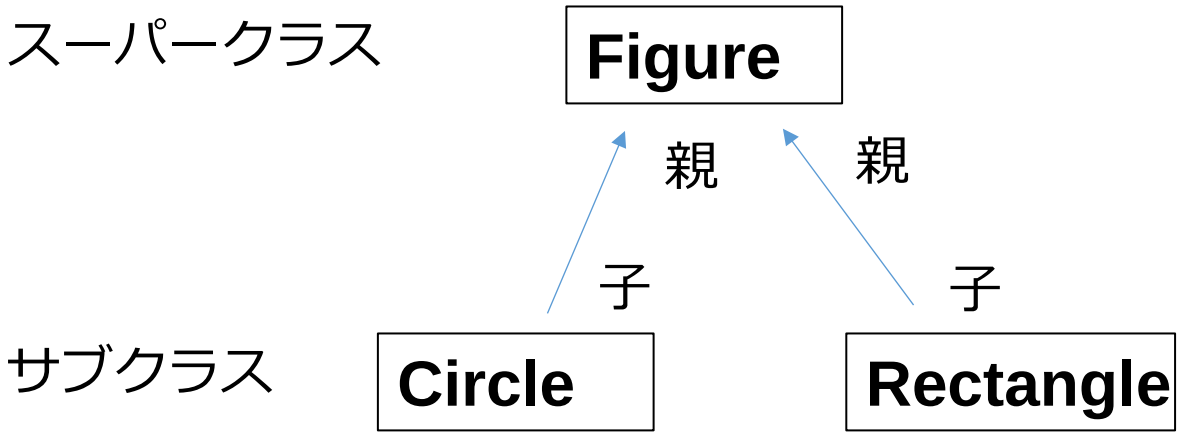


幅 1, 高さ 2, 場所（6, 4）
色 black

**長方形**
**(Rectangle)**

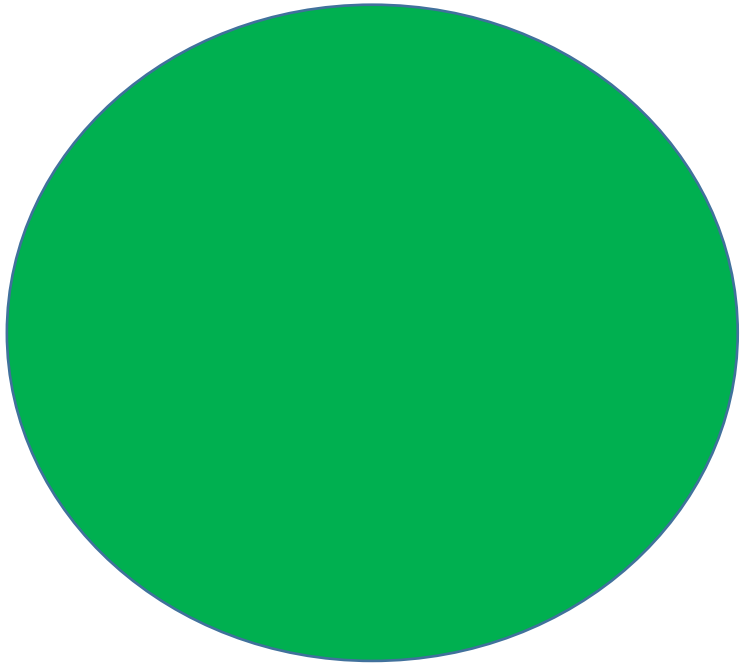
# クラス階層は何のため？

- **似通ったクラス Circle, Rectangle** を使いたい．**プログラムのミスを減らす**ため，**スーパークラス**を考える
- 将来，図形の種類を増やすときにも有効




**クラス Circle, クラス Rectangle が似ている．**

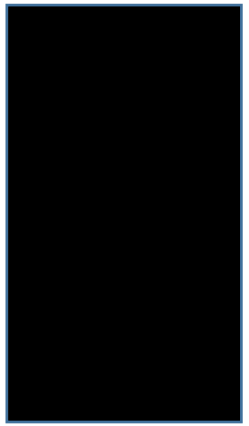
**共通する機能を、スーパークラス Figure にまとめる．**



半径 3，場所（2, 4）
色 green

**円（Circle）**

**Circle クラス**でもあり，
**Figure クラス**でもある



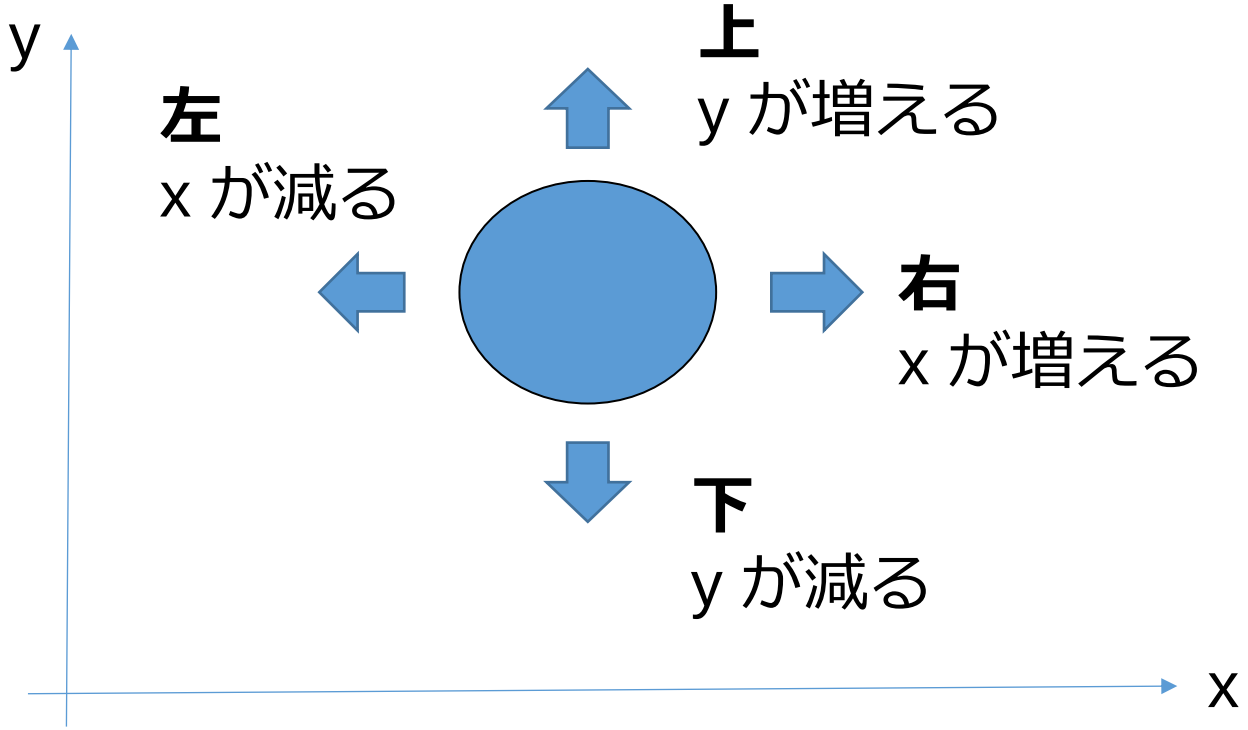
幅 1, 高さ 2, 場所（6, 4）
色 black

**長方形**
**(Rectangle)**

**Rectangle クラス**でもあり，
**Figure クラス**でもある

# Java のメソッド

- **上下左右の移動**
- **オブジェクト**の属性 x, y を増減
- そのためのメソッド move を **Figure クラスに**定義

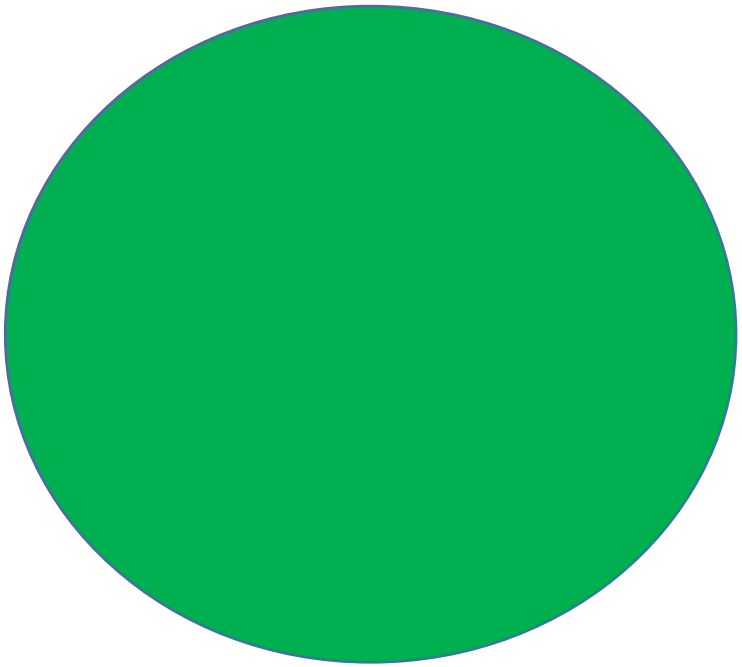


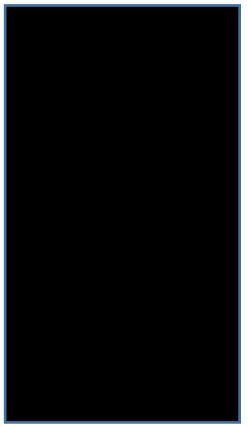
# Java のメソッド

- **面積を求める**
- そのためのメソッド size を定義

  円: 半径 × 半径 × 3.14

  長方形: 縦 × 横





# メソッドのオーバーライド

- **同じ名前**の**メソッド**が，**クラス**によって**違った意味**を持つ

メソッド **area**

クラス Figure では　　**0**

クラス Circle では　　**円の面積**

クラス Rectangle では **長方形の面積**

# 演習

資料：**16 ～ 18**

**【トピックス】**

- **クラス階層**
- **メソッドのオーバーライド**

# ① Java Tutor のエディタで次のプログラムを入れる


```
1  class Figure {
2    double x;
3    double y;
4    String color;
5    public Figure(double x, double y, String color) {
6      this.x = x;
7      this.y = y;
8      this.color = color;
9    }
10   public void move(double xx, double yy) {
11     this.x = this.x + xx;
12     this.y = this.y + yy;
13   }
14   public double area() {
15     return 0;
16   }
17 }
18
19 class Circle extends Figure {
20   double r;
21   public Circle(double x, double y, String color, double r) {
22     super(x, y, color);
23     this.r = r;
24   }
25   public double area() {
26     return this.r * this.r * 3.14;
27   }
28 }
```

メソッド move

メソッド area

メソッド area

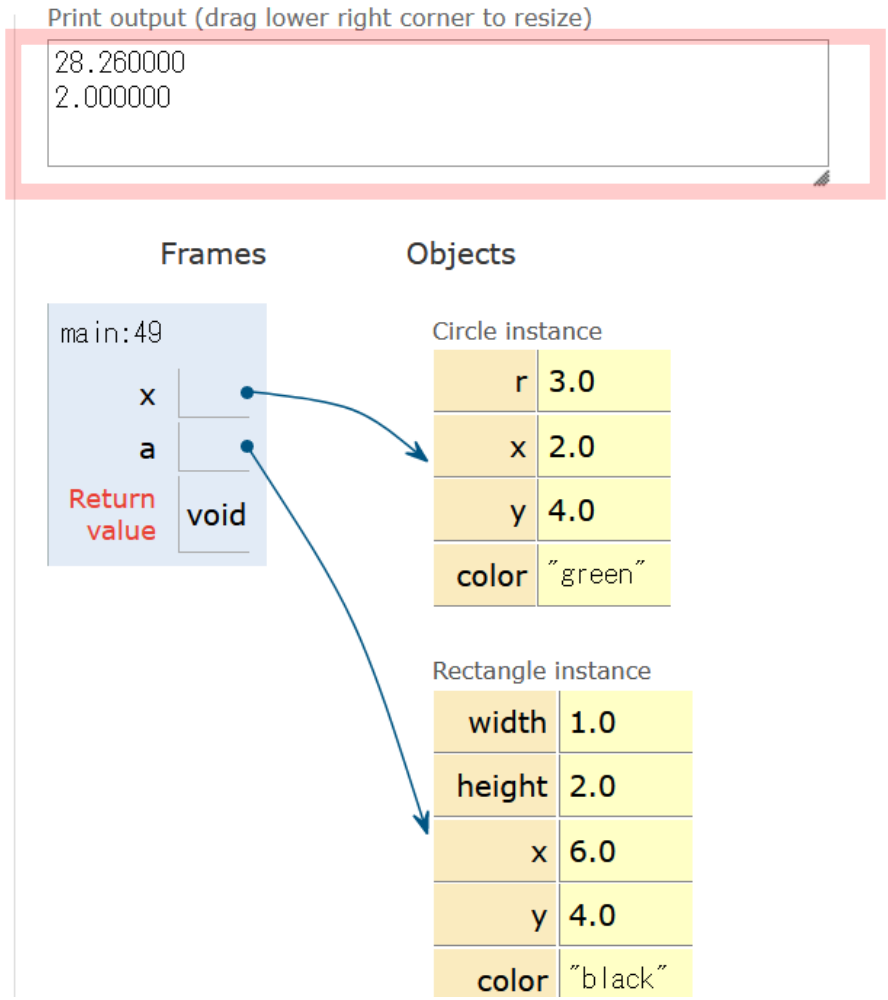
# 続き

```
30  class Rectangle extends Figure {
31    double width;
32    double height;
33    public Rectangle(double x, double y, String color, double w, double h)
34      super(x, y, color);
35      this.width = w;
36      this.height = h;
37    }
38    public double area() {
39      return this.width * this.height;
40    }
41  }
42
43  public class YourClassNameHere {
44      public static void main(String[] args) {
45        Circle x = new Circle(2, 4, "green", 3);
46        Rectangle a = new Rectangle(6, 4, "black", 1, 2);
47        System.out.printf("%f\n", x.area());
48        System.out.printf("%f\n", a.area());
49      }
50  }
```

メソッド area

メソッドを呼び出す部分

# ② 実行し，結果を確認する




「**Visual Execution**」をクリック．そして「**Last**」をクリック．結果を確認．
「**Edit this code**」をクリックすると，エディタの画面に戻る

# 多相性


- さまざまな**サブクラス**の**オブジェクト**を、「**スーパークラスのオブジェクトである**」と思って扱うときに役立つ

# 演習

資料：**21 ～ 23**

**【トピックス】**

- **クラス階層**
- **メソッドのオーバーライド**
- **多相性**

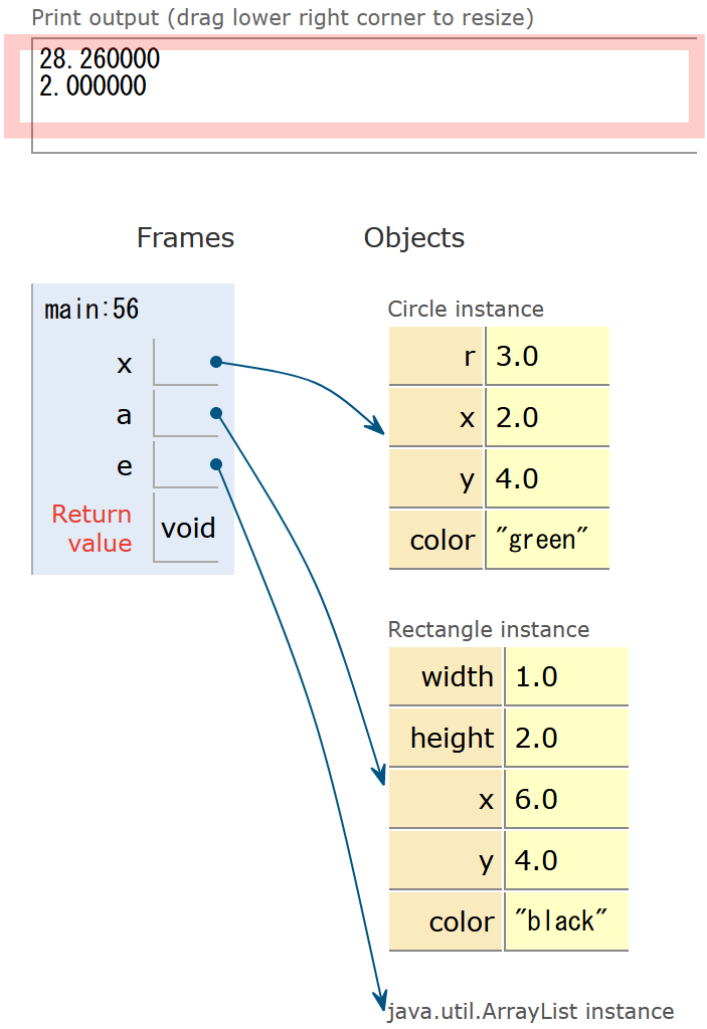
# ① Java Tutor のエディタで次のプログラムを入れる


```
1  import java.util.ArrayList;
2  import java.util.Iterator;
3
4  class Figure {
5    double x;
6    double y;
7    String color;
8    public Figure(double x, double y, String color) {
9      this.x = x;
10     this.y = y;
11     this.color = color;
12   }
13   public void move(double xx, double yy) {
14     this.x = this.x + xx;
15     this.y = this.y + yy;
16   }
17   public double area() {
18     return 0;
19   }
20 }
21
22 class Circle extends Figure {
23   double r;
24   public Circle(double x, double y, String color, double r) {
25     super(x, y, color);
26     this.r = r;
27   }
28   public double area() {
29     return this.r * this.r * 3.14;
30   }
31 }
```

# 続き

```
33  class Rectangle extends Figure {
34    double width;
35    double height;
36    public Rectangle(double x, double y, String color, double w, double h)
37      super(x, y, color);
38      this.width = w;
39      this.height = h;
40    }
41    public double area() {
42      return this.width * this.height;
43    }
44  }
45
46  public class YourClassNameHere {
47      public static void main(String[] args) {
48        Circle x = new Circle(2, 4, "green", 3);
49        Rectangle a = new Rectangle(6, 4, "black", 1, 2);
50        ArrayList<Figure> e = new ArrayList<Figure>();
51        e.add(x);
52        e.add(a);
53        for(Figure f : e) {
54          System.out.printf("%f\n", f.area());
55        }
56      }
57  }
```

# ② 実行し，結果を確認する




「**Visual Execution**」をクリック．そして「**Last**」をクリック．結果を確認．
「**Edit this code**」をクリックすると，エディタの画面に戻る

```
public static void main(String[] args) {
  Circle x = new Circle(2, 4, "green", 3);
  Rectangle a = new Rectangle(6, 4, "black", 1, 2);
  ArrayList<Figure> e = new ArrayList<Figure>();
  e.add(x);
  e.add(a);
  for(Figure f : e) {
    System.out.printf("%f\n", f.area());
  }
}
```


- リストに、要素を追加
- このリストは「**ArrayList<Figure>**」
- x や a は、**Figure クラスのオブジェクトとして、リストに追加**される
- **多相性**により、**move メソッド**（面積の算出）に問題ない．**x では円の面積、a では長方形の面積**


# 11-2. インターフェイスと実装

さまざまな**クラス**の**オブジェクト**を、「**同じ種類のオブジェクトである**」かのように扱う**２つの方法**

① **スーパークラス**と**サブクラス、多相性** (11-1. で説明)

**継承あり**

② **インターフェイス** (11-2. で説明)

**Java 処理系は、クラスがインターフェイスに準拠するかチェックを行う**

**（「継承」は無関係）**

# ２つの**クラス**

Matome

属性

number

unit

**数 (number)と単価 (unit)**

Bara

属性

price

**価格 (price)**

**共通する属性がない**

メソッド

total

total

**名前は同じ**だが，
**中身が違う**

# インターフェイス


複数のクラス A, B が

**名前が同じメソッド**を持つとき

A, B は，共通する**インターフェイス**を持つ
と考えることができる
（**メソッド**の中身は違っても構わない）

**Java** 処理系には，**クラスがインターフェイスに準拠するかをチェックする機能**がある.

インターフェースは Java, C# 言語などが持つ機能

# インターフェイスの Java プログラム例


インターフェイス名

```
interface Product {
   int total();
}
```

中身の**ない**メソッド

# クラス定義で，インターフェイス準拠の指定


クラス Matome がインターフェイス Product に**準拠**するとき

```
1  interface Product {
2    int total();
3  }
4
5  class Matome implements Product {
6    int number;
7    int unit;
8    public Matome(int number, int unit) {
9      this.number = number;
10     this.unit = unit;
11   }
12   public int total() {
13     return this.number * this.unit;
14   }
15 }
```

# クラス定義で，インターフェイス準拠の指定

クラス Bara がインターフェイス Product に**準拠**するとき

```
1  interface Product {
2    int total();
3  }
4
5  class Bara implements Product {
6    int price;
7    public Bara(int price) {
8      this.price = price;
9    }
10   public int total() {
11     return this.price;
12   }
13 }
```

# インターフェイスの機能①

メソッドの定義を忘れるなどで、

**インターフェイスに適合しない**ときは、

警告メッセージが出る

```
1  interface Product {
2    int total();
3  }
4
5  class Bara implements Product {
6    int price;
7    public Bara(int price) {
8      this.price = price;
9    }
10 }
11
12 public class YourClassNameHere {
13     public static void main(String[] args) {
14       Matome a = new Matome(10, 100);
15       System.out.printf("%d\n", a.total());
16     }
17 }
```

Error: Bara is not abstract and does not override abstract method total() in Product

# 演習

資料：**34 ～ 36**

**【トピックス】**

**・インターフェイス**

# ① Java Tutor のエディタで次のプログラムを入れる


```
1  interface Product {
2     int total();
3  }
4
5  class Bara implements Product {
6     int price;
7     public Bara(int price) {
8        this.price = price;
9     }
10    public int total() {
11       return this.price;
12    }
13 }
```

インターフェイス

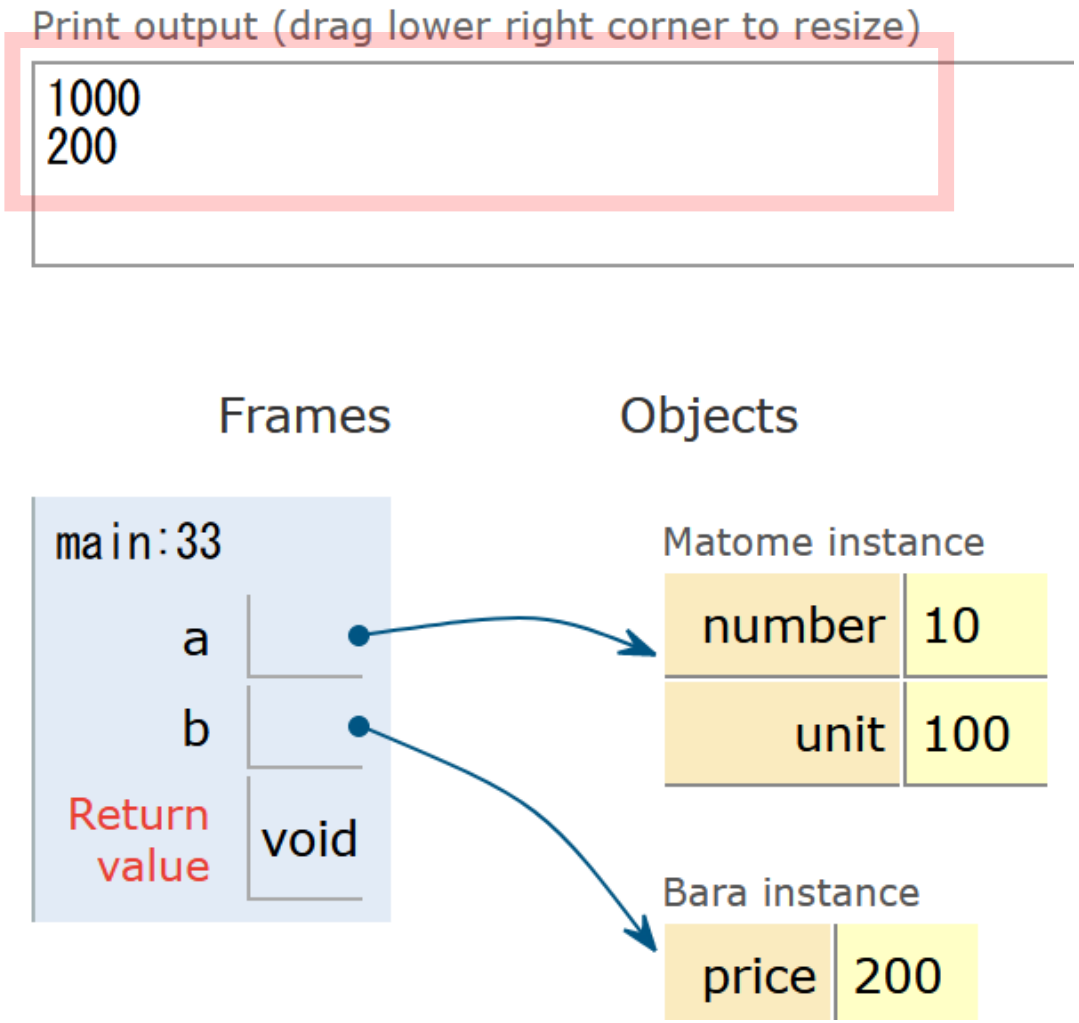
クラス定義

続き

```
15 class Matome implements Product {
16   int number;
17   int unit;
18   public Matome(int number, int unit) {
19     this.number = number;
20     this.unit = unit;
21   }
22   public int total() {
23     return this.number * this.unit;
24   }
25 }
26
27 public class YourClassNameHere {
28     public static void main(String[] args) {
29       Matome a = new Matome(10, 100);
30       Bara b = new Bara(200);
31       System.out.printf("%d\n", a.total());
32       System.out.printf("%d\n", b.total());
33     }
34 }
```

クラス定義

# ② 実行し，結果を確認する




「**Visual Execution**」をクリック．そして「**Last**」をクリック．結果を確認．
「**Edit this code**」をクリックすると，エディタの画面に戻る

# インターフェイスの機能②

コレクションとの組み合わせ

ArrayList<**インターフェイス名**>

のような使い方が可能

```
ArrayList<Product> e = new ArrayList<Product>();
e.add(a);
e.add(b);
for(Product p : e) {
  System.out.printf("%d\n", p.total());
}
```

# 演習

資料：**39 ～ 41**

**【トピックス】**

- **インターフェイス**
- **コレクション**

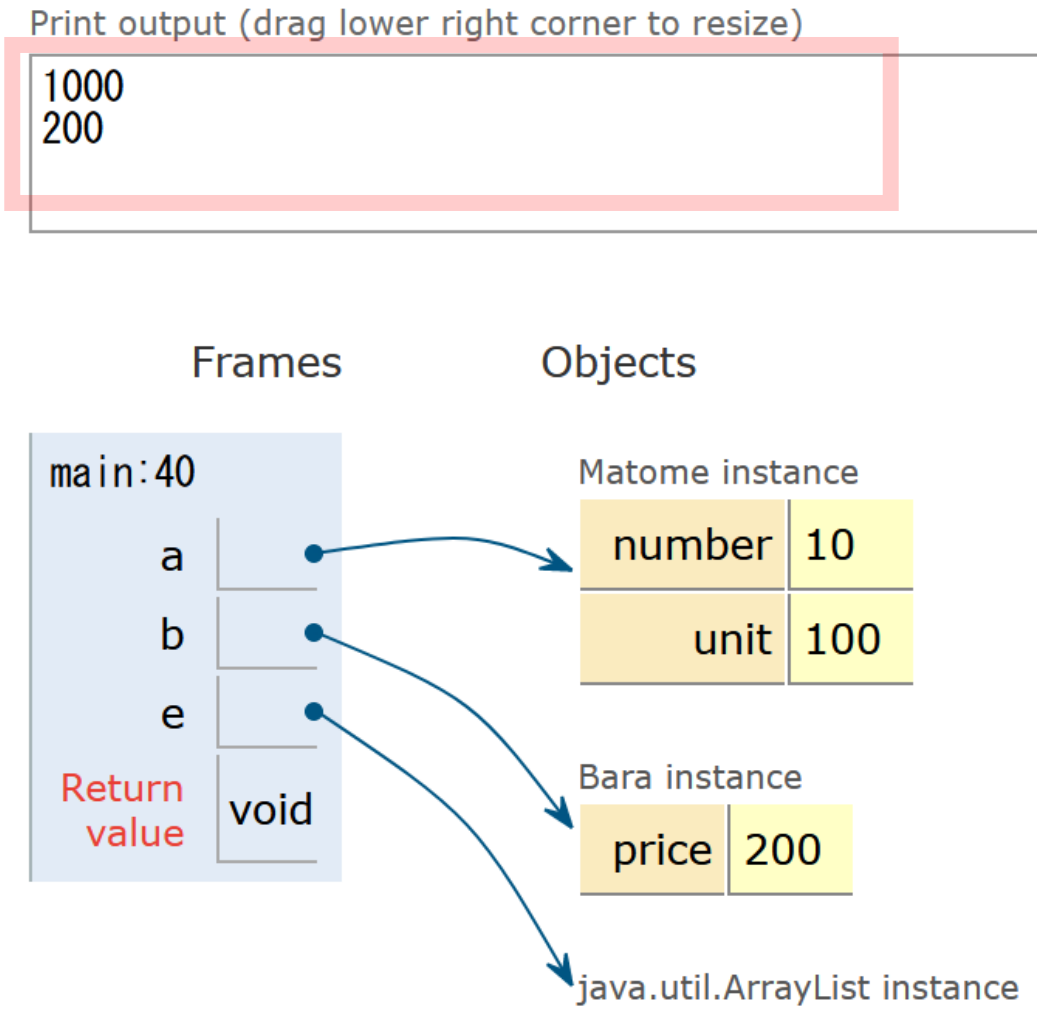
# ① Java Tutor のエディタで次のプログラムを入れる


```
1  import java.util.ArrayList;
2  import java.util.Iterator;
3
4  interface Product {
5    int total();
6  }
7
8  class Bara implements Product {
9    int price;
10   public Bara(int price) {
11     this.price = price;
12   }
13   public int total() {
14     return this.price;
15   }
16 }
```

続き

```
18 class Matome implements Product {
19   int number;
20   int unit;
21   public Matome(int number, int unit) {
22     this.number = number;
23     this.unit = unit;
24   }
25   public int total() {
26     return this.number * this.unit;
27   }
28 }
29
30 public class YourClassNameHere {
31     public static void main(String[] args) {
32       Matome a = new Matome(10, 100);
33       Bara b = new Bara(200);
34       ArrayList<Product> e = new ArrayList<Product>();
35       e.add(a);
36       e.add(b);
37       for(Product p : e) {
38         System.out.printf("%d\n", p.total());
39       }
40     }
41 }
```

# ② 実行し，結果を確認する




「**Visual Execution**」をクリック．そして「**Last**」をクリック．結果を確認．
「**Edit this code**」をクリックすると，エディタの画面に戻る

# インターフェイス

- **インターフェイス**は，ある**メソッド**が**実装済みであること保証**する仕組み

- コレクションで「ArrayList<**インターフェイス名**>」のように書くこともできる

- さまざまな**クラス**の**オブジェクト**を、「**同じ種類のオブジェクトである**」かのように扱う仕組みの１つ


# 11-3. デザインパターン

# デザインパターンとは


- **デザインパターン**とは，プログラムで頻出されるとされるパターンのこと

- **デザインパターン**を知り，活用することが，プログラムを簡単に確実に作成できる手段であると唱える人も

- **デザインパターン**の種類は 23 である（GoFの23パターン）という説も

# デザインパターンの例

「**Template Method**」というデザインパターンの例.

**サブクラス**ごとに**メソッドの中身が違う．スーパークラスでは、メソッドの中身は書かない**というパターン

```
1  abstract class X {
2    abstract void nice();
3    public void hello() {
4      nice();
5    }
6  }
7
8  class Y extends X {
9    void nice() {
10     System.out.println("nice !");
11   }
12 }
13
14 class Z extends X {
15   void nice() {
16     System.out.println("Ummmmmm");
17   }
18 }
19
20
21 public class YourClassNameHere {
22     public static void main(String[] args) {
23       X a = new Y();
24       X b = new Z();
25       a.hello();
26       b.hello();
27     }
28 }
```

Print output (dra

```
nice !
Ummmmmm
```

# 関連ページ

- **Java プログラミング入門**

GDB online を使用

https://www.kkaneko.jp/pro/ji/index.html

- **Java の基本**

Java Tutor, GDB online を使用

https://www.kkaneko.jp/pro/pi/index.html

- **Java プログラム例**

https://www.kkaneko.jp/pro/java/index.html