Computer Language

OOP 3: Casting and Overriding

Agenda

- Casting
- Method Overriding

Class: Up/Downcasting

- Type conversion between classes
  - Similar to promotion/casting concept for primitive types

- Upcasting
  - Type conversion from sub-class to super-class

```
class Person { … }
class Student extends Person { … }

Student s = new Student();
Person p = s; // Upcasting, automatic conversion
```

  - Upcasting reference can only access the members of a superclass

Class: Up/Downcasting (cont'd)

- Upcasting
  - Type conversion from sub-class to super-class
  - Upcasting reference can only access the members of a superclass

```java
class Person{
   String name;
   String id;

   public Person(String name){
      this.name = name;
   }
}

class Student extends Person{
   String grade;
   String department;

   public Student(String name){
      super(name);
   }
}
```

```java
public class UpcastingEx {
   public static void main(String[] args) {
      Person p;
      Student s = new Student("Jinwoo");
      p = s; //upcasting
      System.out.println(p.name);
      //p.grade = "F";
      //p.department = "ITM";
   }
}
```

Class: Up/Downcasting (cont'd)

- Downcasting
  - Type conversion from super-class to sub-class
  - MUST be explicitly made by a developer

```
class Person { … }
class Student extends Person { … }
…
Person p = new Student("Jinwoo"); // upcasting
…
Student s = (Student) p; // downcasting (casting from Person to Student)
```

  - Why downcasting?
    - When we wish to use the members of a subclass!

Class: Up/Downcasting (cont'd)

- Downcasting
  - Type conversion from super-class to sub-class
  - MUST be explicitly made by a developer

```
public class UpcastingEx {
   public static void main(String[] args) {
      Person p = new Student("Jinwoo");
      System.out.println(p.name);
      //p.grade = "F";
      //System.out.println(p.grade);
      Student s = (Student) p;
      System.out.println(s.name);
      s.grade = "A";
      System.out.println(s.grade);
      //p.grade = "F";
      //p.department = "ITM";
   }
}
```

Downcasting (from Person to Student)

Class: Up/Downcasting (cont'd)

- A lot of subclasses from a single superclass available
  - Invalid downcasting results in an error!

```
Parent parent = new Parent();
Child child = (Child) parent;
```

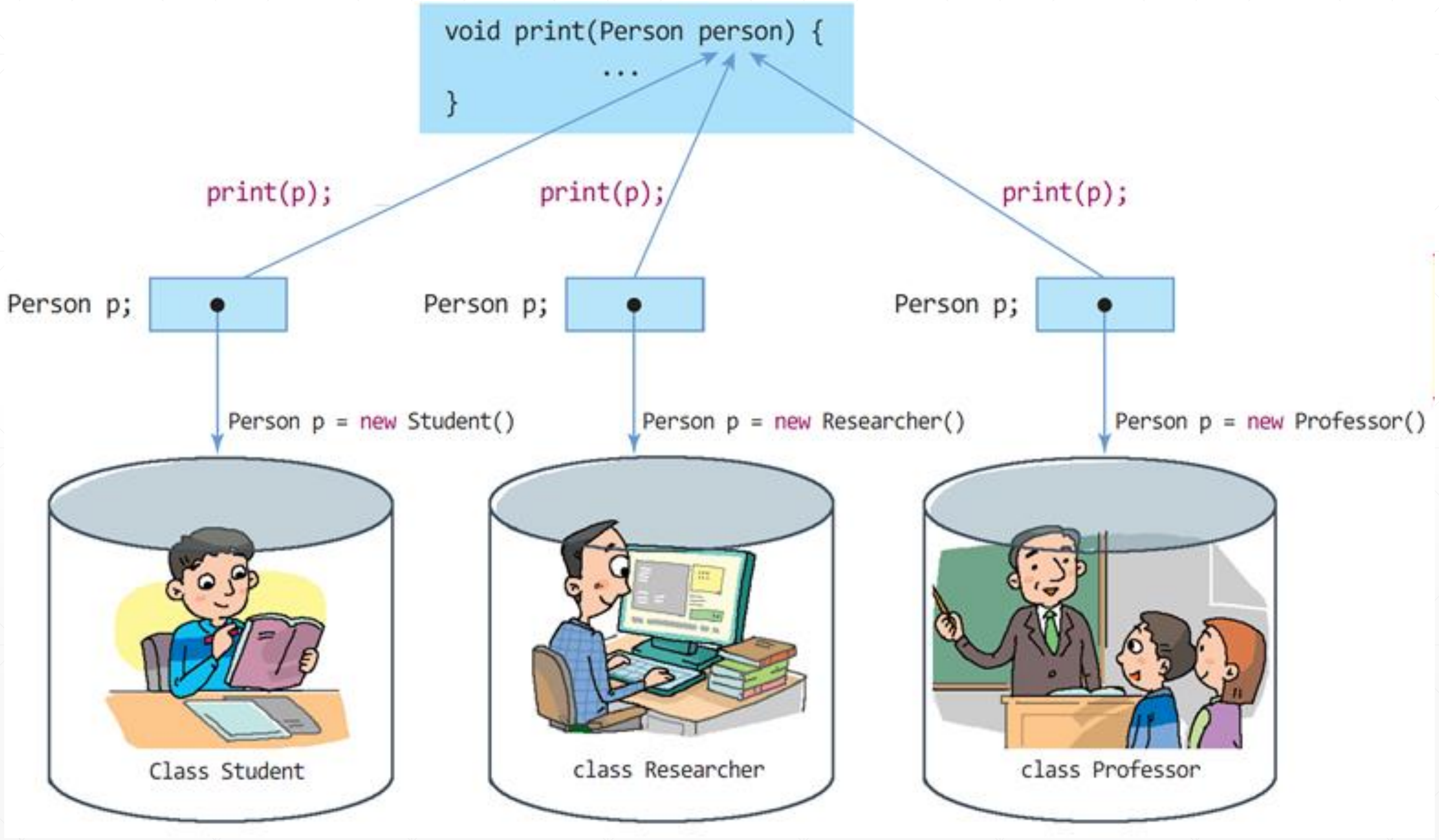
$\longleftarrow$ Impossible!

  - It is impossible to inter the actual type of a upcasting reference

- instanceof operator
  - Used to determine the type of an object
  - Returns true / false

```
objRef instanceof Classtype
```

Class: Up/Downcasting (cont'd)



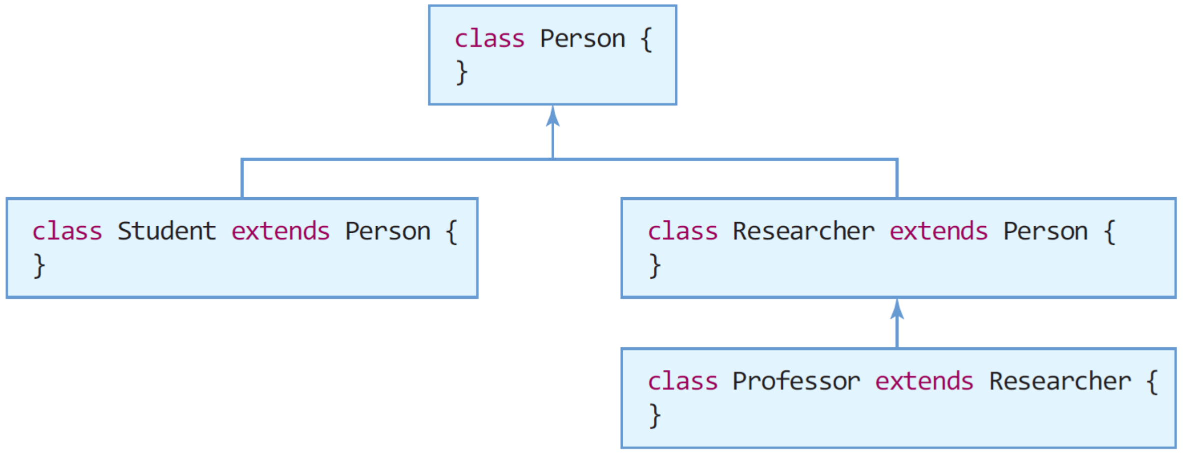
Class: Up/Downcasting (cont'd)

- Example of using instanceof operator

```
Person jee= new Student();
Person kim = new Professor();
Person lee = new Researcher();
if (jee instanceof Person)          // true
if (jee instanceof Student)         // true
if (kim instanceof Student)         // false
if (kim instanceof Professor)       // true
if (kim instanceof Researcher)      // true
if (lee instanceof Professor)       // false
```

```
if(3 instanceof int)                // Error!
```

```
if("java" instanceof String)        // true
```

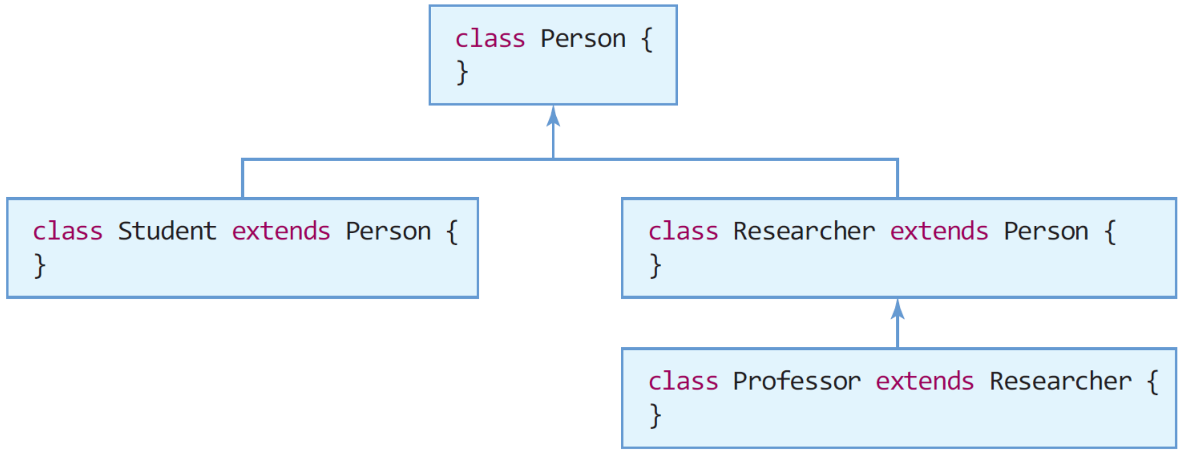


```
class Person {
}

class Student extends Person {
}

class Researcher extends Person {
}

class Professor extends Researcher {
}
```

Class: Up/Downcasting (cont'd)

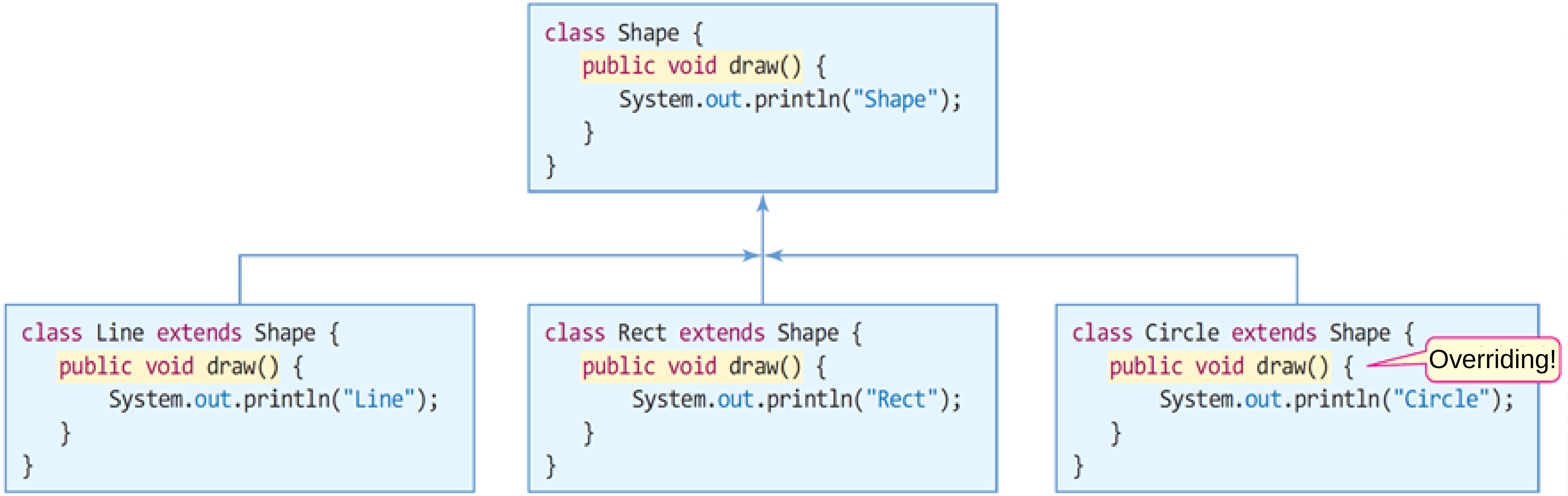
- Example of using instanceof operator

```
class Person { }
class Student extends Person { }
class Researcher extends Person { }
class Professor extends Researcher { }

public class InstanceOfEx {
   static void print(Person p) {
      if(p instanceof Person)
         System.out.print("Person ");
      if(p instanceof Student)
         System.out.print("Student ");
      if(p instanceof Researcher)
         System.out.print("Researcher ");
      if(p instanceof Professor)
         System.out.print("Professor ");
      System.out.println();
   }
   public static void main(String[] args) {
      System.out.print("new Student() ->₩t"); print(new Student());
      System.out.print("new Researcher() ->₩t"); print(new Researcher());
      System.out.print("new Professor() ->₩t"); print(new Professor());
   }
}
```



Method Overriding

- Redefinition of superclass's method in the subclass
  - Same method signature, but different behaviors



```java
class Shape {
   public void draw() {
      System.out.println("Shape");
   }
}
```

```java
class Line extends Shape {
   public void draw() {
      System.out.println("Line");
   }
}
```

```java
class Rect extends Shape {
   public void draw() {
      System.out.println("Rect");
   }
}
```

```java
class Circle extends Shape {
   public void draw() {
      System.out.println("Circle");
   }
}
```

Method Overriding (cont’d)

- Redefinition of superclass’s method in the subclass
  - Same method signature, but different behaviors

- Achieves polymorphism with inheritance
  - Same interface, but different behaviors
    - Line class draws a line using draw() interface
    - Circle class draws a circle using draw() interface
    - Rect class draws a rectangle using draw() interface

Method Overriding (cont'd)

- Example of Polymorphism using method overriding

```
class Shape {
   public void draw() {
      System.out.println("Shape");
   }
}

class Line extends Shape {
   public void draw() { // method overriding!
      System.out.println("Line");
   }
}

class Rect extends Shape {
   public void draw() {// method overriding!
      System.out.println("Rect");
   }
}

class Circle extends Shape {
   public void draw() {// method overriding!
      System.out.println("Circle");
   }
}
```

```
public class MethodOverridingEx {
   static void paint(Shape p) {
      p.draw(); // call overridden draw()

   }

   public static void main(String[] args) {
      Line line = new Line();
      paint(line);
      paint(new Shape());
      paint(new Line());
      paint(new Rect());
      paint(new Circle());
   }
}
```

Method Overriding (cont'd)

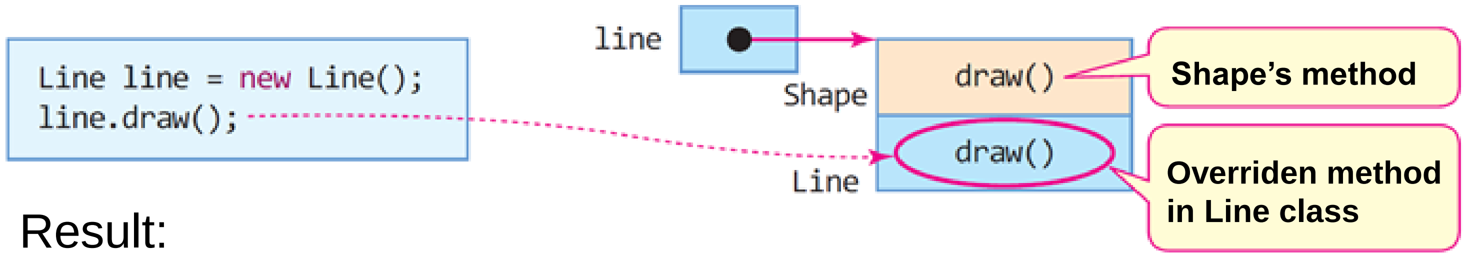
- Which method should be invoked?
  - For input parameter with Shape type, there can be a lot of variations!
  - When this association made?

```
public class MethodOverridingEx {
   static void paint(Shape p) {
      p.draw(); // call overridden draw()

   }

   public static void main(String[] args) {
      Line line = new Line();
      paint(line);
      paint(new Shape());
      paint(new Line());
      paint(new Rect());
      paint(new Circle());
   }
}
```

Shape's draw()?
Line's draw()?
Rect's draw()?
Circle's draw()?

Method Overriding (cont'd)

- Which method should be invoked?
  - Calling an overridden method from the subclass

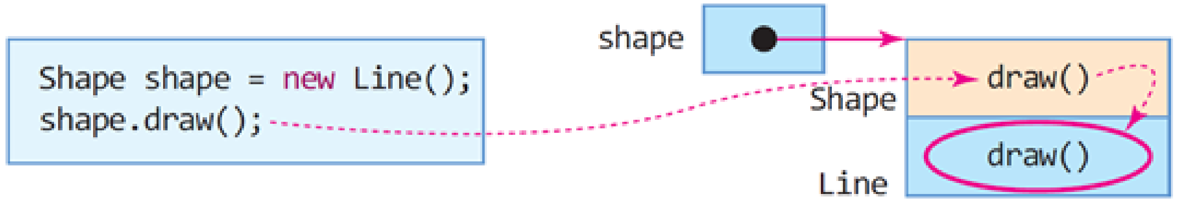


```
Line line = new Line();
line.draw();
```

Result:

```
Line
```

  - Calling an overridden method from the (upcasting) superclass



```
Shape shape = new Line();
shape.draw();
```

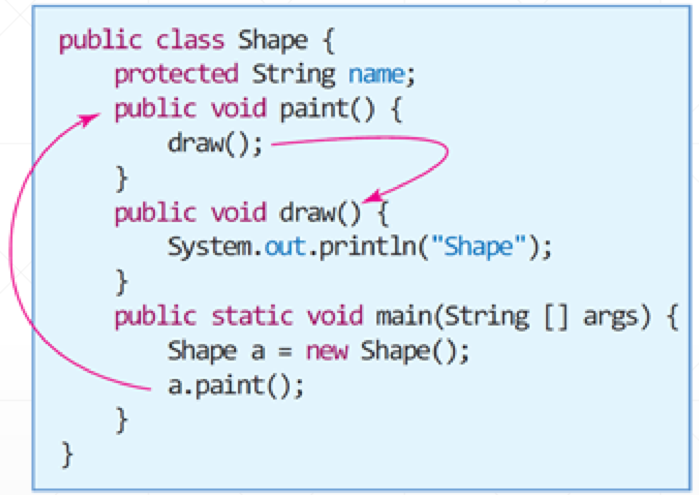
Dynamic binding

Result:

```
Line
```

Method Overriding (cont'd)

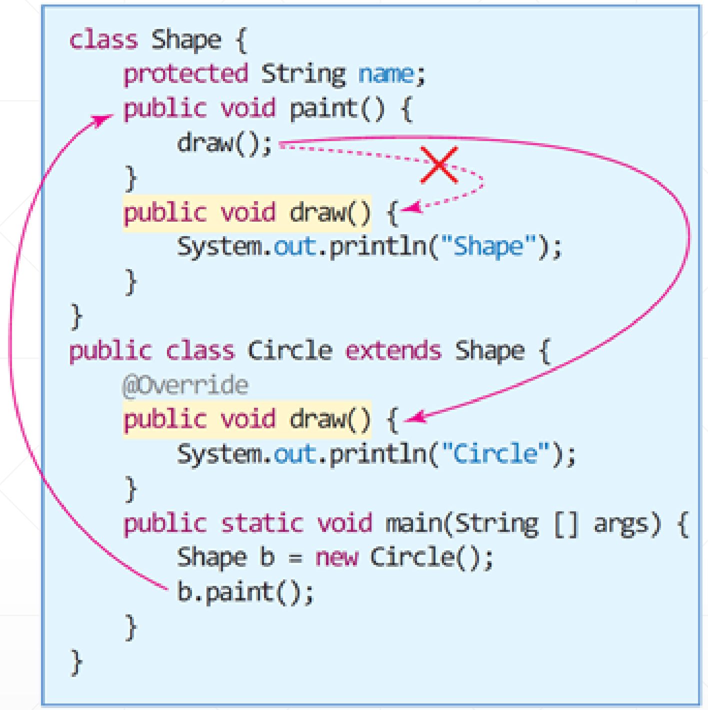
- Dynamic binding
  - Runtime association of method calling
  - "Who should be invoked?" is determined at runtime



```
public class Shape {
    protected String name;
    public void paint() {
        draw();
    }
    public void draw() {
        System.out.println("Shape");
    }
    public static void main(String [] args) {
        Shape a = new Shape();
        a.paint();
    }
}
```

Result:

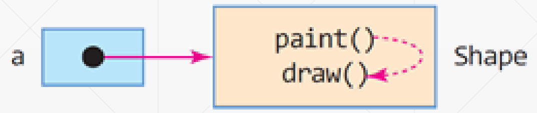
```
Shape
```

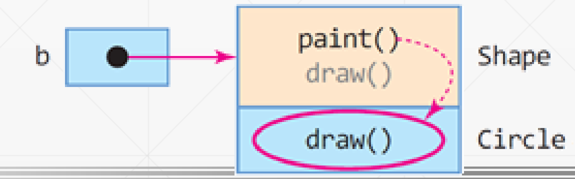


```
class Shape {
    protected String name;
    public void paint() {
        draw();
    }
    public void draw() {
        System.out.println("Shape");
    }
}
public class Circle extends Shape {
    @Override
    public void draw() {
        System.out.println("Circle");
    }
    public static void main(String [] args) {
        Shape b = new Circle();
        b.paint();
    }
}
```

Result:

```
Circle
```





Method Overriding (cont'd)

- Static binding
  - Compile-time association of method calling
  - "Who should be invoked?" is determined at compile time (e.g, static method)

```
class Shape {
static void clear(){ System.out.println("Clear!"); }
   void draw() { System.out.println("Shape"); }
}

class Line extends Shape {
   static void clear(){ System.out.println("Line Clear!"); }
   void draw() { System.out.println("Line"); }
}

class Rect extends Shape {
   static void clear(){ System.out.println("Rect Clear!"); }
   void draw() { System.out.println("Rect"); }
}

class Circle extends Shape {
   static void clear(){ System.out.println("Circle Clear!"); }
   void draw() { System.out.println("Circle"); }
}
```

```
public class MethodOverridingEx {
   static void paint(Shape p){ p.draw();  }
   static void clear(Shape p){ p.clear();  }

   public static void main(String[] args) {
      Line line = new Line();
      paint(line);
      paint(new Shape());
      paint(new Line());                    Dynamic binding
      paint(new Rect());
      paint(new Circle());

      clear(line);
      clear(new Shape());
      clear(new Line());                    Static binding
      clear(new Rect());
      clear(new Circle());
   }
}
```

Method Overriding (cont'd)

- Example)
  - An array containing various payment methods
  - Process a series of payments using abstraction and polymorphism

```java
class Payment {
   void pay(int money) { System.out.println("Payment!"); }
}

class Cash extends Payment {
   void pay(int money) { System.out.println("Success!"+ money+" Won paid"); }
}

class Bitcoin extends Payment {
   void pay(int money) { System.out.println("Fail! Coin destroyed!"); }
}

class Credit extends Payment {
   void pay(int money) { System.out.println("Success! Payment made with your card!"); }
}
```

Method Overriding (cont'd)

- Example)
  - An array containing various payment methods
  - Process a series of payments using abstraction and polymorphism

```java
public class MethodOverridingEx {
   static void purchase(Payment[] pay){
      for (Payment s: pay){
         s.pay(1000);
      }
   }

   public static void main(String[] args) {
      Payment[] myPayments = new Payment[3];
      myPayments[0] = new Cash();
      myPayments[1] = new Bitcoin();
      myPayments[2] = new Credit();

      purchase(myPayments);
   }
}
```

Method Overriding (cont'd)

- Method Overloading vs Method Overriding

	Overloading	Overriding
Declaration	Multiple definition of methods with the same name	Re-defining superclass's method in the subclass
Relationship	In the same class	Inheritance
Purpose	Improved usability through the methods with the same name Compile-time polymorphism	Re-define subclass specific behaviors Runtime polymorphism
Condition	Same method name Different number/type of arguments	Method signature (name, arguments, return type) must be same
binding	Static binding	Dynamic binding

Q&A

- Next week
  - Midterm exam (Closed written test)