03 Inheritance and Polymorphism

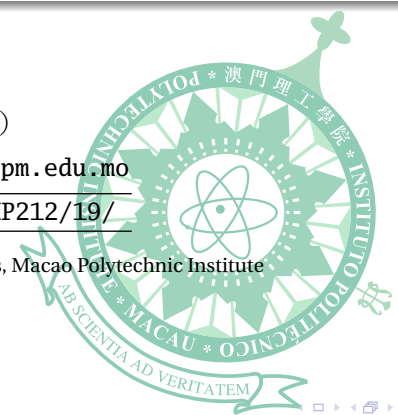
Instructor: Ke Wei（柯韋）

A319 ✆ Ext. 6452 ✉ wke@ipm.edu.mo

http://brouwer.ipm.edu.mo/COMP212/19/

Bachelor of Science in Computing, School of Applied Sciences, Macao Polytechnic Institute

September 12(17), 2019



Outline

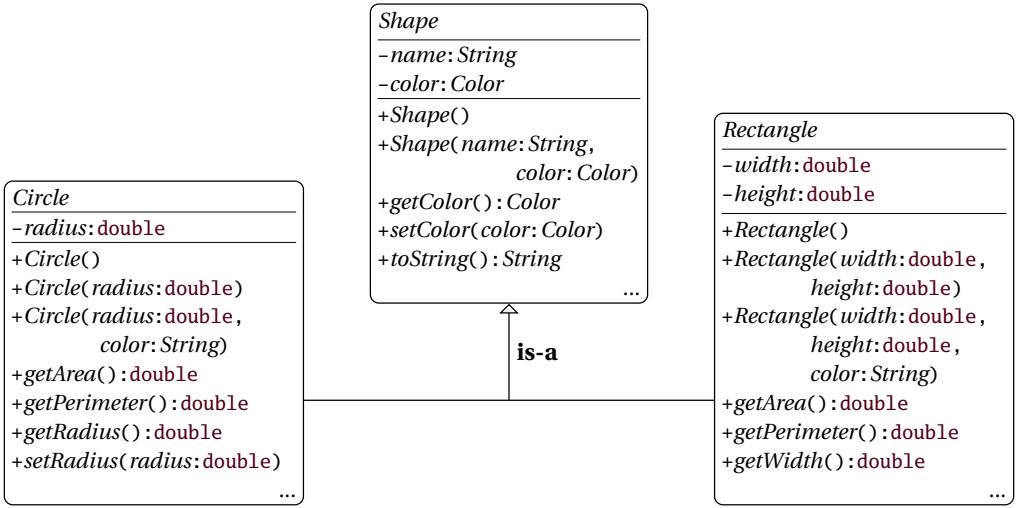
Common and Unique Features

- Some classes have common features, such as circles, triangles and rectangles all having perimeters and areas.
- The common features should be specified only once to avoid redundancy and inconsistency.
- Focusing on the common features, circles, triangles and rectangles can all be regarded as outline shapes.
- Different shapes each have their unique features, such as radii, widths and heights.
- The unique features must be specified separately.

Superclasses and Subclasses

- A *superclass* captures the common features among similar classes.
- A *subclass* introduces unique features, in addition to the common features copied from the superclass.
- An object of a subclass *is also an* object of a superclass.
- A subclass copying features from the superclass is called *inheritance*.
- Objects of a superclass possibly being objects of various subclasses is called *polymorphism*.

Inheritance Hierarchy



Declaring a Subclass

- The `extends` keyword is used to declare a subclass.

  ```
  class Circle extends Shape { ... }
  ```

  where, *Circle* is the subclass, and *Shape* is the superclass of *Circle*.
- A subclass can have *only one* superclass in Java. This is called the single inheritance model.
- However, multiple subclasses can share the same superclass.
- In a subclass, all the methods and attributes from the superclass are inherited.
- However, whether a particular member can be seen follows the visibility rule.
  For example, you can apply a `public` method of *Shape* on an object of *Circle*, but you cannot use the `private` field *color* directly outside the definition of *Shape*.

  ```
  Circle c = new Circle(); Color co = c.getColor();
  ```

Constructors of Superclasses

- Constructors of the superclass are *not* inherited. (Why not?)
- A constructor is used to construct an instance of a class. The construction must be complete.
- Although a (complete) object of *Circle* can be regarded as an object of *Shape*, the *Shape* class does not know how to make a *Circle*.
- Constructors of the superclass can be invoked in constructors of a subclass, to initialize the superclass portion, using the `super` keyword. Like `this(...)` calls, `super(...)` calls in constructors must be the first.

  Circle`() { super(`*Color*`.RED); `*radius*` = 1.0; }`

- If neither `super` nor `this` constructor is explicitly invoked, the default constructor of the superclass will be automatically invoked.

  Circle`() { `*radius*` = 1.0; }` $\Longrightarrow$ `{ super(); `*radius*` = 1.0; }`

- A constructor of the superclass is *always* invoked, either explicitly or implicitly.

Constructor Chaining

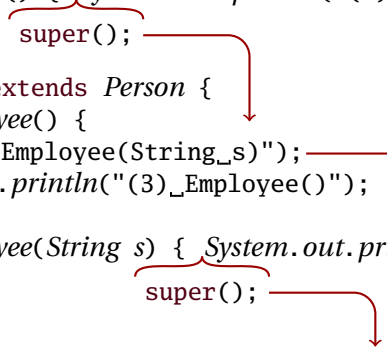
```
1  public class Faculty extends Employee {
2     public Faculty() { System.out.println("(4) Faculty()"); }
3  }
4
5  class Employee extends Person {
6     public Employee() {
7        this("(2) Employee(String s)");
8        System.out.println("(3) Employee()");
9     }
10    public Employee(String s) { System.out.println(s); }
11 }
12
13 class Person {
14    public Person() { System.out.println("(1) Person()"); }
15 }
```

Constructor Chaining

```
1  public class Faculty extends Employee {
2     public Faculty() { System.out.println("(4) Faculty()"); }
3  }                    super();
4
5  class Employee extends Person {
6     public Employee() {
7        this("(2) Employee(String s)");
8        System.out.println("(3) Employee()");
9     }
10    public Employee(String s) { System.out.println(s); }
11 }                             super();
12
13 class Person {
14    public Person() { System.out.println("(1) Person()"); }
15 }
```

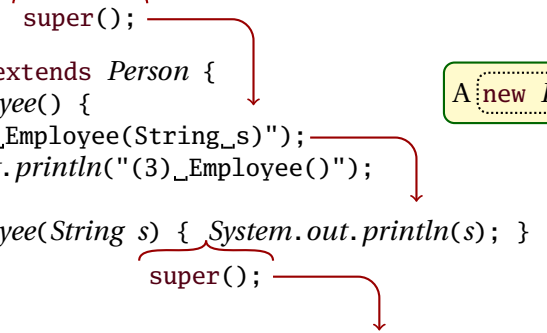
Constructor Chaining

```
1  public class Faculty extends Employee {
2     public Faculty() { System.out.println("(4) Faculty()"); }
3  }
4
5  class Employee extends Person {
6     public Employee() {
7        this("(2) Employee(String s)");
8        System.out.println("(3) Employee()");
9     }
10    public Employee(String s) { System.out.println(s); }
11 }
12
13 class Person {
14    public Person() { System.out.println("(1) Person()"); }
15 }
```



Constructor Chaining

```
1  public class Faculty extends Employee {
2     public Faculty() { System.out.println("(4) Faculty()"); }
3  }
4
5  class Employee extends Person {
6     public Employee() {
7        this("(2) Employee(String s)");
8        System.out.println("(3) Employee()");
9     }
10    public Employee(String s) { System.out.println(s); }
11 }
12
13 class Person {
14    public Person() { System.out.println("(1) Person()"); }
15 }
```



A `new Faculty()` prints (1) — (4).

Impact of a Superclass without a Default Constructor

- For a class, a default constructor is implicitly defined *only* when *no* constructor is explicitly defined.
- For a class, if some constructor is explicitly defined, then all constructors must be explicitly defined.
- Find out the errors in the program:

```
1 public class Apple extends Fruit {  }
2
3
4 public class Fruit {
5    public Fruit(String name) {
6        System.out.println("Fruit's constructor is invoked by:" + name);
7    }
8 }
```

Impact of a Superclass without a Default Constructor

- For a class, a default constructor is implicitly defined *only* when *no* constructor is explicitly defined.
- For a class, if some constructor is explicitly defined, then all constructors must be explicitly defined.
- Find out the errors in the program:

```
1  public class Apple extends Fruit {  }
2
3                       public Apple() {super();}
4  public class Fruit {
5     public Fruit(String name) {
6        System.out.println("Fruit's constructor is invoked by:" + name);
7     }
8  }
```

Impact of a Superclass without a Default Constructor

- For a class, a default constructor is implicitly defined *only* when *no* constructor is explicitly defined.
- For a class, if some constructor is explicitly defined, then all constructors must be explicitly defined.
- Find out the errors in the program:

```
1 public class Apple extends Fruit {  }
2                     public Apple() {super("apple");}
3
4 public class Fruit {
5    public Fruit(String name) {
6       System.out.println("Fruit's constructor is invoked by:" + name);
7    }
8 }
```

Overriding Methods of Superclasses

- A subclass inherits methods from the superclass.
- Sometimes it is necessary for the subclass to *replace* the implementation of a method defined in the superclass. This is referred to as method *overriding*.

```
public class Circle extends Shape {
...
    @Override
    public String toString() {
        return super.toString() + "\nradius is " + radius;
    }
}
```

- A method can be overridden only if it is visible. Thus a `private` method cannot be overridden.
- Always use the `@Override` annotation to check overriding.

Polymorphism and Dynamic Binding

- An object of a *subclass* can be *used* wherever an object of the *superclass* is *required*. Thus a reference to a superclass object may refer to objects of a different class. This feature is known as polymorphism.
- For a *Shape* *x*, *x* may refer to an object of either *Shape*, *Circle*, *Rectangle* or *Triangle*, each of the classes may have their own implementation of method *toString*, due to method overriding.
- Which implementation is used will be determined dynamically, depending on *the class of the object* pointed to by *x* at *runtime*. This capability is known as *dynamic binding*.
- Method implementations are bound to and selected by instances. We often call non-static methods *instance methods*.
- As a consequence, `static` methods cannot be overridden, because they do not belong to instances, they belong to classes.

An Example of Dynamic Binding

```
1  public class DynBindDemo {
2      public static void main(String[] args) {
3          m(new Circle());
4          m(new Rectangle());
5          m(new Triangle());
6          m(new Shape());
7      }
8
9      private static void m(Shape x) {
10         System.out.println(x.toString());
11     }
12 }
```

A single method *m* can output all kinds of shapes, via their overridden *toString* methods, bound at runtime.

Method Matching vs. Binding

- Whether a method can be invoked literally on an instance is a *syntax* issue — matching the method invocation to some method signature.

Book `b = isInClass? new TextBook() : new RecreationBook();`
`b.setDate(2010,9,21);`

Whether `b.setDate(2010,9,21)` is legal
depends on if *Book* has a method `setDate(int,int,int)`.

- When a method can be literally invoked on a reference, which implementation to select is a *semantic* issue — binding a particular implementation according to the instance that the reference actually points to at runtime.

Whether to invoke the *setDate* in *TextBook*
or the *setDate* in *RecreationBook*
depends on what kind of instance *b* is pointing to.

Casting References

- Casting can be used to treat a reference of one class as if the reference is of another class *within* an inheritance hierarchy, at *compile time*.

```
Shape s = (Shape) new Circle(); //upcasting is automatic and safe.
```

- Casting a reference of a superclass to a reference of a subclass is *downcasting*, sometime this is necessary.

```
Shape s = new Circle();
s.setRadius(1.0);          — Syntactically wrong, s is a reference of Shape.
Circle c = s;              — Syntactically wrong, downcasting is not automatic.
Circle c = (Circle)s;      — Syntactically Correct, explicit downcasting is allowed.
                             But it can fail if s does not point to a Circle at runtime.
c.setRadius(1.0);          — Correct.
```

- Casting a reference does *not* change anything, especially not the object pointed to by the reference.

The instanceof Operator

- Use the `instanceof` operator to test whether a reference is pointing to an instance of a class:

```
1  Shape s = new Circle();
2  boolean isCircle = s instanceof Circle;       // should be true.
3  boolean isShape = s instanceof Shape;         // should be true.
4  ...
5  if ( s instanceof Circle ) {
6      System.out.println("The circle radius is " +
7          ((Circle)s).getRadius());
8  } else if ( s instanceof Rectangle ) {
9      System.out.println("The rectangle width is " +
10         ((Rectangle)s).getWidth());
11 }
```

- An instance of a subclass is also an instance of it superclass.

The *equals* Method

- The $a.equals(b)$ method compares the *contents* of two objects a, b.
- The *equals* method is defined in the universal superclass *Object*.
- Any class that implements the content equality should override *equals* method to provide the class's own implementation.
- The default implementation of the *equals* method in the *Object* class is as follows:

```
1 public boolean equals(Object obj) {
2     return (this == obj);
3 }
```

- The == comparison operator is used for comparing two primitive data type values or for determining whether two objects have the same *reference*.

Implementing the *equals* Method

Two rectangles are equal when their widths and heights are respectively equal.

```
1  class Rectangle extends Shape {
2      ...
3      @Override
4      public boolean equals(Object obj) {
5          if ( obj instanceof Rectangle ) {
6              Rectangle r = (Rectangle)obj;
7              return width == r.width && height == r.height;
8          } else return false;
9      }
10 }
```

Practice: Inheritance and Method Overriding

1. Create Java project named *Shape*.
2. Follow the coursework to create the *Shape* class with a constructor that has one — *String name*.
3. Declare the *toString*, *getArea* and *getPerimeter* methods in *Shape*.
4. Define three subclasses of *Shape* — *Circle*, *Rectangle*, *Triangle*.
5. Override the *toString*, *getArea* and *getPerimeter* methods in the subclasses.
6. Create a test class — *TestShape* with the *main* method.
7. Create an array of *Shape* and initialize it with different shapes.
8. Compute the area and perimeter of each shape in the array.
9. Give one good case and one bad case on type downcasts.
10. Submit your source files with comments in `Shape.zip`.