

Inheritance and Polymorphism

Luis Tarrataca

`luis.tarrataca@gmail.com`

CEFET-RJ

1 Introduction

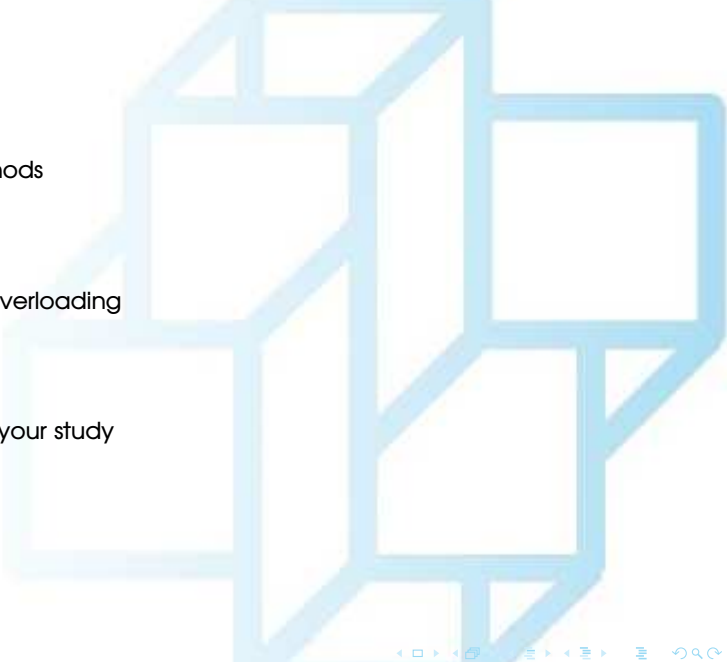
2 Superclasses and Subclasses

3 Using the super Keyword

Calling Superclass Constructors

Constructor Chaining

Calling Superclass Methods

- 
- 4 Overriding Methods
 - 5 Overriding vs. Overloading
 - 6 Where to focus your study

Introduction

What did the previous chapter talked about? Any ideas?

What did the previous chapter talked about? Any ideas?

- Apply class abstraction to develop software;
- Explore \neq 's between procedural and OO paradigm;
- Express relationships between classes;
- Design programs using the object-oriented paradigm

What will this chapter discuss? Any ideas?

What will this chapter discuss? Any ideas?

Several concepts (1/3):

- How to define a subclass from a superclass through inheritance;
- How to invoke superclass's constructors and methods;
- How to override instance methods in the subclass;
- How to distinguish differences between overriding and overloading;

What will this chapter discuss? Any ideas?

Several concepts (2/3):

- Explore the toString() method in the Object class;
- Discover polymorphism and dynamic binding;
- Describe casting and explain why explicit downcasting is necessary;
- Explore the equals method in the Object class;

What will this chapter discuss? Any ideas?

Several concepts (3/3):

- Enable data / methods in a superclass from subclasses:
 - Protected visibility modifier;
- Prevent class extending and method overriding using the final modifier;

Lets start with a simple question:

What are the \neq 's between procedural and OO programming paradigms?
Any ideas?

Lets start with a simple question:

What are the \neq 's between procedural and OO programming paradigms?
Any ideas?

Procedural paradigm focus:

- Designing methods;

OO paradigm focus:

- Combines the power of the procedural paradigm:
 - Whilst integrating data with operations into objects.
- Software reutilization;

Lets focus on the latter one:

Today we will see a concept that allows for software reutilization? Any ideas?

Lets focus on the latter one:

Today we will see a concept that allows for software reutilization? Any ideas?

Inheritance:

- Allows you to define new classes from existing classes;
- *E.g.* consider classes: circles, rectangles, and triangles.
 - Classes have many common features;
 - Best way to design these classes? Avoid redundancy?:
 - Inheritance;

Superclasses and Subclasses

Recall that:

- Classes are use to model objects of the same type;

However:

- \neq classes may have some common properties / behaviours:

What can we do with the common properties / behaviours? Any ideas?

What can we do with the common properties / behaviours? Any ideas?

Define a **generalized** class:

- That can be shared by other classes;

Define **specialized** class:

- **Extending** generalized class;
- Specialized classes inherit properties / methods from general class;

Lets see more with an example

Example

Suppose you want to design classes such as circles and rectangles.

- These can be seen as geometric objects:
 - They share **common** attributes:
 - Color;
 - Filled;
 - They share **common** methods:
 - Corresponding setter / getter methods;

How can we model this situation? Any ideas?

How can we model this situation? Any ideas?

Create general class `GeometricObject`:

- Used to model all geometric objects
- Contains attributes `color` and `fill`:
 - And appropriate getter and setter methods;
- Assume that class also contains `dateCreated` property;
- Add a **`toString()`** method:
 - String representation of the object;

How can we represent class GeometricObject through UML? Any ideas?

How can we represent class `GeometricObject` through UML? Any ideas?

GeometricObject

-color: String
 -filled: boolean
 -dateCreated: java.util.Date

+GeometricObject()
 +GeometricObject(color: String, filled: boolean)
 +getColor(): String
 +setColor(color: String): void
 +isFilled(): boolean
 +setFilled(filled: boolean): void
 +getDateCreated(): java.util.Date
 +toString(): String

The color of the object (default: white).

Indicates whether the object is filled with a color (default: false).

The date when the object was created.

Creates a `GeometricObject`.

Creates a `GeometricObject` with the specified color and filled values.

Returns the color.

Sets a new color.

Returns the filled property.

Sets a new filled property.

Returns the dateCreated.

Returns a string representation of this object.

Figure: The `GeometricObject` class.(Source: (Liang, 2014))

How can we represent class Circle / Rectangle through UML? Any ideas?

How can we represent class Circle / Rectangle through UML? Any ideas?

- Define Circle class extending GeometricObject class:
- Triangular arrow pointing to the superclass:
 - Represents **inheritance** relationship;

Terminology:

- Consider two classes: C1 **extends / inherits** from C2
 - C1 is called a **subclass** or **child class**
 - C2 is called a **superclass** or **parent class**

Ok, but in practice what does **inheritance** mean? Any ideas?

Ok, but in practice what does **inheritance** mean? Any ideas?

Subclass:

- Inherits accessible attributes / methods from superclass;
- May add new data fields / methods.

Circle class:

- Inherits accessible attributes / methods from GeometricObject class;
- Adds new data field: radius
 - And associated getter and setter methods;
- Adds methods:
 - `getArea()` - returns area;
 - `getPerimeter()` - returns perimeter;
 - `getDiameter()` - returns diameter;

Rectangle class:

- Inherits accessible attributes / methods from GeometricObject class;
- Adds new data fields: width and height
 - And associated getter and setter methods;
- Adds methods:
 - `getArea()` - returns area;
 - `getPerimeter()` - returns perimeter;

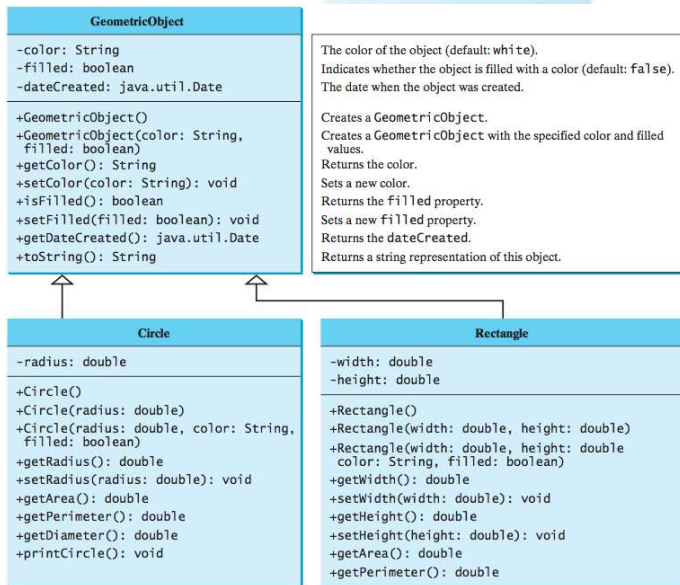


Figure: The GeometricObject class is the superclass for Circle and Rectangle. (Source: (Liang, 2014))

How is inheritance be implemented in Java? Any ideas?


```
public class GeometricObject {
    private String color = "white";
    private boolean filled;
    private java.util.Date dateCreated;
    /** Construct a default geometric object */
    public SimpleGeometricObject() {
        dateCreated = new java.util.Date();
    }

    /** Construct a geometric object with the specified color and filled value */
    public SimpleGeometricObject(String color, boolean filled) {
        dateCreated = new java.util.Date();
        this.color = color;
        this.filled = filled;
    }

    /** Return color */
    public String getColor() { return color; }

    /** Set a new color */
    public void setColor(String color) { this.color = color; }

    /** Return filled. Since filled is boolean,
        its getter method is named isFilled */
    public boolean isFilled() { return filled; }

    /** Set a new filled */
    public void setFilled(boolean filled) { this.filled = filled; }

    /** Get dateCreated */
    public java.util.Date getDateCreated() { return dateCreated; }

    /** Return a string representation of this object */
    public String toString() { return "created on " + dateCreated + "\ncolor: " + color + " and filled: " + filled; }
}
```

```
public class Circle extends GeometricObject{
    private double radius;

    public CircleFromSimpleGeometricObject() {}
    public CircleFromSimpleGeometricObject(double radius) { this.radius = radius; }
    public CircleFromSimpleGeometricObject(double radius, String color, boolean filled) {
        this.radius = radius;
        setColor( color );
        setFilled ( filled );
    }

    /** Return radius */
    public double getRadius() { return radius; }

    /** Set a new radius */
    public void setRadius(double radius) { this.radius = radius; }

    /** Return area */
    public double getArea() { return radius * radius * Math.PI;}

    /** Return diameter */
    public double getDiameter() { return 2 * radius; }

    /** Return perimeter */
    public double getPerimeter() { return 2 * radius * Math.PI; }

    /** Print the circle info */
    public void printCircle() { System.out.println ("The circle is created " + getDateCreated() + " and the radius is " + radius); }
}
```

Important point:

- Overloaded constructor `Circle(double radius, String color, boolean filled)`:
 - Implemented by invoking `setColor` / `setFilled` methods:
 - Public methods defined in superclass `GeometricObject`

From the previous slide:

Could the constructor change the attributes from the superclass directly?
Any ideas?

Would the following code be correct? Any ideas?

```
public CircleFromSimpleGeometricObject( double radius, String color, boolean
    filled ) {
    this . radius = radius ;
    this . color = color ;
    this . filled = filled ;
}
```

Would the following code be correct? Any ideas?

```
public CircleFromSimpleGeometricObject( double radius, String color, boolean
    filled ) {
    this .radius = radius ;
    this .color = color ;
    this . filled = filled ;
}
```

Wrong: Color and filled are **private** attributes:

- Cannot be accessed in any class other than in the GeometricObject;
- Only way to read / write color and filled is through getter / setter methods.

```
public class Rectangle extends GeometricObject {
    private double width;
    private double height;

    public RectangleFromSimpleGeometricObject() {}
    public RectangleFromSimpleGeometricObject( double width, double height) {
        this .width = width;
        this .height = height;
    }
    public RectangleFromSimpleGeometricObject( double width, double height, String color, boolean filled) {
        this .width = width;
        this .height = height;
        setColor(color);
        setFilled ( filled );
    }

    /** Return width */
    public double getWidth() { return width; }

    /** Set a new width */
    public void setWidth(double width) { this .width = width; }

    /** Return height */
    public double getHeight() { return height; }

    /** Set a new height */
    public void setHeight(double height) { this .height = height; }

    /** Return area */
    public double getArea() { return width * height; }

    /** Return perimeter */
    public double getPerimeter() { return 2 * (width + height); }
}
```

Important points regarding inheritance (1/2):

- Subclass is **not** a subset of its superclass:
 - Usually contains more information / methods than superclass;
- **Private** superclass attributes are **not** accessible outside the class:
 - Only accessible through superclass public setters / getters;

Important points regarding inheritance (2/2):

- Inheritance is used to model the is-a relationship:
 - Subclass and superclass must have the is-a relationship;
- Java class may inherit directly from only one superclass:
 - A.k.a single inheritance;
 - Other programming languages allow:
 - Subclass to be derived from several classes.

Using the **super** Keyword

Subclass inherits accessible attributes / methods from superclass:

But does the subclass inherit the superclass constructors? Any ideas?

Can the superclass's constructors be invoked from a subclass? Any ideas?

Using the **super** Keyword

Subclass inherits accessible attributes / methods from superclass:

But does the subclass inherit the superclass constructors? Any ideas?

No, superclass constructors are not inherited...

Can the superclass's constructors be invoked from a subclass? Any ideas?

Yes, through the **super** keyword...

super keyword:

- Refers to the superclass of the class;
- Can be used in two ways:
 - Call a superclass constructor;
 - Call a superclass method;

Calling Superclass Constructors

Unlike attributes / methods:

- Superclass constructors are not inherited by a subclass;

However, superclass constructors:

- Can be invoked from subclasses using **super** keyword:

super(), or **super(parameters)**;

- Statement **super()** invokes:
 - No-argument superclass constructor;
- Statement **super(arguments)** invokes:
 - Superclass constructor that matches arguments;

How can we adapt previous Circle constructors to use **super**? Any ideas?

```
public class Circle extends GeometricObject{
    private double radius;

    ...

    public CircleFromSimpleGeometricObject(double radius, String color, boolean
        filled ) {
        this .radius = radius;
        setColor( color );
        setFilled ( filled );
    }

    ...
}
```

How can we adapt previous Circle constructors to use **super**? Any ideas?

```
public class Circle extends GeometricObject{
    private double radius;

    ...
    public CircleFromSimpleGeometricObject(double radius,
        String color, boolean filled) {
        this .radius = radius;
        setColor( color );
        setFilled ( filled );
    }
    ...
}
```

```
public class Circle extends GeometricObject{
    private double radius;

    ...
    public CircleFromSimpleGeometricObject(double radius,
        String color, boolean filled) {
        super( color, filled )
        this .radius = radius;
    }
    ...
}
```

Important:

- **super** call must be 1st statement in the constructor;
- Invoking superclass constructor name in subclass causes a syntax error.

Constructor Chaining

If superclass constructor is **not** invoked explicitly:

- Compiler automatically puts **super()** as the first statement in the constructor

```
public ClassName() {  
    // some statements  
}
```

Equivalent

```
public ClassName() {  
    super();  
    // some statements  
}
```

```
public ClassName(double d) {  
    // some statements  
}
```

Equivalent

```
public ClassName(double d) {  
    super();  
    // some statements  
}
```

When constructing an object of a subclass:

- Subclass constructor first invokes its superclass constructor;
- Superclass constructor invokes its parent-class constructor;
- ...
- Process continues until last constructor in inheritance chain is called

Conclusion: constructing an instance of a class invokes:

- Constructors of all the superclasses along the inheritance chain.

This is called **constructor chaining**.

What is the output of the following code? Any ideas?

```
public class Faculty extends Employee {
    public static void main(String[] args) {
        new Faculty();
    }

    public Faculty(){
        System.out.println("(4) Performs Faculty's tasks");
    }
}

class Employee extends Person {
    public Employee(){
        this("(2) Invoke Employee's overloaded constructor");
        System.out.println("(3) Performs Employee's tasks ");
    }

    public Employee(String s) {
        System.out.println(" s ");
    }
}

class Person{
    public Person(){
        System.out.println("(1) Performs Person's tasks");
    }
}
```

What is the output of the following code? Any ideas?

-
- (1) Performs Person's tasks
 - (2) Invoke Employee's overloaded constructor
 - (3) Performs Employee's tasks
 - (4) Performs Faculty's tasks
-

Why does the program produces the previous output? Any ideas?

Why does the program produces the previous output? Any ideas?

- 1 new Faculty() invokes Faculty's no-arg constructor;
- 2 Faculty is a subclass of Employee
 - Employee's no-arg constructor is invoked;
- 3 Employee is a subclass of Person:
 - Person's no-arg constructor is invoked before any statements

Why does the program produces the previous output? Any ideas?

- 1 new Faculty() invokes Faculty's no-arg constructor;
- 2 Faculty is a subclass of Employee
 - Employee's no-arg constructor is invoked;
- 3 Employee is a subclass of Person:
 - Person's no-arg constructor is invoked before any statements

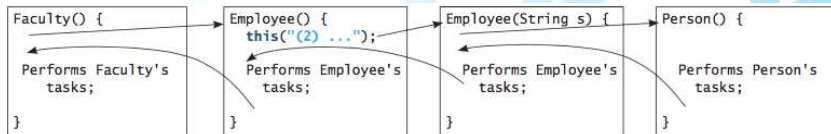


Figure: (Source: (Liang, 2014))

Example

What about the following code? Any ideas?

```
public class Apple extends Fruit {  
}  
  
class Fruit {  
    public Fruit (String name) {  
        System.out.println (" Fruit 's constructor is invoked");  
    }  
}
```

Example

What about the following code? Any ideas?

```
public class Apple extends Fruit {  
}  
  
class Fruit {  
    public Fruit (String name) {  
        System.out.println (" Fruit 's constructor is invoked");  
    }  
}
```

No constructor is explicitly defined in Apple:

- Apple's default no-arg constructor is defined implicitly;
- Since Apple is a subclass of Fruit:
 - Apple's default constructor automatically invokes Fruit's no-arg constructor:
 - Fruit does not have a no-arg constructor;
 - Therefore, the program cannot be compiled.

What can be done to solve the previous problem? Any ideas?

Design tip:

- If possible, provide a no-arg constructor for every class:
 - Makes class easy to extend and avoids errors.

Calling Superclass Methods

If **super** keyword is used to represent the **superclass**:

How can we access the superclass methods? Any ideas?

Calling Superclass Methods

If **super** keyword is used to represent the **superclass**:

How can we access the superclass methods? Any ideas?

Through the syntax:

```
super.method( parameters );
```

printCircle() method (Slide 31) could be rewritten:

```
public void printCircle () {  
    System.out.println ("The circle is created " +  
        super.getDateCreated() + " and the radius is " + radius);  
}
```

However, in this case, this is not necessary:

- getDateCreated is a method in the GeometricObject class:
 - Inherited by the Circle class.

However, in some cases:

- **super** keyword is necessary;

Overriding Methods

What is the concept of method overriding? Any ideas?

Overriding Methods

What is the concept of method overriding? Any ideas?

Method Overriding:

- Subclass provides implementation for inherited superclass method(s);
- Do not confuse with previously seen **method overloading**:
 - “Sobrecarga” in Portuguese =P

Lets see an example =>

toString() method in GeometricObject class (Slide 30):

- returns the string representation of a geometric object.

Idea: have toString method specific to Circle class:

How can we define a toString method specific to Circle class? Any ideas?

How can we define a toString method specific to Circle class? Any ideas?

Simple: Redefine the method in the subclass:

```
public class CircleFromSimpleGeometricObject extends SimpleGeometricObject {  
  
    // Override the toString method defined in the superclass  
    public String toString () {  
        return super.toString () + "\nradius is " + radius ;  
    }  
}
```

Important points (1/2):

- Instance method can be overridden only if it is accessible:
 - Private method cannot be overridden:
 - Not accessible outside its own class
 - If a method defined in a subclass is private in its superclass:
 - Two methods are completely unrelated.

Important points (2/2):

- Static methods can be inherited:
 - **However**, they **cannot** be overridden;
 - If a static method defined in the superclass is redefined in a subclass
 - Method defined in the superclass is hidden
 - Hidden static methods can be invoked: **SuperClassName.staticMethodName**

Overriding vs. Overloading

What is the \neq between Overloading and Overriding? Any ideas?

Overriding vs. Overloading

What is the \neq between Overloading and Overriding? Any ideas?

Overloading methods:

- Define multiple methods with the same name but different signatures.

Overriding methods:

- Provide a new implementation for a method in the subclass;

Example

What is the \neq between these two codes? Any ideas?

```
public class Test {
    public static void main(String[] args) {
        A a = new A();
        a.p(10);
        a.p(10.0);
    }
}

class B {
    public void p(double i) {
        System.out.println(i * 2);
    }
}

class A extends B {
    // This method overrides the method in B
    public void p(double i) {
        System.out.println(i);
    }
}
```

Listing 1: Overriding

```
public class Test {
    public static void main(String[] args) {
        A a = new A();
        a.p(10);
        a.p(10.0);
    }
}

class B {
    public void p(double i) {
        System.out.println(i * 2);
    }
}

class A extends B {
    // This method overloads the method in B
    public void p(int i) {
        System.out.println(i);
    }
}
```

Listing 2: Overloading

Test class in **overriding** example:

- a.p(10) and a.p(10.0) invoke p(double i) method defined in class A:
 - Displays 10

Test class in **overloading** example:

- a.p(10) invokes p(int i) method defined in class A:
 - Displays 10
- a.p(10.0) invokes p(double i) method defined in class B:
 - Displays 20

Overridden methods:

- Are in different classes related by inheritance;

Overloaded methods:

- Can be either in the same class or different classes related by inheritance.;

To avoid mistakes:

- Use Java special override annotation syntax:
 - Place `@Override` before the method in the subclass

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

Where to focus your study

After this class you should be able to (1/3):

- How to define a subclass from a superclass through inheritance;
- How to invoke superclass's constructors and methods;
- How to override instance methods in the subclass;
- How to distinguish differences between overriding and overloading;

Where to focus your study

After this class you should be able to (2/3):

- Explore the `toString()` method in the `Object` class;
- Discover polymorphism and dynamic binding;
- Describe casting and explain why explicit downcasting is necessary;
- Explore the `equals` method in the `Object` class;

Where to focus your study

After this class you should be able to (3/3):

- Enable data / methods in a superclass from subclasses:
 - Protected visibility modifier;
- Prevent class extending and method overriding using the final modifier;

References I



Liang, Y. (2014).

Introduction to Java Programming.

Pearson Education.