Chapter 10
Inheritance and Polymorphism

Dr. Hikmat Jaber
Motivations

Suppose you will define classes to model circles, rectangles, and triangles. These classes have many common features. What is the best way to design these classes so to avoid redundancy? The answer is to use inheritance.

- **Inheritance** is the process of deriving new class from an existing class.
- Inheritance is an important feature for software reusability.
# Superclasses and Subclasses

## GeometricObject

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-color: String</td>
<td>The color of the object (default: white).</td>
</tr>
<tr>
<td>-filled: boolean</td>
<td>Indicates whether the object is filled with a color (default: false).</td>
</tr>
<tr>
<td>-dateCreated: java.util.Date</td>
<td>The date when the object was created.</td>
</tr>
</tbody>
</table>

**Methods**

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

**Circle**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-radius: double</td>
<td>A Circle with a specified radius.</td>
</tr>
</tbody>
</table>

**Methods**

- +Circle()
- +Circle(radius: double)
- +Circle(radius: double, color: String, filled: boolean)
- +getRadius(): double
- +setWidth(radius: double): void
- +getArea(): double
- +getPerimeter(): double
- +printCircle(): void

**Rectangle**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-width: double</td>
<td>A Rectangle with a specified width.</td>
</tr>
<tr>
<td>-height: double</td>
<td>A Rectangle with a specified height.</td>
</tr>
</tbody>
</table>

**Methods**

- +Rectangle()
- +Rectangle(width: double, height: double)
- +Rectangle(width: double, height: double, color: String, filled: boolean)
- +getWidth(): double
- +setWidth(width: double): void
- +getHeight(): double
- +setHeight(height: double): void
- +getArea(): double
- +getPerimeter(): double
import java.util.Date;

public class GeometricObject {
    private String color = "white";
    private boolean filled;
    private Date dateCreated;

    public GeometricObject() { dateCreated = new Date(); }
    public GeometricObject(String Color, boolean filled) {
        dateCreated = new Date();
        this.color = Color;
        this.filled = filled;
    }
    public String getColor() { return color; }
    public void setColor(String color) { this.color = color; }
    public boolean isFilled() { return filled; }
    public void setFilled(boolean filled) { this.filled = filled; }
    public 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 Circle() {
    }

    public Circle(double radius) {
        this.radius = radius;
    }

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

    public double getRadius() {
        return radius;
    }

    public void setRadius(double radius) {
        this.radius = radius;
    }

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

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

    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);
    }
}

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

    public Rectangle() {}
    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }
    public Rectangle(double width, double height, String color, boolean filled) {
        this.width = width;
        this.height = height;
        setColor(color);
        setFilled(filled);
    }
    public double getWidth() { return width; }
    public void setWidth(double width) { this.width = width; }
    public double getHeight() { return height; }
    public void setHeight(double height) { this.height = height; }
    public double getArea() { return width * height; }
    public double getPerimeter() { return 2 * (width + height); }
}
public class TestCircleRectangle {
    public static void main(String[] args) {
        Circle circle = new Circle(1);
        System.out.println("A circle " + circle.toString());
        System.out.println("The radius is " + circle.getRadius());
        System.out.println("The area is " + circle.getArea());
        System.out.println("The diameter is " + circle.getDiameter());

        Rectangle rectangle = new Rectangle(2, 4);
        System.out.println("A rectangle " + rectangle.toString());
        System.out.println("The area is " + rectangle.getArea());
        System.out.println("The perimeter is " + rectangle.getPerimeter());
    }
}

Output:
Are superclass’s Constructor Inherited?

No. They are not inherited.

They are invoked explicitly or implicitly.

Explicitly using the super keyword.

A constructor is used to construct an instance of a class. Unlike properties and methods, a superclass's constructors are not inherited in the subclass. They can only be invoked from the subclasses' constructors, using the keyword super. If the keyword super is not explicitly used, the superclass's no-arg constructor is automatically invoked.
A constructor may invoke an overloaded constructor or its superclass’s constructor. If none of them is invoked explicitly, the compiler puts `super()` as the first statement in the constructor. For example,

```java
public A(double d) {
  // some statements
}
```

is equivalent to

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

Similarly,

```java
public A() {
}
```

is equivalent to

```java
public A() {
  super();
}
```

```java
public A(double d) {
  // some statements
}
```

is equivalent to

```java
public A(double d) {
  super();
  // some statements
}
```
Using the Keyword `super`

The keyword `super` refers to the superclass of the class in which `super` appears. This keyword can be used in two ways:

- To call a superclass constructor
- To call a superclass method
CAUTION

You must use the keyword `super` to call the superclass constructor. Invoking a superclass constructor’s name in a subclass causes a syntax error. Java requires that the statement that uses the keyword `super` appear first in the constructor.
Constructor Chaining

Constructing an instance of a class invokes all the superclasses’ constructors along the inheritance chain. This is called constructor chaining.

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

    public Faculty() {
        System.out.println("(4) Faculty's no-arg constructor is invoked");
    }
}

class Employee extends Person {
    public Employee() {
        this("(2) Invoke Employee’s overloaded constructor");
        System.out.println("(3) Employee's no-arg constructor is invoked");
    }

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

class Person {
    public Person() {
        System.out.println("(1) Person's no-arg constructor is invoked");
    }
}
```
public class Faculty extends Employee {
    public static void main(String[] args) {
        new Faculty();
    }

    public Faculty() {
        System.out.println("(4) Faculty's no-arg constructor is invoked");
    }
}

class Employee extends Person {
    public Employee() {
        this("(2) Invoke Employee's overloaded constructor");
        System.out.println("(3) Employee's no-arg constructor is invoked");
    }

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

class Person {
    public Person() {
        System.out.println("(1) Person's no-arg constructor is invoked");
    }
}

1. Start from the main method
public class Faculty extends Employee {
    public static void main(String[] args) {
        new Faculty();
    }

    public Faculty() {
        System.out.println("(4) Faculty's no-arg constructor is invoked");
    }
}

class Employee extends Person {
    public Employee() {
        this("(2) Invoke Employee’s overloaded constructor");
        System.out.println("(3) Employee's no-arg constructor is invoked");
    }

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

class Person {
    public Person() {
        System.out.println("(1) Person's no-arg constructor is invoked");
    }
}
public class Faculty extends Employee {
    public static void main(String[] args) {
        new Faculty();
    }

    public Faculty() {
        System.out.println("(4) Faculty's no-arg constructor is invoked");
    }
}

class Employee extends Person {
    public Employee() {
        this("(2) Invoke Employee's overloaded constructor");
        System.out.println("(3) Employee's no-arg constructor is invoked");
    }

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

class Person {
    public Person() {
        System.out.println("(1) Person's no-arg constructor is invoked");
    }
}
public class Faculty extends Employee {
    public static void main(String[] args) {
        new Faculty();
    }

    public Faculty() {
        System.out.println("(4) Faculty's no-arg constructor is invoked");
    }
}

class Employee extends Person {
    public Employee() {
        this("(2) Invoke Employee’s overloaded constructor");
        System.out.println("(3) Employee's no-arg constructor is invoked");
    }

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

class Person {
    public Person() {
        System.out.println("(1) Person's no-arg constructor is invoked");
    }
}
public class Faculty extends Employee {
    public static void main(String[] args) {
        new Faculty();
    }

    public Faculty() {
        System.out.println("(4) Faculty's no-arg constructor is invoked");
    }
}

class Employee extends Person {
    public Employee() {
        this("(2) Invoke Employee’s overloaded constructor");
        System.out.println("(3) Employee's no-arg constructor is invoked");
    }

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

class Person {
    public Person() {
        System.out.println("(1) Person's no-arg constructor is invoked");
    }
}
public class Faculty extends Employee {
    public static void main(String[] args) {
        new Faculty();
    }

    public Faculty() {
        System.out.println("(4) Faculty's no-arg constructor is invoked");
    }
}

class Employee extends Person {
    public Employee() {
        this("(2) Invoke Employee’s overloaded constructor");
        System.out.println("(3) Employee's no-arg constructor is invoked");
    }

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

class Person {
    public Person() {
        System.out.println("(1) Person's no-arg constructor is invoked");
    }
}
public class Faculty extends Employee {
    public static void main(String[] args) {
        new Faculty();
    }

    public Faculty() {
        System.out.println("(4) Faculty's no-arg constructor is invoked");
    }
}

class Employee extends Person {
    public Employee() {
        this("(2) Invoke Employee’s overloaded constructor");
        System.out.println("(3) Employee's no-arg constructor is invoked");
    }

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

class Person {
    public Person() {
        System.out.println("(1) Person's no-arg constructor is invoked");
    }
}

7. Execute println
public class Faculty extends Employee {
    public static void main(String[] args) {
        new Faculty();
    }

    public Faculty() {
        System.out.println("(4) Faculty's no-arg constructor is invoked");
    }
}

class Employee extends Person {
    public Employee() {
        this("(2) Invoke Employee’s overloaded constructor");
        System.out.println("(3) Employee's no-arg constructor is invoked");
    }

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

class Person {
    public Person() {
        System.out.println("(1) Person's no-arg constructor is invoked");
    }
}
public class Faculty extends Employee {
    public static void main(String[] args) {
        new Faculty();
    }

    public Faculty() {
        System.out.println("(4) Faculty's no-arg constructor is invoked");
    }
}

class Employee extends Person {
    public Employee() {
        this("(2) Invoke Employee's overloaded constructor");
        System.out.println("(3) Employee's no-arg constructor is invoked");
    }

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

class Person {
    public Person() {
        System.out.println("(1) Person's no-arg constructor is invoked");
    }
}
Example on the Impact of a Superclass without no-arg Constructor

Find out the errors in the program:

```java
public class Apple extends Fruit {
}

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

**Corrected Program:**

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

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

A subclass extends properties and methods from the superclass. You can also:

✧ Add new properties
✧ Add new methods
✧ Override the methods of the superclass
Calling Superclass Methods

You could rewrite the `printCircle()` method in the `Circle` class as follows:

```java
public void printCircle() {
    System.out.println("The circle is created " +
        super.getDateCreated() + " and the radius is " + radius);
}
```
A subclass inherits methods from a superclass. Sometimes it is necessary for the subclass to modify the implementation of a method defined in the superclass. This is referred to as *method overriding*.

```java
public class Circle extends GeometricObject {
    // Other methods are omitted

    /** Override the toString method defined in GeometricObject */
    public String toString() {
        return super.toString() + "\nradius is " + radius;
    }
}
```
NOTE

An instance method can be overridden only if it is accessible. Thus a private method cannot be overridden, because it is not accessible outside its own class. If a method defined in a subclass is private in its superclass, the two methods are completely unrelated.
Like an instance method, a **static method** can be **inherited**. **However**, a **static method cannot be overridden**. If a static method defined in the superclass is redefined in a subclass, the method defined in the superclass is hidden.
Overriding vs. Overloading

```java
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);
    }
}
```

```java
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(double i) {
        System.out.println(i);
    }
}
```
Every class in Java is descended from the `java.lang.Object` class. If no inheritance is specified when a class is defined, the superclass of the class is `Object`.

```java
public class Circle {
  ...
}
```

Equivalent

```java
public class Circle extends Object {
  ...
}
```
The `toString()` method in Object

The `toString()` method returns a string representation of the object. The default implementation returns a string consisting of a class name of which the object is an instance, the at sign (@), and a number representing this object.

```java
Loan loan = new Loan();
System.out.println(loan.toString());
```

The code displays something like `Loan@15037e5`. This message is not very helpful or informative. Usually you should override the `toString` method so that it returns a digestible string representation of the object.
Polymorphism, Dynamic Binding and Generic Programming

```java
public class PolymorphismDemo {
    public static void main(String[] s) {
        m(new GraduateStudent());
        m(new Student());
        m(new Person());
        m(new Object());
    }
    public static void m(Object x) {
        System.out.println(x.toString());
    }
}
class GraduateStudent extends Student {
}
class Student extends Person {
    public String toString() {
        return "Student";
    }
}
class Person extends Object {
    public String toString() {
        return "Person";
    }
}
```

Method m takes a parameter of the Object type. You can invoke it with any object.

An object of a subtype can be used wherever its supertype value is required. This feature is known as **polymorphism**.

When the method `m(Object x)` is executed, the argument `x`’s `toString` method is invoked. `x` may be an instance of `GraduateStudent`, `Student`, `Person`, or `Object`. Classes `GraduateStudent`, `Student`, `Person`, and `Object` have their own implementation of the `toString` method. Which implementation is used will be determined dynamically by the Java Virtual Machine at runtime. This capability is known as **dynamic binding**.
Dynamic Binding

Dynamic binding works as follows: Suppose an object \( o \) is an instance of classes \( C_1, C_2, ..., C_{n-1}, \) and \( C_n \), where \( C_1 \) is a subclass of \( C_2 \), \( C_2 \) is a subclass of \( C_3 \), ..., and \( C_{n-1} \) is a subclass of \( C_n \). That is, \( C_n \) is the most general class, and \( C_1 \) is the most specific class. In Java, \( C_n \) is the `Object` class. If \( o \) invokes a method \( p \), the JVM searches the implementation for the method \( p \) in \( C_1, C_2, ..., C_{n-1} \) and \( C_n \), in this order, until it is found. Once an implementation is found, the search stops and the first-found implementation is invoked.

```
\[ C_n \leftarrow C_{n-1} \leftarrow \ldots \leftarrow C_2 \leftarrow C_1 \]
```

Since \( o \) is an instance of \( C_1 \), \( o \) is also an instance of \( C_2, C_3, \ldots, C_{n-1}, \) and \( C_n \).
Matching a method signature and binding a method implementation are two issues. The compiler finds a matching method according to parameter type, number of parameters, and order of the parameters at compilation time. A method may be implemented in several subclasses. The Java Virtual Machine dynamically binds the implementation of the method at runtime.
Polymorphism allows methods to be used generically for a wide range of object arguments. This is known as generic programming. If a method’s parameter type is a superclass (e.g., Object), you may pass an object to this method of any of the parameter’s subclasses (e.g., Student or String). When an object (e.g., a Student object or a String object) is used in the method, the particular implementation of the method of the object that is invoked (e.g., toString) is determined dynamically.

```java
generic Programming

class Person extends Object {
    public String toString() {
        return "Person";
    }
}
class Student extends Person {
    public String toString() {
        return "Student";
    }
}
class GraduateStudent extends Student {
}
public class PolymorphismDemo {
    public static void main(String[] args) {
        m(new GraduateStudent());
        m(new Student());
        m(new Person());
        m(new Object());
    }
    public static void m(Object x) {
        System.out.println(x.toString());
    }
}
```
Casting Objects

You have already used the casting operator to convert variables of one primitive type to another. *Casting* can also be used to convert an object of one class type to another within an inheritance hierarchy. In the preceding section, the statement

```java
m(new Student());
```

assigns the object `new Student()` to a parameter of the `Object` type. This statement is equivalent to:

```java
Object o = new Student(); // Implicit casting
m(o);
```

The statement `Object o = new Student()`, known as *implicit casting*, is legal because an instance of `Student` is automatically an instance of `Object`. 
### Why Casting Is Necessary?

Suppose you want to assign the object reference o to a variable of the Student type using the following statement:

```java
Student b = o;
```

A compilation error would occur. Why does the statement `Object o = new Student()` work and the statement `Student b = o` doesn’t? This is because a Student object is always an instance of Object, but an Object is not necessarily an instance of Student. Even though you can see that o is really a Student object, the compiler is not so clever to know it. To tell the compiler that o is a Student object, use an explicit casting. The syntax is similar to the one used for casting among primitive data types. Enclose the target object type in parentheses and place it before the object to be cast, as follows:

```java
Student b = (Student)o; // Explicit casting
```
Casting from Superclass to Subclass

Explicit casting must be used when casting an object from a superclass to a subclass. This type of casting may not always succeed.

Apple x = (Apple)fruit;

Orange x = (Orange)fruit;
Use the `instanceof` operator to test whether an object is an instance of a class:

```java
Object myObject = new Circle();
... // Some lines of code
/** Perform casting if myObject is an instance of Circle */
if (myObject instanceof Circle) {
    System.out.println("The circle diameter is "+
                       ((Circle)myObject).getDiameter());
...}
```
To help understand casting, you may also consider the analogy of fruit, apple, and orange with the `Fruit` class as the superclass for `Apple` and `Orange`. An apple is a fruit, so you can always safely assign an instance of `Apple` to a variable for `Fruit`. However, a fruit is not necessarily an apple, so you have to use explicit casting to assign an instance of `Fruit` to a variable of `Apple`. 
Example: Demonstrating Polymorphism and Casting

This example creates two geometric objects: a circle, and a rectangle, invokes the `displayGeometricObject` method to display the objects. The `displayGeometricObject` displays the area and diameter if the object is a circle, and displays area if the object is a rectangle.
public class CastingDemo {
    public static void main(String[] args) {
        Object object1 = new Circle(1);
        Object object2 = new Rectangle(1, 1);

        // Display circle and rectangle
        displayObject(object1);
        displayObject(object2);
    }

    /** A method for displaying an object */
    public static void displayObject(Object object) {
        if (object instanceof Circle) {
            System.out.println("The circle area is " + ((Circle)object).getArea());
            System.out.println("The circle diameter is " + ((Circle)object).getDiameter());
        } else if (object instanceof Rectangle) {
            System.out.println("The rectangle area is " + ((Rectangle)object).getArea());
        }
    }
}
The `equals()` method compares the contents of two objects. The default implementation of the `equals` method in the `Object` class is as follows:

```java
public boolean equals(Object obj) {
    return (this == obj);
}
```

For example, the `equals` method is overridden in the `Circle` class.

```java
public boolean equals(Object o) {
    if (o instanceof Circle) {
        return radius == ((Circle)o).radius;
    }
    else
        return false;
}
```
NOTE

The `==` comparison operator is used for comparing two primitive data type values or for determining whether two objects have the same references. The `equals` method is intended to test whether two objects have the same contents, provided that the method is modified in the defining class of the objects. The `==` operator is stronger than the `equals` method, in that the `==` operator checks whether the two reference variables refer to the same object.
The **protected** Modifier

- The *protected* modifier can be applied on data and methods in a class. A protected data or a protected method in a public class can be accessed by any class in the same package or its subclasses, even if the subclasses are in a different package.

- `private`, `default`, `protected`, `public`

Visibility increases

- `private`, none (if no modifier is used), `protected`, `public`
## Accessibility Summary

<table>
<thead>
<tr>
<th>Modifier on members in a class</th>
<th>Accessed from the same class</th>
<th>Accessed from the same package</th>
<th>Accessed from a subclass</th>
<th>Accessed from a different package</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>protected</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>default</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>private</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Visibility Modifiers

```java
package p1;

class C1 {
    public int x;
    protected int y;
    int z;
    private int u;

    protected void m() {
    }
}

class C2 {
    C1 o = new C1();
    can access o.x;
    can access o.y;
    can access o.z;
    cannot access o.u;
    can invoke o.m();
}

class C3 extends C1 {
    can access x;
    can access y;
    can access z;
    cannot access u;
    can invoke m();
}

class C4 extends C1 {
    can access x;
    can access y;
    cannot access z;
    cannot access u;
    can invoke m();
}

class C5 {
    C1 o = new C1();
    can access o.x;
    cannot access o.y;
    cannot access o.z;
    cannot access o.u;
    cannot invoke o.m();
}
```
A subclass may override a protected method in its superclass and change its visibility to public. However, a subclass cannot weaken the accessibility of a method defined in the superclass. For example, if a method is defined as public in the superclass, it must be defined as public in the subclass.
The modifiers are used on classes and class members (data and methods), except that the \texttt{final} modifier can also be used on local variables in a method. A final local variable is a constant inside a method.
The final Modifier

- The **final** class cannot be extended:
  ```java
  final class Math {
      ...
  }
  ```

- The **final** variable is a constant:
  ```java
  final static double PI = 3.14159;
  ```

- The **final** method cannot be overridden by its subclasses.