PROGRAMMING IN JAVA

Lecture 8

Inheritance and Polymorphism

Superclasses, Subclasses, `extends`, `super`, Method Overriding, Upcasting, Dynamic Binding, and Abstract Classes

Lecturer: Arshyn Altybay

Department of Computer Engineering,
Almaty Technological University, Almaty, Kazakhstan

1. Learning Objectives

After completing this lecture, students should be able to:

1. explain the concept of inheritance in object-oriented programming;
2. identify superclasses and subclasses;
3. create subclasses using the `extends` keyword;
4. explain which class members are inherited;
5. use the `super` keyword to call superclass constructors and methods;
6. explain constructor execution order in an inheritance hierarchy;
7. override inherited methods correctly;
8. distinguish method overriding from method overloading;
9. use superclass references to refer to subclass objects;
10. explain upcasting and dynamic method dispatch;
11. apply runtime polymorphism in practical programs;
12. use `instanceof` safely;
13. perform safe downcasting when it is genuinely necessary;
14. define and use abstract classes and abstract methods;
15. recognize situations in which inheritance should not be used.

2. Introduction to Inheritance

Inheritance is one of the fundamental principles of object-oriented programming.

It allows one class to acquire fields and methods from another class.

The existing class is commonly called the:

- **superclass**,
- **parent class**, or
- **base class**.

The new class is commonly called the:

- **subclass**,
- **child class**, or
- **derived class**.

Inheritance is useful when several classes share common characteristics.

For example:

Illustration

```
Person
  |
  +-- Student
  |
  +-- Teacher
```

Both students and teachers are persons. Therefore common data such as `name` and `age` can be placed in the `Person` superclass.

3. The `extends` Keyword

A Java class inherits from another class using `extends`.

```java
class Person {
    String name;
}

class Student extends Person {
    double gpa;
```

```
}
```

The `Student` class inherits accessible members of `Person`.

An object of `Student` therefore contains both:

- the inherited state from `Person`;
- the state declared directly in `Student`.

Example 1. Basic Inheritance

```
1  public class Main {
2
3      static class Person {
4          String name;
5          int age;
6
7          void displayPerson() {
8              System.out.println(
9                      "Name: " + name
10             );
11             System.out.println(
12                     "Age: " + age
13             );
14         }
15     }
16
17     static class Student extends Person {
18         double gpa;
19
20         void displayStudent() {
21             displayPerson();
22             System.out.println(
23                     "GPA: " + gpa
24             );
25         }
26     }
27
28     public static void main(String[] args) {
29
30         Student student = new Student();
31
32         student.name = "Aruzhan";
33         student.age = 19;
34         student.gpa = 3.8;
35
```

Example 1 (continued)

```
36          student.displayStudent();
37      }
38  }
```

Expected output

```
Name: Aruzhan
Age: 19
GPA: 3.8
```

The subclass can use inherited members when their access level allows it.

4. What Is Inherited?

A subclass inherits accessible members of its superclass.

However, not every member behaves in the same way.

Member	Inheritance behavior
public field	inherited and accessible
protected field	inherited and accessible under protected rules
package-private field	accessible only when package rules allow it
private field	part of object state, but not directly accessible in subclass
public method	inherited
protected method	inherited
private method	not directly inherited as an accessible member
constructor	not inherited
static method	associated with class; can be hidden, not overridden polymorphically

A very important rule is:

Constructors are not inherited.

5. Private Fields and Inheritance

A subclass cannot directly access a private field declared in its superclass.

This is correct encapsulation.

```
class Person {

    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

The subclass should use the superclass public or protected behavior rather than breaking encapsulation.

Example 2. Inheritance with Encapsulation

```
1  public class Main {
2
3      static class Person {
4
5          private String name;
6          private int age;
7
8          public Person(
9                  String name,
10                 int age) {
11
12             this.name = name;
13             this.age = age;
14         }
15
16         public String getName() {
17             return name;
18         }
19
20         public int getAge() {
21             return age;
```

Example 2 (continued)

```
        }
    }

    static class Student extends Person {

        private double gpa;

        public Student(
                String name,
                int age,
                double gpa) {

            super(name, age);
            this.gpa = gpa;
        }

        public void display() {
            System.out.println(
                    getName() + ", " +
                    getAge() + ", GPA = " +
                    gpa
            );
        }
    }

    public static void main(String[] args) {

        Student student =
                new Student(
                        "Dias", 20, 3.5
                );

        student.display();
    }
}
```

Expected output

```
Dias, 20, GPA = 3.5
```

6. The `super` Keyword

The keyword `super` refers to the superclass part of the current object.

It is commonly used to:

1. call a superclass constructor;
2. call an overridden superclass method;
3. access an inherited superclass member when a subclass member hides it.

7. Calling a Superclass Constructor

A subclass constructor may call a superclass constructor using:

```
super(arguments);
```

Important rule:

`super(...)` must be the first statement in the constructor.

If no explicit call is written, Java attempts to insert:

```
super();
```

This works only if the superclass has an accessible no-argument constructor.

Example 3. Constructor Execution Order

```
public class Main {

    static class Person {

        public Person() {
            System.out.println(
                    "Person constructor"
            );
        }
    }

    static class Student extends Person {

        public Student() {
            System.out.println(
                    "Student constructor"
            );
        }
    }

    public static void main(String[] args) {

        new Student();
    }
}
```

Expected output

```
Person constructor
Student constructor
```

The superclass constructor executes before the subclass constructor body.

8. Multi-Level Inheritance

Java allows multi-level inheritance.

Illustration

```
Person
  |
Student
  |
GraduateStudent
```

Example:

```
class Person { }

class Student extends Person { }

class GraduateStudent extends Student { }
```

A `GraduateStudent` is also a `Student` and a `Person`.

9. Method Overriding

A subclass may provide a new implementation of an inherited instance method.

This is called **method overriding**.

Example:

```
class Animal {

    public void sound() {
        System.out.println("Animal sound");
    }
```

```
}

class Dog extends Animal {

    @Override
    public void sound() {
        System.out.println("Woof");
    }
}
```

The annotation `@Override` is strongly recommended because the compiler checks whether a valid override actually exists.

10. Rules for Method Overriding

For a valid override:

- the method name must be the same;
- the parameter list must be the same;
- the return type must be compatible;
- access cannot be made more restrictive;
- the method must be an instance method that can be overridden;
- a `final` method cannot be overridden.

Example 4. Overriding

```
1  public class Main {
2
3      static class Employee {
4
5          public double salary() {
6              return 200000;
7          }
8      }
9
10     static class Manager extends Employee {
11
12         @Override
13         public double salary() {
14             return 350000;
15         }
16     }
```

Example 4 (continued)

```
17
18      public static void main(String[] args) {
19
20          Employee employee =
21                  new Employee();
22
23          Manager manager =
24                  new Manager();
25
26          System.out.println(
27                  employee.salary()
28          );
29
30          System.out.println(
31                  manager.salary()
32          );
33      }
34  }
```

Expected output

```
200000.0
350000.0
```

11. Calling the Overridden Superclass Method

Sometimes a subclass should extend the behavior of the superclass rather than replace it completely.

The subclass can call:

```
super.methodName();
```

Example 5. Using `super.method()`

```
1  public class Main {
2
3      static class Person {
4
5          public void display() {
6              System.out.println(
7                      "Person information"
8              );
9          }
10     }
11
12     static class Student extends Person {
13
14         @Override
15         public void display() {
16
17             super.display();
18
19             System.out.println(
20                     "Student information"
21             );
22         }
23     }
24
25     public static void main(String[] args) {
26
27         Student student =
28                 new Student();
29
30         student.display();
31     }
32 }
```

Expected output

```
Person information
Student information
```

12. Overloading vs Overriding

These two concepts are often confused.

Property	Overloading	Overriding
Location	usually same class	superclass/subclass
Method name	same	same
Parameters	different	same
Binding	compile time	runtime for instance methods
Purpose	several ways to call a method	specialized subclass behavior
Inheritance required?	No	Yes

13. Polymorphism

Polymorphism means "many forms."

In Java, runtime polymorphism commonly occurs when a superclass reference refers to an object of a subclass.

Example:

```
Animal animal = new Dog();
```

The variable type is `Animal`, but the actual object is `Dog`.

If `sound()` is overridden, Java chooses the method according to the actual object at runtime.

14. Upcasting

Assigning a subclass object reference to a superclass variable is called **upcasting**.

Example:

```
Dog dog = new Dog();

Animal animal = dog;
```

Upcasting is safe and normally automatic.

Why?

Because every `Dog` is an `Animal` if `Dog extends Animal`.

Example 6. Runtime Polymorphism

```
1  public class Main {
2
3      static class Animal {
4
5          public void sound() {
6              System.out.println(
7                      "Some animal sound"
8              );
9          }
10     }
11
12     static class Dog extends Animal {
13
14         @Override
15         public void sound() {
16             System.out.println("Woof");
17         }
18     }
19
20     static class Cat extends Animal {
21
22         @Override
23         public void sound() {
24             System.out.println("Meow");
25         }
26     }
27
28     public static void main(String[] args) {
```

Example 6 (continued)

```
29
30          Animal a1 = new Dog();
31          Animal a2 = new Cat();
32
33          a1.sound();
34          a2.sound();
35      }
36  }
```

Expected output

```
Woof
Meow
```

Although both variables have type `Animal`, Java calls the implementation belonging to the actual object.

This behavior is called:

- dynamic binding;
- late binding; or
- dynamic method dispatch.

Example 7. Polymorphic Array

```
1  public class Main {
2
3      static class Shape {
4
5          public double area() {
6              return 0.0;
7          }
8      }
9
10     static class Circle extends Shape {
11
12         private double radius;
13
14         public Circle(double radius) {
15             this.radius = radius;
16         }
17
18         @Override
19         public double area() {
20             return Math.PI *
```

Example 7 (continued)

```
                    radius * radius;
        }
    }

    static class Rectangle
            extends Shape {

        private double width;
        private double height;

        public Rectangle(
                double width,
                double height) {

            this.width = width;
            this.height = height;
        }

        @Override
        public double area() {
            return width * height;
        }
    }

    public static void main(String[] args) {

        Shape[] shapes = {
            new Circle(2.0),
            new Rectangle(3.0, 4.0),
            new Circle(1.0)
        };

        for (Shape shape : shapes) {
            System.out.printf(
                    "%.2f%n",
                    shape.area()
            );
        }
    }
}
```

Expected output

```
12.57
12.00
```

```
3.14
```

The loop does not need to know which exact subclass it is processing.

This is one of the major practical advantages of polymorphism.

15. Static Type vs Runtime Type

Consider:

```
Animal animal = new Dog();
```

There are two relevant types:

- **static type:** `Animal`;
- **runtime type:** `Dog`.

The static type determines which members are accessible through the variable.

The runtime type determines which overridden instance method executes.

Example 8. Static Type Restriction

```
class Animal {

    public void sound() {
        System.out.println("Sound");
    }
}

class Dog extends Animal {

    public void fetch() {
        System.out.println("Fetching");
    }
}
```

Then:

```
Animal animal = new Dog();

animal.sound();

// animal.fetch(); // compile-time error
```

The actual object is a `Dog`, but the variable type is `Animal`. The compiler only allows methods known through the static type.

16. Downcasting

Converting a superclass reference back to a subclass reference is called **downcasting**.

Example:

```
Animal animal = new Dog();

Dog dog = (Dog) animal;
```

Unlike upcasting, downcasting can fail at runtime if the object is not actually an instance of the target subclass.

17. Unsafe Downcasting

Example:

```
Animal animal = new Cat();

Dog dog = (Dog) animal;
```

This compiles, but at runtime it throws:

Illustration

```
ClassCastException
```

Therefore downcasting should be used carefully.

18. The `instanceof` Operator

The operator `instanceof` tests whether an object is compatible with a given type.

Example:

```
if (animal instanceof Dog) {
    Dog dog = (Dog) animal;
    dog.fetch();
}
```

Modern Java also supports pattern matching for `instanceof`.

```
if (animal instanceof Dog dog) {
    dog.fetch();
}
```

This avoids a separate explicit cast.

Example 9. Safe Type-Specific Behavior

```
public class Main {

    static class Animal {

        public void sound() {
            System.out.println("Sound");
        }
    }

    static class Dog extends Animal {

        @Override
        public void sound() {
            System.out.println("Woof");
        }

        public void fetch() {
            System.out.println(
                    "Dog is fetching"
            );
        }
    }

    public static void main(String[] args) {
```

Example 9 (continued)

```
25
26          Animal animal = new Dog();
27
28          animal.sound();
29
30          if (animal instanceof Dog dog) {
31              dog.fetch();
32          }
33      }
34  }
```

Expected output

```
Woof
Dog is fetching
```

19. Avoiding Excessive Type Checking

Although `instanceof` is useful, excessive type checking may indicate poor design.

For example, this is often undesirable:

```
if (shape instanceof Circle) {
    // ...
} else if (shape instanceof Rectangle) {
    // ...
} else if (shape instanceof Triangle) {
    // ...
}
```

A better design may define a polymorphic method such as:

```
shape.area();
```

Then each subclass provides its own implementation.

20. Abstract Classes

Sometimes a superclass represents a general concept that should not be instantiated directly.

For example, “Shape” is a useful concept, but a generic shape may not have one meaningful area formula.

Java supports **abstract classes**.

```
abstract class Shape {
}
```

An abstract class cannot be instantiated directly:

```
// Shape s = new Shape(); // compile-time error
```

21. Abstract Methods

An abstract method has no implementation in the abstract superclass.

Example:

```
abstract class Shape {

    public abstract double area();
}
```

A concrete subclass must implement the abstract method.

Example 10. Abstract Shape Hierarchy

```
1  public class Main {
2
3      static abstract class Shape {
4
5          public abstract double area();
6
7          public void describe() {
8              System.out.println(
9                      "This is a shape."
10             );
11         }
```

Example 10 (continued)

```
12      }
13
14      static class Circle extends Shape {
15
16          private double radius;
17
18          public Circle(double radius) {
19              this.radius = radius;
20          }
21
22          @Override
23          public double area() {
24              return Math.PI *
25                      radius * radius;
26          }
27      }
28
29      static class Rectangle
30              extends Shape {
31
32          private double width;
33          private double height;
34
35          public Rectangle(
36                  double width,
37                  double height) {
38
39              this.width = width;
40              this.height = height;
41          }
42
43          @Override
44          public double area() {
45              return width * height;
46          }
47      }
48
49      public static void main(String[] args) {
50
51          Shape s1 = new Circle(3);
52          Shape s2 =
53                  new Rectangle(4, 5);
54
55          s1.describe();
56
```

Example 10 (continued)

```
57          System.out.printf(
58                  "Circle area = %.2f%n",
59                  s1.area()
60          );
61
62          System.out.printf(
63                  "Rectangle area = %.2f%n",
64                  s2.area()
65          );
66      }
67  }
```

Expected output

```
This is a shape.
Circle area = 28.27
Rectangle area = 20.00
```

An abstract class may contain both:

- abstract methods;
- concrete methods.

22. Abstract Class with Shared State

An abstract class may contain fields and constructors.

Example:

```
abstract class Employee {

    private String name;

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

    public String getName() {
        return name;
    }

    public abstract double salary();
```

```
}
```

This is useful when several subclasses share both data and behavior.

Example 11. Polymorphic Payroll System

```
public class Main {

    static abstract class Employee {

        private String name;

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

        public String getName() {
            return name;
        }

        public abstract double salary();
    }

    static class FullTimeEmployee
            extends Employee {

        private double monthlySalary;

        public FullTimeEmployee(
                String name,
                double monthlySalary) {

            super(name);
            this.monthlySalary =
                    monthlySalary;
        }

        @Override
        public double salary() {
            return monthlySalary;
        }
    }

    static class HourlyEmployee
            extends Employee {

```

Example 11 (continued)

```
41          private double rate;
42          private int hours;
43
44          public HourlyEmployee(
45                  String name,
46                  double rate,
47                  int hours) {
48
49              super(name);
50              this.rate = rate;
51              this.hours = hours;
52          }
53
54          @Override
55          public double salary() {
56              return rate * hours;
57          }
58      }
59
60      public static void main(String[] args) {
61
62          Employee[] employees = {
63              new FullTimeEmployee(
64                      "Aruzhan", 400000),
65              new HourlyEmployee(
66                      "Dias", 5000, 80)
67          };
68
69          for (Employee employee :
70                  employees) {
71
72              System.out.println(
73                      employee.getName() +
74                      ": " +
75                      employee.salary()
76              );
77          }
78      }
79  }
```

Expected output

```
Aruzhan: 400000.0
Dias: 400000.0
```

The client code processes all employees uniformly through the superclass type.

23. The `Object` Class

Every Java class ultimately inherits from `java.lang.Object`.

If a class does not explicitly extend another class:

```
class Student {
}
```

it implicitly extends `Object`.

Important methods inherited from `Object` include:

- `toString()`
- `equals(Object obj)`
- `hashCode()`
- `getClass()`

These methods are important throughout Java programming.

Example 12. Overriding `toString()`

```
1  public class Main {
2
3      static class Student {
4
5          private String name;
6          private double gpa;
7
8          public Student(
9                  String name,
10                 double gpa) {
11
12             this.name = name;
13             this.gpa = gpa;
14         }
15
16         @Override
17         public String toString() {
18             return "Student{name='" +
19                     name +
20                     "', gpa=" +
```

Example 12 (continued)

```
                    gpa + "}";
        }
    }

    public static void main(String[] args) {

        Student student =
                new Student(
                        "Ali", 3.7
                );

        System.out.println(student);
    }
}
```

Expected output

```
Student{name='Ali', gpa=3.7}
```

24. Final Classes and Final Methods

A class declared `final` cannot be subclassed.

```
final class Utility {
}
```

A final method cannot be overridden.

```
class Parent {

    public final void display() {
        System.out.println("Fixed");
    }
}
```

These features are useful when a developer wants to prevent extension or behavior replacement.

25. Inheritance and Access Modifiers

The interaction between inheritance and access modifiers is important.

Modifier	Practical inheritance meaning
private	subclass cannot directly access member
package-private	accessible only when package rules allow it
protected	designed partly for subclass access
public	accessible wherever the class is accessible

In well-designed programs, private fields are often preferred even in superclasses, with protected or public methods providing controlled behavior.

26. The "Is-A" Relationship

Inheritance should normally represent a genuine **is-a** relationship.

Examples:

- A student **is a** person.
- A manager **is an** employee.
- A circle **is a** shape.

But:

- A car **has an** engine.
- A university **has** students.
- A computer **has a** processor.

These are **has-a** relationships and are normally better represented using composition rather than inheritance.

27. Inheritance vs Composition

Inheritance:

Illustration

```
Student is a Person
```

Composition:

Illustration

```
Car has an Engine
```

Example:

```
class Engine {
}

class Car {

    private Engine engine;

    public Car() {
        engine = new Engine();
    }
}
```

Do not use inheritance merely to reuse a few methods. The relationship should make conceptual sense.

28. Common Mistakes

28.1 Mistake 1: Trying to Access Private Superclass Fields

Incorrect:

```
class Person {
    private String name;
}

class Student extends Person {
```

```
    void show() {
        // System.out.println(name);
    }
}
```

Use an appropriate superclass method such as `getName()`.

28.2 Mistake 2: Forgetting `super(...)`

If the superclass has no no-argument constructor, the subclass must explicitly call an appropriate superclass constructor.

28.3 Mistake 3: Confusing Overloading and Overriding

Different parameters imply overloading. The same signature in a subclass normally implies overriding.

28.4 Mistake 4: Unsafe Downcasting

Never assume a superclass reference contains a particular subclass object.

28.5 Mistake 5: Excessive `instanceof`

If every operation uses long `instanceof` chains, polymorphism may not be used effectively.

28.6 Mistake 6: Inheritance for Code Reuse Only

Inheritance should model an is-a relationship, not just provide access to some convenient code.

Example 13. Transportation Fare System

```
1 public class Main {
2
3     static abstract class Transport {
4
5         private String name;
6
7         public Transport(String name) {
8             this.name = name;
```

Example 13 (continued)

```
9           }
10
11          public String getName() {
12              return name;
13          }
14
15          public abstract double fare(
16                  double distance);
17      }
18
19      static class Bus extends Transport {
20
21          public Bus() {
22              super("Bus");
23          }
24
25          @Override
26          public double fare(
27                  double distance) {
28
29              return 80 +
30                      15 * distance;
31          }
32      }
33
34      static class Taxi extends Transport {
35
36          public Taxi() {
37              super("Taxi");
38          }
39
40          @Override
41          public double fare(
42                  double distance) {
43
44              return 500 +
45                      120 * distance;
46          }
47      }
48
49      static class Train extends Transport {
50
51          public Train() {
52              super("Train");
53          }
```

Example 13 (continued)

```
54
55          @Override
56          public double fare(
57                  double distance) {
58
59              return 50 +
60                      10 * distance;
61          }
62      }
63
64      public static void main(String[] args) {
65
66          Transport[] transports = {
67              new Bus(),
68              new Taxi(),
69              new Train()
70          };
71
72          double distance = 10;
73
74          for (Transport transport :
75                  transports) {
76
77              System.out.println(
78                      transport.getName() +
79                      ": " +
80                      transport.fare(
81                              distance)
82              );
83          }
84      }
85  }
```

Expected output

```
Bus: 230.0
Taxi: 1700.0
Train: 150.0
```

The main method depends only on the abstraction `Transport`. Each subclass supplies its own fare calculation.

Example 14. University Member Hierarchy

```
public class Main {

    static abstract class
            UniversityMember {

        private String name;

        public UniversityMember(
                String name) {

            this.name = name;
        }

        public String getName() {
            return name;
        }

        public abstract String role();
    }

    static class Student
            extends UniversityMember {

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

        @Override
        public String role() {
            return "Student";
        }
    }

    static class Teacher
            extends UniversityMember {

        public Teacher(String name) {
            super(name);
        }

        @Override
        public String role() {
            return "Teacher";
        }
```

Example 14 (continued)

```
45      }
46
47      public static void main(String[] args) {
48
49          UniversityMember[] members = {
50              new Student("Aruzhan"),
51              new Teacher("Dr. Aliyev")
52          };
53
54          for (UniversityMember member :
55                  members) {
56
57              System.out.println(
58                      member.getName() +
59                      " - " +
60                      member.role()
61              );
62          }
63      }
64  }
```

Expected output

```
Aruzhan - Student
Dr. Aliyev - Teacher
```

29. Design Guidelines

When using inheritance:

1. use inheritance only when the subclass truly is a specialized form of the superclass;
2. keep superclass fields encapsulated;
3. prefer meaningful protected/public behavior over exposing state;
4. use `@Override`;
5. make superclass constructors initialize superclass state;
6. avoid deep inheritance hierarchies without a strong reason;
7. use abstract classes when a shared concept should not be instantiated directly;
8. prefer polymorphic methods over repeated type checks;
9. avoid unnecessary downcasting;
10. consider composition when the relationship is “has-a” rather than “is-a”.

30. Summary

In this lecture, we learned that:

- inheritance allows a subclass to specialize a superclass;
- Java uses `extends` for class inheritance;
- constructors are not inherited;
- `super(...)` calls a superclass constructor;
- superclass construction occurs before subclass construction;
- method overriding provides specialized behavior;
- `@Override` helps the compiler verify overrides;
- upcasting allows a superclass reference to refer to a subclass object;
- runtime polymorphism chooses overridden methods according to the actual object;
- the static type controls what members are visible through a variable;
- downcasting should be used carefully;
- `instanceof` can be used for safe type checks;
- abstract classes represent incomplete general concepts;
- abstract methods must be implemented by concrete subclasses;
- every Java class ultimately derives from `Object`;
- inheritance should model an is-a relationship;
- composition is often better for has-a relationships.

31. Review Questions

1. What is inheritance?
2. What is a superclass?
3. What is a subclass?
4. Which keyword is used to create a subclass?
5. Are constructors inherited?
6. Can a subclass directly access private fields of its superclass?
7. What is the purpose of the `super` keyword?
8. What rule applies to `super(...)` inside a constructor?
9. In what order are superclass and subclass constructors executed?
10. What is method overriding?
11. Why should `@Override` be used?
12. What is the difference between overriding and overloading?
13. What is polymorphism?
14. What is upcasting?
15. Why is upcasting normally safe?
16. What is dynamic binding?
17. What is the difference between static type and runtime type?
18. What is downcasting?
19. Why can downcasting fail?
20. What exception may occur after an invalid cast?
21. What does `instanceof` do?
22. What is an abstract class?
23. Can an abstract class be instantiated?
24. What is an abstract method?
25. Can an abstract class contain concrete methods?
26. Can an abstract class contain fields and constructors?
27. What is the `Object` class?
28. What does a final class prevent?
29. What does a final method prevent?
30. What is an is-a relationship?
31. What is a has-a relationship?
32. When should composition be preferred over inheritance?

32. Programming Exercises

Beginner Level

1. Create a `Person` superclass and a `Student` subclass.
2. Add private fields to `Person` and initialize them through a constructor.
3. Use `super(...)` from the `Student` constructor.
4. Create an `Animal` superclass and override `sound()` in `Dog`.
5. Create a `Vehicle` superclass and `Car` subclass.
6. Demonstrate constructor execution order.
7. Override `toString()` in a `Book` class.
8. Create a superclass reference that refers to a subclass object.
9. Create an array of superclass references containing several subclass objects.
10. Create an abstract `Shape` class with one abstract method.

Intermediate Level

1. Build the hierarchy `Person -> Student -> GraduateStudent`.
2. Build `Employee -> Manager` and `Employee -> Developer`.
3. Override a salary calculation in several employee subclasses.
4. Create a polymorphic array of employees and calculate total salary.
5. Create abstract `Shape` with `area()` and `perimeter()` methods.
6. Implement `Circle`, `Rectangle`, and `Triangle`.
7. Create `Account`, `SavingsAccount`, and `CurrentAccount`.
8. Demonstrate safe downcasting with `instanceof`.
9. Refactor a type-checking program to use polymorphism instead.
10. Create a transportation system with polymorphic fare calculations.

Advanced Level

1. Develop a complete payroll system using an abstract employee superclass.
2. Develop a geometry application that computes total area for mixed shape objects.
3. Design a university-member hierarchy containing students, teachers, and administrators.
4. Create a banking hierarchy with different interest and fee rules.
5. Create a product hierarchy with different discount policies.
6. Build a hotel-room hierarchy with different price calculations.
7. Develop a game-character hierarchy with polymorphic attack behavior.
8. Create a media hierarchy containing books, audio books, and videos.
9. Refactor duplicated code from several classes into a meaningful superclass.
10. Build an application in which adding a new subclass requires no change to the main processing loop.

33. Homework

Task: Build a Polymorphic University Payroll Application

Create the following hierarchy:

Illustration



Requirements:

1. `Employee` must be abstract.
2. It must contain:
   - private employee ID;
   - private name;
   - constructor;
   - getters;
   - abstract method `calculateSalary()`.
3. `FullTimeEmployee` must calculate salary from a fixed monthly salary.
4. `PartTimeEmployee` must calculate salary from hourly rate and hours worked.

5. `ContractEmployee` must calculate salary from a fixed contract amount.
6. Create an array or list-like array structure of type `Employee[]`.
7. Store at least six mixed employee objects in the array.
8. Display each employee's name and calculated salary using polymorphism.
9. Calculate total payroll without using an `instanceof` chain.
10. Override `toString()` meaningfully.
11. Add one subclass of your own design.

Submission requirements:

- all Java source files;
- program output;
- a short class-hierarchy diagram;
- explanation of where method overriding is used;
- explanation of where upcasting occurs;
- explanation of why polymorphism makes the program easier to extend.

34. Key Terms

Inheritance
: Mechanism by which a subclass acquires accessible members of a superclass.

Superclass
: The more general parent class.

Subclass
: A specialized class that extends another class.

`extends`
: Java keyword used for class inheritance.

`super`
: Reference to the superclass part of the current object.

Method overriding
: Providing a subclass implementation of an inherited method.

Polymorphism
: Ability to treat different subclass objects through a common superclass type.

Upcasting
: Converting a subclass reference to a superclass reference.

Downcasting
: Converting a superclass reference to a subclass reference.

Dynamic binding
: Selecting an overridden instance method according to the runtime object type.

`instanceof`
: Operator used to test whether an object is compatible with a type.

Abstract class
: A class that cannot be instantiated directly and may define abstract methods.

Abstract method
: A method declared without an implementation.

Is-a relationship
: Relationship appropriately modeled by inheritance.

Has-a relationship
: Relationship usually modeled by composition.