Object Oriented and Java Programming

Course 4

Qiong Liu

qiong.liu@cyu.fr
CY Tech, CY Cergy Paris University

October 2024

Contents

Outline

(1-1) Inheritance and Polymorphism: Key Concepts in Object-Oriented Programming

Introduction:

- Inheritance and polymorphism are essential for building flexible and maintainable program architectures.
- Inheritance:
  - Enables reuse of pre-defined classes, reducing redundant code.
- Polymorphism:
  - Allows dynamic adjustment of object behavior, reducing dependency between objects.

(1-1) Inheritance Basics

Definition:

- Inheritance allows a class (child) to acquire the properties and methods of another class (parent).
- Establishes a parent-child relationship.

(1-1) Inheritance Basics

Definition:

- Inheritance allows a class (child) to acquire the properties and methods of another class (parent).
- Establishes a parent-child relationship.

Syntax:

- `class ChildClass extends ParentClass { }`

(1-1) Inheritance Basics

Example:

```
class Animal {
    void eat() { System.out.println("This animal eats
        food."); }
}
class Dog extends Animal {
    void bark() { System.out.println("The dog barks.")
        ; }
}
public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.eat(); // Inherited from Animal
        myDog.bark(); // Defined in Dog
    }
}
```

(1-1) Inheritance Basics

Pay attention:

- Java supports only single inheritance, meaning each child class can have only one parent class.
- The following example is invalid in Java:
  - `class ChildClass extends ParentClass1, ParentClass2 { }`

(1-2) `java.lang.Object`

In Java:

- All classes implicitly inherit from the `java.lang.Object` class.
- `Object` is the root of the class hierarchy.
- Every class, has `Object` as a parent. (we do not need to state it)

(1-2) java.lang.Object

In Java:

- All classes implicitly inherit from the `java.lang.Object` class.
- `Object` is the root of the class hierarchy.
- Every class, has `Object` as a parent. (we do not need to state it)

Example:

```
class MyClass {
    // Automatically extends java.lang.Object
}
public class Main {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        System.out.println(obj.toString()); // Method
            from Object
    }
}
```

(1-2) Methods of `java.lang.Object`

Commonly Used Methods:

- `toString()` – Returns a string representation of the object.
- `equals(Object obj)` – Checks whether two objects are equal.
- `hashCode()` – Returns a hash code for the object.
- `clone()` – Creates a copy of the object (if the class implements `Cloneable`).
- `finalize()` – Called before the object is garbage collected.
- `getClass()` – Returns the runtime class of the object.
- `notify()`, `notifyAll()`, and `wait()` – Used in multi-threaded.

The `toString()` Method

Description:

- Defined in `java.lang.Object`.
- Returns a string representation of the object.
- Default implementation returns: `<ClassName>@<hashcode>`

The toString() Method

Description:

- Defined in `java.lang.Object`.
- Returns a string representation of the object.
- Default implementation returns: `<ClassName>@<hashcode>`

Example:

```
class MyClass {}

public class Main {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        System.out.println(obj.toString()); // Output:
            MyClass@1b6d3586
    }
}
```

(1-2) The toString() Method

Customizing toString():

```
class MyClass {
    @Override
    public String toString() {
        return "This is MyClass";
    }
}
```

(1-2) The `equals()` Method

In Java, there are two ways to compare objects : `==` vs `equals`

- `==`: Compares reference equality.
  - Checks if two references point to the same memory location.
- `equals`: Compares logical equality.
  - Checks if two objects are logically equivalent based on the class's definition of equality.
  - Default implementation in `java.lang.Object` behaves like `==`, but it can be overridden.

(1-2) The equals() Method

Example: Compare Users Based on Identifier

```
class User {
    String name,identifier;
    User(String name, String identifier) {
        this.name = name; this.identifier=identifier;
    }
    @Override
    public boolean equals(Object obj) {
        if (obj instanceof User) {
            User other = (User) obj;
            return this.identifier.equals(other.
                identifier);
        }
        return false;
    }}
User u1 = new User("Alice", "123");
User u2 = new User("Bob", "123");
System.out.println(u1.equals(u2)); //true
```

(1-3) Object Type Casting: Upcasting and Downcasting

What is Object Type Casting?

- Changing an object's reference type within the inheritance hierarchy.
- Two types:
  - Upcasting: subclass to superclass
  - Downcasting:superclass to subclass

(1-3) Upcasting

Definition:

- Converting a subclass object into a superclass reference. Always **safe**.

Key Points:

- The parent class reference can only access fields and methods defined in the parent class.
- Subclass-specific fields and methods are **not accessible**.

(1-3) Upcasting

Definition:

- Converting a subclass object into a superclass reference. Always **safe**.

Key Points:

- The parent class reference can only access fields and methods defined in the parent class.
- Subclass-specific fields and methods are **not accessible**.

```
class Animal {
    public void eat() { System.out.println("Animal eats."); }
}
class Dog extends Animal {
    public void bark() { System.out.println("Dog barks."); }
}
public class Main {
     public static void main(String[] args) {
          Animal animal = new Dog(); // Upcasting
          animal.eat();  // Allowed (defined in Animal)
          // animal.bark(); // Error! Not accessible
     }}
```

(1-3) Downcasting

Definition:

- Converting a superclass reference back into a subclass reference.
- This requires an **explicit cast** and is **not always safe**.

Key Points:

- Downcasting is necessary to access subclass-specific fields or methods.
- If the object is not actually an instance of the subclass, a `ClassCastException` will occur.

(1-3) Downcasting

Example:

```
class Animal {
    void eat() { System.out.println("Animal eats."); }
}
class Dog extends Animal {
    void bark() { System.out.println("Dog barks."); }
}
public class Main {
    public static void main(String[] args) {
        Animal animal = new Dog(); // Upcasting
        Dog dog = (Dog) animal;    // Downcasting
        dog.bark();  // Allowed
    }}
```

Important:

```
// Unsafe Downcasting
Animal animal = new Animal();
Dog dog = (Dog) animal; // Throws ClassCastException!
```

(1-4) `instanceof` keyword

What is `instanceof`?

- A keyword in Java used for type checking.
- Checks if an object is:
  - An instance of a specific class.
  - An instance of a subclass.
  - An implementation of an interface.
  - Always returns `false` for `null`.

Syntax:

```
object instanceof ClassOrInterface
```

(1-4) instanceof keyword

Example

```
class Animal {}
class Dog extends Animal {}
class Cat extends Animal {}

public class TestInstanceof {
    public static void main(String[] args) {
        Animal myAnimal = new Dog();

        if (myAnimal instanceof Dog) {
            System.out.println("This is a Dog.");
        } else if (myAnimal instanceof Cat) {
            System.out.println("This is a Cat.");
        } else {
            System.out.println("Unknown type.");
        }
    }
}
```

(1-5) Method Overloading

Method Overloading

Method overloading allows a class to define more than one method with the **same name** as long as their parameter lists differ.

- In java, a class can only have one constructor method, which is determined by the class name.
- If we want to instantiate objects in different ways, we need multiple constructors.
- Achieved by having:
  - Different number of parameters.
  - Different types of parameters.

(1-5) Method Overloading

Example:

```
public class ArrayMuni {
    private int[] arrayList;

    //Constructor
    public ArrayMuni(int n) { arrayList = new int[n];}
    public ArrayMuni() {  arrayList = new int[0];}
    public ArrayMuni(int[] initialArray) {
    arrayList = initialArray.clone();}

    public static void main(String[] args) {
        // Example usage
        ArrayMuni arr = new ArrayMuni(5);
        ArrayMuni arr1 = new ArrayMuni(new int[]{-3,
            5, -3, 6, -2, 4, 11, -5, 4});
     }
}
```

(1-6) Polymorphism

Polymorphism

Polymorphism allows a single interface to represent different types of objects; it enables a single method to perform different tasks depending on the object that invokes it.

Types of Polymorphism:

1. Compile-time Polymorphism (Method Overloading):
   - Same method name with different parameter lists.[see (1-5)]
2. Runtime Polymorphism (Method Overriding):
   - A subclass provides a specific implementation of a method already defined in its superclass.

(1-6) Polymorphism

Method Overriding:

```
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}
class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}
public class TestPolymorphism {
    public static void main(String[] args) {
        Animal myAnimal = new Dog();
        myAnimal.sound();  // Output: Dog barks
    }
}
```

(1-7) Abstract Class

Abstract Class

An abstract class is used as a base class for other classes; we declare it with the `abstract` keyword.

Features:

- Can have both abstract methods (no body) and concrete methods (with body).
- Subclasses must implement abstract methods or be declared abstract.

Example:

```
abstract class Animal {
    abstract void sound(); // Abstract method
    void eat() {           // Concrete method
        System.out.println("This animal eats food.");
    }
}
class Dog extends Animal {}
```

Outline

What is an Interface?

- An interface is like a **contract** in Java.
- It defines methods that a class must have, but it doesn't say *how* to implement them.
- Classes that "sign" the contract by **implementing** the interface must provide the method details.
- **Think of it as a plan or a set of rules.**

Example:

```
public interface Animal {
    void makeSound();
}
class Dog implements Animal {
    @Override
    public void makeSound() {
        System.out.println("Bark");
    }
}
```

Why Use an Interface?

Why Use an Interface?

- Interfaces help define a common structure for different classes.
- They allow a class to follow multiple rules or contracts.
- They make your code more flexible, organized, and easier to maintain.

Syntax Example:

```
public interface Paintable {
    void draw();
}
```

- Public: The interface is always public; no other modifier is allowed.
- Interface: A keyword used to define an interface.
- Paintable: The name of the interface.

Implement multi interfaces

A class can inherit and implement at the same time:

```
public class parallelogram extends Circle implements
    Paintable {
     ...
}
```

Important: In Java, class can not inherit multiple classes, but can implement multiple interfaces.

```
Class className implements interface1, interface2,...
    interfacenn{

}
```

Implement multi interfaces

Define interfaces:

```
interface Animal {
void eat();}
interface Pet {
    void play();}
```

Implementing Multiple Interfaces:

```
class Dog implements Animal, Pet {
    @Override
    public void eat() {
        System.out.println("Dog is eating");
    }
    @Override
    public void play() {
        System.out.println("Dog is playing");
    }

    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.eat(); myDog.play();
    }}
```

Interface Inheritance in Java

What is Interface Inheritance?

- Just like classes, interfaces can also inherit other interfaces using the `extends` keyword.
- An interface can extend one or more interfaces.
- The extending interface inherits all the abstract methods of its parent interfaces.

Example:

```
interface Animal {
    void eat();
}
interface Pet extends Animal {
    void play();
}
```

Multiple Inheritance in Interfaces

What is Multiple Inheritance in Interfaces?

- Interfaces in Java can inherit multiple parent interfaces using the `extends` keyword.
- This is a unique feature since Java prohibits multiple inheritance for classes.
- A child interface inherits all methods from its parent interfaces.

Syntax:

```
interface Parent1 {
    void method1();
}
interface Parent2 {
    void method2();
}

interface Child extends Parent1, Parent2 {
    void method3();
}
```

Exercise 1: Inheritance, Polymorphism, and Interfaces

Exercise 1:

- Create a Flyable interface with an abstract method fly() to represent flying.
- Create an abstract class Insect:
  - It has an int variable legs to represent the number of legs.
  - It has a constructor that accepts a parameter to initialize legs.
  - It declares an abstract method reproduce().
- Create a class Butterfly:
  - It extends Insect and implementsFlyable.
  - Implements both fly() and reproduce() methods.
- Finally, create a Test Class to output:
  - "Butterfly has 4 legs."
  - "Butterfly can fly in the air."
  - "Butterfly reproduces by laying eggs."

Exercise 2: Triangle Verification and Abstract Class

Problem: try to find if it is a triangle?

- Create an abstract class **Shape**:
  - Declare an abstract method `calculatePerimeter()` to compute the perimeter.
- Create a `Triangle` class:
  - It `extends` `Shape`.
  - Declare three sides: a,b, c.
  - Add a method to check if the sides can form a triangle:
    - The sum of any two sides must be greater than the third side.
  - Override the `calculatePerimeter()` method to compute the perimeter.
- Create a `Test Class` to output:
  - For sides 3, 4, 5: "Sides 3, 4, 5 can form a triangle. The perimeter is 12.0."
  - For sides 1, 4, 5: "Sides 1, 4, 5 cannot form a triangle because the sum of any two sides must be greater than the third side."

Outline

Packages

Package(use lowercase!)

A package is a namespace for organizing Java classes and interfaces.

- Packages help avoid **name conflicts** by grouping related classes together.
- They allow developers to manage large projects by structuring the code logically.

Example of Name Conflict Without Packages:

- Two classes named Login for different functionalities:
  - One for user authentication.
  - Another for admin authentication.
- With packages:
  - `com.user.Login` for user login.
  - `com.admin.Login` for admin login.

Using Package

Importing a Package:

Use the import keyword to avoid typing the full path every time.

```
// Fully Qualified Name
com.user.Login userLogin = new com.user.Login();
// Using Import
import com.user.Login;
Login userLogin = new Login();
```

Static Import:

- Use import static to import **static members** of a class.
- Simplifies access to constants or utility methods.

```
import static java.lang.Math.PI;
import static java.lang.Math.sqrt;
// Now you can directly use:
System.out.println(PI);
System.out.println(sqrt(16));
```

Inner Class

Inner classes

Inner classes are classes defined inside another class.

Two types of inner classes:

- Member Inner Class: A class defined as a non-static member of another class.

```
1 class OuterClass{
2     class InnerClass{
3
4     }
5 }
```

- Anonymous Inner Class: A class defined without a name for instant use.

```
1 new FatherClass/FatherInterface(){
2 ...
3 };
```

Member Inner Classes

What Is a Member Inner Class?

- A member inner class is a non-static class defined inside another class.
- It behaves like a member of the outer class.
- It can access private members of the outer class.

```
class Outer {
    private String message = "Hello from Outer!";

    // Member Inner Class
    class Inner {
        void display() {
            System.out.println(message); // Access outer class
                private member
        }
    }
}
public class Main {
    public static void main(String[] args) {
        Outer outer = new Outer();
        Outer.Inner inner = outer.new Inner();
        inner.display(); // Output: Hello from Outer!
    }
```

Exercise 3: Member Inner Class

Problem Description:

- Create a class named Car:
  - A private field brand to store the car's brand name.
  - A constructor to initialize the brand.
  - A method start() that prints "[brand] is starting...".
- Create a member inner class named Engine inside the Car class:
  - It has private fields model (for the engine model) and type (for engine type, such as "electric" or "essence").
  - Add a constructor to initialize model and type.
  - Add a method displayEngineDetails() to print the engine model and type.
- In the main method:
  - Create a Car object with brand "Tesla".
  - Call the start() method.
  - Create an Engine object with model "Model 3" and type "electric".
  - Call displayEngineDetails() to show engine details.

Expected Output:

Using `this` Keyword for Inner and Outer Class References

Problem:

- Sometimes, an inner class and its outer class can have variables with the same name.
- Use the `this` keyword to:
  - Refer to the inner class's variable with `this.x`.
  - Refer to the outer class's variable with `OuterClassName.this.x`.

Example:

```
class Outer {
    int x = 10; // Outer class variable
    class Inner {
        int x = 20; // Inner class variable

        void display() {
            System.out.println("Inner x: " + this.i);
            System.out.println("Outer x: " + Outer.this.i);
        }
    }
}
```

Anonymous Inner Class

Anonymous Inner Class

An anonymous inner class is a special type of inner class that does not have a name and is instantiated and declared all at once.

Characteristics:

- Typically used for event handling or callback mechanisms.

Syntax:

```
InterfaceName obj = new InterfaceName() {

    @Override
    public void methodName() {
        // Implementation
    }
};
ClassName obj = new ClassName() {
    @Override
    public void methodName() {
```

Anonymous Inner Class

Example

```
abstract class Dog{
String Color;
public abstract void move();
}
public class Demo{
public static void main(String args[]){
    Dog dog1 = new Dog(){
    @Override
    public void move(){
        System.out.println("the dog is run");
    }
    };
    dog1.Color = "black";
    dog1.move();
    }
}
```

Outline

Exception handling

In programming, exceptions may be caused by several issues:

- User input bad data
- User want to open a non-existent file or document.
- Dividing by zero or other illegal operations, interrupting the normal flow of instructions.

For example: can 0 be a divisor?

```
public class Baulk{
    public static void main(String[] args){
    int result = 3 / 0;
    System.out.pringln(result);
    }
}
```

Output:

- `Exception in thread "main"`
  `java.lang.ArithmeticException: / by zero`

Exception Handling mechanism in Java

Exception Handling mechanism:

- In Java, an exception is an instance of a class representing an error condition.
- When a method encounters an error, it creates an **exception object** and passes it to the runtime system.
- This mechanism separates error-handling code from the main logic, improving code clarity and maintainability.

Exception Handling with `try-catch`

How to Handle Exceptions in Java:

- Use a `try` block to enclose code that might throw an exception.
- Use a `catch` block to handle specific exceptions and prevent program crashes.
- Use a `finally` block to execute cleanup code, whether or not an exception occurs.

Exception Handling with `try-catch`

How to Handle Exceptions in Java:

- Use a `try` block to enclose code that might throw an exception.
- Use a `catch` block to handle specific exceptions and prevent program crashes.
- Use a `finally` block to execute cleanup code, whether or not an exception occurs.

Example: Handling Array Overflow

```
public class ExceptionHandling {
    public static void main(String[] args) {
        try {
            int[] numbers = {1, 2, 3};
            System.out.println(numbers[5]); // Invalid index
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Error: " + e.getMessage());
        } finally {
            System.out.println("Execution completed.");
        }
    }
}
```

Common Exception Classes in Java

- All exceptions inherit from `java.lang.Throwable`.
- Two main categories:
  - `Error` – Serious issues (e.g., `OutOfMemoryError`).
  - `Exception` – Issues that can be handled in the program.

Common Exception Classes in Java

- All exceptions inherit from `java.lang.Throwable`.
- Two main categories:
  - `Error` – Serious issues (e.g., `OutOfMemoryError`).
  - `Exception` – Issues that can be handled in the program.

Checked Exceptions:

- Must be handled (via `try-catch` or `throws`).
  - `IOException`: Issues with file handling or I/O operations.
  - `SQLException`: Errors interacting with a database.
  - `ClassNotFoundException`: Class not found during runtime.

Unchecked Exceptions:

- Runtime exceptions; do not require explicit handling.
  - `ArithmeticException`: E.g., dividing by zero.
  - `NullPointerException`: Accessing a null object.
  - `ArrayIndexOutOfBoundsException`: Accessing invalid array indices.
  - `IllegalArgumentException`: Invalid method arguments.

Common Exception Classes in Java

Errors:

- Cannot be handled in the application.
  - `OutOfMemoryError`: JVM runs out of memory.
  - `StackOverflowError`: Infinite recursion causes stack overflow.

you can find more cases at: `https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html`.

Custom Exceptions

You can also create your own Custom Exception.

Example: Custom Exception for Age Validation

```
class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message); // Pass message to the Exception class
    }
}
public class CustomExceptionExample {
    public static void checkAge(int age) throws InvalidAgeException
         {
        if (age < 18) {
            throw new InvalidAgeException("Age must be above 18.");
        }
        System.out.println("Valid age: " + age);
    }
    public static void main(String[] args) {
        try {
            checkAge(16);
        } catch (InvalidAgeException e) {
            System.out.println("Exception caught: " + e.getMessage
                ());
        }
```

Throwing Exceptions in Methods

Why Throw Exceptions?

- Sometimes a method may encounter an exception but cannot or should not handle it immediately.
- `throws`: Declares that the method may throw a specific exception.
- `throw`: Used inside the method to actually throw an exception.

Throwing Exceptions in Methods

Example: Method Throwing an Exception

```
class ExceptionExample {
    public static void divide(int a, int b) throws
        ArithmeticException {
        if (b == 0) {
            throw new ArithmeticException("Division by zero is not
                allowed.");
        }
        System.out.println("Result: " + (a / b));
    }
    public static void main(String[] args) {
        try {
            divide(10, 0);
        } catch (ArithmeticException e) { // Catches the exception
            System.out.println("Exception caught: " + e.getMessage
                ());
        }
    }}
```

Exercise 4: Exception Handling

1. Implement a simple integer calculator that supports addition, subtraction, multiplication, and division between two integers. Use `try ... catch` to handle the `InputMismatchException` if the user inputs invalid data.
2. Custom Exception: A supermarket limits the purchase of low-cost items. For example, eggs cost €1 per kg, with a purchase limit of 3 kg per person. If a user attempts to exceed this limit, throw a custom exception. Otherwise, calculate the total cost of the purchase.
3. Write a program for user information input:
   - Prompt the user to enter their name and age.
   - If the entered age is invalid (e.g., `0.5`), throw an exception and prompt the user to re-enter a valid age.
   - Once valid input is received, display the user's information.