Object Oriented System Design

Omid Abbaszadeh

University of Zanjan
o.abbaszadeh@znu.ac.ir

Syllabus

- Introduction (1 session)
- Object oriented computing concepts (2 sessions)
  - Abstract data types, classes, methods
  - Message passing
  - Inheritance, Polymorphism, Dynamic binding
- A review of UML (1 session)
- Software Development Methodologies (2 sessions)
- Principle of object-oriented design (3 sessions)
  - Basic principles
  - GRASP patterns
  - Design by Contract
- GoF patterns and MVC pattern (5 sessions)
  - Creational patterns
  - Structural patterns
  - Behavioral patterns
- Software testing (4 sessions)
  - Acceptance testing, Integration testing
  - Unit testing, Functional testing
  - Performance testing, Stress testing, Usability testing
- Refactoring and Code Reviews (3 sessions)

Outline

- Class
  - Variables (Fields, Variables, and Parameters)
  - Methods - Overloading
  - Access Modifiers
  - Constructors
  - Nested class
  - Class related Keywords: this, final, equals, hashcode, toString, ...
- Interface and Abstract class
  - Default methods (Java 8)
  - Functional interface (Java 8)
- Object
  - References, Identity, and State
- Inheritance and Polymorphism
  - super
  - Overriding
- Typecasting (Upcasting and Downcasting)
- Serialization and Externalization
- ORM (Object Relational Mapping)

Class

Definition (Class)

A class is a **blueprint** or **prototype** from which *objects* are created.

```java
public class Bicycle {

    public int cadence;
    public int gear;
    public int speed;

    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }

    public void setCadence(int newValue) { cadence = newValue; }

    public void setGear(int newValue) { gear = newValue; }

    public void applyBrake(int decrement) { speed -= decrement; }

    public void speedUp(int increment) { speed += increment; }

}
```

Variables

Definition (Variables)

There are several kinds of variables:

- *Member variables* in a class - these are called fields.
- *Variables in a method or block of code* - these are called local variables.
- *Variables in method declarations* - these are called parameters.

Access modifiers for fields and methods:

Access Levels

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

Variables

In the spirit of encapsulation, it is common to make fields private.

```
public class Bicycle {

    private int cadence;
    private int gear;
    private int speed;

    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }

    public int getCadence() { return cadence; }

    public void setCadence(int newValue) { cadence = newValue; }

    public int getGear() { return gear; }

    public void setGear(int newValue) { gear = newValue; }

    public int getSpeed() { return speed; }

    public void applyBrake(int decrement) { speed -= decrement; }
```

Variables

All variables must have a type.

You can use **primitive types** such as:

- int
- float
- boolean
- etc.

Or you can use **reference types**, such as:

- Integer
- Boolean
- Float
- strings
- arrays
- etc.

Variables

Some reasons why we use **primitive data types** in Java:

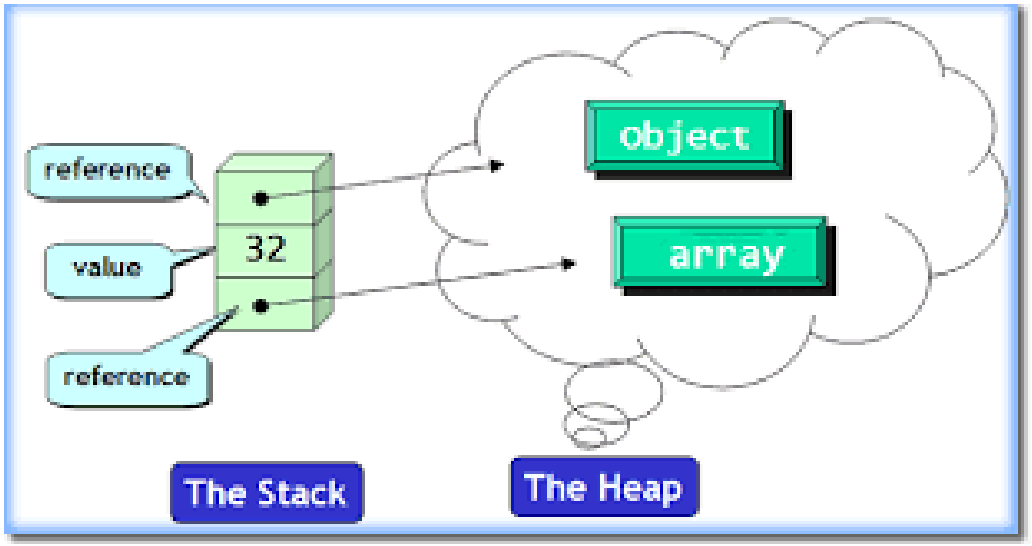
1. Efficiency:
   - Primitive data types are more efficient than objects because they are stored directly in memory, without the overhead of creating an object.
2. Interoperability:
   - Many programming languages use primitive data types, so using them in Java can make it easier to interact with other languages.
3. Language features:
   - Java provides a number of language features that are designed to work with primitive data types, such as arithmetic operators, comparison operators, and logical operators.
4. **Memory management**:
   - **Primitive data types are allocated on the stack, which makes them easier to manage than objects, which are allocated on the heap. The stack is automatically managed by the JVM, so there's no need to worry about memory allocation or garbage collection.**

Variables

Some reasons why we use **Reference data types (e.g. Integer)** in Java:

1. When working with collections:
   - Many collections in Java, such as ArrayList and LinkedList, require objects to be stored in them, not primitives. In this case, you would need to use the Integer class to wrap your int values.
2. When working with generics:
   - Generics in Java do not support primitive types, so if you want to use a primitive value as a generic parameter, you would need to use the corresponding wrapper class. For example, if you want to create a List of int values, you would need to use List Integer.
3. When working with APIs that require objects:
   - Some APIs in Java require objects instead of primitives. For example, the Collections.sort() method requires a List of Comparable objects, not a list of primitive values. In this case, you would need to use the Integer class to wrap your int values.
4. When dealing with null values:
   - Primitives in Java cannot be null, but Integer objects can be. This can be useful in scenarios where you need to represent the absence of a value.

Stack and Heap in JVM



Stack and Heap in JVM

Here are some reasons why the JVM uses this structure:

- Efficiency:
  - The stack is a simple and efficient data structure that allows for quick allocation and deallocation of memory. It also allows for **efficient access to recently-used memory locations**. The heap, on the other hand, is a more complex data structure that allows for **dynamic allocation of memory**, but it is less efficient than the stack.
- Separation of concerns:
  - The stack is used to store primitive values and object references, while the heap is used to store objects and their instance variables. This separation of concerns helps to keep the code clean and easy to manage.
- Garbage collection:
  - The JVM uses a garbage collector to manage the memory used by objects on the heap. The garbage collector periodically scans the heap for objects that are no longer being used and frees up their memory. This automatic memory management helps to prevent memory leaks and other memory-related errors.

In general, the use of stack and heap memory is a common pattern in computer systems and is used in many programming languages, not just Java.

Methods

Here is an example of a typical method declaration:

```
public int sampleMethod(int param1, double param2) throws IOException,AssertionError {
    //Calculation
    // ...
    return 0;
}
```

Generally, method declarations have six components, in order:

- Modifiers
- The return type
- The parameter list in parenthesis
- An exception list
- The method body

The signature of the method declared above is:
sampleMethod(int, double)

Method Overloading

Method overloading provides a core Object-Oriented Programming (OOP) feature called **Polymorphism**. Polymorphism refers to the ability of objects to take on many forms.

```
public class Bicycle {

    public int sampleMethod(int param1, double param2) {
        //Calculation
        // ...
        return 0;
    }
    public int sampleMethod(String param1, double param2) {
        //Calculation
        // ...
        return 0;
    }    public int sampleMethod(float param1, double param2) {
        //Calculation
        // ...
        return 0;
    }
```

Method Overloading

When overloading a method, we should be careful about the following problems:

- Ambiguity:
  - Overloading a method with similar parameter types can lead to ambiguity. When a method is called with arguments, the compiler tries to match the argument types with the parameter types of the overloaded methods. If the compiler is unable to determine the correct method to call, it will raise a compile-time error.
- Changing behavior:
  - Overloading a method with different parameter types can lead to unexpected behavior if the method implementation is not designed to handle all possible argument types. We should ensure that each overloaded method behaves as expected and handles all possible input types.
- Method signature:
  - Overloaded methods must have different method signatures, which includes the method name and the parameter list. The return type of the method cannot be used to differentiate between overloaded methods. Therefore, it's important to ensure that each overloaded method has a unique signature.

Constructor

Definition (Constructor)

A constructor is a special **method** that is used to initialize objects. The constructor has the same name as the class and **does not have a return type, not even void**.

- Constructors are used to initialize objects and set their initial state.
- Constructors can be **overloaded** to provide different ways of creating and initializing objects. This allows for flexibility in object creation and initialization.
- When an object is created using the new keyword, a constructor is called automatically to initialize the object's state.

Constructor

Code ...

Nested Class (Inner Class)

Definition

An inner class or nested class is a class that is **declared inside the class or interface**.

Nested class provides:

- Encapsulation:
  - Nested classes can be used to encapsulate related functionality within a single class. This can improve code organization and make the code more modular and easier to maintain.
- Information Hiding:
  - By nesting a class within another class, we can control the visibility of the nested class. We can make the nested class private, for example, so that it is only accessible from within the containing class.
- Better Design:
  - Sometimes it makes sense to have a class that is only used by another class. By nesting the class, we can avoid polluting the global namespace with unnecessary class names and make the code more organized.

Constructor

Code ...

this

Within an instance method or a constructor, **this is a reference to the current object** — the object whose method or constructor is being called.

```java
public class Point {
    public int x = 0;
    public int y = 0;

    //constructor
    public Point(int a, int b) {
        x = a;
        y = b;
    }
}
```

```java
public class Point {
    public int x = 0;
    public int y = 0;

    //constructor
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

this

You can also use the **this** keyword to call another constructor in the same class. Doing so is called an **explicit constructor invocation**

```java
public class Rectangle {
    private int x, y;
    private int width, height;

    public Rectangle() {
        this( x: 0,  y: 0,  width: 1,  height: 1);
    }
    public Rectangle(int width, int height) {
        this( x: 0,  y: 0, width, height);
    }
    public Rectangle(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
}
```

final

The **final** keyword in java is used to restrict the user. The java **final** keyword can be used in many context. Final can be:

- variable
- method
- class

If you make any variable as final, you cannot change the value of final variable(**It will be constant**).

```
final int MAX_VALUE = 100;
```

If you make any method as final, you cannot **override** it.

```
public final void doSomething() {
    // Method implementation here
}
```

equals

In Java, the **equals()** method is used to determine whether two objects are **semantically** equal.

```java
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}
```

In Java, two objects with the same value are not necessarily equal because the default implementation of the **equals()** method in the Object class only checks for reference equality.

equals

We can do this by overriding the **equals()** method in the Person class to compare the name and age fields of the objects. Here's an example:

```
public class Person {
    // fields and constructor omitted for brevity

    @Override
    public boolean equals(Object obj) {
        if (obj == this) {
            return true;
        }
        if (!(obj instanceof Person)) {
            return false;
        }
        Person other = (Person) obj;
        return this.name.equals(other.name) && this.age == other.age;
    }
}
```

toString

The toString() method returns the String representation of the object.

```java
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString() {
        return "Person{name='" + name + "', age=" + age + "}";
    }
}
```

Outline

- Class
  - Variables (Fields, Variables, and Parameters)
  - Methods - Overloading
  - Access Modifiers
  - Constructors
  - Nested class
  - Class related Keywords: this, final, equals, hashcode, toString, ...
- Interface and Abstract class
  - Default methods (Java 8)
  - Functional interface (Java 8)
- Object
  - References, Identity, and State
- Inheritance and Polymorphism
  - super
  - Overriding
- Typecasting (Upcasting and Downcasting)
- Serialization and Externalization
- ORM (Object Relational Mapping)

Interface

You can also use the **this** keyword to call another constructor in the same class. Doing so is called an **explicit constructor invocation**

```java
public class Rectangle {
    private int x, y;
    private int width, height;

    public Rectangle() {
        this( x: 0,  y: 0,  width: 1,  height: 1);
    }
    public Rectangle(int width, int height) {
        this( x: 0,  y: 0, width, height);
    }
    public Rectangle(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
}
```

Interface

You can also use the **this** keyword to call another constructor in the same class. Doing so is called an **explicit constructor invocation**

```java
public class Rectangle {
    private int x, y;
    private int width, height;

    public Rectangle() {
        this( x: 0,  y: 0,  width: 1,  height: 1);
    }
    public Rectangle(int width, int height) {
        this( x: 0,  y: 0, width, height);
    }
    public Rectangle(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
}
```

Abstract class

You can also use the **this** keyword to call another constructor in the same class. Doing so is called an **explicit constructor invocation**

```java
public class Rectangle {
    private int x, y;
    private int width, height;

    public Rectangle() {
        this( x: 0,  y: 0,  width: 1,  height: 1);
    }
    public Rectangle(int width, int height) {
        this( x: 0,  y: 0, width, height);
    }
    public Rectangle(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
}
```