Java

(OOPS , wrapper class , Dictionaries , Call by Reference for objects)

OOPS (Object Oriented Programming System) concepts:

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of objects, which can contain data and code that manipulates that data. Java is a strongly object-oriented programming language, and it supports the following core OOP concepts:

1. **Encapsulation**
2. **Inheritance**
3. **Polymorphism**
4. **Abstraction**

Let's discuss each of these concepts in detail with examples.

1. Encapsulation

Encapsulation is the bundling of data (fields) and methods (functions) that operate on the data into a single unit or class. It restricts direct access to some of an object's components, which is a means of preventing accidental interference and misuse of the data.

Scenario: A banking application that handles customer accounts, transactions, and balances.

Need: Encapsulation helps protect sensitive information, such as account balances and personal details, from unauthorized access and modifications.

```
public class Person {
    // Private fields
    private String name;
    private int age;
    // Public constructor
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
```

```
    // Public getter for name
    public String getName() {
        return name;
    }
    // Public setter for name
    public void setName(String name) {
        this.name = name;
    }
    // Public getter for age
    public int getAge() {
        return age;
    }
    // Public setter for age
    public void setAge(int age) {
        if (age > 0) {
            this.age = age;
        }
    }
    // Method to display person details
    public void display() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
}
public class Main {
    public static void main(String[] args) {
```

```
        Person person = new Person("John", 25);
        person.display();

        person.setAge(30);
        person.display();
    }
}
```

2. Inheritance

Inheritance is a mechanism in which one class (subclass or derived class) inherits the fields and methods of another class (superclass or base class). It promotes code reuse and establishes a relationship between classes.

Scenario: A vehicle management system for a rental company that handles different types of vehicles, such as cars, bikes, and trucks.

Need: Inheritance allows the creation of a base class Vehicle with common attributes and methods, and subclasses for specific vehicle types with additional features.

```
// Base class
public class Animal {
    protected String name;
    public Animal(String name) {
        this.name = name;
    }
    public void eat() {
        System.out.println(name + " is eating.");
    }
}
// Derived class
```

```
public class Dog extends Animal {
    public Dog(String name) {
        super(name);
    }
    public void bark() {
        System.out.println(name + " is barking.");
    }
}
public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog("Buddy");
        dog.eat();   // Inherited method
        dog.bark();  // Own method
    }
}
```

3. Polymorphism

Polymorphism allows methods to do different things based on the object it is acting upon. It can be achieved in two ways: method overloading (compile-time polymorphism) and method overriding (runtime polymorphism).

Scenario: A payment processing system that supports multiple payment methods, such as credit cards, debit cards, and PayPal.

Need: Polymorphism allows the system to handle different payment methods through a common interface, simplifying the process of adding new payment methods.

Example (Method Overloading):

```
public class MathOperations {
    // Overloaded method for addition
    public int add(int a, int b) {
```

```
        return a + b;
    }
    public double add(double a, double b) {
        return a + b;
    }
}
public class Main {
    public static void main(String[] args) {
        MathOperations math = new MathOperations();
        System.out.println(math.add(5, 3));       // Calls int add(int, int)
        System.out.println(math.add(5.0, 3.0));   // Calls double add(double, double)
    }
}
```

Example (Method Overriding):

```
// Base class
public class Animal {
    public void makeSound() {
        System.out.println("Animal sound");
    }
}
// Derived class
public class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Woof");
```

```
    }
}
public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Dog();
        myAnimal.makeSound();  // Calls the overridden method in Dog
    }
}
```

4. Abstraction

Abstraction is the concept of hiding the complex implementation details and showing only the essential features of the object. It can be achieved using abstract classes and interfaces.

Scenario: An employee management system for a company that handles different types of employees, such as full-time, part-time, and contractors.

Need: Abstraction allows defining a common interface or abstract class for different employee types, ensuring that each type implements necessary methods while hiding implementation details.

Example (Abstract Class):

```
// Abstract class
abstract class Animal {
    public abstract void makeSound();
    public void sleep() {
        System.out.println("Zzz...");
    }
}
// Concrete class
public class Dog extends Animal {
```

```
    @Override
    public void makeSound() {
        System.out.println("Woof");
    }
}
public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.makeSound();  // Calls the overridden method in Dog
        dog.sleep();      // Calls the concrete method in Animal
    }
}
```

Example (Interface):

```
// Interface
interface Animal {
    void makeSound();
}
// Implementing class
public class Dog implements Animal {
    @Override
    public void makeSound() {
        System.out.println("Woof");
    }
}
public class Main {
```

```
    public static void main(String[] args) {
        Animal myDog = new Dog();
        myDog.makeSound();  // Calls the method in Dog
    }
}
```

Java's object-oriented programming features help in creating modular, reusable, and maintainable code. Encapsulation helps protect data, inheritance promotes code reuse, polymorphism allows flexibility in using methods, and abstraction hides complexity while exposing only the necessary parts of an object.

Dictionaries in Java:

In Java, dictionaries are typically represented using the Map interface and its various implementations, such as HashMap, TreeMap, and LinkedHashMap. These classes are part of the Java Collections Framework and provide the functionality to store key-value pairs.

Example: Using HashMap in Java

A HashMap is an implementation of the Map interface that provides constant-time performance for basic operations such as get and put. It does not guarantee any specific order of the elements.

Here's a basic example of how to work with a HashMap:

```
import java.util.HashMap;
import java.util.Map;

public class Main {
    public static void main(String[] args) {
        // Creating a HashMap
        Map<String, Integer> dictionary = new HashMap<>();
        // Adding key-value pairs to the HashMap
        dictionary.put("Apple", 1);
```

```
        dictionary.put("Banana", 2);
        dictionary.put("Cherry", 3);
        // Retrieving a value by key
        System.out.println("Value for key 'Apple': " + dictionary.get("Apple"));
        // Checking if a key exists
        if (dictionary.containsKey("Banana")) {
            System.out.println("The key 'Banana' exists in the dictionary.");
        }
        // Checking if a value exists
        if (dictionary.containsValue(3)) {
            System.out.println("The value 3 exists in the dictionary.");
        }
        // Iterating over keys and values
        for (Map.Entry<String, Integer> entry : dictionary.entrySet()) {
            System.out.println("Key: " + entry.getKey() + ", Value: " + entry.getValue());
        }
        // Removing a key-value pair
        dictionary.remove("Cherry");
        // Printing the final state of the HashMap
        System.out.println("Final dictionary: " + dictionary);
    }
}
```

Other Map Implementations

TreeMap: Maintains a sorted order of keys.

```
Map<String, Integer> treeMap = new TreeMap<>();
```

LinkedHashMap: Maintains the insertion order of keys.

```
Map<String, Integer> linkedHashMap = new LinkedHashMap<>();
```

Use Cases for Different Implementations

- **HashMap**: Best for general-purpose use where order is not important and you need fast access to elements.
- **TreeMap**: Useful when you need the keys to be sorted in natural order or by a comparator.
- **LinkedHashMap**: Useful when you need to maintain the insertion order or access order of elements.

Wrapper Class:

In Java, a wrapper class is a class that wraps (encapsulates) a primitive data type within an object. Java provides wrapper classes for each of the primitive data types, allowing them to be used as objects in Java collections and providing additional functionalities that primitive types do not have, such as methods and constructors.

Here are the wrapper classes for primitive data types in Java:

1. **Byte**: java.lang.Byte
2. **Short**: java.lang.Short
3. **Integer**: java.lang.Integer
4. **Long**: java.lang.Long
5. **Float**: java.lang.Float
6. **Double**: java.lang.Double
7. **Boolean**: java.lang.Boolean
8. **Character**: java.lang.Character

Purpose of Wrapper Classes:

1. **Conversion**: Wrapper classes provide methods to convert primitive data types to objects (valueOf()) and objects back to primitive types (xxxValue() methods).
2. **Nullability**: Wrapper classes can hold null values, whereas primitive types cannot.
3. **Compatibility**: They allow primitive types to be used in collections that require objects, such as ArrayList or HashMap.
4. **Additional Methods**: Wrapper classes provide useful methods for various operations, such as parsing, comparing, and converting values.

```
public class WrapperExample {

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

```
        // Using Integer wrapper class
        Integer num1 = new Integer(10); // using constructor
        Integer num2 = Integer.valueOf(20); // using valueOf()
        // Converting wrapper object to primitive type
        int sum = num1.intValue() + num2.intValue();
        System.out.println("Sum: " + sum);
        // Auto-boxing and auto-unboxing
        Integer num3 = 30; // Auto-boxing: primitive to wrapper
        int num4 = num3;   // Auto-unboxing: wrapper to primitive
        // Checking equality
        System.out.println("Equality check: " + (num3.equals(num4))); // true
        // Converting String to Integer
        String str = "123";
        Integer parsed = Integer.parseInt(str);
        System.out.println("Parsed integer: " + parsed);
        // Null values
        Integer nullValue = null;
        System.out.println("Null value: " + nullValue); // Prints null
    }
}
```

Key Points:

- **Auto-boxing**: The process of converting a primitive type to its corresponding wrapper class automatically.
- **Auto-unboxing**: The process of converting a wrapper class object back to its corresponding primitive type automatically.
- **Parsing**: Wrapper classes provide methods like parseInt() (for Integer) to convert a String to its corresponding primitive type.

Wrapper classes are integral in scenarios where primitive data types need to be treated as objects, such as in collections or when null values are required. They provide flexibility and additional functionalities beyond what primitive types alone can offer in Java.

Feature	Primitive Data Types	Wrapper Classes
Storage and Use	Directly store values	Wrap primitive types in objects
Memory and Performance	Efficient, less memory overhead	Less efficient, more memory overhead
Nullability	Cannot be null	Can be null
Methods and Functionalities	No methods	Provide methods for conversion, comparison, etc.
Usage in Collections	Cannot be used in collections	Can be used in collections
Conversion Methods	Not applicable	Provide methods like parseInt, valueOf, etc.
Constants	No constants	Have constants like MAX_VALUE, MIN_VALUE
Autoboxing/Unboxing	Not applicable	Support autoboxing and unboxing
Instantiation	Direct assignment	Requires object creation (e.g., new Integer(5))
Equality Comparison	Use == for comparison	Use .equals() for comparison
Default Values	Have default values (e.g., 0 for int)	Can be null
Type	Primitive	Reference type (object)
Examples	int, float, char	Integer, Float, Character

Additional functionalities that wrapper class provide:

Wrapper classes in Java provide additional functionalities that are not available with primitive data types. These functionalities include:

1. **Conversion Methods**

Wrapper classes provide methods to convert between different types and to and from String.

- parseInt(String s): Converts a String to an int.
- toString(): Converts the value to a String.
- valueOf(String s): Converts a String to an object of the wrapper class.

```
int num = Integer.parseInt("123")

String str = Integer.toString(123)

Integer numObject = Integer.valueOf("123")
```

-

2. **Utility Methods**

Wrapper classes provide utility methods for operations like comparison and value extraction.

- compareTo(): Compares two values.
- equals(): Checks if two objects are equal.
- xxxValue(): Returns the value as a specific primitive type.

```
Integer num1 = 10;

Integer num2 = 20;

int comparison = num1.compareTo(num2); // returns a negative value because num1 < num2

boolean isEqual = num1.equals(num2); // false

int intValue = num1.intValue(); // 10
```

3. **Constants**

Wrapper classes provide useful constants like MAX_VALUE and MIN_VALUE.

```
int maxInt = Integer.MAX_VALUE

int minInt = Integer.MIN_VALUE

System.out.println("Max int: " + maxInt)

System.out.println("Min int: " + minInt)
```

4. **Autoboxing and Unboxing**

Autoboxing automatically converts primitive types to their corresponding wrapper classes. Unboxing converts wrapper class objects to their corresponding primitive types.

```
Integer num = 10 // Autoboxing

int n = num  // Unboxing
```

5. **Nullability**

Wrapper classes can be assigned null, allowing the representation of the absence of a value, which is useful in collections and other data structures.

```
Integer num = null;

System.out.println("Nullable Integer: " + num); // Prints null
```

Call (Pass) by reference in java :

In Java, the concept of "call by reference" can be a bit misleading because Java strictly uses "call by value". This means that when you pass an argument to a method, Java passes the value of the argument rather than a reference to the argument itself. However, when it comes to objects, the value passed is the reference to the object, which might give the impression of "call by reference".

Call by Value with Primitives

For primitive types (int, double, etc.), Java passes a copy of the value. Changes made to the parameter inside the method do not affect the original value.

```
public class CallByValue {

  public static void main(String[] args) {

    int num = 10;
```

```
        modifyPrimitive(num);
        System.out.println("After modifyPrimitive: " + num); // Output: 10
    }
    public static void modifyPrimitive(int n) {
        n = 20;
    }
}
```

Call by Value with Objects

For objects, Java passes the reference (address) of the object. This means that changes made to the object's fields inside the method will affect the original object. However, reassigning the reference to a new object inside the method will not change the original reference.

```
class MyObject {
    int value;

    MyObject(int value) {
        this.value = value;
    }
}

public class CallByReference {
    public static void main(String[] args) {
        MyObject obj = new MyObject(10);
        modifyObject(obj);
        System.out.println("After modifyObject: " + obj.value); // Output: 20
        reassignObject(obj);
```

```
        System.out.println("After reassignObject: " + obj.value); // Output: 20
    }
    public static void modifyObject(MyObject o) {
        o.value = 20;
    }
    public static void reassignObject(MyObject o) {
        o = new MyObject(30);
    }
}
```

The method reassignObject receives a copy of the reference to obj.

Reassigning o to a new MyObject does not change the reference obj in main, so the original object remains unchanged.

Java uses call by value for both primitives and objects. For primitives, it passes a copy of the value, and for objects, it passes a copy of the reference. **This means changes to the object's fields will reflect in the original object, but reassigning the reference inside the method will not affect the original reference.**