# WEEK-1

### **1. Call by Value**

In Java, all primitive data types are **passed by value**, meaning a copy of the value is passed to the method.

**Example Code:**

public class CallByValueExample {

public static void main(String[] args) {

int num = 10;

System.out.println("Before method call: " + num);

modifyValue(num);

System.out.println("After method call: " + num);

}

public static void modifyValue(int value) {

value = 20;

System.out.println("Inside method: " + value);

}

}

**Explanation:**

* The main method initializes an integer num with a value of 10.
* The modifyValue method is called with num as an argument.
* Inside modifyValue, the parameter value is changed to 20, but this does not affect the original num variable in main.
* The output demonstrates that the original num remains unchanged.

### **2. Call by Reference**

Java passes **objects by reference**, meaning the reference (memory address) of the object is passed to the method.

**Example Code:**

class MyObject {

int num;

MyObject(int num) {

this.num = num;

}

}

public class CallByReferenceExample {

public static void main(String[] args) {

MyObject obj = new MyObject(10);

System.out.println("Before method call: " + obj.num);

modifyObject(obj);

System.out.println("After method call: " + obj.num);

}

public static void modifyObject(MyObject obj) {

obj.num = 20;

System.out.println("Inside method: " + obj.num);

}

}

**Explanation:**

* A class MyObject is defined with an integer member num.
* In the main method, an instance of MyObject is created with num set to 10.
* The modifyObject method is called with obj as an argument.
* Inside modifyObject, the num field of the obj instance is changed to 20.
* The output shows that the original object's num value is modified.

### 3. Scope and Lifetime of Variable

Scope refers to the region of the program **where a variable is accessible**, and **lifetime refers to the duration** a variable exists in memory.

**Example Code:**

public class ScopeAndLifetimeExample {

public static void main(String[] args) {

int outer = 10; // outer scope

System.out.println("Outer variable: " + outer);

if (true) {

int inner = 20; // inner scope

System.out.println("Inner variable: " + inner);

}

// The following line would cause an error because 'inner' is out of scope

System.out.println("Inner variable outside block: " + inner);

}

}

**Explanation:**

* The outer variable is declared in the main method, making it accessible throughout the method.
* The inner variable is declared inside the if block, making it accessible only within that block.
* The last line would cause an error because inner is not accessible outside its block, demonstrating its limited scope and lifetime.

### 4. Objects

Objects are instances of classes in Java, representing real-world entities. They encapsulate data and behavior.

**Example Code:**

public class ObjectExample {

public static void main(String[] args) {

// Creating an object of class Car

Car myCar = new Car("Toyota", "Camry", 2020);

// Accessing object properties and methods

System.out.println("Make: " + myCar.make);

System.out.println("Model: " + myCar.model);

myCar.start();

myCar.drive();

}

}

class Car {

String make;

String model;

int year;

// Constructor

Car(String make, String model, int year) {

this.make = make;

this.model = model;

this.year = year;

}

// Method to start the car

void start() {

System.out.println("The car is starting.");

}

// Method to drive the car

void drive() {

System.out.println("The car is moving.");

}

}

**Explanation:**

* An object of the class Car is created in the main method using the new keyword.
* Properties (make, model, year) and methods (start, drive) of the Car class are accessed using the dot notation.

### 5. Subtyping

Subtyping refers to the relationship between types where a value of one type can be substituted for a value of another type.

**Example Code:**

class Animal {

void makeSound() {

System.out.println("Some sound");

}

}

class Dog extends Animal {

void makeSound() {

System.out.println("Woof");

}

}

public class SubtypingExample {

public static void main(String[] args) {

Animal myDog = new Dog(); // Subtyping

myDog.makeSound(); // Outputs "Woof"

}

}

**Explanation:**

* The Dog class extends the Animal class and overrides the makeSound() method.
* In the main method, an object of Dog is assigned to a reference variable of type Animal, demonstrating subtyping.
* Despite the reference being of type Animal, the overridden method makeSound() of Dog is called.

### 6. Dynamic Lookup

Dynamic lookup is the process of determining the method to invoke at runtime, based on the actual type of the object.

**Example Code:**

class Parent {

void print() {

System.out.println("Parent's print method");

}

}

class Child extends Parent {

void print() {

System.out.println("Child's print method");

}

}

public class DynamicLookupExample {

public static void main(String[] args) {

Parent obj = new Child(); // Dynamic lookup

obj.print(); // Outputs "Child's print method"

}

}

**Explanation:**

* The Child class extends the Parent class and overrides the print() method.
* In the main method, an object of Child is assigned to a reference variable of type Parent, demonstrating dynamic lookup.
* Despite the reference being of type Parent, the overridden method print() of Child is called.

### 7. Inheritance

Inheritance is the mechanism where a new class inherits properties and behavior from an existing class.

**Example Code:**

class Animal {

void eat() {

System.out.println("Animal is eating");

}

}

class Dog extends Animal {

void bark() {

System.out.println("Dog is barking");

}

}

public class InheritanceExample {

public static void main(String[] args) {

Dog myDog = new Dog();

myDog.eat(); // Inherited method

myDog.bark(); // Method of subclass

}

}

**Explanation:**

* The Dog class extends the Animal class, inheriting its eat() method.
* Dog class defines its own method bark().
* An object of Dog class can access both eat() (inherited) and bark() methods.

### 8. Modularity in OOPs

Modularity is a fundamental concept in object-oriented programming (OOP) that involves **breaking down a software** system into distinct, manageable, and interchangeable modules.

**Example:**

Imagine you are building a software application for an online bookstore. To manage the complexity, you can divide the application into several modules, each responsible for a different aspect of the bookstore's functionality:

1. **User Management Module**:
   * **Responsibility**: Handles user registration, login, profile management, and authentication.
   * **Classes**: User, UserProfile, AuthenticationService.
2. **Product Management Module**:
   * **Responsibility**: Manages the inventory of books, including adding new books, updating book details, and searching for books.
   * **Classes**: Book, Inventory, SearchService.
3. **Order Management Module**:
   * **Responsibility**: Handles the process of placing orders, tracking order status, and managing payment.
   * **Classes**: Order, OrderItem, PaymentService.
4. **Review Management Module**:
   * **Responsibility**: Manages customer reviews and ratings for books.
   * **Classes**: Review, Rating, ReviewService.