**Eugene Kaguri**

**23/02507**

**Section 1:**

1. **Explain the differences between primitive and reference data types.**

**Primitive vs. Reference Data Types:**

* **Primitive Data Types:** Primitive data types are basic data types that store simple values like numbers, characters, and boolean values**. Examples** include `int`, `char`, `boolean`, `byte`, `short`, `long`, `float`, and `double`. These data types are stored directly in the stack memory and have a fixed size. For instance, an `int` in Java is always 4 bytes.

**E.g**;

int age = 25;

char initial = 'A';

boolean isStudent = true;

* **Reference Data Types:** Reference data types store references or memory addresses that point to objects or data structures in the heap memory. They do not hold the actual value but rather a reference to the value. Examples include classes, arrays, and interfaces.

**Eg;**

**String name = "John";**

**int[] numbers = {1, 2, 3, 4, 5};**

1. **Define the scope of a variable (hint: local and global variable)**

**Scope of a Variable**

In programming, the scope of a variable defines where in the code the variable can be accessed and used. Java categorizes variables into local variables, instance variables (member variables), and static variables (class variables), each with its own scope and accessibility rules.

* **Local Variables:**

**Definition:** Local variables are declared within a method, constructor, or block of code.

**Scope:** Limited to the method, constructor, or block where they are declared.

**Lifetime:** Exists only during the execution of the method, constructor, or block.

**Example:**

public class Example {

public void someMethod() {

// Local variable

int localVar = 10;

System.out.println(localVar); // Accessible here

}

}

* **Instance Variables (Non-static Member Variables):**

**Definition:** Declared within a class but outside any method, constructor, or block, without the static keyword.

**Scope:** Each instance (object) of the class has its own copy.

**Lifetime:** Exists as long as the instance of the class exists.

**Access:** Accessed using an object of the class.

**Example:**

public class Example {

// Instance variable

private int instanceVar;

// Constructor to initialize instance variable

public Example(int value) {

this.instanceVar = value;

}

// Instance method using instance variable

public void display() {

System.out.println(instanceVar);

}

}

* **Static Variables (Class Variables or "Global" Variables):**

**Definition:** Declared with the static keyword within a class, outside any method, constructor, or block. In Java, static variables are often referred to as "global" variables because they can be accessed from anywhere in the application using the class name.

**Scope:** Shared among all instances of the class.

**Lifetime:** Exists as long as the class is loaded in memory.

**Access:** Accessed using the class name or through an object of the class.

**Example:**

public class Example {

// Static variable

public static int staticVar;

// Static method

public static void staticMethod() {

// Accessing static variable

staticVar = 10;

}

}

* **Method Parameters:**

**Definition:** Variables that are passed to methods as arguments.

**Scope:** Limited to the method where they are defined.

**Lifetime:** Exists only during the execution of the method.

**Example:**

public class Example {

public void someMethod(int param) {

// param is a method parameter

System.out.println(param); // Accessible here

}

}

1. Why is initialization of variables required.

**Importance of Initialization of Variables**

In programming, initialization refers to assigning an initial value to a variable when it is declared. This practice is essential for several reasons:

* **Avoiding Garbage Values:**

-When a variable is declared but not initialized, it may contain garbage values—random or undefined data left in the variable's memory location from previous operations.

-Initializing variables ensures that they start with a known and predictable value, preventing unintended behaviors or errors caused by undefined data.

* **Ensuring Predictable Behavior:**

-Initialization sets a starting point for variables, establishing their initial state or value.

-This predictability is crucial in programming logic and calculations, where variables need to have defined values to produce correct results.

* **Compiler and Language Requirements:**

-Many programming languages, including Java, require variables to be initialized before they can be used in calculations or logic operations.

-Failure to initialize a variable can result in compile-time errors or warnings, enforcing good programming practices and preventing potential runtime errors.

* **Maintaining Code Readability and Debugging:**

-Initializing variables explicitly makes code more readable and understandable for other developers.

-It clarifies the intent of the variable and its usage, reducing confusion and making debugging easier when unexpected behaviors arise.

* **Supporting Program Logic and Flow:**

-Initialization is fundamental to defining the flow and logic of programs.

-It allows variables to participate in conditional statements, loops, and other control structures based on their initial values.

**Example in Java:**

java

public class InitializationExample {

public static void main(String[] args) {

// Variable initialization

int x = 10; // Initialized with a value

// Using initialized variable

int y = x + 5; // y = 10 + 5 = 15

System.out.println("The value of y is: " + y);

}

}

1. Differentiate between static, instance and local variables.

* **Local Variables:** Local variables are declared within a method, constructor, or block of code.

**Scope:** Limited to the method, constructor, or block where they are declared.

**Lifetime:** Exists only during the execution of the method, constructor, or block.

**Access:** Accessed only within the method, constructor, or block where they are declared.

java

public class Example {

public void someMethod() {

// Local variable

int localVar = 10;

System.out.println(localVar); // Accessible here

}

}

* **Instance Variables (Member Variables):** Instance variables are declared within a class but outside any method, constructor, or block. Each instance of the class has its own copy of instance variables.

**Scope:** Accessible by all methods within the class through an instance of the class.

**Lifetime:** Exists as long as the instance of the class exists.

java

public class Example {

// Instance variable

private int instanceVar;

public void setInstanceVar(int value) {

this.instanceVar = value;

}

public void display() {

System.out.println(instanceVar); // Accessible here

}

}

* **Static Variables (Class Variables):**

**-** Static variables are declared with the static keyword within a class but outside any method, constructor, or block. They are shared among all instances of the class.

**Scope:** Accessible by all methods within the class and can be accessed without creating an instance of the class.

**Lifetime:** Exists as long as the class is loaded in memory.

**Access:** Accessed using the class name or through an object of the class.

java

public class Example {

// Static variable

private static int staticVar;

public static void setStaticVar(int value) {

staticVar = value;

}

public static void display() {

System.out.println(staticVar); // Accessible here

}

}

1. **Differentiate between widening and narrowing casting in java.**

**Widening casting** in Java automatically converts smaller data types to larger ones, while **narrowing casting** requires explicit conversion from larger to smaller types, potentially risking data loss.

* **Widening Casting (Automatic/Implicit Casting):** Widening casting refers to the automatic conversion of a smaller data type to a larger data type. It happens implicitly and does not require explicit code.

**Order:** The conversion follows the order: byte → short → int → long → float → double.

**Data Loss:** There is no loss of data in widening casting because the larger data type can accommodate all values of the smaller data type.

**Syntax:** It is done automatically by the Java compiler.

**Example:**

public class WideningCasting {

public static void main(String[] args) {

int num = 100;

double doubleNum = num; // Automatic casting: int to double

System.out.println(doubleNum); // Output: 100.0

}

}

**2. Narrowing Casting (Manual/Explicit Casting):** Narrowing casting refers to the explicit conversion of a larger data type to a smaller data type. It requires explicit code because there is a risk of data loss.

**Order:** The conversion follows the reverse order of widening casting: double → float → long → int → short → byte.

**Data Loss:** There can be a loss of data or precision because the smaller data type might not be able to accommodate all the values of the larger data type.

**Syntax:** It must be done manually by the programmer using a cast operator.

**Example:**

public class NarrowingCasting {

public static void main(String[] args) {

double doubleNum = 100.04;

int num = (int) doubleNum; // Manual casting: double to int

System.out.println(num); // Output: 100

}

}

1. **the following table shows data type, its size, default value and the range. Filling in the missing values.**

|  |  |  |  |
| --- | --- | --- | --- |
| **TYPE** | **SIZE (IN BYTES)** | **DEFAULT** | **RANGE** |
| boolean | 1 bit | false | true, false |
| Char | 2 | '\u0000' | ‘\0000’ to ‘\ffff’ |
| Byte | 1 | 0 | -128 to 127 |
| Short | 2 | 0 | -215 to +215-1 |
| Int | 4 | 0 | 2,147,483,648 to 2,147,483,647 |
| Long | 8 | 0L | - -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| Float | 4 | 00.0f | ±1.4E-45 to ±3.4028235E+38 |
| Double | 8 | 0.0 | -1.8E+308 to +1.8E+308 |

1. **Define class as used in OOP**

* A **class** in object-oriented programming (OOP) is a blueprint or template for creating objects. It serves as a fundamental building block in OOP, encapsulating data (properties or attributes) and functionality (methods or behaviors) into a single cohesive unit.
* Classes enable the creation of multiple instances (objects) that share the same structure and behavior defined within the class. This promotes code reusability, enhances organization, and facilitates the implementation of concepts like inheritance and polymorphism, which are essential for designing complex and scalable software systems.

1. **Explain the importance of classes in Java programming.**

 **Blueprint for Objects**: Classes in Java serve as templates or blueprints for creating objects. They define the structure (attributes) and behavior (methods) that objects will have.

 **Encapsulation**: They encapsulate data (attributes) and behavior (methods) into cohesive units, promoting code organization and reusability by allowing objects to be easily replicated and modified.

 **Code Reusability**: By defining classes, developers can create multiple instances (objects) with similar characteristics and functionalities. This promotes code reuse, reducing redundancy and improving maintainability.

 **Object-Oriented Principles**: Classes facilitate the implementation of object-oriented principles such as inheritance, polymorphism, and encapsulation.

-**Inheritance**: Classes can inherit attributes and behaviors from other classes, allowing for hierarchical relationships and code reuse.

-**Polymorphism**: Through polymorphism, different classes can be treated as instances of the same superclass, providing flexibility and extensibility in code design.

-**Encapsulation**: Encapsulation ensures that the internal state of an object is protected from outside interference, promoting data integrity and security.

 **Structured Development**: They form the foundation for structured and efficient software development in Java. They enable modular design, where each class represents a distinct functionality or entity, leading to easier maintenance and scalability of software systems.