WYCLIFFE MUINDE 23\03845.

JAVA PROGRAMMING. DIT 409. ASSIGNMENT.

**Section 1:**

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

Primitive and reference data types are two fundamental ways to store data in computer programs. Here's how they differ:

**Storage:**

* **Primitive data types:** These hold the actual value directly in the variable itself. They are simple and fixed in size, meaning they can only store a specific range of values depending on the type (e.g., integers, characters).
* **Reference data types:** These variables don't store the actual data. Instead, they store a memory address, which acts like a pointer, referencing the location in memory where the actual data is stored. This allows for more complex data structures like objects and arrays.

**Examples:**

* **Primitive:**
  + int age = 25; (stores the value 25 directly in the variable)
  + char initial = 'A'; (stores the character 'A' directly)
* **Reference:**
  + String name = "John"; (stores the memory address of the string "John" in memory, not the string itself)
  + int[] numbers = {1, 2, 3}; (stores the memory address of the array containing the numbers 1, 2, and 3)

**Impact on Assignment:**

* Assigning a value to another primitive variable creates a copy of the value. Changes to one variable won't affect the other.
* Assigning a reference variable to another one creates a copy of the reference (memory address). Both variables now point to the same data in memory, so changes made through one variable will be reflected in the other.

**Choosing the Right Type:**

* Use primitives for simple data like numbers, characters, or boolean values (true/false).
* Use references for complex data structures like objects, arrays, or any data that needs to be shared and modified across different parts of your program.

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

 **Local Scope:**

* A local variable is declared within a specific block of code, typically inside a function, loop, or conditional statement.
* Its scope is limited to that block. You can only access and modify the local variable within the block it's declared in.
* Local variables help prevent naming conflicts with variables from other parts of your program and promote better code organization.

 **Global Scope:**

* A global variable is declared outside of any function or block of code.
* Its scope is the entire program. You can access and modify the global variable from anywhere in your code.
* While convenient for sharing data across the program, overuse of global variables can lead to naming conflicts and make code harder to maintain. It's generally recommended to use local variables whenever possible.

1. Why is initialization of variables required.

**Predictability:** Uninitialized variables contain indeterminate values, meaning they could hold leftover data from previous program runs or random bits in memory. Using such values in calculations or logic can lead to unexpected behavior and bugs that are difficult to track down. Initialization ensures you know exactly what value the variable holds before using it.

**Preventing Errors:** Some programming languages enforce mandatory initialization of variables before use. This helps catch potential errors early on during compilation or code analysis, preventing the program from even running with undefined values.

**Clarity and Readability:** By initializing a variable with a meaningful starting value, you improve the readability and maintainability of your code. It makes it clear what the variable's purpose is and what data it's expected to hold. This is especially important when revisiting code or collaborating with other programmers.

**Default Values:** While not always the case, some languages might assign default values (often 0 or null) to uninitialized variables. However, relying on these defaults can be risky, as they might not align with your intended use for the variable and lead to errors.

Overall, initializing variables is a good practice that promotes reliable, predictable, and maintainable code. It eliminates the uncertainty of uninitialized values and sets a clear starting point for your program's data manipulation.

1. Differentiate between static, instance and local variables.

In object-oriented programming, there are three main types of variables that differ in their scope, lifetime, and how they are accessed: static, instance (also called class), and local variables. Here's a breakdown of their key differences:

**Scope:**

* **Static Variables:**
  + Declared with the static keyword within a class but outside of any method or block.
  + Scope is limited to the class itself, not individual objects.
  + There's only one copy of a static variable shared among all instances of the class.
* **Instance Variables:**
  + Declared within a class but outside of any method (often referred to as class member variables).
  + Scope is specific to each object (instance) of the class.
  + Each object has its own copy of the instance variable, with potentially different values.
* **Local Variables:**
  + Declared within a method, loop, or conditional block.
  + Scope is limited to the block of code where they are declared.
  + Local variables exist only during the execution of that block and are destroyed once the block finishes.

**Lifetime:**

* **Static Variables:**
  + Created at the start of the program and persist throughout the program's execution.
  + They are destroyed only when the program terminates.
* **Instance Variables:**
  + Created when an object is instantiated and destroyed when the object is garbage collected.
* **Local Variables:**
  + Created when the block of code they are declared in starts and destroyed when the block finishes execution.

**Access:**

* **Static Variables:**
  + Accessed directly using the class name followed by the variable name (e.g., ClassName.staticVariableName).
  + Don't require an object reference.
* **Instance Variables:**
  + Accessed through an object reference followed by the variable name (e.g., objectName.instanceVariableName).
* **Local Variables:**
  + Only accessible within the block of code where they are declared.

**Use Cases:**

* **Static Variables:**
  + Useful for constants (values that don't change) shared across all objects of the class (e.g., conversion factors, counters).
  + Can be used to store class-level information like the number of objects created.
* **Instance Variables:**
  + Represent the unique properties or attributes of each object (e.g., an Account object might have instance variables for accountNumber and balance).
* **Local Variables:**
  + Used for temporary data specific to a particular method or block of code (e.g., loop counters, calculation results).

1. Differentiate between widening and narrowing casting in java.

In Java, casting refers to manually converting a value from one data type to another. There are two main types of casting based on the direction of conversion: widening and narrowing.

**Widening Casting (Implicit Conversion):**

* Also known as **upcasting**.
* Occurs when converting a variable from a smaller data type to a larger data type.
* Since the larger type can hold a wider range of values, no data loss occurs.
* **Example:** Assigning an int value (smaller) to a long variable (larger).
* Widening casting happens **implicitly** in Java without any explicit code needed.
* It's generally safe as there's no information loss.

**Narrowing Casting (Explicit Conversion):**

* Also known as **downcasting**.
* Involves converting a variable from a larger data type to a smaller data type.
* Since the smaller type has a limited range, there's a potential for data loss if the larger value doesn't fit within the smaller type's range.
* **Example:** Assigning a long value to an int variable.
* Narrowing casting requires **explicit conversion** using the target data type in parentheses before the value.
* It's important to be cautious with narrowing casting, as data loss can occur if the larger value exceeds the smaller type's capacity.

**Key Points:**

* Widening - Converting smaller to larger (safe, implicit).
* Narrowing - Converting larger to smaller (risky, explicit).
* Always consider the data range of the target type before narrowing to avoid data loss.

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 |  | true, false |
| Char | 2 | ‘\u0000’ to ‘\uffff’ | ‘\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 | - 922,337,203,8,477,808 to 922,337,685,477,5807 |
| Float | 4 | 00.0f | Approx. +\_ 3.402823347E + 38F(6-7 sf) |
| Double | 8 | 0.0d | -1.8E+308 to +1.8E+308 |

1. Define class as used in OOP

In Object-Oriented Programming (OOP), a class acts as a blueprint or template for creating objects. It defines the properties (attributes) and functionalities (methods) that objects of that class will share. Here's a breakdown of what a class encompasses:

**Structure:**

* A class typically consists of two main components:
  + **Attributes (Member Variables):** These variables define the characteristics or data an object will hold. For example, an Animal class might have attributes like name, species, and age.
  + **Methods (Member Functions):** These are functions defined within the class that represent the actions or behaviors an object can perform. Examples include eat(), sleep(), or makeSound() methods for an Animal class.

**Functionality:**

* Classes provide a way to group related data and behavior together, promoting modularity and code organization.
* You can create multiple objects (instances) from a single class, each with its own set of attribute values. This allows for representing real-world entities or concepts with their unique characteristics.

1. Explain the importance of classes in Java programming.

**1. Reusability:** Classes promote code reuse by encapsulating data (attributes) and functionalities (methods) together. You can define a class once and create multiple objects (instances) of that class, each inheriting the same functionality without rewriting code. This saves time, reduces redundancy, and improves code maintainability.

**2. Encapsulation:** Classes enforce encapsulation, a crucial concept in OOP. By bundling data and methods within a class, you control access to the data. You can define public methods to allow interaction with the data and private methods to handle internal logic, protecting the data's integrity. This prevents accidental modifications and promotes data security.

**3. Organization:** Classes help organize your code by grouping related functionalities and data structures. This makes your code more modular, easier to understand, and maintain. Complex programs can be broken down into smaller, well-defined classes, improving readability and reducing the cognitive load for programmers.

**4. Inheritance:** Classes enable inheritance, a powerful feature that allows new classes (subclasses) to inherit properties and methods from existing classes (superclasses). This promotes code reuse and simplifies the creation of specialized object types. You can build upon existing functionality without starting from scratch, leading to more efficient and maintainable code.

**5. Abstraction:** Classes facilitate abstraction, another core OOP principle. A class allows you to expose essential functionalities (methods) while hiding the underlying implementation details. This enables users to interact with objects without worrying about the internal workings, simplifying code usage and promoting maintainability.

**6. Object-Oriented Approach:** Classes are the foundation for object-oriented programming in Java. They allow you to model real-world entities and their behaviors as objects, making your code more intuitive and easier to reason about. Objects represent things like animals, accounts, or products, with their attributes and actions defined within the class.

In summary, classes are essential for writing clean, reusable, maintainable, and well-structured code in Java. They empower you to leverage the benefits of object-oriented programming, leading to efficient and scalable software development.

Section 2:

1. Write a Java program that asks the user to enter their sur name and current age then print the number of characters of their sir name and even or odd depending on their age number.

Example of Expected result:

If sir name is Saruni and age is 29, output will be;

then the number of characters is 6.

Your current age is an odd number.

import java.util.Scanner;

public class SurnameAge {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter your surname: ");

String surname = scanner.nextLine();

System.out.print("Enter your current age: ");

int age = scanner.nextInt();

System.out.println("The number of characters in your surname is " + surname.length() + ".");

System.out.println("Your current age is an " + (age % 2 == 0 ? "even" : "odd") + " number.");

}

}

1. Write Java program to ask student to enter the marks of the five units they did last semester, compute the average and display it on the screen. (Average should be given in two decimal places).

import java.util.Scanner;

public class StudentMarks {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

double totalMarks = 0.0;

// Loop to get marks for five units

for (int i = 1; i <= 5; i++) {

System.out.print("Enter the marks for unit " + i + ": ");

double marks = scanner.nextDouble();

totalMarks += marks;

}

// Calculate and format average with two decimal places

double average = totalMarks / 5.0;

average = Math.round(average \* 100.0) / 100.0; // Round to two decimal places

System.out.println("The average marks for the five units is: " + average);

}

}

1. Write a program that will help kids learn divisibly test of numbers of integers. The program should check whether the given integer is divisible by integers in the range of 0-9. For example, if a number (955) is divisible by five, the program should print, the number is divisible by 5 because it ends with a 5, and 900 is divisible by 5 because it ends with a 0(zero).

import java.util.Scanner;

public class DivisibilityTester {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter an integer: ");

int number = scanner.nextInt();

System.out.println("Divisibility Tests:");

// Divisibility by 2 (even numbers)

if (number % 2 == 0) {

System.out.println(" - Divisible by 2: Even numbers (ends in 0, 2, 4, 6, or 8).");

}

// Divisibility by 5 (ends in 0 or 5)

if (number % 5 == 0) {

System.out.println(" - Divisible by 5: Ends in 0 or 5.");

}

// Divisibility by 3 (sum of digits is divisible by 3)

int sumOfDigits = 0;

int temp = number;

while (temp > 0) {

sumOfDigits += temp % 10;

temp /= 10;

}

if (sumOfDigits % 3 == 0) {

System.out.println(" - Divisible by 3: Sum of digits is divisible by 3.");

}

// Divisibility by 9 (sum of digits is divisible by 9) - similar to 3

if (sumOfDigits % 9 == 0) {

System.out.println(" - Divisible by 9: Sum of digits is divisible by 9.");

}

// Divisibility by 4 (last two digits divisible by 4)

if (number >= 10 && number % 4 == 0) {

System.out.println(" - Divisible by 4: Last two digits are divisible by 4.");

}

// Check for divisibility by 7, 8, and 11 using pre-defined rules (more complex)

// You can add these checks if you want to cover moredivisors, but explanations might be more intricate.

}

}

1. Write a Java program to display all the multiples of 2, 3 and 7 within the range 71 to 150.

public class MultiplesInRange {

public static void main(String[] args) {

int start = 71;

int end = 150;

// Loop through numbers in the range

for (int num = start; num <= end; num++) {

// Check for divisibility by 2, 3, and 7

if (num % 2 == 0 && num % 3 == 0 && num % 7 == 0) {

System.out.print(num + " ");

}

}

System.out.println(); // Add a newline for better formatting

}

}

1. Create a calculator using java to help user perform the basic operations (+, -, \* and /).
   1. User should be asked to enter a number, then an operation, the program computes the operation and display the output to the computer screen.

import java.util.Scanner;

public class BasicCalculator {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

// Get the first number

System.out.print("Enter the first number: ");

double firstNumber = scanner.nextDouble();

// Get the operation

System.out.print("Enter the operation (+, -, \*, /): ");

char operation = scanner.next().charAt(0);

// Get the second number

System.out.print("Enter the second number: ");

double secondNumber = scanner.nextDouble();

// Perform the operation and display the result

double result;

switch (operation) {

case '+':

result = firstNumber + secondNumber;

System.out.println("Result: " + firstNumber + " + " + secondNumber + " = " + result);

break;

case '-':

result = firstNumber - secondNumber;

System.out.println("Result: " + firstNumber + " - " + secondNumber + " = " + result);

break;

case '\*':

result = firstNumber \* secondNumber;

System.out.println("Result: " + firstNumber + " \* " + secondNumber + " = " + result);

break;

case '/':

if (secondNumber == 0) {

System.out.println("Error: Division by zero is not allowed.");

} else {

result = firstNumber / secondNumber;

System.out.println("Result: " + firstNumber + " / " + secondNumber + " = " + result);

}

break;

default:

System.out.println("Invalid operation entered.");

}

}

}