**Java Assignment 1**

1. Why Java is considered as platform independent language.

* Bytecode: Java code is compiled into bytecode (.class files) that is not specific to any operating system or hardware architecture.
* Java Virtual Machine (JVM): The JVM translates bytecode into machine-specific code, allowing Java programs to run on any platform that has a JVM.
* Platform-independent libraries: Java's libraries and APIs provide a platform-independent interface to system resources, making it easy to write cross-platform code.
* No native code: Java programs do not contain native code, eliminating the need for platform-specific code.

1. Explain features of Java.

Features of the Java

* 1. Simple
     1. Easy syntax : Java's syntax is designed to be easy to read and write, with a focus on simplicity and clarity.
     2. Fewer lines of code : Compared to other languages like C++, Java requires fewer lines of code to accomplish the same task.
  2. Object Oriented
     1. Java supports the principles of object-oriented programming (OOP) like encapsulation, inheritance, and polymorphism.
     2. This allows developers to write reusable and modular code.
  3. Distributed
     1. Java has built-in support for distributed computing, making it easy to write programs that can run on multiple machines.
     2. This is achieved through Java's Remote Method Invocation (RMI) API.
  4. Interpreted
     1. Java code is compiled into an intermediate format called bytecode, which is then interpreted by the Java Virtual Machine (JVM).
     2. This allows Java code to be platform-independent and run on any machine with a JVM.
  5. Robust
     1. Java has a strong focus on reliability and robustness.
     2. The language has built-in checks for common programming errors like null pointer exceptions and array out-of-bounds errors.
  6. Secure
     1. Java has a strong focus on security, with built-in features like memory management and data type checking.
     2. The language also has a sandboxed environment for running untrusted code.
  7. Architecture neutral
     1. Java code is compiled into bytecode, which is architecture-neutral.
     2. Write Once, Run Anywhere (WORA)
     3. This means that Java code can run on any machine with a JVM, regardless of the underlying architecture.
  8. Portable
     1. Java code is portable across different platforms, including Windows, macOS, and Linux.
     2. This is achieved through the JVM, which provides a layer of abstraction between the Java code and the underlying platform.
  9. High Performance
     1. Java has a just-in-time (JIT) compiler that can dynamically compile bytecode into native machine code.
     2. This provides a significant performance boost and allows Java code to run at speeds comparable to native code.
  10. Multithread
      1. Java has built-in support for multithreading, which allows developers to write programs that can run multiple threads concurrently.
      2. This provides a significant performance boost and allows Java programs to take advantage of multi-core processors.
  11. Dynamic
      1. Java is a dynamically-typed language, which means that the data type of a variable is determined at runtime rather than at compile time.
      2. This provides greater flexibility and allows developers to write more dynamic code.

3. Explain the following terms in Java along with example:

* Keywords

Reserved words with special meanings in Java, such as if, else, while, etc.

|  |  |  |
| --- | --- | --- |
| Keyword | Description | Category |
| abstract | Abstract class or method | Access Modifier |
| assert | Assert statement | Statement |
| boolean | Boolean data type | Data Type |
| break | Break statement | Statement |
| byte | Byte data type | Data Type |
| case | Case statement | Statement |
| catch | Catch block | Exception Handling |
| char | Char data type | Data Type |
| class | Class definition | Class |
| const | Constant | Access Modifier |
| continue | Continue statement | Statement |
| default | Default statement | Statement |
| do | Do-while loop | Loop |
| double | Double data type | Data Type |
| else | Else statement | Statement |
| enum | Enum definition | Enum |
| extends | Extends keyword | Inheritance |
| final | Final keyword | Access Modifier |
| finally | Finally block | Exception Handling |
| float | Float data type | Data Type |
| for | For loop | Loop |
| goto | Goto statement (not used) | Statement |
| if | If statement | Statement |
| implements | Implements keyword | Interface |
| import | Import statement | Package |
| instanceof | Instanceof operator | Operator |
| int | Int data type | Data Type |
| interface | Interface definition | Interface |
| long | Long data type | Data Type |
| native | Native method | Access Modifier |
| new | New keyword | Operator |
| package | Package statement | Package |
| private | Private access modifier | Access Modifier |
| protected | Protected access modifier | Access Modifier |
| public | Public access modifier | Access Modifier |
| return | Return statement | Statement |
| short | Short data type | Data Type |
| static | Static keyword | Access Modifier |
| strictfp | Strictfp keyword | Access Modifier |
| super | Super keyword | Inheritance |
| switch | Switch statement | Statement |
| synchronized | Synchronized keyword | Access Modifier |
| this | This keyword | Operator |
| throw | Throw statement | Exception Handling |
| throws | Throws keyword | Exception Handling |
| transient | Transient keyword | Access Modifier |
| try | Try block | Exception Handling |
| void | Void data type | Data Type |
| volatile | Volatile keyword | Access Modifier |
| while | While loop | Loop |

* Literals

literals are values that are represented directly in the code, rather than being computed or stored in a variable. They are used to initialize variables, pass arguments to methods, and return values from method

* + Integer Literals
    - Decimal Integer Literal: A decimal integer literal is a sequence of digits without a leading zero. Examples: 123, 456, 789.
    - Hexadecimal Integer Literal: A hexadecimal integer literal is a sequence of digits preceded by 0x or 0X. Examples: 0x7B, 0XFF, 0x100.
    - Octal Integer Literal: An octal integer literal is a sequence of digits preceded by 0. Examples: 0123, 077, 0100.
    - Binary Integer Literal: A binary integer literal is a sequence of binary digits (0s and 1s) preceded by 0b or 0B. Examples: 0b1011, 0B1100, 0b1001. (Java 7 and later)
  + Floating-Point Literals
    - Decimal Floating-Point Literal: A decimal floating-point literal is a sequence of digits with a decimal point. Examples: 123.45, 456.78, 789.90.
    - Exponential Floating-Point Literal: An exponential floating-point literal is a sequence of digits with an exponent part. Examples: 1.23e-4, 4.56E+7, 7.89e+10.
  + Boolean Literals
    - True Literal: The true literal represents a boolean value of true.
    - False Literal: The false literal represents a boolean value of false.
  + Character Literals
    - Single Character Literal: A single character literal is a single character enclosed in single quotes. Examples: 'a', '1', '!'.
    - Unicode Character Literal: A Unicode character literal is a Unicode code point enclosed in single quotes and preceded by \\u. Examples: '\\u0000', '\\u0041', '\\u20AC'.
    - Escape Sequence Literal: An escape sequence literal is a sequence of characters that represents a special character. Examples:
    - \\' (single quote)
    - \\" (double quote)
    - \\\\ (backslash)
    - \\n (newline)
    - \\r (carriage return)
    - \\t (tab)
    - \\b (backspace)
    - \\f (form feed)
  + String Literals
    - Double-Quoted String Literal: A double-quoted string literal is a sequence of characters enclosed in double quotes. Examples: "Hello, World!", "This is a string.".
    - Unicode String Literal: A Unicode string literal is a sequence of Unicode code points enclosed in double quotes. Examples: "\\u0048\\u0065\\u006c\\u006c\\u006f", "\\u20AC\\u20AC\\u20AC".
  + Null Literal
    - Null Literal: The null literal represents a null reference.
* Identifiers

Names given to variables, methods, classes, interfaces, and packages in Java, such as x, calculateArea, String, etc.

1.Case-sensitive: Java identifiers are case-sensitive, meaning that "Hello" and "hello" are treated as two different identifiers.

2.Unique: Identifiers must be unique within their scope.

3.Start with a letter, underscore, or dollar sign: Identifiers can start with a letter (a-z or A-Z), an underscore (\_), or a dollar sign ($).

4.Contain letters, digits, underscores, or dollar signs: After the first character, identifiers can contain letters (a-z or A-Z), digits (0-9), underscores (\_), or dollar signs ($).

5.Cannot contain special characters: Identifiers cannot contain special characters, such as !, @, #, etc.

4.Explain operator precedence and associativity.

* + Operator Precedence

Operator precedence determines the order in which operators are evaluated when multiple operators are present in an expression. The operators with higher precedence are evaluated first.

|  |  |
| --- | --- |
| Precedence | Operators |
| 1 | Postfix: ++, -- |
| 2 | Prefix: ++, --, +, -, !, ~ |
| 3 | Multiplicative: \*, /, % |
| 4 | Additive: +, - |
| 5 | Shift: <<, >>, >>> |
| 6 | Relational: <, >, <=, >= |
| 7 | Equality: ==, != |
| 8 | Bitwise: &, ^, ` |
| 9 | Logical: &&, ` |
| 10 | Assignment: =, +=, -=, \*=, /=, %= |

* + Operator Associativity:  
      
    Operator associativity determines the direction of evaluation when multiple operators with the same precedence are present in an expression.

|  |  |
| --- | --- |
| Associativity | Operators |
| Left-associative | +, -, \*, /, %, <<, >>, >>>, &, ^, ` |
| Right-associative | =, +=, -=, \*=, /=, %= |

5.Illustrate the precedence of different operators in Java with the help of a chart.

|  |  |  |
| --- | --- | --- |
| Precedence | Operator | Description |
| 1 | ++ -- | Postfix increment/decrement |
|  | ++ -- | Prefix increment/decrement |
|  | + - | Unary plus/minus |
|  | ! ~ | Logical NOT, bitwise NOT |
| 2 | \* / % | Multiplication, division, modulus |
| 3 | + - | Addition, subtraction |
| 4 | << >> >>> | Left shift, right shift, unsigned right shift |
| 5 | < > <= >= | Relational operators |
| 6 | == != | Equality operators |
| 7 | & | Bitwise AND |
| 8 | ^ | Bitwise XOR |
| 9 | ` | ` |
| 10 | && | Logical AND |
| 11 | ` |  |
| 12 | = += -= \*= /= %= | Assignment operators |

6. Explain primitive data types in java along with their size, range and default values.

|  |  |  |  |
| --- | --- | --- | --- |
| Data Type | Size (bits) | Range | Default Value |
| byte | 8 | -128 to 127 | 0 |
| short | 16 | -32,768 to 32,767 | 0 |
| int | 32 | -2,147,483,648 to 2,147,483,647 | 0 |
| long | 64 | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 | 0L |
| float | 32 | Approximately ±3.4e+38 | 0.0f |
| double | 64 | Approximately ±1.8e+308 | 0.0d |
| char | 16 | Unicode characters (0 to 65,535) | '\u0000' |
| Boolean | 1 (technically, but implemented as a byte) | true or false | false |

7. WAP to demonstrate how to create variables of different types.

public class Variable {

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

*// Byte variable*

byte byteVar = 12;

System.out.println("Byte variable: " + byteVar);

*// Short variable*

short shortVar = 3123;

System.out.println("Short variable: " + shortVar);

*// Int variable*

int intVar = 211231231;

System.out.println("Int variable: " + intVar);

*// Long variable*

long longVar = 9213L;

System.out.println("Long variable: " + longVar);

*// Float variable*

float floatVar = 31223.1239f;

System.out.println("Float variable: " + floatVar);

*// Double variable*

double doubleVar = 1231.2312;

System.out.println("Double variable: " + doubleVar);

*// Char variable*

char charVar = 's';

System.out.println("Char variable: " + charVar);

*// Boolean variable*

boolean booleanVar = true;

System.out.println("Boolean variable: " + booleanVar);

*// String variable*

String stringVar = "Assignment of java";

System.out.println("String variable: " + stringVar);

}

}

8. Explain implicit type conversion, explicit type conversion and type promotion in Java.

* Type Conversion and casting

Type conversion is the process of changing the data type of a value from one type to another.

* + Implicit Type Conversion (Widening)

(or Java’s Automatic Conversions)

Implicit type conversion, also known as widening, is a process in Java where a smaller data type is automatically converted to a larger data type. This process is done by the compiler and does not require any explicit casting.

* + - Rules for Widening Conversion
      * byte → short → int → long → float → double
      * char → int → long → float → double
      * boolean cannot be widened to any other type.
    - Sequence for Widening/Automatic Conversion
      * byte to short: If the value is within the range of short, it is widened to short.
      * short to int: If the value is within the range of int, it is widened to int.
      * int to long: If the value is within the range of long, it is widened to long.
      * long to float: If the value is within the range of float, it is widened to float.
      * float to double: If the value is within the range of double, it is widened to double.
  + Explicit Type Conversion (Narrowing)

(or Narrow or Explicit conversion)

Explicit type conversion, also known as narrowing, is a process in Java where a larger data type is explicitly converted to a smaller data type. This process requires an explicit cast operator and can result in data loss.

* + - Rules for Explicit Typecasting
      * double → float → long → int → short → byte
      * float → long → int → short → byte
      * long → int → short → byte
      * int → short → byte
      * short → byte
      * char → byte or short
    - Sequence for Narrowing Conversion
      * double to float: If the value is within the range of float, it is narrowed to float.
      * float to long: If the value is within the range of long, it is narrowed to long.
      * long to int: If the value is within the range of int, it is narrowed to int.
      * int to short: If the value is within the range of short, it is narrowed to short.
      * short to byte: If the value is within the range of byte, it is narrowed to byte.
* Type Promotion

Type promotion is the process of automatically converting the data type of a value to a wider data type in an expression. This process is done by the compiler to ensure that the expression is evaluated correctly.

* + Rules for Type Promotion
    - Byte and Short: These types are promoted to int.
    - Int: This type is promoted to long if the other operand is long. Otherwise, it remains int.
    - Long: This type is promoted to float if the other operand is float. Otherwise, it remains long.
    - Float: This type is promoted to double if the other operand is double.
    - Double: This type remains double.
  + Type Promotion in Arithmetic Expressions

In arithmetic expressions, the following type promotions occur:

* + - Addition, Subtraction, Multiplication, and Division: The operands are promoted to the wider type.
    - Unary Plus and Unary Minus: The operand is promoted to the wider type.
  + Type Promotion in Comparison Expressions

In comparison expressions, the following type promotions occur:

* + - Equality and Inequality Operators: The operands are promoted to the wider type.
    - Relational Operators: The operands are promoted to the wider type.

9. WAP to demonstrate explicit type conversion.

class Explicit {

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

double x = 10.5;

int y = (int) x;

System.out.println("Print y:"+" "+y+" "+"Print x:"+" "+x);

}

}

10. WAP to demonstrate implicit type conversion.

class Implicit {

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

int x = 10;

double y = x;

System.out.println("Print y:"+" "+y+" "+"Print x:"+" "+x);

}

}

11. Explain arithmetic operators in java.

* Arithmetic Operators

Arithmetic operators are used to perform mathematical operations on numeric values.

* + 1. Addition: +
    2. Subtraction: -
    3. Multiplication: \*
    4. Division: /
    5. Modulus (Remainder): %
    6. Increment: ++
    7. Decrement: --
    8. Addition assignment: +=
    9. Subtraction assignment: -=
    10. Multiplication assignment: \*=
    11. Division assignment: /=
    12. Modulus assignment: %=

12. WAP to demonstrate arithmetic operators in java.

public class ArithmeticOperators {

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

int a = 5;

int b = 3;

*// Addition Operator*

int sum = a + b;

System.out.println("Addition: " + a + " + " + b + " = " + sum);

*// Subtraction Operator*

int difference = a - b;

System.out.println("Subtraction: " + a + " - " + b + " = " + difference);

*// Multiplication Operator*

int product = a \* b;

System.out.println("Multiplication: " + a + " \* " + b + " = " + product);

*// Division Operator*

int quotient = a / b;

System.out.println("Division: " + a + " / " + b + " = " + quotient);

*// Modulus Operator*

int remainder = a % b;

System.out.println("Modulus: " + a + " % " + b + " = " + remainder);

*// Unary Plus Operator*

int unaryPlus = +a;

System.out.println("Unary Plus: +" + a + " = " + unaryPlus);

*// Unary Minus Operator*

int unaryMinus = -a;

System.out.println("Unary Minus: -" + a + " = " + unaryMinus);

*// Increment Operator (prefix)*

int prefixIncrement = ++a;

System.out.println("Prefix Increment: ++" + a + " = " + prefixIncrement);

*// Increment Operator (postfix)*

a = 5; *// reset a*

int postfixIncrement = a++;

System.out.println("Postfix Increment: " + a + "++ = " + postfixIncrement);

*// Decrement Operator (prefix)*

int prefixDecrement = --a;

System.out.println("Prefix Decrement: --" + a + " = " + prefixDecrement);

*// Decrement Operator (postfix)*

a = 5; *// reset a*

int postfixDecrement = a--;

System.out.println("Postfix Decrement: " + a + "-- = " + postfixDecrement);

}

}

13. WAP to demonstrate increment (pre and post) and decrement (pre and post) operators.

class IncrementDecrement {

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

int a = 10;

int b = 10;

System.out.println("Pre-increment operator:");

System.out.println("a = " + a);

System.out.println("++a = " + ++a);

System.out.println("a = " + a);

System.out.println("\nPost-increment operator:");

System.out.println("b = " + b);

System.out.println("b++ = " + b++);

System.out.println("b = " + b);

System.out.println("\nPre-decrement operator:");

System.out.println("a = " + a);

System.out.println("--a = " + --a);

System.out.println("a = " + a);

System.out.println("\nPost-decrement operator:");

System.out.println("b = " + b);

System.out.println("b-- = " + b--);

System.out.println("b = " + b);

}

}

14. WAP to demonstrate compound assignment operator.

public class CompoundAssignmentOperators {

public static void main(String[] args) {

int a = 10;

int b = 2;

System.out.println("Arithmetic Compound Assignment:");

System.out.println("a = " + a + ", b = " + b);

a += b; // a = a + b

System.out.println("a += b: " + a);

a -= b; // a = a - b

System.out.println("a -= b: " + a);

a \*= b; // a = a \* b

System.out.println("a \*= b: " + a);

a /= b; // a = a / b

System.out.println("a /= b: " + a);

a %= b; // a = a % b

System.out.println("a %= b: " + a);

int c = 5;

int d = 3;

System.out.println("\\nBitwise Compound Assignment:");

System.out.println("c = " + c + ", d = " + d);

c &= d; // c = c & d

System.out.println("c &= d: " + c);

c |= d; // c = c | d

System.out.println("c |= d: " + c);

c ^= d; // c = c ^ d

System.out.println("c ^= d: " + c);

int e = 10;

System.out.println("\\nShift Compound Assignment:");

System.out.println("e = " + e);

e <<= 2; // e = e << 2

System.out.println("e <<= 2: " + e);

e >>= 1; // e = e >> 1

System.out.println("e >>= 1: " + e);

e >>>= 1; // e = e >>> 1

System.out.println("e >>>= 1: " + e);

}

}

15. Explain relational operators in java.   
  
Relational Operators:

Relational operators are used to compare two values and return a boolean result (true or false).

1. Equal To (==)

* Syntax: a == b
* Description: Returns true if a is equal to b, false otherwise.

2. Not Equal To (!=)

* Syntax: a != b
* Description: Returns true if a is not equal to b, false otherwise.

3. Greater Than (>

* Syntax: a > b
* Description: Returns true if a is greater than b, false otherwise.

4. Less Than (<)

* Syntax: a < b
* Description: Returns true if a is less than b, false otherwise.

5. Greater Than or Equal To (>=)

* Syntax: a >= b
* Description: Returns true if a is greater than or equal to b, false otherwise.

6. Less Than or Equal To (<=)

* Syntax: a <= b
* Description: Returns true if a is less than or equal to b, false otherwise

16. WAP to demonstrate relational operators in java.

class RelationalOperators {

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

int a = 5;

int b = 3;

System.out.println("a == b: " + (a == b));

System.out.println("a != b: " + (a != b));

System.out.println("a > b: " + (a > b));

System.out.println("a < b: " + (a < b));

System.out.println("a >= b: " + (a >= b));

System.out.println("a <= b: " + (a <= b));

}

}

17. Explain logical operators in Java.

Logical Operators:

Logical operators are used to combine conditional statements and return a boolean result (true or false). They are commonly used in conditional statements and loops.

1. Logical AND (&&)

* Syntax: a && b
* Description: Returns true if both a and b are true, false otherwise.

2. Logical OR (||)

* Syntax: a || b
* Description: Returns true if either a or b is true, false otherwise.

3. Logical NOT (!)

* Syntax: !a
* Description: Returns true if a is false, false otherwise.

18. WAP to demonstrate logical operators in java.

class LogicalOperator {

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

boolean a = true;

boolean b = false;

System.out.println("Logical AND:");

System.out.println(a + " && " + a + " = " + (a && a));

System.out.println(a + " && " + b + " = " + (a && b));

System.out.println(b + " && " + a + " = " + (b && a));

System.out.println(b + " && " + b + " = " + (b && b));

System.out.println("\nLogical OR:");

System.out.println(a + " || " + a + " = " + (a || a));

System.out.println(a + " || " + b + " = " + (a || b));

System.out.println(b + " || " + a + " = " + (b || a));

System.out.println(b + " || " + b + " = " + (b || b));

System.out.println("\nLogical NOT:");

System.out.println("! " + a + " = " + !a);

System.out.println("! " + b + " = " + !b);

}}

19. Explain short circuit operators in Java.

Short-Circuit Logical Operators:

Java also provides short-circuit logical operators that evaluate the second operand only if necessary.

1. Short-Circuit AND (&&)

* Syntax: a && b
* Description: Evaluates b only if a is true.

2. Short-Circuit OR (||)

* Syntax: a || b
* Description: Evaluates b only if a is false.

20. WAP to demonstrate short circuit operators.   
  
public class ShortCircuitOperators {

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

*// Short-circuit AND (&&)*

System.out.println("Short-circuit AND (&&):");

boolean a = true;

boolean b = false;

System.out.println(a + " && " + b + ": " + (a && b));

a = false;

b = true;

System.out.println(a + " && " + b + ": " + (a && b));

*// Short-circuit OR (||)*

System.out.println("\nShort-circuit OR (||):");

a = true;

b = false;

System.out.println(a + " || " + b + ": " + (a || b));

a = false;

b = true;

System.out.println(a + " || " + b + ": " + (a || b));

}

}

21. Explain conditional operator along with syntax and example.

* Ternary (Conditional) Operator
  + Conditional expression: condition ? value\_if\_true : value\_if\_false

public class ConditionalOperator {

public static void main(String[] args) {

int x = 10;

int y = 20;

// Using conditional operator to find the maximum value

int max = (x > y) ? x : y;

System.out.println("Maximum value: " + max);

}

}  
  
the conditional operator (x > y) ? x : y evaluates the boolean expression x > y. If the expression is true, it returns the value of x; otherwise, it returns the value of y. The result is then assigned to the variable max.

22. WAP to find greater of two numbers using conditional operator.

public class GreaterNumber {

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

int num1 = 10;

int num2 = 20;

int greater = (num1 > num2) ? num1 : num2;

System.out.println("Greater number: " + greater);

}

}

23. Explain conditional statements (if, if else, nested if else, else if ladder, switch case)

Conditional Statements

Conditional statements are used to execute different blocks of code based on certain conditions or decisions. Java provides several types of conditional statements:

1. If Statement

The if statement is used to execute a block of code if a certain condition is true.

Syntax:

if (condition) {

// code to be executed

}

2.If-Else Statement

The if-else statement is used to execute one block of code if a certain condition is true, and another block of code if the condition is false.

Syntax:

if (condition) {

// code to be executed if condition is true

} else {

// code to be executed if condition is false

}

1. Nested If-Else Statement

The nested if-else statement is used to execute different blocks of code based on multiple conditions.

Syntax:

if (condition1) {

if (condition2) {

// code to be executed if condition1 and condition2 are true

} else {

// code to be executed if condition1 is true and condition2 is false

}

} else {

// code to be executed if condition1 is false

}

1. Else-If Ladder

The else-if ladder is used to execute different blocks of code based on multiple conditions.

Syntax:

if (condition1) {

// code to be executed if condition1 is true

} else if (condition2) {

// code to be executed if condition1 is false and condition2 is true

} else if (condition3) {

// code to be executed if condition1 and condition2 are false and condition3 is true

} else {

// code to be executed if all conditions are false

}

1. Switch Case Statement

The switch case statement is used to execute different blocks of code based on the value of a variable.

Syntax:

switch (variable) {

case value1:

// code to be executed if variable is equal to value1

break;

case value2:

// code to be executed if variable is equal to value2

break;

default:

// code to be executed if variable is not equal to any of the values

break;

}

24. WAP to find greater of two numbers using if else statement.

public class GreaterNumberIfElse {

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

int num1 = 10;

int num2 = 20;

if (num1 > num2) {

System.out.println(num1 + " is greater than " + num2);

} else if (num2 > num1) {

System.out.println(num2 + " is greater than " + num1);

} else {

System.out.println("Both numbers are equal");

}

}

}

25. WAP to find whether the inputted number is even or odd.  
  
public class EvenOrOdd {

public static void main(String[] args) {

int num = 10;

if (num % 2 == 0) {

System.out.println(num + " is even");

} else {

System.out.println(num + " is odd");

}

}

}

26. WAP to find greatest among three numbers using if else.

public class GreatestAmongThree {

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

int num1 = 10;

int num2 = 20;

int num3 = 30;

if (num1 > num2 && num1 > num3) {

System.out.println(num1 + " is the greatest");

} else if (num2 > num1 && num2 > num3) {

System.out.println(num2 + " is the greatest");

} else {

System.out.println(num3 + " is the greatest");

}

}

}

27. Explain how to use Scanner class for user input. Discuss

different methods for taking user input of Scanner class. WAP to demonstrate it.

Using the Scanner Class for User Input

The Scanner class is a simple text scanner that can parse primitive types and strings using regular expressions. It is a useful tool for getting user input in Java.

Creating a Scanner Object

To use the Scanner class, you need to create a Scanner object and pass the input source to its constructor. The most common input source is System.in, which represents the keyboard.

Scanner scanner = new Scanner(System.in);

Methods for Taking User Input

The Scanner class provides several methods for taking user input:

1. next():

This method reads the next token (word or number) from the input source.

String name = scanner.next();

2. nextLine():

This method reads the entire line from the input source.

String address = scanner.nextLine();

3. nextInt():

This method reads the next integer from the input source.

int age = scanner.nextInt();

4. nextDouble():

This method reads the next double-precision floating-point number from the input source.

double height = scanner.nextDouble();

5. nextBoolean():

This method reads the next boolean value from the input source.

boolean isAdmin = scanner.nextBoolean();

Example Program

import java.util.Scanner;

public class ScannerEx {

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

Scanner scanner = new Scanner(System.in);

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

String name = scanner.next();

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

int age = scanner.nextInt();

System.out.print("Enter your height (in meters): ");

double height = scanner.nextDouble();

System.out.print("Are you an admin? (true/false): ");

boolean isAdmin = scanner.nextBoolean();

System.out.println("Name: " + name);

System.out.println("Age: " + age);

System.out.println("Height: " + height + " meters");

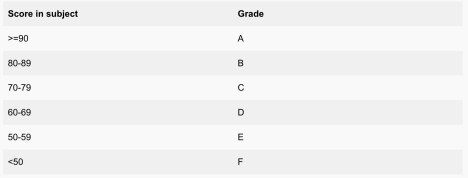
System.out.println("Admin: " + isAdmin);

scanner.close();

}

}

28. Write a Java Program Find out Students Grades using Switch Case



public class StudentGrade {

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

int[] marks = {95, 88, 75, 62, 58};

int totalMarks = 0;

for (int mark : marks) {

totalMarks += mark;

}

double percentage = (totalMarks / (double) (marks.length \* 100)) \* 100;

System.out.println("Marks in 5 subjects: ");

for (int i = 0; i < marks.length; i++) {

System.out.println("Subject " + (i + 1) + ": " + marks[i]);

}

System.out.println("Total Marks: " + totalMarks);

System.out.println("Percentage: " + percentage);

char grade = getGrade(percentage);

System.out.println("Grade: " + grade);

}

public static char getGrade(double *percentage*) {

int gradeCode = (int) (*percentage* / 10);

switch (gradeCode) {

case 10:

case 9:

return 'A';

case 8:

return 'B';

case 7:

return 'C';

case 6:

return 'D';

case 5:

return 'E';

default:

return 'F';

}

}

}

29. WAP to check whether the inputted character is Vowel or Consonant

import java.util.Scanner;

public class VowelOrConsonant {

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

Scanner scanner = new Scanner(System.in);

System.out.print("Enter a character: ");

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

boolean isVowel = isVowel(ch);

if (isVowel) {

System.out.println(ch + " is a vowel");

} else if (Character.isLetter(ch)) {

System.out.println(ch + " is a consonant");

} else {

System.out.println(ch + " is not a letter");

}

scanner.close();

}

public static boolean isVowel(char *ch*) {

char[] vowels = {'a', 'e', 'i', 'o', 'u', 'A', 'E', 'I', 'O', 'U'};

for (int i = 0; i < vowels.length; i++) {

if (*ch* == vowels[i]) {

return true;

}

}

return false;

}

}