Assignment 2 Java

1. Why Java is considered a platform-independent language?

Java is considered platform-independent because it uses the **“Write Once, Run Anywhere” (WORA)** approach. Java code is compiled into **bytecode** by the Java Compiler, which is not specific to any platform. The bytecode is executed by the **Java Virtual Machine (JVM)**, which is platform-specific. Since every platform has its own JVM, the same bytecode can run on different platforms without modification.

2. Features of Java

1. **Platform Independent**: Uses bytecode, which can run on any platform with a JVM.
2. **Object-Oriented**: Supports concepts like inheritance, polymorphism, encapsulation, and abstraction.
3. **Simple**: Easy to learn with a clean syntax.
4. **Secure**: Provides built-in security features like bytecode verification and access control.
5. **Portable**: Code is portable across platforms as it doesn’t rely on hardware-specific features.
6. **Robust**: Handles runtime errors with strong exception handling and memory management.
7. **Multithreaded**: Allows concurrent execution of programs, improving performance.
8. **High Performance**: Uses Just-In-Time (JIT) compiler for optimized performance.
9. **Dynamic**: Supports dynamic loading of classes and linking.
10. **Distributed**: Facilitates building distributed applications using tools like RMI and EJB.

3. Explanation of Terms in Java

i. Keyword

- **Definition**: Reserved words in Java that have a predefined meaning and cannot be used as identifiers.
- **Example**:

```
public class Example {
public static void main(String[] args) {
int number = 10;  // 'int' and 'public' are keywords
}
}
```

ii. Identifier

- **Definition**: The name used to identify variables, methods, classes, or objects in Java. Identifiers must begin with a letter, underscore _, or dollar sign $.
- **Example**:

```
public class Example {  // 'Example' is an identifier for the class name
public static void main(String[] args) {
int number = 10;  // 'number' is an identifier for the variable
}
}
```

iii. Literals

- **Definition**: Constant values assigned to variables in Java. These include numbers, characters, strings, and boolean values.
- **Types**: Integer, Floating-point, Character, String, Boolean.
- **Example**:

```
public class Example {
public static void main(String[] args) {
int number = 10;          // Integer literal
double price = 99.99;     // Floating-point literal
char grade = 'A';         // Character literal
String name = "Java";     // String literal
boolean isJavaFun = true; // Boolean literal
}
```

```
}
```

4. Operator Precedence and Associativity

Operator Precedence: It determines the order in which operators are evaluated in an expression. Operators with higher precedence are evaluated before operators with lower precedence.

Associativity: When two operators of the same precedence appear in an expression, associativity determines the direction (left-to-right or right-to-left) in which the operators are evaluated.

- **Left-to-Right Associativity**: Most operators like +, -, *, / are evaluated from left to right.
- **Right-to-Left Associativity**: Assignment operators (=, +=, etc.) and some unary operators are evaluated from right to left.

Example:

```
int result = 10 + 5 * 2;  // Multiplication (*) is evaluated before addition (+)
System.out.println(result);  // Output: 20
int value = 5;
value = value + 2 * 3;  // * is evaluated first, then +
System.out.println(value);  // Output: 11
```

5. Precedence of Different Operators in Java (Chart)

Precedence Level Operator Associativity

Here's the operator precedence chart in an easy-to-copy format:

Precedence Level	Operator	Associativity
1 (Highest)	(), [], . (Parentheses, Array access, Member access)	Left-to-right
2	++, -- (Post-increment/decrement)	Left-to-right
3	++, -- (Pre-increment/decrement), +, - (Unary), ! (Logical NOT), ~ (Bitwise Complement)	Right-to-left
4	*, /, % (Multiplication, Division, Modulus)	Left-to-right

5	**+, - (Addition, Subtraction)**	**Left-to-right**
6	**<<, >>, >>> (Shift operators)**	**Left-to-right**
7	**<, >, <=, >=, instanceof**	**Left-to-right**
8	**==, != (Equality, Inequality)**	**Left-to-right**
9	**& (Bitwise AND)**	**Left-to-right**
10	**^ (Bitwise XOR)**	**Left-to-right**
11	**| (Bitwise OR)**	**Left-to-right**
12	**&& (Logical AND)**	**Left-to-right**
13	**|| (Logical OR)**	**Left-to-right**
14	**?: (Ternary Conditional)**	**Right-to-left**
15 (Lowest)	**=, +=, -=, *=, /=, etc. (Assignment operators)**	**Right-to-left**

6. Primitive Data Types in Java

Primitive data types are the building blocks for data manipulation in Java. These are predefined by the language and represent simple values.

Data Type	Size	Range	Default Value
byte	1 byte	-128 to 127	0
short	2 bytes	-32,768 to 32,767	0
int	4 bytes	-2,147,483,648 to 2,147,483,647	0
long	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	
float	4 bytes	Approximately ±3.40282347E+38F	0.0f
Double	8 bytes	Approximately ±1.79769313486231570E+308	0.0d
char	2 bytes	0 to 65,535 (Unicode)	‘\u0000’
Boolean	1 bit	true or false	false

Example:

```
public class PrimitiveDataTypes {
public static void main(String[] args) {
byte b = 100;          // Example of byte
short s = 1000;        // Example of short
int i = 10000;         // Example of int
long l = 100000L;      // Example of long
float f = 10.5f;       // Example of float
double d = 10.555;     // Example of double
char c = 'A';          // Example of char
boolean bool = true;   // Example of boolean
System.out.println("byte: " + b);
System.out.println("short: " + s);
System.out.println("int: " + i);
System.out.println("long: " + l);
System.out.println("float: " + f);
System.out.println("double: " + d);
System.out.println("char: " + c);
System.out.println("boolean: " + bool);
}
}
```

8. Implicit Type Conversion, Explicit Type Conversion, and Type Promotion in Java

In Java, type conversion refers to converting a value from one data type to another. This can happen automatically (implicit) or manually (explicit). Additionally, type promotion occurs when smaller data types are automatically converted to larger ones in certain operations. Here's a detailed explanation:

1. Implicit Type Conversion (Widening Conversion)

- **Definition**: Implicit type conversion occurs automatically when Java converts a smaller data type to a larger data type. It is also called **widening conversion**.
- **Why it happens**: Java does this automatically to avoid data loss when converting between data types that can store a larger range of values.
- **When it occurs**: It happens when you assign a smaller data type (e.g., byte, short, char) to a larger data type (e.g., int, long, float, double), or in

arithmetic operations.

Example:

```
int num = 10;      // int is a 32-bit data type
double result = num;    // Implicit conversion from int to double (widening)
System.out.println(result);    // Output: 10.0
```

- Here, the int value 10 is automatically converted to a double without any explicit cast.

Key Points:

- The conversion is safe as there's no risk of losing information.
- Smaller types like byte, short, and char are promoted to int during arithmetic operations.

2. Explicit Type Conversion (Narrowing Conversion)

- **Definition**: Explicit type conversion, also called **narrowing conversion** or **type casting**, happens when you manually convert a larger data type to a smaller one.
- **Why it happens**: It is done by the programmer when they want to assign a larger data type (e.g., double, float) to a smaller data type (e.g., int, byte). This requires manual intervention because there is a risk of data loss.
- **How it's done**: You use a **cast operator** to explicitly convert a larger type to a smaller type.

Example:

```
double num = 9.8;     // double type has more precision
int result = (int) num;    // Explicit conversion (casting) from double to int
System.out.println(result);    // Output: 9
```

- In this example, the double value 9.8 is cast to int, which results in truncating the decimal part and returning 9.

Key Points:

- Data loss can occur during explicit casting (e.g., truncating a decimal point when converting from double to int).
- You must use the **cast operator** (type) to convert from one type to another.

3. Type Promotion (Automatic Conversion in Operations)

- **Definition**: Type promotion happens automatically in expressions where different data types are involved. Smaller data types are promoted to larger data types to ensure that the operation is performed accurately without losing data.
- **Why it happens**: Java promotes smaller types like byte, short, and char to int to handle operations in a consistent way, since int is the default type for arithmetic calculations.

Promotion Order: The promotion follows this hierarchy:

- **byte → short → int → long → float → double**

Example:

```
byte a = 10;
byte b = 20;
int result = a + b;    // Both bytes are promoted to int during the addition
System.out.println(result);    // Output: 30
```

- Here, both byte variables a and b are promoted to int during the addition because int is the default type for arithmetic operations.

If one of the operands is a long, float, or double, the result is promoted to the highest type in the operation.

Example:

```
int a = 10;
double b = 5.5;
double result = a + b;    // 'int' is promoted to 'double' during addition
System.out.println(result);    // Output: 15.5
```

- In this case, the int value a is promoted to double to match the double operand b, and the result is a double.

Key Points:

- Type promotion is automatic and happens when there is a mix of different types in an expression.
- The result of an operation will always be of the highest type involved in the operation.

Summary of Differences:

- **Implicit Type Conversion**: Automatic conversion from a smaller type to a larger type. No need for explicit casting, and no loss of data. Example: int to double.

- **Explicit Type Conversion**: Manual conversion from a larger type to a smaller type. Requires explicit casting, and there's potential for data loss. Example: double to int.
- **Type Promotion**: Automatic conversion during arithmetic operations where smaller types (byte, short, char) are promoted to int, and larger types (long, float, double) may promote the result to a higher type.

9. Arithmetic Operators in Java

Arithmetic operators are used to perform basic mathematical operations.

- **Addition (+)**: Adds two numbers.
- **Subtraction (-)**: Subtracts one number from another.
- **Multiplication (*)**: Multiplies two numbers.
- **Division (/)**: Divides one number by another. In integer division, any fractional part is discarded.
- **Modulus (%)**: Returns the remainder of a division operation.

10. Relational Operators in Java

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

- **Equal to (==)**: Checks if two values are equal.
- **Not equal to (!=)**: Checks if two values are not equal.
- **Greater than (>)**: Checks if one value is greater than another.
- **Less than (<)**: Checks if one value is less than another.
- **Greater than or equal to (>=)**: Checks if one value is greater than or equal to another.
- **Less than or equal to (<=)**: Checks if one value is less than or equal to another.

11. Logical Operators in Java

Logical operators are used to combine multiple conditions (boolean expressions) or evaluate logical values. They operate on boolean values and return a boolean result.

1. **Logical AND (&&)**:

- Returns true if **both** operands are true.
- If either operand is false, the result is false.
- Example:

```
boolean a = true, b = false;
System.out.println(a && b);  // Output: false
```

2. **Logical OR (||)**:

- Returns true if **at least one** of the operands is true.
- If both operands are false, the result is false.
- Example:

```
boolean a = true, b = false;
System.out.println(a || b);  // Output: true
```

3. **Logical NOT (!)**:

- Reverses the value of a boolean.
- If the operand is true, it returns false, and vice versa.
- Example:

```
boolean a = true;
System.out.println(!a);  // Output: false
```

12. Short-Circuit Operators in Java

Short-circuit operators are a special type of logical operators (&& and ||) that stop evaluating the expression as soon as the result is determined.

1. **Short-circuit AND (&&)**:

- If the first operand is false, the second operand is not evaluated because the overall result will always be false.
- Example:

```
int x = 10, y = 5;
if (x < y && x / y > 1) {  // Second condition won't be evaluated
System.out.println("Condition is true");
```

```
} else {
System.out.println("Condition is false");  // Output: Condition is false
}
```

2. **Short-circuit OR (||)**:

- If the first operand is true, the second operand is not evaluated because the overall result will always be true.
- Example:

```
int x = 10, y = 5;
if (x > y || x / y > 1) {  // Second condition won't be evaluated
System.out.println("Condition is true");  // Output: Condition is true
}
```

Key Benefits of Short-circuit Operators:

- They improve performance by avoiding unnecessary evaluations.
- Prevent errors like division by zero (since the second operand isn't evaluated if the first is sufficient to determine the result).

13. Conditional Operator in Java

The **conditional operator** (also known as the **ternary operator**) is used to evaluate a condition and return one of two values based on whether the condition is true or false. It is the only ternary operator in Java because it operates on three operands.

```
Syntax:
condition ? value_if_true : value_if_false;
```

How it works:

- If the condition evaluates to true, the operator returns value_if_true.
- If the condition evaluates to false, the operator returns value_if_false.

Example:

```
int a = 10, b = 20;
int max = (a > b) ? a : b;  // If a > b, max = a; otherwise, max = b
System.out.println("Maximum: " + max);  // Output: Maximum: 20
```

Key Points:

- It is a shorthand for if-else statements.
- It is often used for concise conditional assignments.

14. Conditional Statements in Java

Conditional statements are used to execute specific blocks of code based on certain conditions.

1. if Statement

- Executes a block of code **only if the condition is true**.

- **Syntax** :

```
if (condition) {
// Code to execute if condition is true
}
```

- **Example** :

```
int num = 10;
if (num > 5) {
System.out.println("The number is greater than 5");
}
```

2. if-else Statement

- Executes one block of code if the condition is true and another if the condition is false.

- **Syntax** :

```
if (condition) {
// Code to execute if condition is true
} else {
// Code to execute if condition is false
}
```

- **Example** :

```
int num = 10;
if (num % 2 == 0) {
System.out.println("Even number");
} else {
System.out.println("Odd number");
}
```

3. Nested if-else Statement

- An if or if-else statement inside another if or if-else.

- **Syntax** :

```
if (condition1) {
```

```
if (condition2) {
// Code to execute if both conditions are true
} else {
// Code to execute if condition1 is true but condition2 is false
}
} else {
// Code to execute if condition1 is false
}
```

- **Example**:

```
int num = 20;
if (num > 0) {
if (num % 2 == 0) {
System.out.println("Positive Even number");
} else {
System.out.println("Positive Odd number");
}
} else {
System.out.println("Negative number");
}
```

4. else-if Ladder

- Used to check multiple conditions sequentially.
- **Syntax**:

```
if (condition1) {
// Code if condition1 is true
} else if (condition2) {
// Code if condition2 is true
} else {
// Code if none of the conditions are true
}
```

- **Example**:

```
int marks = 85;
if (marks >= 90) {
System.out.println("Grade A");
} else if (marks >= 75) {
System.out.println("Grade B");
} else if (marks >= 50) {
```

```
System.out.println("Grade C");
} else {
System.out.println("Fail");
}
```

5. switch Case Statement

- Used to execute one block of code out of many options.
- **Syntax**:

```
switch (expression) {
case value1:
// Code for value1
break;
case value2:
// Code for value2
break;
...
default:
// Code if no cases match
}
```

- **Example**:

```
int day = 3;
switch (day) {
case 1:
System.out.println("Monday");
break;
case 2:
System.out.println("Tuesday");
break;
case 3:
System.out.println("Wednesday");
break;
default:
System.out.println("Invalid day");
}
```

15. Using Scanner Class for User Input

The Scanner **class** is part of the java.util package and is used to take input from the user.

Steps to Use Scanner Class:

1. Import the Scanner class:

```
import java.util.Scanner;
```

2. Create a Scanner object:

```
Scanner scanner = new Scanner(System.in);
```

3. Use methods of the Scanner class to take input.

Methods of Scanner Class for User Input:

1. nextInt(): Reads an integer.
2. nextFloat(): Reads a floating-point number.
3. nextDouble(): Reads a double-precision number.
4. next(): Reads a single word (string).
5. nextLine(): Reads an entire line (string).
6. nextBoolean(): Reads a boolean value (true or false).

Example Program: Using Scanner to Take Input

```
import java.util.Scanner;
public class UserInputDemo {
public static void main(String[] args) {
// Create a Scanner object
Scanner scanner = new Scanner(System.in);
// Taking integer input
System.out.print("Enter an integer: ");
int num = scanner.nextInt();
// Taking float input
System.out.print("Enter a float: ");
float decimal = scanner.nextFloat();
// Taking string input
System.out.print("Enter your name: ");
scanner.nextLine(); // Consume the leftover newline character
String name = scanner.nextLine();
// Taking boolean input
System.out.print("Are you a student? (true/false): ");
boolean isStudent = scanner.nextBoolean();
```

```
// Printing the inputs
System.out.println("\nHere is what you entered:");
System.out.println("Integer: " + num);
System.out.println("Float: " + decimal);
System.out.println("Name: " + name);
System.out.println("Student: " + isStudent);
// Close the Scanner object
scanner.close();
}
}
```

Explanation of the Code:

1. nextInt() reads an integer value.
2. nextFloat() reads a float value.
3. nextLine() is used to read a full line (after nextInt() or nextFloat(), it is needed to consume the leftover newline character).
4. nextBoolean() reads a boolean value (true or false).

This program demonstrates how to take multiple types of input from the user and display them. Let me know if you need further clarification!