Lecture 2: Introduction to Java Programming

1. Getting Started with Java

Java is one of the most popular programming languages for building cross-platform applications, known for its reliability, portability, and scalability. It runs on the **Java Virtual Machine (JVM)**, which makes it platform-independent—write once, run anywhere. In this section, we'll cover how to set up a development environment and write your first Java program.

Setting Up the Development Environment

To begin programming in Java, you'll need to:

- Install the **Java Development Kit (JDK)**.
- Use an Integrated Development Environment (IDE) such as **IntelliJ IDEA**.

Creating Your First Java Project in IntelliJ IDEA

- **Steps**:
    1. Open IntelliJ IDEA.
    2. Select "Create New Project".
    3. Choose the Java SDK and create a new class file.

Writing and Running a Hello World Program

- The **Hello World** program is traditionally the first program every beginner writes.

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

- **Explanation**:
    - `public class Main`: Defines the class named `Main`.
    - `public static void main(String[] args)`: The `main` method is the entry point for Java applications.
    - `System.out.println()`: Prints output to the console.

---

2. Declaring Your First Class

A **class** in Java is a blueprint from which individual objects are created. It can contain fields (attributes) and methods (behavior).

Structure of a Java Class

- **Syntax**:

```
public class Car {
    private String model;  // Field
    public void start() {  // Method
        System.out.println("Car is starting.");
    }
}
```

- **Important Keywords**:
  - `public`: Access modifier allowing the class or method to be accessed from other classes.
  - `private`: Restricts access to members of the class.
  - `static`: Allows methods or variables to be called without creating an object.

Creating a Class in Java

- Define the class, declare variables and methods inside the class. Use **access modifiers** to control visibility.

---

3. The `main` Method

The **`main` method** serves as the starting point of any Java program. Without this method, a Java application cannot run.

Syntax of the `main` Method

```
public static void main(String[] args) {
    // code to be executed
}
```

- `public`: The method is accessible from anywhere.
- `static`: It belongs to the class, not instances of the class.
- `void`: It does not return any value.
- `String[] args`: The program can accept arguments from the command line.

---

4. Code Blocks and Statements

A **code block** in Java is a group of one or more statements enclosed in curly braces `{}`.

Example of Code Block

```
if (age >= 18) {
    System.out.println("You are an adult.");
}
```

Here, the code inside the `{}` braces is a **code block** that runs only when the condition (`age >= 18`) is true.

- **Statements** are complete instructions in Java (e.g., method calls or variable declarations).

---

5. Variables in Java

Variables in Java are used to store data values. Every variable has:

- **Type**: Defines what kind of data it stores.
- **Name**: Identifier for the variable.
- **Value**: The actual data stored.

Types of Variables:

1. **Local Variables**: Declared inside methods or blocks and only accessible there.
2. **Instance Variables**: Declared inside a class but outside methods, each object gets its own copy.
3. **Class Variables**: Declared with the `static` keyword inside a class, shared among all instances.

---

6. Primitive Data Types

Java provides **8 primitive data types**:

1. **byte** (1 byte) – Stores whole numbers from -128 to 127.
2. **short** (2 bytes) – Stores whole numbers from -32,768 to 32,767.
3. **int** (4 bytes) – Stores whole numbers from -2^31 to 2^31-1.
4. **long** (8 bytes) – Stores large whole numbers.
5. **float** (4 bytes) – Stores fractional numbers, single precision.
6. **double** (8 bytes) – Stores fractional numbers, double precision.
7. **char** (2 bytes) – Stores a single character using Unicode.
8. **boolean** (1 bit) – Stores `true` or `false`.

Example:

```
int age = 25;
char grade = 'A';
boolean isJavaFun = true;
```

---

7. Wrapper Classes

Wrapper classes are used to convert primitive data types into objects. They provide utility methods and constants for the primitive types.

The `Integer` class provides constants to represent the **maximum** and **minimum** values that an `int` data type can hold. This is useful when you need to check for boundaries or handle potential overflow/underflow conditions.

Example Program: Using Integer.MAX_VALUE and Integer.MIN_VALUE

```
public class IntegerLimitsExample {
    public static void main(String[] args) {
        // Displaying the maximum value an int can hold
        System.out.println("Maximum value of int: " + Integer.MAX_VALUE);

        // Displaying the minimum value an int can hold
        System.out.println("Minimum value of int: " + Integer.MIN_VALUE);

        // Example usage: Checking boundary conditions
        int max = Integer.MAX_VALUE;
        int min = Integer.MIN_VALUE;

        System.out.println("Is " + max + " the maximum int value? " + (max ==
Integer.MAX_VALUE));
        System.out.println("Is " + min + " the minimum int value? " + (min ==
Integer.MIN_VALUE));
    }
}
```

Explanation:

- `Integer.MAX_VALUE` gives the largest integer value Java can represent with the `int` type (`2^31 - 1`).
- `Integer.MIN_VALUE` gives the smallest integer value (`-2^31`).
- This example prints these values and shows how they can be compared.

Section 8: Integer Overflow and Underflow

Integer overflow occurs when a calculation exceeds the maximum value an integer can store, causing it to wrap around to the minimum value. **Underflow** happens when it drops below the minimum and wraps around to the maximum.

Example Program: Demonstrating Integer Overflow and Underflow

```
public class OverflowUnderflowExample {
    public static void main(String[] args) {
        // Integer overflow: exceeding Integer.MAX_VALUE
        int maxInt = Integer.MAX_VALUE;
        System.out.println("Max int value: " + maxInt);
        int overflowInt = maxInt + 1;  // This causes overflow
        System.out.println("After overflow: " + overflowInt);  // Wraps to
Integer.MIN_VALUE

        // Integer underflow: dropping below Integer.MIN_VALUE
        int minInt = Integer.MIN_VALUE;
        System.out.println("Min int value: " + minInt);
        int underflowInt = minInt - 1;  // This causes underflow
        System.out.println("After underflow: " + underflowInt);  // Wraps to
Integer.MAX_VALUE
    }
}
```

Explanation:

- Adding 1 to `Integer.MAX_VALUE` causes the integer to overflow and wrap around to `Integer.MIN_VALUE`.
- Subtracting 1 from `Integer.MIN_VALUE` causes underflow and wraps around to `Integer.MAX_VALUE`.

9. Type Casting

In Java, **type casting** is the process of converting one data type into another. It can happen either **implicitly** (automatic conversion by the compiler) or **explicitly** (manually by the programmer). Type casting is particularly important when performing operations between different data types or when precision needs to be controlled.

Types of Type Casting

1. Implicit Type Casting (Widening)

- This occurs automatically when you convert a smaller data type to a larger one, as there is no risk of losing information.
- **Example**: Converting an `int` to a `double`.

Example: Implicit Casting

```
public class ImplicitCastingExample {
    public static void main(String[] args) {
        int myInt = 100;  // int is a smaller data type
        double myDouble = myInt;  // Automatic conversion (widening)

        System.out.println("Integer value: " + myInt);  // Outputs: 100
        System.out.println("Double value after implicit casting: " + myDouble);
// Outputs: 100.0
    }
}
```

Explanation:

- In the example, the `int` value `100` is implicitly cast to `double`. There is no need for explicit casting because Java handles the conversion automatically.

2. Explicit Type Casting (Narrowing)

- **Explicit casting** is required when converting a larger data type to a smaller one, as there might be a loss of precision or data.
- **Example**: Converting a `double` to an `int`.

Example: Explicit Casting

```
public class ExplicitCastingExample {
    public static void main(String[] args) {
        double myDouble = 9.78;  // double is a larger data type
        int myInt = (int) myDouble;  // Manual casting: double to int
(narrowing)

        System.out.println("Double value: " + myDouble);  // Outputs: 9.78
        System.out.println("Integer value after explicit casting: " + myInt);
// Outputs: 9 (fractional part is truncated)
    }
}
```

Explanation:

- In the above example, we manually cast a `double` value (9.78) to an `int`. Since `int` cannot store decimal values, the fractional part (`.78`) is **truncated**, and only the integer part (`9`) is kept.

Why Type Casting Is Important

Type casting is necessary when performing operations between variables of different data types or when precision must be controlled. For example, when you divide two integers, the result is an integer, but casting them to `double` can yield a more precise result.

Type Casting and Expressions

When performing arithmetic operations between different data types, Java promotes smaller types to larger types automatically (widening). However, for narrowing conversions, you need to cast explicitly.

Example: Integer Division vs. Floating Point Division

```
public class DivisionExample {
    public static void main(String[] args) {
        int num1 = 5;
        int num2 = 2;

        // Integer division
        int intResult = num1 / num2;  // This results in 2, because it's
integer division
        System.out.println("Integer division: " + intResult);

        // Floating point division with casting
        double floatResult = (double) num1 / num2;  // Explicit casting to
double
        System.out.println("Floating point division: " + floatResult);  //
Outputs: 2.5
    }
}
```

Explanation:

- **Integer Division**: When two integers are divided, the result is truncated to an integer (e.g., `5 / 2 = 2`).
- **Floating Point Division**: By explicitly casting `num1` to `double`, the result becomes `2.5`, demonstrating the need for type casting in preserving precision.

Common Scenarios Requiring Type Casting

1. Converting `int` to `double` for Precision in Calculations

```
int num1 = 7;
int num2 = 3;
```

```
double result = (double) num1 / num2;  // Casting one operand to double for
precision
System.out.println("Precise result: " + result);  // Outputs: 2.3333333
```

2. Avoiding Loss of Data in Narrowing Conversion

```
double largeNumber = 123456.789;
int smallerNumber = (int) largeNumber;  // Explicit cast, loss of decimal part
System.out.println("After casting to int: " + smallerNumber);  // Outputs:
123456
```

Type Promotion in Expressions

In expressions involving multiple data types, smaller data types (like `int`) are promoted to larger types (like `double`) automatically.

Example: Mixed Data Types in Expressions

```
public class TypePromotionExample {
    public static void main(String[] args) {
        int x = 5;
        double y = 4.5;

        double result = x + y;  // x is promoted to double
        System.out.println("Result of adding int and double: " + result);  //
Outputs: 9.5
    }
}
```

Explanation:

- In this example, when adding `int` and `double`, Java automatically promotes the `int` to a `double` before performing the operation to prevent loss of precision.

10. Floating Point Numbers and Precision

The **float** and **double** types are used for decimal values. `float` is single precision (less accurate but smaller), while `double` is double precision (more accurate).

Example:

```
float pi = 3.14f;  // 'f' is necessary for float literals
double largePi = 3.14159265358979;
```

---

11. Arithmetic Operators and Expressions

Java supports standard arithmetic operators:

- `+` (addition)
- `-` (subtraction)
- `*` (multiplication)
- `/` (division)
- `%` (modulus)

Example:

```
int sum = 10 + 5;
int product = 10 * 5;
```

---

12. Char and Boolean

- **`char`**: Stores a single Unicode character.
  - Example: `char letter = 'A';`
- **`boolean`**: Stores true or false values.
  - Example: `boolean isJavaFun = true;`

---

13. Strings in Java

In Java, **Strings** are a crucial part of any program that deals with text. Unlike primitive data types (like `int`, `char`, `boolean`), `String` is a **class** in Java. It is used to handle and manipulate sequences of characters. While Strings may seem like a simple data type, they are essential in most Java applications, making them an important concept to understand thoroughly.

What Is a String?

A **String** in Java is a sequence of characters. It is an **object** of the `String` class, and strings are **immutable**, meaning that once created, their values cannot be changed.

Declaring a String:

```
String message = "Hello, World!";
```

In this example:

- `"Hello, World!"` is a **string literal**, and `message` is a reference to the `String` object.

String Immutability:

- **Immutability** means that once a `String` object is created, it cannot be modified. Any operation that changes the string creates a new `String` object.

Example:

```
String str1 = "Hello";
String str2 = str1;  // Both str1 and str2 refer to the same object
str1 = "Goodbye";  // Now, str1 refers to a new object, but str2 remains
unchanged

System.out.println(str2);  // Outputs: Hello
```

Why Immutability?

- **Thread Safety**: Immutable objects can be shared among multiple threads without synchronization.
- **Caching**: Since strings are immutable, Java can cache and reuse `String` objects (e.g., string literals in a pool).

Commonly Used String Methods

1. **`length()`**:
   - Returns the number of characters in the string.
   - **Example**:

     ```
     java
     Copy code
     String str = "Java Programming";
     int len = str.length();  // Returns 16
     ```

2. **`charAt(int index)`**:
   - Returns the character at the specified index (0-based index).
   - **Example**:

     ```
     java
     Copy code
     char ch = str.charAt(5);  // Returns 'P'
     ```

3. **`substring(int beginIndex, int endIndex)`**:
   - Extracts a part of the string, starting at `beginIndex` and ending at `endIndex - 1`.
   - **Example**:

```
java
Copy code
String subStr = str.substring(0, 4);  // Returns "Java"
```

4. **`toLowerCase()` and `toUpperCase()`**:
   - Converts all characters in the string to lowercase or uppercase.
   - **Example**:

```
java
Copy code
String lower = str.toLowerCase();  // Returns "java programming"
String upper = str.toUpperCase();  // Returns "JAVA PROGRAMMING"
```

5. **`contains(String sequence)`**:
   - Checks if the string contains a particular sequence of characters.
   - **Example**:

```
boolean result = str.contains("Java");  // Returns true
```

6. **`equals(Object obj)`** and **`equalsIgnoreCase(String anotherString)`**:
   - Compares two strings for equality. `equalsIgnoreCase()` compares strings without considering case.
   - **Example**:

```
String a = "hello";
String b = "Hello";
boolean isEqual = a.equals(b);  // Returns false
boolean isEqualIgnoreCase = a.equalsIgnoreCase(b);  // Returns true
```

7. **`replace(char oldChar, char newChar)`**:
   - Replaces all occurrences of `oldChar` with `newChar`.
   - **Example**:

```
String replaced = str.replace('a', 'o');  // Returns "Jovo
Progromming"
```

String Concatenation

In Java, strings can be concatenated (joined) using the `+` operator or the `concat()` method.

Example:

```
String firstName = "John";
String lastName = "Doe";
String fullName = firstName + " " + lastName;  // Outputs: "John Doe"
```

Alternatively:

```
String fullName = firstName.concat(" ").concat(lastName);
```

String Comparison

1. **`==` vs `equals()`**:
   - **`==`** compares the **reference** of the strings (whether they point to the same object).
   - **`equals()`** compares the **content** of the strings.

Example:

```
String a = new String("hello");
String b = "hello";
boolean isEqualReference = (a == b);  // Returns false (different objects)
boolean isEqualContent = a.equals(b);  // Returns true (same content)
```

String Pool

In Java, string literals are stored in a special **String Pool**. When a string literal is created, Java checks if the same literal already exists in the pool. If it does, the new reference points to the existing string in the pool, rather than creating a new one.

Example:

```
String str1 = "Hello";
String str2 = "Hello";
boolean isSame = (str1 == str2);  // Returns true (both refer to the same
object in the pool)
```

StringBuilder and StringBuffer

Since strings are immutable, performing numerous string operations (e.g., concatenation) can lead to performance issues because each operation creates a new `String` object. **`StringBuilder`** and **`StringBuffer`** classes provide mutable alternatives.

StringBuilder Example:

```
StringBuilder sb = new StringBuilder("Hello");
sb.append(" World");  // Modifies the original object
System.out.println(sb);  // Outputs: "Hello World"
```

- **StringBuilder** is not synchronized, meaning it is faster but not thread-safe.
- **StringBuffer** is synchronized, making it thread-safe but slower.

Working with Strings - Sample Program

Here's a program demonstrating some of the key String methods:

```
public class StringDemo {
    public static void main(String[] args) {
        // String Initialization
        String sentence = "Java Programming is fun";

        // 1. Length of the string
        System.out.println("Length: " + sentence.length());

        // 2. Character at a specific index
        System.out.println("Character at index 5: " + sentence.charAt(5));

        // 3. Substring extraction
        System.out.println("Substring (0, 4): " + sentence.substring(0, 4));

        // 4. Convert to uppercase
        System.out.println("Uppercase: " + sentence.toUpperCase());

        // 5. Check if the string contains a word
        System.out.println("Contains 'Programming': " +
sentence.contains("Programming"));

        // 6. String comparison
        String anotherSentence = "JAVA programming is fun";
        System.out.println("Equals (case-sensitive): " +
sentence.equals(anotherSentence));
        System.out.println("Equals (ignore case): " +
sentence.equalsIgnoreCase(anotherSentence));

        // 7. Replace characters in a string
        System.out.println("Replace 'a' with 'o': " + sentence.replace('a',
'o'));

        // 8. String concatenation
        String greeting = "Hello";
        String name = "Alice";
        System.out.println("Concatenation: " + greeting + ", " + name + "!");
    }
}
```

Explanation:

- The program demonstrates various methods such as finding the length of a string, extracting a substring, converting a string to uppercase, and replacing characters.
- It also showcases the difference between `equals()` and `equalsIgnoreCase()` when comparing strings.