Unit 2: Tokens, Expressions and Control Structures [5 Hrs.]

1. Primitive Data Types

Java has built-in data types that represent basic values.

Integers

- **int**: Stores whole numbers (32-bit).
- **long**: Stores large integers (64-bit).

Explanation:

- `int` is commonly used to store integer values, while `long` is used for larger numbers.
- Example: `int x = 10;`

Code Example:

```
public class IntegerExample {
    public static void main(String[] args) {
        int num = 10;
        long bigNum = 10000000000L; // 'L' suffix to denote long type

        System.out.println("Integer: " + num);   // Output: Integer: 10
        System.out.println("Long Integer: " + bigNum);  // Output: Long Integer: 10000000000
    }
}
```

Output:

```
Integer: 10
Long Integer: 10000000000
```

Floating-Point Types

- **float**: Stores single-precision floating-point numbers (32-bit).
- **double**: Stores double-precision floating-point numbers (64-bit).

Explanation:

- `float` is used when lower precision is enough. `double` provides more precision for calculations involving decimals.

Code Example:

```
public class FloatExample {
    public static void main(String[] args) {
        float pi = 3.14F;  // 'F' denotes float
        double e = 2.71828;  // Default is double
        System.out.println("Float: " + pi);  // Output: Float: 3.14
        System.out.println("Double: " + e);  // Output: Double: 2.71828
    }
}
```

Output:

```
Float: 3.14
Double: 2.71828
```

Characters

- **char**: Stores single characters in 16-bit Unicode format.

Explanation:

- `char` is used to store individual characters enclosed in single quotes.

Code Example:

```
public class CharExample {
    public static void main(String[] args) {
        char grade = 'A';
        System.out.println("Character: " + grade);  // Output: Character: A
    }
}
```

Output:

```
Character: A
```

Booleans

- **boolean**: Stores true or false.

Explanation:

- `boolean` is used for conditional statements and logical operations.

Code Example:

```
public class BooleanExample {
    public static void main(String[] args) {
        boolean isJavaFun = true;
        System.out.println("Is Java fun? " + isJavaFun);  // Output: Is Java fun? true
    }
}
```

Output:

```
Is Java fun? true
```

---

2. Variables, Declarations, and Constants

- **Variable**: A storage location with a name and type.
- **Constant**: A variable whose value cannot change.

Explanation:

- Variables are defined with a specific type, and constants are defined with `final` keyword to prevent reassignment.

Code Example:

```
public class VariableExample {
    public static void main(String[] args) {
        int age = 25;  // Variable declaration and assignment
        final int MAX_AGE = 100;  // Constant declaration
        System.out.println("Age: " + age);  // Output: Age: 25
        System.out.println("Max Age: " + MAX_AGE);  // Output: Max Age: 100
    }
}
```

Output:

```
Age: 25
Max Age: 100
```

---

3. Type Conversion and Casting

- **Implicit Conversion (Widening)**: Automatic conversion from smaller to larger type.
- **Explicit Conversion (Narrowing)**: Manual conversion from larger to smaller type.

Explanation:

- **Implicit**: Happens automatically when assigning a smaller type to a larger type.
- **Explicit**: Requires casting (e.g., from `double` to `int`).

Code Example:

```
public class TypeConversionExample {
    public static void main(String[] args) {
        int num = 10;
        double result = num;  // Implicit conversion (int to double)
        System.out.println("Converted to double: " + result);  // Output: Converted to
double: 10.0

        double pi = 3.14;
        int intPi = (int) pi;  // Explicit conversion (double to int)
        System.out.println("Converted to int: " + intPi);  // Output: Converted to
int: 3
    }
}
```

Output:

```
Converted to double: 10.0
Converted to int: 3
```

---

4. Arrays of Primitive Data Types

An array allows you to store multiple values of the same type.

Explanation:

- Arrays are fixed in size and store values of a single type.

- Example: `int[] arr = {1, 2, 3, 4, 5};`

Code Example:

```
public class ArrayExample {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};  // Array of integers
        for (int i = 0; i < arr.length; i++) {
            System.out.println(arr[i]);  // Output: 1 2 3 4 5
        }
    }
}
```

Output:

```
1
2
3
4
5
```

---

5. Control Statements

Branching: if, switch

- **if**: Executes code if the condition is true.
- **switch**: Allows checking multiple conditions in a concise manner.

Explanation:

- **if** is used for single conditions, while **switch** is used when there are multiple options.

Code Example (if):

```
public class IfExample {
    public static void main(String[] args) {
        int age = 20;
        if (age >= 18) {
            System.out.println("Adult");  // Output: Adult
        }
    }
}
```

Code Example (switch):

```
public class SwitchExample {
    public static void main(String[] args) {
        int day = 3;
        switch (day) {
            case 1: System.out.println("Monday"); break;
            case 2: System.out.println("Tuesday"); break;
            default: System.out.println("Invalid day");
        }
```

```
    }
}
```

Output for if:

```
Adult
```

Output for switch:

```
Invalid day
```

Looping: while, do-while, for

- **while**: Executes code as long as the condition is true.
- **do-while**: Executes code at least once and then checks the condition.
- **for**: Executes code for a specific number of times.

Explanation:

- **while** checks the condition before executing the code.
- **do-while** guarantees at least one execution.
- **for** is used when the number of iterations is known in advance.

Code Example (while):

```java
public class WhileExample {
    public static void main(String[] args) {
        int i = 0;
        while (i < 5) {
            System.out.println(i);  // Output: 0 1 2 3 4
            i++;
        }
    }
}
```

Code Example (do-while):

```java
public class DoWhileExample {
    public static void main(String[] args) {
        int i = 0;
        do {
            System.out.println(i);  // Output: 0 1 2 3 4
            i++;
        } while (i < 5);
    }
}
```

Code Example (for):

```java
public class ForExample {
    public static void main(String[] args) {
        for (int i = 0; i < 5; i++) {
            System.out.println(i);  // Output: 0 1 2 3 4
        }
```

```
    }
}
```

Output for while, do-while, and for:

```
0
1
2
3
4
```

Jumping Statements: break, continue, return

- **break**: Exits from a loop or switch statement.
- **continue**: Skips the current iteration and moves to the next.
- **return**: Exits from a method and optionally returns a value.

Explanation:

- **break** stops loop execution.
- **continue** skips the current loop iteration.
- **return** ends method execution and optionally returns a value.

Code Example (break):

```java
public class BreakExample {
    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            if (i == 5) break;
            System.out.println(i);  // Output: 0 1 2 3 4
        }
    }
}
```

Code Example (continue):

```java
public class ContinueExample {
    public static void main(String[] args) {
        for (int i = 0; i < 5; i++) {
            if (i == 2) continue;
            System.out.println(i);  // Output: 0 1 3 4
        }
    }
}
```

Code Example (return):

```java
public class ReturnExample {
    public static void main(String[] args) {
        int result = add(5, 3);
        System.out.println(result);  // Output: 8
    }

    public static int add(int a, int b) {
        return a + b;
```

```
    }
}
```

Output for break, continue, and return:

```
0
1
2
3
4

0
1
3
4

8
```

---

6. User-Defined Data Types

In Java, user-defined data types are types created by the programmer to suit specific needs. These include **classes**, **interfaces**, and **enums**.

- **Class**: A blueprint for creating objects. A class defines properties (fields) and behaviors (methods).
- **Interface**: A contract that specifies a set of methods that a class must implement.
- **Enum**: A special class that represents a group of constants.

Code Example:

```java
// Defining a class
class Person {
    String name;
    int age;

    // Constructor to initialize Person object
    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Method to display details
    void display() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
}

public class UserDefinedDataType {
    public static void main(String[] args) {
        Person p = new Person("Alice", 30);  // Creating a Person object
        p.display();  // Output: Name: Alice, Age: 30
```

```
    }
}
```

Output:

```
Name: Alice, Age: 30
```

---

7. Identifiers and Literals

- **Identifiers**: Names given to variables, methods, classes, etc. Identifiers must start with a letter, underscore, or dollar sign.
- **Literals**: Constant values directly represented in the code (e.g., numbers, strings).

Code Example:

```
public class IdentifierLiteralExample {
    public static void main(String[] args) {
        int num = 5;  // Literal: 5
        String name = "John";  // Literal: "John"
        System.out.println("Number: " + num + ", Name: " + name);  // Output: Number: 5, Name: John
    }
}
```

Output:

```
Number: 5, Name: John
```

---

8. Default Variable Initialization

Java initializes instance variables to default values if not explicitly initialized:

- **int**: 0
- **boolean**: false
- **Object references**: null

Code Example:

```
public class DefaultInitializationExample {
    int num;  // Default value is 0
    boolean flag;  // Default value is false

    public static void main(String[] args) {
        DefaultInitializationExample obj = new DefaultInitializationExample();
        System.out.println("Default int value: " + obj.num);  // Output: Default int value: 0
        System.out.println("Default boolean value: " + obj.flag);  // Output: Default boolean value: false
    }
}
```

Output:

```
Default int value: 0
Default boolean value: false
```

---

9. Command-Line Arguments

Command-line arguments are values passed to the program when it is run from the command line.

Syntax: `java ClassName arg1 arg2`

Code Example:

```
public class CommandLineExample {
    public static void main(String[] args) {
        if (args.length > 0) {
            System.out.println("Command-line argument: " + args[0]);  // Output:
Command-line argument: Hello
        } else {
            System.out.println("No command-line arguments passed.");
        }
    }
}
```

Command: `java CommandLineExample Hello`

Output:

```
Command-line argument: Hello
```

---

10. Comment Syntax

Java supports both single-line and multi-line comments.

- **Single-line comment**: `// comment`
- **Multi-line comment**: `/* comment */`

Code Example:

```
public class CommentExample {
    public static void main(String[] args) {
        // This is a single-line comment
        /* This is a
           multi-line comment */
        System.out.println("Hello World!");  // Output: Hello World!
    }
}
```

Output:

```
Hello World!
```

---

11. Garbage Collection

Garbage Collection (GC) is a process of automatically reclaiming memory from objects that are no longer in use.

- **Garbage Collector**: Java automatically manages memory. The garbage collector reclaims memory by deleting unused objects.
- You cannot directly control the garbage collection process in Java, but you can suggest it using `System.gc()`.

Code Example:

```
public class GarbageCollectionExample {
    public static void main(String[] args) {
        String str = new String("Hello");
        str = null;  // The object is eligible for garbage collection
        System.gc();  // Suggests garbage collection
        System.out.println("Garbage collection triggered");
    }
}
```

Output:

```
Garbage collection triggered
```

---

12. Expressions

An expression in Java is a combination of variables, operators, and method calls that are evaluated to produce a value.

Example Expression: `(x + y) * z`

Code Example:

```
public class ExpressionExample {
    public static void main(String[] args) {
        int x = 10, y = 20, z = 30;
        int result = (x + y) * z;  // Expression
        System.out.println("Result: " + result);  // Output: Result: 900
    }
}
```

Output:

```
Result: 900
```

---

13. Using Operators

Java provides several operators that perform different operations:

Arithmetic Operators: `+`, `-`, `*`, `/`, `%`

Bitwise Operators: `&`, `|`, `^`, `~`, `<<`, `>>`

Relational Operators: `==`, `!=`, `<`, `>`, `<=`, `>=`

Logical Operators: `&&`, `||`, `!`

Assignment Operators: `=`, `+=`, `-=`, `*=`, `/=`, `%=`

Conditional Operator (Ternary): `condition ? trueValue : falseValue`

Shift Operators: `<<`, `>>`, `>>>`

Auto-Increment and Auto-Decrement: `++`, `--`

Code Example (Arithmetic and Ternary Operator):

```
public class OperatorsExample {
    public static void main(String[] args) {
        int x = 10, y = 20;
        int sum = x + y;  // Arithmetic
        int max = (x > y) ? x : y;  // Ternary
        System.out.println("Sum: " + sum);  // Output: Sum: 30
        System.out.println("Max: " + max);  // Output: Max: 20
    }
}
```

Output:

```
Sum: 30
Max: 20
```
