UNIT – I

Object Oriented Programming: Basic concepts, Principles, Program Structure in Java: Introduction, Writing Simple Java Programs, Elements or Tokens in Java Programs, Java Statements, Command Line Arguments, User Input to Programs, Escape Sequences, Comments, Programming Style.

Data Types, Variables, and Operators : Introduction, Data Types in Java, Declaration of Variables, Data Types, Type Casting, Scope of Variable Identifier, Literal Constants, Symbolic Constants, Formatted Output with printf() Method, Static Variables and Methods, Attribute Final, **Introduction to Operators,** Precedence and Associativity of Operators, Assignment Operator (=), Basic Arithmetic Operators, Increment (++) and Decrement (- -) Operators, Ternary Operator, Relational Operators, Boolean Logical Operators, Bitwise Logical Operators.

Control Statements: Introduction, if Expression, Nested if Expressions, if–else Expressions, Ternary Operator?:, Switch Statement, Iteration Statements, while Expression, do–while Loop, for Loop, Nested for Loop, For–Each for Loop, Break Statement, Continue Statement

---

NEED FOR OBJECT ORIENTED PROGRAMMING

Object Oriented Programming (OOP) is an approach to program organisation and development. The major objective of OOP is to eliminate the flaws in Procedure Oriented Programming approach.

Procedural programming involves the executing a step – by – step list of instructions in order complete a given task. In this approach, the whole program forms a single part having a sequential flow. It decomposes the task into variables, data structures and sub routines. These procedures can be invoked at any given time in order to complete its intended function. Some of the procedural languages include Pascal, COBOL, FORTRAN and C. Although procedural oriented programs are extremely powerful, they do have some limitations:

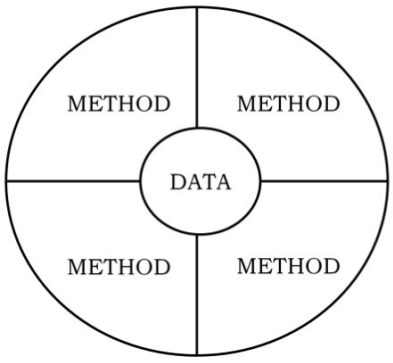
1. In this approach, functions have unrestricted access to global data. Therefore, the data is exposed to the whole program and security cannot be ensured.
2. Further, the importance is given to the operations on data rather than the data itself.
3. It is quite difficult to create user – defined data type and this tends to reduce the extensibility of the program.

On the other hand, object oriented approach to the application development enables us to design programs quickly and carry out modification easily. This approach makes it easier for programmers to change and update the programs as individual objects can be modified without affecting other aspects of the program. As software programs have grown larger

over the years, OOP has made developing these large programs more manageable. Object Oriented Programming offers several advantages and some of those are as follows:

- **Reusability:** In OOP programs, functions and modules can be reused by other users without any modification. It enables programmers to build the programs from previously tested standard working modules that communicate with one another. Thus, it saves development time and leads to high productivity as we do not need to write the code from the scratch.
- **Modular Structure:** OOP provides a clear and modular structure for program development. This enables us to define abstract data type where implementation details can be kept hidden.
- **Easy to maintain and upgrade:** Object – oriented systems can be easily upgraded. It is easy to maintain and modify the existing code. Because of the modularity, it is easier to locate and correct the problem rather than search for the problem in the entire program.
- **Inheritance:** This allows us to eliminate redundant code and use existing classes.
- **Data Hiding:** The programmer can hide the data and functions in as class from other classes. It helps in building the secure programs.

- OOP treats data as a critical element and it does not allow to flow freely around the software system. It ties data more closely to the functions that operate on it and protects from unintentional usage.
- OOP allows us to decompose a problem into a number of entities called objects and then build data and functions around these entities. The data of an object can be accessed only by those methods associated with the objects. This is shown in following diagram:



PRINCIPLES OF OBJECT ORIENTED PROGRAMMING

Object-oriented programming (OOP) is a fundamental programming paradigm based on the concept of "objects". The main aim of OOP is to bind together the data and the methods that operate on them so that no other part of the code can access this data.

The core concept of the object-oriented approach is to break complex problems into smaller objects.

OOPs Concepts:

- Class & Objects
- Data Abstraction and Encapsulation
- Inheritance
- Polymorphism
- Dynamic Binding
- Message Passing

Class:

A class is a user-defined data type. It consists of data members and member functions, which can be accessed and used by creating an instance of that class. It represents the set of properties or methods that are common to all objects of one type. A class is like a blueprint for an object.

For Example: Consider the Class of Cars. There may be many cars with different names and brands but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range, etc. So here, Car is the class, and wheels, speed limits, mileage are their properties.

Object:

It is a basic unit of Object-Oriented Programming and represents the real-life entities. An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated. An object has an identity, state, and behavior. Each object contains data and code to manipulate the data. Objects can interact without having to know details of each other’s data or code, it is sufficient to know the type of message accepted and type of response returned by the objects.

For example “Dog” is a real-life Object, which has some characteristics like color, Breed, Bark, Sleep, and Eats.

An object is represented as follows:

Identity:
Name of the Dog
State/Attribute:
Breed
Age
Color
Behaviours:
Bark
Sleep
Eat

Data Abstraction:

Data abstraction is one of the most essential and important features of object-oriented programming. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

Encapsulation:

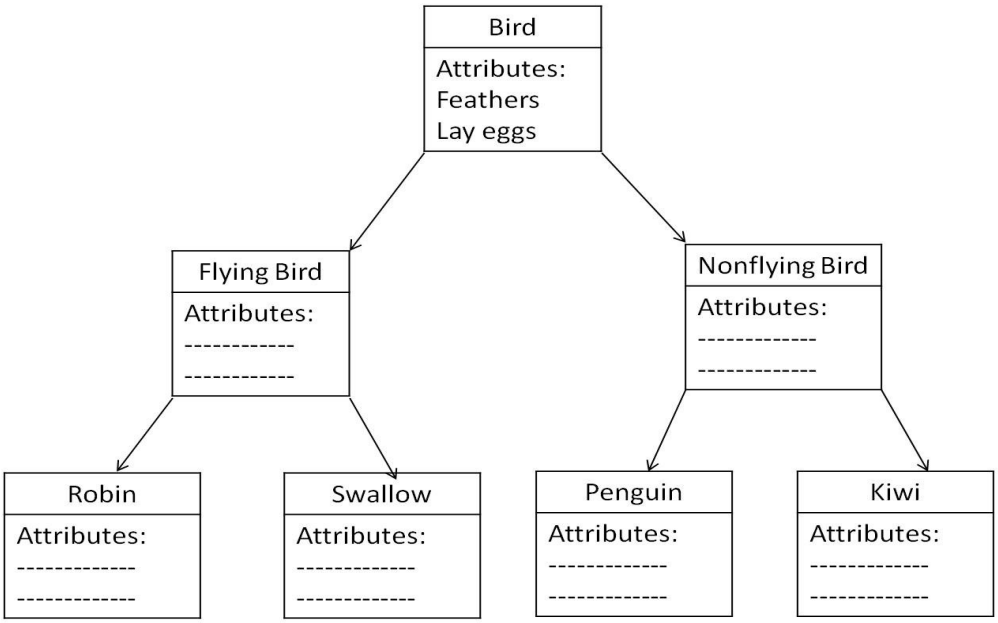
Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates. In Encapsulation, the variables or data of a class are hidden from any other class and can be accessed only through any member function of their class in which they are declared.

Inheritance: Inheritance is a mechanism where a new class is created based on an existing class. Existing class is called Super Class/Base Class/Parent Class. New Class is called Sub Class/Derived Class/Child Class.

The derived class inherits the features (methods and properties) of the base class, allowing the code reusability and establishing a hierarchical relationship between classes.

Subclasses can be allowed to provide a new implementation for methods defined in the super class. This allows subclasses to extend the behavior of inherited methods to suit their specific needs.

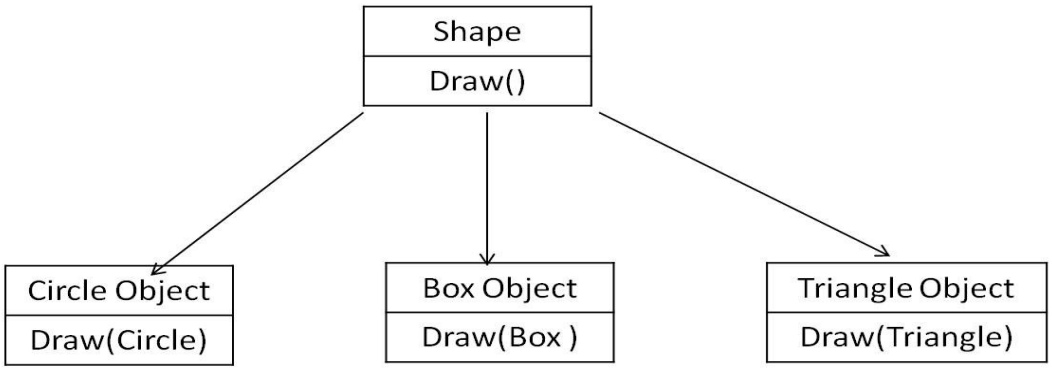
Inheritance Example:



Polymorphism: The word polymorphism means having many forms. For example, an operation may exhibit different behaviour in different instances. The behaviour depends upon the types of data used in the operation.

For example, consider the operation of addition. For two numbers, the operations will generate the sum, but for two strings the operation would produce a third string by concatenation.

Consider the following figure, it illustrates that a single function name can be used to handle different number and different type of arguments:



Dynamic Binding: Binding refers to the process of linking a method call to its corresponding method definition. Binding can happen either at runtime or at compile time. In dynamic binding, the code to be executed in response to the function call is decided at runtime. Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run time.

Message Passing: An object oriented program consists of set of objects that communicate with eachother. Objects communicate with one another by sending and receiving information to each other.

A message for an object is a request for execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results.

Message passing involves specifying the name of the object, the name of the function, and the information to be sent.

Additional Features:

Reuse, Coupling and Cohesion in OOP:

Software engineering lays emphasis on developing high quality software that is easy to use and maintain. To achieve these goals and assess the quality, some metrics have been defined to assess the software design of OOP. This includes cohesion, coupling and reusability.

1. **Cohesion** of a module indicates the extent to which the components of a module are related to each other. In the context of object – oriented programming, if it is applied to classes, it implies the extent to which the class is dependent on other classes in the system. When applied to the methods, it implies the degree to which methods within the class are related to one another and work together to provide well – bounded behaviour.

   In an effective object – oriented design, cohesion is preferred to be high as cohesion promotes encapsulation.
2. **Coupling** is the degree of inter – dependence between the pairs of modules. It is a measure of the strength of association established by a connection from one module to another. Classes are said to be coupled in the following ways:
   1. When a message is passed between the objects
   2. When the methods are declared in one class and accessed the methods or attributes defined in another class.

   A well designed object oriented design should have low coupling, in turn making of independent modules can be possible
3. **Reusability** is the ability to reuse the previously defined objects and classes. In Java, reusability is achieved through Inheritance. We can reuse the already defined objects by virtue of inheritance.

BENEFITS OF OOP

OOP offers many benefits to both the programmer and the user. OOP promises the greater programming productivity, better quality of software and lesser maintenance cost. The principal advantages are:

1. Through inheritance, code can be reused.
2. Data Hiding helps the programmer to build the secure programs
3. We can build programs from the standard working modules rather than having to start writing code from the scratch, this leads to high productivity.
4. It is possible to have multiple objects to co-exist without any interference.
5. It is easy to partition the work in a project based on objects
6. The data-centred design approach enables us to capture more details of a model in an implementation form
7. Object Oriented Systems can be easily upgraded from small to larger systems
8. Message Passing techniques for communication between objects make the interface with external systems much simpler
9. Software complexity can be easily managed.

JAVA PROGRAM STRUCTURE

A Java Program contains one or more classes of which only one class defines a main method. Classes contain data and methods that operate on data of the class. Methods may contain data type declarations and executable statements.

A Java Program contains following sections, of which some are required/some are optional:

Documentation Section
Package Statement
Import Statements
Interface Statements
Class Definitions

Main Method Class
{
Main Method Definition
}

Documentation Section

The documentation section comprises a set of comment lines giving the name of the program, the author and other details which would like to refer further. Comments are used to describe the code. This greatly helps in maintaining the program.

Comment lines are written in two styles:

- To write a single line comment, then that comment statement must be prefixed by // symbol.
- To write multiple lines of comment, then that comment statement must be begins with the symbol /* and ends with the symbol */.

Package Statement

The first statement allowed in Java file is a **package** statement. This statement declares a **package** name and informs the compiler that the classes defined here belong to this package.

```
package <package_name>;
```

package is keyword, <package_name> is the name of the package where the name of the package contains only lowercase letters. This statement is optional.

Import Statements

Import statement is used to instruct the Java Interpreter to load the classes from package specified. An import statement is written using the keyword **import** as follows:

```
import package_name.ClassName;
or
import package_name.*;
```

One is allowed to use the import statement in either of the ways. This statement could gives access to the classes of named packages. This section is optional.

Interface Statements

This section is also optional. An interface is like a class but includes only data constants and abstract methods. This is used to implement the OOP concept - Polymorphism

Class Definitions

This section is also optional. Classes are the primary and essential elements of a Java Program. A Java program may contain multiple class definitions. These classes are used to map the objects of real world problems. The number of classes used depends on the complexity of the problem.

Main Method Class

Every Java program requires a main () method as its starting point, and the class is essential part of the program, so a simple java program contains only this part. The main() method creates objects of various classes and establishes communication between them. On reaching the end of the main(), the program terminates.

JAVA TOKENS

Basically, a Java program is a collection of classes, in turn each class contains data members and methods, methods in turn contain executable/declaration statements, where the statements in turn contain expressions, which describe the actions carried out on data where the expression is a combination of operands and operators, which are indivisible (atomic). These smallest individual elements in Java program are known as tokens.

Java language includes five types of tokens. They are:

- Keywords
- Identifiers
- Literals
- Operators
- Separators

All these tokens are made up of characters in the Java character set. All the characters in this set are defined as Unicode character set, which is an emerging standard. This Unicode is a 16-bit character coding system and currently supports 34,000 defined characters derived from 24 characters.

Keywords

Keywords are the reserved words, they have specific meaning. They are allowed to use for that purpose only as they are reserved. All the keywords are written in lower case letters, as Java is case – sensitive, they must be in lower case otherwise it cannot met its purpose.

In Java there are 50 keywords, those are follows:

abstract	assert	boolean	break
byte	case	catch	char
class	const	continue	default
do	double	else	enum
extends	final	finally	float
for	goto	if	implements
import	instanceof	int	interface
long	native	new	package
private	protected	public	return
short	static	strictfp	super
switch	synchronized	this	throw
throws	transient	try	void
volatile	while		

Identifiers

Identifiers are the programmer designed tokens. They are used for naming classes, methods, variables, objects, labels, packages and interfaces in a program. Java identifiers follow the following rules:

- They can have alphabets, digits, and the underscore characters and dollar sign characters.
- They must not begin with a digit.
- Uppercase and lowercase letters are distinct
- They can be of any length

Identifier must be meaningful, short enough to be quickly and easily typed and long enough to be descriptive and easily read.

Literals

Literals in Java are a sequence of characters; include digits, letters and other characters that represent constant values to be stored in variables. Java language specifies the following types of literals:

- Integer Literals
- Floating Point Literals
- Character Literals
- String Literals
- Boolean Literals
- Null Literal

Each of them has a type associated with it. The type describes how the values behave and how the values are stored.

Operators

An operator is a symbol that takes one or more arguments and operates on them to produce the results. Operators are of many types and they are:

1. Arithmetic Operators (+, -, *, /, %)
2. Relational Operators (>, <, ==, !=, >=, <=)
3. Logical Operators (&&, ||, !)
4. Assignment Operators (=, +=, -=, *=, /=)
5. Increment and Decrement (++, --)
6. Bitwise Operators (&, |, ~, ^, <<, >>, >>>)
7. Conditional Operator (?:)
8. Special Operators (instanceof)

Separators

Separators are the special symbols used to indicate where groups of code are divided and arranged. They basically define the shape and function of our code. The following are list of separators:

- Parenthesis ( )
- Braces { }
- Brackets [ ]
- Semicolon ;
- Comma ,
- Period .

JAVA STATEMENTS

Java Statements comprises set of instructions. These statements specify the sequence of actions to be performed when some method or constructor is invoked. The statements are executed in the sequence in the order in which they appear. It is also possible to carry out the execution without following the sequence using special statements.

Some of the important statements are:

Statement	Description
Empty Statement	These are used during the program development
Variable Declaration Statement	It defines a variable that can be used to store the values
Labelled Statement	A block of statements is given a label. The labels should not be keywords, previously
Expression Statement	Most of the statements came under this category. There are seven types of expressions that include assignment, method call, pre-increment, post increment, pre-decrement and post decrement statements.
Control Statement	This comprises selection, iteration and jump statements
Selection Statement	In these statements, one of the various statements is selected when a certain condition expression is true. There are three types of selection statements include if, if-else and switch
Iteration Statement	These involve the use of loops until some condition for the termination of loop is satisfied. There are three types of iteration statements which include while, do-while, for

Jump Statement	In these Statements, the control is transferred to the beginning or end of the current block or to a labelled statement. These statements include break, continue, return and throw
Synchronization Statement	These are used with multi – threading
Guarding Statement	These are used to carry out the code safely that may cause exceptions. These statements include try-catch block and finally block

COMMAND LINE ARGUMENTS

- Java command-line argument is an argument i.e. passed at the time of running the Java program.
- In Java, the command line arguments passed from the console can be received in the Java program and they can be used as input.
- Command Line arguments are passed as space-separated values. We can pass both strings and primitive data types(int, double, float, char, etc) as command-line arguments.
- These arguments convert into a string array and are provided to the main() function as a string array argument.

When command-line arguments are given, JVM wraps up these command-line arguments into the args[ ] array that we pass into the main() function. We can check these arguments using args.length method. JVM stores the first command-line argument at args[0], the second at args[1], the third at args[2], and so on.

// Java Program to Check for Command Line Arguments

```
class ArgumentsDemo {
	public static void main(String[] args)   {
		if (args.length > 0)
		{
		System.out.println("The command line"+ " arguments are:");
		for (String val : args)
			System.out.println(val);
		}
		else
		System.out.println("No command line "+ "arguments found.");
	}
}
```

USER INPUT TO PROGRAMS

In Java, the Scanner class of package java.util is used to read input from various sources such as the keyboard (standard input), files, or strings. It provides methods to parse primitive types and strings from the input stream.

To use the Scanner class:

4. Import the class Scanner
5. Create the object of Scanner class
6. Call the methods:
    1. To read a line of input as a string, use nextLine()
    2. To read an integer input, use nextInt():
    3. To read a floating-point number, use nextFloat() or nextDouble():
    4. To read a boolean value (true or false), use nextBoolean():
    5. To read a single character, use next().charAt(0):
7. Close the Scanner object

Here's a complete example that demonstrates reading different types of input using Scanner:

```
import java.util.Scanner;
public class ScannerExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter your name: ");
        String name = scanner.nextLine();
        System.out.print("Enter your age: ");
        int age = scanner.nextInt();
        System.out.print("Enter your height (in meters): ");
        float height = scanner.nextFloat();
        System.out.print("Are you married? (true/false): ");
        boolean married = scanner.nextBoolean();
        System.out.println("\n--- User Information ---");
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("Height: " + height + " meters");
        System.out.println("Married: " + (married ? "Yes" : "No"));
        scanner.close();
    }
```

}

Output:

Enter your name: Sharma

Enter your age: 22

Enter your height (in meters): 5.8

Are you married? (true/false): true

--- User Information ---

Name: Sharma

Age: 22

Height: 5.8 meters

Married: Yes

ESCAPE SEQUENCES

In Java, backslashes (\) tell your code to do something special with certain characters. These special characters are like secret codes that make your text behave differently.

Common Escape Sequences

1. Newline (`\n`) - Moves the cursor to the next line.

2. Tab (`\t`) - Inserts a horizontal tab.

3. Backspace (`\b`) - Moves the cursor one position back, deleting the character before it.

4. Double Quote (`\"`) - Represents a double quote character in a string.

5. Single Quote (\') - Represents a single quote character in a string.

6. Backslash (`\\`) - Represents a backslash character.

Here's a small example demonstrating some of these escape sequences:

```
public class EscapeSequencesExample
{
   public static void main(String[] args)
   {
      System.out.println("Hello, World!");    // Prints Hello, World!
      System.out.println("Hello\nWorld");     // Prints Hello and World on separate lines
      System.out.println("Hello\tWorld");     // Prints Hello and World separated by a tab
      System.out.println("She said, \"Hello!\""); // Prints She said, "Hello!"
      System.out.println("Path: C:\\Program Files\\Java");  // Prints Path: C:\Program Files\Java
   }
}
```

Output:

Hello, World!

Hello

World

Hello World

She said, "Hello!"

Path: C:\Program Files\Java

These escape sequences help in formatting output and including special characters in strings that otherwise would be hard to incorporate.

JAVA COMMENTS

- Comments in Java are like notes you write within your code to explain the code.
- They are ignored by the computer when it runs the program, but they are very helpful for humans who need to read and understand the code.

Here are the two main types of comments in Java:

1. Single-line comments: These start with two forward slashes (//) and extend to the end of the line. Anything you write after the // will be ignored by the computer.

Example:

```
int age = 25; // This stores the user's age
```

2. Multi-line comments: These are used for longer explanations and start with /* and end with */. You can write on multiple lines within these markers.

Example:

```
/*
 This function calculates the area of a circle.
 It takes the radius as input and returns the area.      */
public double calculateArea(int radius)
{
        return 3.14 * radius * radius;
}
```

Why use comments?

- *Make code easier to understand:* Comments explain what the code is doing, especially for complex parts.
- *Improve code maintainability:* When you or someone else comes back to the code later, comments help you remember what it does.
- *Help with debugging:* Comments can help you identify where problems might be in your code.

VARIABLES

A variable is a symbolic name or identifier that represents a storage location (memory location) capable of holding a data value.

DECLARATION OF A VARIABLE

In Java, variables are declared with a specific type followed by the variable name. Here are some examples and explanations:

Variable declaration Syntax:

type variable_Name;

where:

- type: Specifies the type of the variable (e.g., int, double, String, boolean, etc.).
- variable_Name: The name of the variable (must be a valid identifier).

Examples:

int age; // Integer Variable

- Here, int is the type and age is the variable name. This declares an integer variable named age that can store integers

String name; //String Variable

- Here, String is the type and name is the variable name. This declares a String variable named name that can store textual data.

double salary; //Double Variable

- Here, double is the type and salary is the variable name. This declares a double variable named salary that can store decimal numbers.

INITIALIZATION OF VARIABLE: Variables can also be initialized (assigned an initial value) at the time of declaration:

type variableName = initialValue;

Example with Initialization:

```
int age = 30;
String name = "John Doe";
double salary = 2500.75;
boolean isStudent = false;
```

Important Notes:

4. *Naming Conventions:* Variable names in Java should follow certain conventions, such as starting with a letter, using camel Case, and avoiding reserved words.
5. *Scope:* Variables have scope, meaning they are only accessible within the block of code where they are declared (except for class-level variables which have broader scope).

Example Code:

```
public class VariableDeclarationExample {
    public static void main(String[] args) {
        // Variable declarations and initializations
        int age = 30;
        String name = "John Doe";
        double salary = 2500.75;
        boolean isStudent = false;

        // Outputting the values of the variables
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("Salary: $" + salary);
        System.out.println("Is Student: " + isStudent);
    }
}
```

DATA TYPES

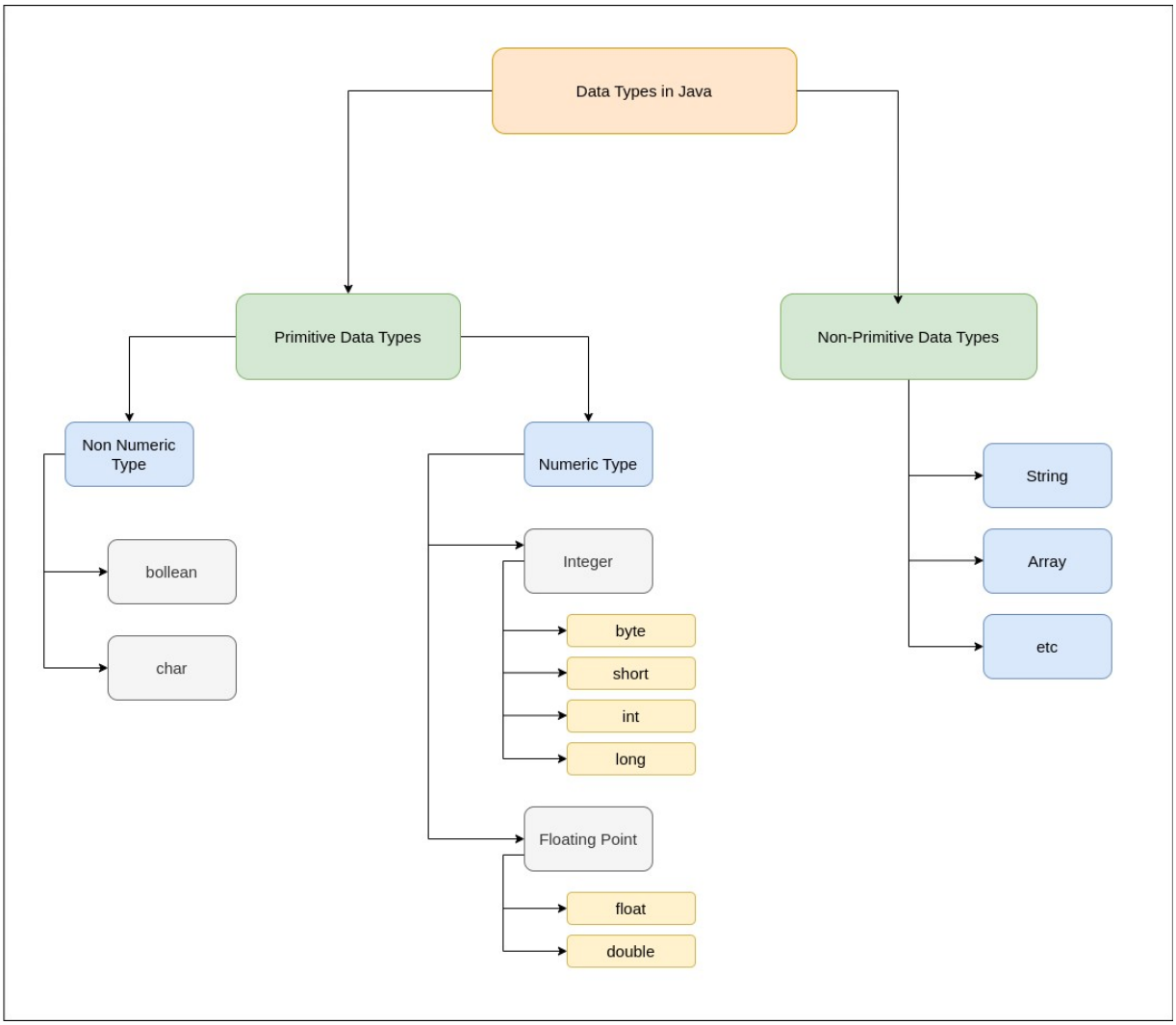
- Data types define the type and value range of the data for the variables, constants, method parameters, returns type, etc.
- The data type tells the compiler about the type of data to be stored and the required memory.
- To store and manipulate different types of data, all variables must have specified data types.

Data Types in Java

Data types in Java are of different sizes and values that can be stored in the variable. In Java, data types are divided into two categories .

Primitive Data Type: such as boolean, char, int, short, byte, long, float, and double

***Non-Primitive Data Type*:** such as String, Array, class, interface.



THE PRIMITIVE TYPES:

Java defines eight *primitive* types of data: **byte**, **short**, **int**, **long**, **char**, **float**, **double**, and **boolean**.

The primitive types are also commonly referred to as *simple* types. These can be put in four groups:

- **Integers:** This group includes **byte**, **short**, **int**, and **long**, which are for whole-valued signed numbers.
- **Floating-point numbers:** This group includes **float** and **double**, which represent numbers with fractional precision.
- **Characters:** This group includes **char**, which represents symbols in a character set, like letters and numbers.
- **Boolean:** This group includes **boolean**, which is a special type for representing true/false values.

Integers

Java defines four integer types: **byte**, **short**, **int**, and **long**. All of these are signed, positive and negative values. Java does not support unsigned, positive-only integers.

The width and ranges of these integer types vary widely, as shown in this table:

Type	Size (in bytes)	Minimum Value	Maximum Value
Byte	1	-128	127
Short	2	-32,768	32,767
Int	4	-2,417,483,648	2,417,483,647
Long	8	-9,223,372,036,854,775,808	9,223,372,036,854,775,807

byte: The smallest integer type is **byte**. This is a signed 8-bit type that has a range from –128 to 127. Byte variables are declared by use of the **byte** keyword. For example, the following declares two **byte** variables called **b** and **c**:

```
byte b, c;
```

short: **Short** is a signed 16-bit type. It has a range from –32,768 to 32,767. It is probably the least-used Java type. Here are some examples of **short** variable declarations:

```
short s;
short t;
```

int: The most commonly used integer type is **int**. It is a signed 32-bit type that has a range from –2,147,483,648 to 2,147,483,647. Here are some examples of **int** variable declarations:

```
int k,m;
int h;
```

long: **Long** is a signed 64-bit type and is useful for those occasions where an **int** type is not large enough to hold the desired value. The range of a **long** is quite large. This makes it useful when big, whole numbers are needed. Here are some examples of **long** variable declarations: long id_number;

Floating-Point Types

Floating-point numbers, also known as *real* numbers, are used when evaluating expressions that require fractional precision.

For example, calculations such as square root, or transcendental such as sine and cosine, results in a value whose precision requires a floating-point type.

There are two kinds of floating-point types, **float** and **double**, which represent single- and double-precision numbers, respectively. Their width and ranges are shown here:

Type	Size (in Bytes)	Minimum Value	Maximum Value
Float	4	3.4e-038	1.7e+0.38
Double	8	3.4e-038	1.7e+0.38

Float

The type **float** specifies a *single-precision* value that uses 32 bits of storage. Variables of type **float** are useful when you need a fractional component, but don't require a large degree of precision. For example, **float** can be useful when representing temperature values, currency values, etc.

Here are some example **float** variable declarations:

float high_temp, low_temp;

Double

Double precision, as denoted by the **double** keyword, uses 64 bits to store a value. All math functions, such as **sin()**, **cos()**, and **sqrt()**, return **double** values. When you need to maintain accuracy over many iterative calculations, or are manipulating large-valued numbers, **double** is the best choice.

Characters

In Java, the data type used to store characters is **char**. Java uses Unicode to represent characters. *Unicode* defines a fully international character set that can represent all of the characters found in all human languages. **It requires 16 bits (2 Bytes)**. The range of a **char** is 0 to 65,536. There are no negative **characters**. Since Java is designed to allow programs to be written for worldwide use, it makes sense that it would use Unicode to represent characters.

Example character variable declarations:

char name;

char book_title;

Booleans

Java has a primitive type, called **boolean**, for logical values. It can have only one of two possible values, **true** or **false**. Its size is only one bit of storage. It stores either 0 or 1. **false** is represented as 0, **true** is represented as 1.

This is the type of value returned by conditional expressions those include relational operators and/or logical operators, as in the case of **a < b**.

```
// Java Program to Demonstrate Primitive Data Types
class DataTypeDemo{
   public static void main(String args[]) {
      char a = 'G';
      int i = 89;
      byte b = 4;
      short s = 56;
      double d = 4.355453532;
      float f = 4.7333434f;
      long l = 12121;
      System.out.println("char: " + a);
      System.out.println("integer: " + i);
      System.out.println("byte: " + b);
      System.out.println("short: " + s);
      System.out.println("float: " + f);
      System.out.println("double: " + d);
      System.out.println("long: " + l);
   }
}
```

NON – PRIMITIVE TYPES:

Non-primitive data types in Java, also known as reference types, include - arrays, classes, objects, strings, interfaces, etc. The Reference Data Types will contain a memory address of variable values.

1. Classes:

- Classes are templates or blueprints used to create objects.
- They define properties (fields) and behaviors (methods) that the objects created from the class can have.
- They support features like inheritance, polymorphism, encapsulation, and abstraction.

2. Objects:

- Objects are instances of classes.
- Objects have states represented by fields (variables) and behaviors represented by methods

3. Arrays:

- Arrays are collections of elements, all of the same type.
- They are fixed in size once they are created.
- Arrays can be of any type, including both primitive and non-primitive types.
- They allow indexed access to their elements.

4. Interfaces:

- Interfaces define abstract methods that a class must implement.
- They provide a way to achieve abstraction in Java.
- An interface only includes method signatures and constants.
- Classes that implement an interface must provide the method bodies.

5. Strings:

- Strings are sequences of characters.
- They are immutable, meaning once a `String` object is created, it cannot be changed.
- The `String` class provides various methods for string manipulation and comparison.
- Strings are stored in a special memory area called the string pool for efficiency.

6. Enums:

- Enums are special data types that enable a variable to be a set of predefined constants.
- They provide a way to define and group sets of constants under a single type.
- Enums are often used to represent a collection of related values.

TYPE CASTING

Type casting is the process of converting a variable from one data type to another. There are two main types of type casting in Java:

1. Implicit Type Casting
2. Explicit Type Casting

1. Implicit (Widening) Type Casting:

- Automatic conversion performed by the Java compiler.
- Occurs when converting a smaller data type to a larger data type.
- No data loss occurs during widening casting.
- In expressions involving mixed data types, smaller types are automatically promoted to larger types. For example, in an expression involving an int and a double, the int is promoted to a double.
- Examples:

  byte to short, int, long, float, double

  short to int, long, float, double

int to long, float, double
long to float, double
float to double

Example:

```
int i = 100;
long l = i; // Implicit casting from int to long
```

2. Explicit (Narrowing) Type Casting:

- Explicit casting is required when you want to forcefully convert a type in an expression.
- Occurs when converting a larger data type to a smaller data type.
- Potential data loss can occur during narrowing casting.
- Requires a cast operator in the syntax.
- Examples:

  double to float, long, int, short, byte
  float to long, int, short, byte
  long to int, short, byte
  int to short, byte
  short to byte

Example:

```
double d = 100.04;
int i = (int) d; // Explicit casting from double to int
```

Understanding type casting is crucial for writing robust and error-free code, especially when dealing with different data types in Java.

SCOPE OF VARIABLE

In Java, the scope of a variable determines where in your code the variable can be accessed or used. This scope is defined by the block of code within which the variable is declared.

Variable scope is crucial for managing data effectively and preventing unintended errors or conflicts in your programs. Here's a detailed explanation of variable scopes in Java:

1. Local Variables

Local variables are declared within a method, constructor, or block of code (such as a loop or conditional statement). They are only accessible within the block in which they are declared.

Example:

```
public class ScopeExample
{
   public void methodA()
        {
              int A = 10; // A is a local variable in methodA
              System.out.println(A); // A is accessible here
         }
  public void methodB()
        {
       //A is not accessible here; it belongs to methodA's scope
         System.out.println(A); // This would cause a compile-time error
        }
}
```

In this example:

- **A** is a local variable within the **methodA()** method. It cannot be accessed from outside **methodA()** or from within **methodB()**

2. Instance Variables (Non-static Fields)

Instance variables are declared within a class but outside any method, constructor, or block. They are initialized when an instance (object) of the class is created and are accessible to all methods, constructors, and blocks in the class.

Example:

```
public class ScopeExample
{
   int x; // Here, x is an instance variable
   public void methodA()
        {
             x = 20; // x can be accessed and modified within methodA
             System.out.println(x);
        }
   public void methodB()
        {
              x= 30; // x can be accessed and modified within methodB
              System.out.println(x);
        }
}
```

In this example:

- **x** is an instance variable of the class **ScopeExample**. It is accessible and can be modified from both **methodA()** and **methodB().**

3. Class Variables (Static Fields)

Class variables, also known as static fields, are declared with the static keyword within a class but outside any method, constructor, or block. They are associated with the class rather than with instances of the class. Class variables are shared among all instances (objects) of the class.

LITERAL CONSTANTS

Understanding Literals:

In Java, Literals are essentially constant values that appear directly within your program source code. You can assign them to variables to store and manipulate the data.
For example, in the statement int age = 25;`, the value `25` is an integer literal.

Types of Literals in Java:

Java offers various literal types to represent different kinds of data:

Integer Literals: Represent whole numbers. They can be written in decimal (e.g., 10), octal (e.g., 077), hexadecimal (e.g., 0xFF), or binary (e.g., 0b1100) formats.

Floating-Point Literals: Represent numbers with decimal points or exponents. Examples include `3.14159` and `1.2e-3` (scientific notation).

Character Literals: Represent single characters enclosed within single quotes (e.g., 'a', '!', '$').

String Literals: Represent sequences of characters enclosed within double quotes (e.g., "Hello, world!").

Boolean Literals: Represent logical values, either `true` or `false`.

SYMBOLIC CONSTANTS

Symbolic constants are fixed values with descriptive names, defined once and used throughout your program. They improve code readability and maintainability compared to using raw numerical values everywhere.

Declaring Symbolic Constants

- Use the final keyword during variable declaration to create a symbolic constant.
- The variable type can be int, double, boolean, String, or any other valid data type.
- We name constants with all capital letters, separated by underscores. (e.g., MAX_VALUE, PI).
- **Example:**

```
final double PI = 3.14159;
final int MAX_VALUE = 100;
final String GREETING = "Hello, world!";
```

Benefits of Symbolic Constants

- **Readability:** Descriptive names make code easier to understand.
- **Maintainability:** If a constant value needs to change, you only modify it in one place.
- **Type Safety:** The compiler ensures the constant's value matches the declared data type.

Conditions to follow:

- Symbolic constants cannot be reassigned a value after initialization.
- They are typically declared at the class level or within a static block.

By effectively using symbolic constants, you can write cleaner, more manageable, and less error-prone Java code.

STATIC VARIABLES AND METHODS

Static Variables:

- In Java, a static variable is a variable declared with the **static** keyword.
- All objects of the class share the same copy of the static variable. This means, there is only one copy of the variable for the entire class, rather than each instance of the class having its own copy.
- If one object modifies the static variable, the change is reflected for all other objects as well.
- Static variables are accessed using the class name.

Example:

```
class Counter {
        public static int count = 0;
        public Counter()
        {
                count++;
         }
          public static void main(String[] args)
         {
                Counter c1 = new Counter();
                 Counter c2 = new Counter();
                 System.out.println(Counter.count);
        }
}
```

In this example, count is a static variable that keeps track of the number of Counter objects created. Even though separate objects are created (c1 and c2), they modify the same count variable.

Static Methods:

- In Java, a static method is a method declared with the **static** keyword.
- Static methods belong to the class itself rather than individual objects of the class.
- Static methods can be called directly using the class name, without creating an object of the class.

For example, the method sqrt() is declared as static method in Math class, and so it is called as Math.sqrt(16);

// Program example to illustrate the using of static methods of class Math:

```
import java.lang.Math;
public class PredefinedNumbers {
        public static void main(String[] args)
        {
        System.out.println("The square root of 16 is: " + Math.sqrt(16));
        System.out.println("The cube root of 27 is: " + Math.cbrt(27));
        System.out.println(“Five Random Numbers are:”);
        for(int i=0;i<5;i++)
                System.out.print((int)(100*Math.random())+"\t");
        }
}
```

Output:

The square root of 16 is: 4.0

The cube root of 27 is: 3.0

Five Random Numbers are: 81 98 55 12 66

MATHEMATICAL FUNCTIONS

The Math class in Java provides a collection of methods for performing basic mathematical operations and functions. These methods are static, so they can be called directly using the class name (Math.methodName()). Here are some commonly used mathematical functions available in the Math class:

1. Exponential and Logarithmic Functions:

- exp(double a): Returns the exponential (e^a).
- log(double a): Returns the natural logarithm (base e) of a.
- log10(double a): Returns the base 10 logarithm of a.
- pow(double a, double b): Returns a raised to the power of b (a^b).

2. Trigonometric Functions:

- sin(double a), cos(double a), tan(double a): Computes the sine, cosine, and tangent of an angle given in radians.
- asin(double a), acos(double a), atan(double a): Computes the arcsine, arccosine, and arctangent of a given value (returning results in radians).
- toRadians(double angdeg): Converts an angle measured in degrees to radians.
- toDegrees(double angrad): Converts an angle measured in radians to degrees.

3. Rounding and Absolute Value:

- ceil(double a): Returns the smallest integer greater than or equal to a.
- floor(double a): Returns the largest integer less than or equal to a.
- round(float a), round(double a): Rounds a floating-point number to the nearest integer.
- abs(int a), abs(long a), abs(float a), abs(double a): Returns the absolute value of a number.

4. Square Root and Exponents:

- sqrt(double a): Returns the square root of a non-negative number.
- cbrt(double a): Returns the cube root of a number.
- hypot(double x, double y): Computes sqrt(x^2 + y^2) without intermediate overflow or underflow.
- pow(double x, double y): Returns the value x to the power of y

5. Constants:

- Math.PI: Represents the mathematical constant pi (π).
- Math.E: Represents the mathematical constant e (Euler's number).

6. Aggregate Functions:

- max(double x, double y): Returns the maximum value out of x and y
- min(double x, double y): Returns the minimum value out of x and y

EXAMPLE PROGRAM:

```
import java.lang.Math;
public class MathFunctionsExample {
    public static void main(String[] args) {
        int x = 10;
        int y = 5;
        double a = 30.5;
        double b = 2.0;
         // Exponential and logarithmic functions
        System.out.println("e raised to the power of " + a + " is: " + Math.exp(a));
        System.out.println("Natural logarithm of " + a + " is: " + Math.log(a));
        System.out.println("Base 10 logarithm of " + a + " is: " + Math.log10(a));
        System.out.println(b + " raised to the power of " + a + " is: " + Math.pow(b, a));
          // Trigonometric functions (angles in radians)
        double angle = Math.PI / 4; // 45 degrees in radians
        System.out.println("Sin of " + angle + " radians is: " + Math.sin(angle));
        System.out.println("Cos of " + angle + " radians is: " + Math.cos(angle));
        System.out.println("Tan of " + angle + " radians is: " + Math.tan(angle));
          // Rounding and absolute value
        double num = -45.67;
        System.out.println("Ceiling of " + num + " is: " + Math.ceil(num));
        System.out.println("Floor of " + num + " is: " + Math.floor(num));
        System.out.println("Rounded value of " + num + " is: " + Math.round(num));
        System.out.println("Absolute value of " + num + " is: " + Math.abs(num));
        // Square root and cube root
        double number = 144.0;
        System.out.println("Square root of " + number + " is: " + Math.sqrt(number));
        System.out.println("Cube root of " + number + " is: " + Math.cbrt(number));
         }
}
```

STATIC IMPORT

Static import in Java is a feature introduced in Java 5 that allows you to directly use static members (fields and methods) of a class without having to prefix them with the class name. This can improve code readability and reduce verbosity, especially when you're working with a class that has many static members you use frequently.

```
/*  Program example to illustrate the using of static methods of class Math:  */
import static java.lang.Math.*;
public class PredefinedNumbers
{
        public static void main(String[] args)
        {
        System.out.println("The square root of 16 is: " +sqrt(16));
        System.out.println("The cube root of 27 is: " + cbrt(27));
        System.out.println(“Five Random Numbers are:”);
        for(int i=0;i<5;i++)
                System.out.print((int)(100*random())+"\t");
        }
}
```

Output:

```
The square root of 16 is: 4.0
The cube root of 27 is: 3.0
Five Random Numbers are: 81 98 55 12 66
```

ATTRIBUTE FINAL

The final keyword in Java is a versatile tool used to restrict modifications and inheritance. It can be applied to variables, methods, and even classes.

Final Variables: When you declare a variable as final, its value becomes fixed after the initial assignment. You cannot reassign a new value to it, later in your code.

So, final is used to create constants, which represent fixed values like PI (3.14159) or MAX_SPEED (100).

Example:

```
final double PI = 3.14159;
PI = 10; // This will cause a compilation error because PI is final
```

Final Methods:

A method declared as final cannot be overridden by subclasses. This prevents changes to the method definition in child classes.

Example:

```
class MathUtil
{
        public final double calculateArea(int radius)
        {
                return PI * radius * radius;
        }
}
class Circle extends MathUtil
{
          // in this class, we cannot override calculateArea(), because it's final
}
```

Final Classes:

A class declared as final cannot be extended by other classes. This prevents unintended inheritance.

Example:

```
final class StringHelper
{
        ----------
        ----------
        ----------
}
```

The below code would cause a compilation error because **StringHelper** *is final:*

```
class MyString extends StringHelper
{
        ----------
        ----------
        ----------
}
```

INTRODUCTION TO OPERATORS

Operators in Java are special symbols or keywords that perform operations on variables and values. Java provides a rich set of operators to manipulate variables. They can be categorized based on the functionality they provide. Here's an overview of the main types of operators in Java:

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Unary Operators
5. Assignment Operators
6. Bitwise Operators
7. Conditional (Ternary) Operator
8. Instanceof Operator

ASSIGNMENT OPERATOR (=)

Assignment operators in Java are used to assign values to variables. The basic assignment operator is =.

Basic Assignment Operator

= This operator assigns the value on its right to the variable on its left.

```
int a = 10; // Assigns the value 10 to the variable a
```

Compound Assignment Operators

Java also provides compound assignment operators, which combine an arithmetic or bitwise operation with assignment. Here are the compound assignment operators:

+= Add and assign
-= Subtract and assign
*= Multiply and assign
/= Divide and assign
%= Modulus and assign
&= Bitwise AND and assign
|= Bitwise OR and assign
^= Bitwise XOR and assign
<<= Left shift and assign
>>= Right shift and assign
>>>= Unsigned right shift and assign

Examples of Compound Assignment Operators

Here are examples demonstrating how these operators work:

```
public class AssignmentOperatorsExample {
    public static void main(String[] args) {
        int a = 10;
        int b = 20;

        a += b; // equivalent to a = a + b; now a is 30
        a -= b; // equivalent to a = a - b; now a is 10
        a *= b; // equivalent to a = a * b; now a is 200
        a /= b; // equivalent to a = a / b; now a is 10
        a %= b; // equivalent to a = a % b; now a is 10

        a &= b; // equivalent to a = a & b; now a is 0
        a |= b; // equivalent to a = a | b; now a is 20
        a ^= b; // equivalent to a = a ^ b; now a is 0

        a = 10; // reset a to 10
        a <<= 2; // equivalent to a = a << 2; now a is 40
        a >>= 2; // equivalent to a = a >> 2; now a is 10
        a >>>= 2; // equivalent to a = a >>> 2; now a is 2
    }
}
```

Explanation of Compound Assignment Operators

1. Addition Assignment (+=):
   a += b is equivalent to a = a + b.
   - Example: If a is 10 and b is 20, then a += b will make the value of a is 30.

2. Subtraction Assignment (-=):
   a -= b is equivalent to a = a - b.
   - Example: If a is 30 and b is 20, then a -= b will make a 10.

3. Multiplication Assignment (*=):
   a *= b is equivalent to a = a * b.
   - Example: If a is 10 and b is 20, then a *= b will make the value of a is 200.

4. Division Assignment (/=):
   a /= b is equivalent to a = a / b.
   - Example: If a is 200 and b is 20, then a /= b will make the value of a is 10.

5. Modulus Assignment (%=):
   a %= b is equivalent to a = a % b.
   - Example: If a is 10 and b is 3, then a %= b will make the value of a is 1.

6. Bitwise AND Assignment (&=):
   a &= b is equivalent to a = a & b.
   - Example: If a is 10 (1010 in binary) and b is 20 (10100 in binary), then a &= b will make the value of a is 0 (0000 in binary).

7. Bitwise OR Assignment (|=):
   a |= b is equivalent to a = a | b.
   - Example: If a is 10 (1010 in binary) and b is 20 (10100 in binary), then a |= b will make the value of a is 30 (11110 in binary).

8. Bitwise XOR Assignment (^=):
   a ^= b is equivalent to a = a ^ b.
   - Example: If a is 10 (1010 in binary) and b is 20 (10100 in binary), then a ^= b will make a value is 30 (11110 in binary).

9. Left Shift Assignment (<<=):
   a <<= b is equivalent to a = a << b.
   - Example: If a is 10 (1010 in binary) and b is 2, then a <<= b will make the value of a is 40 (101000 in binary).

10. Right Shift Assignment (>>=):
    a >>= b is equivalent to a = a >> b.
    - Example: If a is 10 (1010 in binary) and b is 2, then a >>= b will make the value of a is 2 (10 in binary).

11. Unsigned Right Shift Assignment (>>>=):
    a >>>= b is equivalent to a = a >>> b.
    - Example: If a is 10 (1010 in binary) and b is 2, then a >>>= b will make the value of a is 2 (10 in binary).

Assignment operators are a fundamental part of Java programming, allowing for efficient manipulation of variables. The basic assignment operator = is straightforward, while the compound assignment operators provide a concise way to perform arithmetic and bitwise operations combined with assignment.

BASIC ARITHMETIC OPERATORS

Arithmetic operators in Java are used to perform basic mathematical operations such as addition, subtraction, multiplication, division, and modulus. Here is a detailed overview of each arithmetic operator along with examples:

1. Addition (+)

The addition operator adds two operands.

Example:

```
int a = 10;
int b = 20;
int sum = a + b; // 30
System.out.println("Sum: " + sum);
```

2. Subtraction (-)

The subtraction operator subtracts the second operand from the first.

Example:

```
int a = 20;
int b = 10;
int difference = a - b; // 10
System.out.println("Difference: " + difference);
```

3. Multiplication (*)

The multiplication operator multiplies two operands.

Example:

```
int a = 10;
int b = 20;
int product = a * b; // 200
System.out.println("Product: " + product);
```

4. Division (/)

The division operator divides the first operand by the second. If both operands are integers, the result will also be an integer (truncated towards zero). If one or both operands are floating-point numbers, the result will be a floating-point number.

Example:

```
int a = 20;
int b = 10;
int quotient = a / b; // 2
System.out.println("Quotient: " + quotient);
double x = 20.0;
double y = 10.0;
double quotientDouble = x / y; // 2.0
System.out.println("Quotient (double): " + quotientDouble);
```

5. Modulus (%)

The modulus operator returns the remainder when the first operand is divided by the second.

Example:

```
int a = 20;
int b = 7;
int remainder = a % b; // 6
System.out.println("Remainder: " + remainder);
```

These operators are used frequently in Java programming for mathematical calculations and can be applied to both integer and floating-point data types.

Example Program: Basic Arithmetic Operations

```
public class ArithmeticOperatorsExample {
    public static void main(String[] args) {
        // Declare and initialize variables
        int num1 = 25;
        int num2 = 10;
        // Perform arithmetic operations
        int sum = num1 + num2;
        int difference = num1 - num2;
        int product = num1 * num2;
        int quotient = num1 / num2;
        int remainder = num1 % num2;
        // Display the results
        System.out.println("Number 1: " + num1);
        System.out.println("Number 2: " + num2);
```

```
        System.out.println("Sum: " + sum);
        System.out.println("Difference: " + difference);
        System.out.println("Product: " + product);
        System.out.println("Quotient: " + quotient);
        System.out.println("Remainder: " + remainder);
    }
}
```

INCREMENT (++) AND DECREMENT (- -) OPERATORS

In Java, the increment (++) and decrement (--) operators are unary operators that are used to increase or decrease the value of a variable by one, respectively.

These operators can be used in two forms: prefix and postfix.

1. Increment Operator (++)

The increment operator increases the value of a variable by one.

- *Prefix Increment (++variable):* Increments the variable's value before using it in an expression.
- *Postfix Increment (variable++):* Increments the variable's value after using it in an expression.

Example:

```
public class IncrementExample {
    public static void main(String[] args) {
        int a = 5;
        int x = ++a; // a is incremented to 6, then x is assigned 6
        System.out.println("Prefix Increment: a = " + a + ", prefixResult = " +x);

        // Reset a to 5
        a = 5;
        int y = a++; // y is assigned 5, then a is incremented to 6
        System.out.println("Postfix Increment: a = " + a + ", postfixResult = " + y);
    }
}
```

2. Decrement Operator (--)

The decrement operator decreases the value of a variable by one.

- *Prefix Decrement (--variable):* Decrements the variable's value before using it in an expression.
- *Postfix Decrement (variable--):* Decrements the variable's value after using it in an expression.

Example:

```
public class DecrementExample {
    public static void main(String[] args) {
        int a = 5;
        int x = --a; // a is decremented to 4, then x is assigned 4
        System.out.println("Prefix Decrement: a = " + a + ", prefixResult = " + x);

        // Reset a to 5
        a = 5;
        int y = a--; // y is assigned 5, then a is decremented to 4
        System.out.println("Postfix Decrement: a = " + a + ", postfixResult = " +y);
    }
}
```

TERNARY OPERATOR

The ternary operator in Java is a shorthand way of writing an if-else statement. It is also known as the conditional operator. The ternary operator consists of three parts: a condition, a result for true, and a result for false.

The syntax is as follows:

```
result = condition ? value1 : value2;
```

- condition: The expression to be evaluated, which should return a boolean value (true or false).

- value1: The value assigned to the result if the condition is true.

- value2: The value assigned to the result if the condition is false.

Example

Let's look at an example where we use the ternary operator to determine the minimum of two numbers:

```
public class TernaryOperatorExample {
    public static void main(String[] args) {
        int a = 10;
        int b = 20;
        int min = (a < b) ? a : b;
        System.out.println("The minimum value is: " + min);
    }
}
```

In this example:

- The condition (a < b) checks if a is less than b.
- If the condition is true, a is assigned to min.
- If the condition is false, b is assigned to min.

More Examples

Example 1: Finding the Maximum Value

```
public class TernaryOperatorMaxExample {
    public static void main(String[] args) {
        int a = 10;
        int b = 20;
        int max = (a > b) ? a : b;
        System.out.println("The maximum value is: " + max);
    }
}
```

Example 2: Checking for Even or Odd

```
public class TernaryOperatorEvenOddExample {
    public static void main(String[] args) {
        int number = 25;
        String result = (number % 2 == 0) ? "Even" : "Odd";
        System.out.println("The number is: " + result);
    }
}
```

Nested Ternary Operator

The ternary operator can also be nested, although it's generally recommended to avoid excessive nesting for readability purposes.

Example: Nested Ternary Operator

```
public class NestedTernaryOperatorExample {
    public static void main(String[] args) {
        int a = 10;
        int b = 20;
        int c = 15;
        int max = (a > b) ? (a > c ? a : c) : (b > c ? b : c);
        System.out.println("The maximum value is: " + max);
    }
}
```

The ternary operator provides a concise way to make simple conditional assignments. It can improve code readability for simple conditions but should be used judiciously to avoid confusing or complex expressions.

RELATIONAL OPERATORS

Relational operators in Java are used to compare two values. They return a boolean result indicating whether the comparison is true or false. These operators are essential for making decisions in control flow statements like if, for, while, etc.

List of Relational Operators

1. == : Equal to
2. != : Not equal to
3. > : Greater than
4. < : Less than
5. >= : Greater than or equal to
6. <= : Less than or equal to

Examples

1. Equal to (==): Checks if the value of two operands is equal.

```
int a = 10;
int b = 20;
boolean result = (a == b); // false
System.out.println("a == b: " + result);
```

2. Not equal to (!=): Checks if the value of two operands is not equal.

```
int a = 10;
int b = 20;
boolean result = (a != b); // true
System.out.println("a != b: " + result);
```

3. Greater than (>): Checks if the value of the left operand is greater than the value of the right operand.

```
int a = 10;
int b = 20;
boolean result = (a > b); // false
System.out.println("a > b: " + result);
```

4. Les than (<): Checks if the value of the left operand is less than the value of the right operand.

```
int a = 10;
int b = 20;
boolean result = (a < b); // true
System.out.println("a < b: " + result);
```

5. Greater than or equal to (>=): Checks if the value of the left operand is greater than or equal to the value of the right operand.

```
int a = 20;
int b = 20;
boolean result = (a >= b); // true
System.out.println("a >= b: " + result);
```

6. Less than or equal to (<=): Checks if the value of the left operand is less than or equal to the value of the right operand.

```
int a = 10;
int b = 20;
boolean result = (a <= b); // true
System.out.println("a <= b: " + result);
```

BOOLEAN LOGICAL OPERATORS

Boolean logical operators in Java are used to perform logical operations on boolean expressions. They are primarily used in conditional statements and loops to combine multiple boolean expressions and control the flow of the program based on these combined conditions.

List of Boolean Logical Operators

1. && Logical AND
2. || Logical OR
3. ! Logical NOT
4. ^ Logical XOR

Examples

1. Logical AND (&&): The logical AND operator returns true if both operands are true. If either operand is false, it returns false.

```
boolean a = true;
boolean b = false;
boolean result = a && b; // false
System.out.println("a && b: " + result);
result = a && true; // true
System.out.println("a && true: " + result);
```

2. Logical OR (||): The logical OR operator returns true if at least one of the operands is true. If both operands are false, it returns false.

```
boolean a = true;
boolean b = false;
```

```
boolean result = a || b; // true
System.out.println("a || b: " + result);
result = b || false; // false
System.out.println("b || false: " + result);
```

3. Logical NOT (!): The logical NOT operator inverts the value of its operand. If the operand is true, it returns false; if the operand is false, it returns true.

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

4. Logical XOR (^): The logical XOR (exclusive OR) operator returns true if exactly one of the operands is true. If both operands are true or both are false, it returns false.

```
boolean a = true;
boolean b = false;
boolean result = a ^ b; // true
System.out.println("a ^ b: " + result);
result = a ^ true; // false
System.out.println("a ^ true: " + result);
```

Short-Circuit Evaluation

Java uses short-circuit evaluation for && and || operators. This means that the second operand is evaluated only if necessary.

- && (Logical AND): If the first operand is false, the second operand is not evaluated because the result will be false regardless of the second operand.
- || (Logical OR): If the first operand is true, the second operand is not evaluated because the result will be true regardless of the second operand.

Example:

```
public class ShortCircuitEvaluationExample {
   public static void main(String[] args) {
      int x = 5;
      boolean result = (x > 10) && (++x > 5); // false, ++x is not evaluated
      System.out.println("Result: " + result); // Result: false
      System.out.println("x: " + x); // x: 5
      result = (x < 10) || (++x > 5); // true, ++x is not evaluated
      System.out.println("Result: " + result); // Result: true
      System.out.println("x: " + x); // x: 5
   }
}
```

Boolean logical operators in Java (&&, ||, !, ^) are essential for constructing complex conditional statements and controlling the flow of a program. Understanding how these operators work and how to combine them effectively is crucial for writing clear and efficient Java code.

BITWISE LOGICAL OPERATORS

Bitwise logical operators in Java are used to perform bit-level operations on integer types (byte, short, int, and long). These operators treat their operands as sequences of 32 or 64 bits (binary digits) rather than decimal, hexadecimal, or octal numbers. The bitwise logical operators include AND, OR, XOR, and NOT.

Here's a detailed look at each bitwise operator:

1. & Bitwise AND
2. | Bitwise OR
3. ^ Bitwise XOR (exclusive OR)
4. ~ Bitwise NOT (complement)
5. << Left shift
6. >> Right shift
7. >>> Unsigned right shift

Examples

1. Bitwise AND (&): The bitwise AND operator compares each bit of its first operand to the corresponding bit of its second operand. If both bits are 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.

```
int a = 5;  // 0101 in binary
int b = 3;  // 0011 in binary

int result = a & b; // 0001 in binary (1 in decimal)
System.out.println("a & b: " + result);
```

2. Bitwise OR (|): The bitwise OR operator compares each bit of its first operand to the corresponding bit of its second operand. If either bit is 1, the corresponding result bit is set to 1. If both bits are 0, the corresponding result bit is set to 0.

```
int a = 5;  // 0101 in binary
int b = 3;  // 0011 in binary
int result = a | b; // 0111 in binary (7 in decimal)
System.out.println("a | b: " + result);
```

3. Bitwise XOR (^): The bitwise XOR operator compares each bit of its first operand to the corresponding bit of its second operand. If the bits are different, the corresponding result bit is set to 1. If the bits are the same, the corresponding result bit is set to 0.

```
int a = 5;  // 0101 in binary
int b = 3;  // 0011 in binary
int result = a ^ b; // 0110 in binary (6 in decimal)
System.out.println("a ^ b: " + result);
```

4. Bitwise NOT (~): The bitwise NOT operator inverts each bit of its operand. All 0s are changed to 1s and all 1s are changed to 0s.

```
int a = 5;  // 0101 in binary
int result = ~a; // 1010 in binary (inverting all bits)
System.out.println("~a: " + result);
```

5. Left Shift (<<): The left shift operator shifts the bits to the left by the number of positions specified by the right operand. The empty positions on the right are filled with zeros.

```
int a = 5;  // 0101 in binary
int result = a << 1; // 1010 in binary (10 in decimal)
System.out.println("a << 1: " + result);
```

6. Right Shift (>>): The right shift operator shifts the bits to the right by the number of positions specified by the right operand. The empty positions on the left are filled with the sign bit (0 for positive numbers, 1 for negative numbers).

```
int a = 5;  // 0101 in binary
int result = a >> 1; // 0010 in binary (2 in decimal)
System.out.println("a >> 1: " + result);
```

7. Unsigned Right Shift (>>>): The unsigned right shift operator shifts the bits to the right by the number of positions specified by the right operand. The empty positions on the left are filled with zeros.

```
int a = -5;  // 11111111111111111111111111111011 in binary (two's complement)
int result = a >>> 1; // 01111111111111111111111111111101 in binary
System.out.println("a >>> 1: " + result);
```

PRECEDENCE AND ASSOCIATIVITY OF OPERATORS

In Java, operators have a specific order in which they are evaluated, known as precedence, and a specific direction in which they are evaluated, known as associativity. Understanding these concepts is crucial for writing correct and efficient code.

Precedence

Operator precedence determines the order in which operators are evaluated in expressions. Operators with higher precedence are evaluated before operators with lower precedence.

Associativity

Associativity determines the direction of evaluation for operators of the same precedence. Operators can be left-associative, meaning they are evaluated from left to right, or right-associative, meaning they are evaluated from right to left.

There is often confusion when it comes to equations having multiple operators. The problem is which part to solve first. There is a golden rule to follow in these situations.

- If the operators have different precedence, solve the higher precedence first.
- If they have the same precedence, solve according to associativity, that is, either from right to left or from left to right.

Java Operator Precedence and Associativity Table

Here is a table listing Java operators, their precedence (from highest to lowest), and their associativity:

Operators	Associativity	Type
++ --	Right to left	Unary postfix
++ -- + - ! (type)	Right to left	Unary prefix
/ * %	Left to right	Multiplicative
+ -	Left to right	Additive
< <= > >=	Left to right	Relational
== !=	Left to right	Equality
&	Left to right	Boolean Logical AND
^	Left to right	Boolean Logical Exclusive OR
|	Left to right	Boolean Logical Inclusive OR
&&	Left to right	Conditional AND
||	Left to right	Conditional OR
?:	Right to left	Conditional
= += -= *= /= %=	Right to left	Assignment

Example

Consider the expression `a + b * c`. According to the precedence rules:

1. The multiplication (`*`) operator has higher precedence than the addition (`+`) operator.
2. Therefore, `b * c` is evaluated first, then the result is added to `a`.

Understanding operator precedence and associativity is essential for writing unambiguous and correct Java expressions. When in doubt, you can always use parentheses to explicitly specify the desired order of evaluation.

CONTROL STATEMENTS

In Java, Control statements guide the flow of execution of the program based on certain conditions and logic. These statements help in making decisions, looping through a series of statements, and controlling the program's execution flow.

Here are the main types of control statements in Java:

1. Decision Making Statements
2. Loop Statements
3. Jump Statements

DECISION MAKING STATEMENTS

Decision-making statements are crucial in Java programming as they allow a program to make decisions based on certain conditions and execute different actions depending on those conditions.

These statements evaluate boolean expressions and direct the flow of the program accordingly. In Java, the most common decision-making statements include:

1. if statement
2. if – else statement
3. Ternanry Operator
4. if-else-if statement
5. switch statement

IF EXPRESSION:

A simple if expression is used to evaluate a single condition and execute a block of code if that condition is true.

Syntax:

```
if (condition)
{
        // code to be executed if condition is true
}
```

Example

Let's write a program that checks if a number is positive and prints a message if it is.

```
public class SimpleIfExample
{
   public static void main(String[] args)
  {
      int number = 10;
      if (number > 0)
     {
         System.out.println("The number is positive.");
      }
   }
}
```

Explanation

Condition: number > 0

This condition checks if the value of number is greater than 0

Code to be executed if condition is true:

```
System.out.println("The number is positive.");
```

This line will be executed if the condition number > 0 evaluates to true.

In this example, since number is 10 (which is greater than 0), the program will print "The number is positive." to the console.

Another Example

Here's another example where the program checks if a number is even.

```
public class SimpleIfExample {
   public static void main(String[] args) {
      int number = 8;
      if (number % 2 == 0) {
         System.out.println("The number is even.");
      }
   }
}
```

IF–ELSE EXPRESSIONS

The if-else statement allows you to execute one block of code if a condition is true and another block of code if the condition is false. This provides a way to handle binary decision-making in your programs i.e. selecting one alternative between two.

Syntax

```
if (condition)
{
   // code to be executed if condition is true
}
else
{
   // code to be executed if condition is false
}
```

Example 1: Checking if a Number is Positive or Negative

```
public class IfElseExample
{
   public static void main(String[] args)
   {
      int number = -10;
      if (number >= 0) {
         System.out.println("The number is positive.");
```

```
      }
      else  {
         System.out.println("The number is negative.");
      }
   }
}
```

Explanation

Condition: number >= 0

This condition checks if the value of number is greater than or equal to 0.

Code to be executed if condition is true:

```
System.out.println("The number is positive.");
```

Code to be executed if condition is false:

```
System.out.println("The number is negative.");
```

In this example, since number is -10 (which is less than 0), the program will print "The number is negative." to the console.

Example 2: Checking if a Number is Even or Odd

```
public class IfElseExample
{
   public static void main(String[] args)
   {
      int number = 7;
      if (number % 2 == 0)
      {
         System.out.println("The number is even.");
      }
      else
      {
         System.out.println("The number is odd.");
      }
   }
}
```

Here, number is 7 (which is odd), the program will print "The number is odd." to the console.

Example 3: Checking if a Person is Eligible to Vote

```
public class IfElseExample
{
```

```
    public static void main(String[] args)
   {
       int age = 17;
       if (age >= 18)
       {
           System.out.println("You are eligible to vote.");
       }
       else
       {
           System.out.println("You are not eligible to vote.");
       }
   }
}
```

Here, age is 17 (which is less than 18), the program will print "You are not eligible to vote." to the console.

TERNARY OPERATOR: The ternary operator in Java is a shorthand way of writing an if-else statement. It is also known as the conditional operator. The ternary operator consists of three parts: a condition, a result for true, and a result for false.

The syntax is as follows:

result = condition ? value1 : value2;

- condition: The expression to be evaluated, which should return a boolean value (true or false).

- value1: The value assigned to the result if the condition is true.

- value2: The value assigned to the result if the condition is false.

Example: Determining if a Number is Even or Odd

```
public class TernaryExample
{
   public static void main(String[] args)
   {
       int number = 5;
       String result = (number % 2 == 0) ? "Even" : "Odd";
       System.out.println("The number is " + result);
   }
}
```

Explanation

Condition: (number % 2 == 0)

This checks if number is divisible by 2.

valueIfTrue: "Even"

This is returned if the condition is true.

valueIfFalse: "Odd"

This is returned if the condition is false.

IF – ELSE – IF LADDER EXPRESSION

The if-else-if ladder in Java is a way to chain multiple conditions together. It allows the program to test various conditions sequentially and execute the corresponding block of code for the first condition that evaluates to true. If none of the conditions is true, the code in the else block (if provided) will be executed.

Syntax

```
if (condition1) {
   // code to be executed if condition1 is true
}
else if (condition2) {
   // code to be executed if condition2 is true
}
else if (condition3)  {
   // code to be executed if condition3 is true
}
else {
   // code to be executed if none of the above conditions are true
}
```

Example: Checking Grade Based on Marks

```
public class IfElseIfLadderExample
{
   public static void main(String[] args) {
       int marks = 85;
       if (marks >= 90) {
           System.out.println("Grade: A");
      }
      else if (marks >= 80) {
           System.out.println("Grade: B");
      }
     else if (marks >= 70)  {
           System.out.println("Grade: C");
```

```
    }
    else if (marks >= 60)  {
        System.out.println("Grade: D");
     }
     else {
        System.out.println("Grade: F");
      }
   }
}
```

In this example, since marks is 85 (which is greater than or equal to 80 but less than 90), the program will print "Grade: B" to the console.

SWITCH STATEMENT

In Java, the switch statement provides a way to execute different actions based on the value of a variable. It's useful when a single variable or expression has to compare against multiple values.

Here’s an overview of how the switch statement works, its syntax, and some examples:

Syntax

```
switch (expression)
{
   case value1:  // code to be executed if expression equals value1
                 break;
   case value2:  // code to be executed if expression equals value2
                 break;
   --------------
   --------------
   --------------
   default:
      // code to be executed if none of the above cases are true
}
```

- expression: A variable or expression whose value is evaluated.

- case valueX: Each case specifies a possible value of the expression.

- break: Ends the switch statement. Without break, execution continues to the next case (known as fall-through behavior).

- default: Optional. Executes if none of the case values match the expression.

How It Works

- The switch statement evaluates the expression once.
- It then compares the value of the expression with the values specified in each case.
- If a match is found, the corresponding block of code executes.
- If no match is found, and a default block is provided, that block executes.
- If no default block is provided and no matches are found, the switch statement simply ends.

Example : Days of the Week

```
public class SwitchExample {
    public static void main(String[] args) {
        int day = 3;
        String dayName;
        switch (day) {
            case 1:
                dayName = "Monday";
                break;
            case 2:
                dayName = "Tuesday";
                break;
            case 3:
                dayName = "Wednesday";
                break;
            case 4:
                dayName = "Thursday";
                break;
            case 5:
                dayName = "Friday";
                break;
            case 6:
                dayName = "Saturday";
                break;
            case 7:
                dayName = "Sunday";
                break;
            default:
                dayName = "Invalid day";
```

```
            break;
        }
        System.out.println("The day is " + dayName);
    }
}
```

Explanation

- ***day*** Variable holding the value 3.
- The switch statement evaluates ***day***.
- Since ***day*** is 3, it matches case 3.
- Therefore, ***dayName*** is set to "Wednesday".
- The break statement ends the switch statement.

LOOPING STATEMENTS OR ITERATIVE STATEMENTS

Looping statements in Java are used to execute a block of code repeatedly based on a condition. The main types of loops in Java are:

1. while loop
2. do-while loop
3. for loop
4. enhanced for loop (for-each loop)

WHILE LOOP:

A while loop in Java is a control flow statement that allows you to repeatedly execute a block of code as long as a certain condition is true. It's essentially a looping construct that keeps running the code within it until a specified condition becomes false.

Here's how a while loop works:

1. Initialization: You start by setting up a loop counter variable and initializing it with a value.

2. Condition Check: The loop evaluates a condition. If the condition is true, the loop continues.

3. Code Execution: If the condition is true, the code block inside the loop's body executes. This code block can contain any valid Java statements.

4. Update: After the code executes, the loop counter variable typically gets updated (incremented or decremented) to move the loop towards termination.

5. Loop Back: The loop goes back to step 2 and checks the condition again. This cycle continues as long as the condition remains true.

Here's a basic syntax of a while loop in Java:

```
while (condition)
{
  // code to be executed
}
```

Examples:

Printing numbers from 1 to 10:

```
int i = 1;
while (i <= 10)
{
  System.out.println(i);
  i++;
}
```

Summing numbers from 1 to 10:

```
int sum = 0;
int i = 1;
while (i <= 10)
{
  sum += i;
  i++;
}
System.out.println("The sum is: " + sum);
```

DO – WHILE LOOP:

The do-while loop in Java is another looping construct similar to the while loop, but with a key difference in condition checking.

Functionality:

- A **do-while** loop guarantees that the code block inside the loop executes at least once, even if the condition initially evaluates to false. Use do-while loops when you need the code to execute at least once, even if the condition might not be true initially.
- Unlike a **while** loop that checks the condition before executing the code, a do-while loop executes the code first and then checks the condition.

Syntax:

```
do
{
  // code to be executed
        ---------------
        ---------------
        ---------------
} while (condition);
```

Explanation:

1. Code Execution: The code within the curly braces {} executes first, regardless of the condition.

2. Condition Check: After the code execution, the loop evaluates the condition.

3. Loop Continuation: If the condition is true, the entire loop (code execution and condition check) repeats.

4. Loop Termination: The loop continues as long as the condition remains true. Once the condition becomes false, the loop stops.

Example:

```
int number = 0;
do
{
  number++;
  System.out.println(number);
} while (number < 5);
```

FOR LOOP:

A for loop in Java is a control flow statement that allows you to efficiently execute a block of code a specific number of times. It provides a concise way to handle repetitive tasks where you know the number of iterations before hand.

Syntax:

```
for (initialization; condition; increment/decrement)
{
        // code to be executed
}
```

Here is how it works:

1. Initialization: You typically set up a loop counter variable and initialize it with a starting value.

2. Condition: The loop continues to execute as long as this condition remains true.

3. Increment/Decrement: After each iteration, this expression updates the loop counter variable (usually incrementing it).

Example:

```
for (int i = 1; i <= 5; i++)
{
  System.out.println(i);
}
```

Explanation:

1. We declare an integer variable i and initialize it to 1 (initialization).
2. The condition i <= 5 checks if i is less than or equal to 5.
3. Inside the loop, we print the current value of i.
4. After each iteration, i is incremented by 1 (i++).
5. The loop repeats as long as the condition i <= 5 is true. Once i becomes 6, the condition becomes false, and the loop terminates.

FOR – EACH LOOP:

The for-each loop, also known as the enhanced for loop, is a concise way to iterate through elements in arrays and collections (like ArrayList) in Java. It provides a simpler syntax compared to traditional for loops.

Syntax:

```
for (dataType element : array/collection)
{
        // code to be executed for each element
}
```

Example:

```
int[] numbers = {1, 2, 3, 4, 5};
for (int number : numbers)
{
        System.out.println(number);
}
```

Explanation:

1. We have an integer array numbers.
2. The for-each loop declares an integer variable number.
3. It iterates through each element in the numbers array. For each iteration, the current element is assigned to the number variable.

4. Inside the loop, we print the value of number.

5. The loop automatically continues to the next element in the array until it reaches the end.

NESTED FOR LOOP:

A nested for loop in Java refers to a loop structure where you have one for loop placed entirely inside the body of another for loop. This creates a nested iteration process, allowing you to execute a block of code multiple times for different values set by the outer and inner loops. The outer loop controls the outer iterations, and the inner loop controls the inner iterations within each outer loop iteration.

Example:

```
for (int i = 1; i <= 3; i++)  // Outer loop iterates 3 times
{
  for (int j = 1; j <= 2; j++)  // Inner loop iterates 2 times in each outer loop iteration
  {
        System.out.println("Outer loop: " + i + ", Inner loop: " + j);
  }
}
```

Explanation:

1. The outer loop (i) iterates three times for different values 1, 2, 3

2. Inside each iteration of the outer loop, the inner loop (j) iterates twice for the different values 1, 2

3. For each combination of outer and inner loop iterations, the code inside the innermost curly braces executes.

- So, it will print:
  - Outer loop: 1, Inner loop: 1
  - Outer loop: 1, Inner loop: 2
  - Outer loop: 2, Inner loop: 1
  - Outer loop: 2, Inner loop: 2
  - Outer loop: 3, Inner loop: 1
  - Outer loop: 3, Inner loop: 2

Sample Use Cases:

- Multiplication tables: Nested loops can be used to generate multiplication tables, iterating through rows and columns.
- 2D arrays: They are commonly used to process two-dimensional arrays, where the outer loop iterates through rows and the inner loop iterates through elements in each row (columns).

- Patterns: Nested loops can be used to create text or graphical patterns by controlling the output based on the loop iterations.

LOOP CONTROL STATEMENTS

Loop control statements in Java are used to change the normal sequence of loop execution. They provide control over the flow of loops, allowing developers to manipulate the iteration process. The primary loop control statements in Java are break, continue, and return.

BREAK STATEMENT: The break statement in Java is used to exit from a loop before it has completed its normal execution. When a break statement is encountered, the control immediately exits the loop and the program continues with the next statement following the loop.

Within a loop:

```
while (condition)
{
   -----------
   if (Condition)
   {
      break;
   }
   ------------
}
```

Example:

```
for (int i = 0; i < 10; i++) {
   if (i == 5) {
      break; // Exit the loop when i is 5
   }
   System.out.println("Iteration: " + i);
}
System.out.println("Loop terminated.");
```

Output:

```
Iteration: 0
Iteration: 1
Iteration: 2
Iteration: 3
Iteration: 4
Loop terminated.
```

In this example, the loop terminates when i equals 5, and the message "Loop terminated." is printed.

Using break in Nested Loops

When used in nested loops, the break statement will only exit the innermost loop in which it is placed.

Example:

```
for (int i = 0; i < 3; i++) {
   for (int j = 0; j < 3; j++) {
      if (j == 1)
      {
         break; // Exit the inner loop when j is 1
      }
      System.out.println("i: " + i + ", j: " + j);
   }
}
```

Output:

```
i: 0, j: 0
i: 1, j: 0
i: 2, j: 0
```

In this example, the inner loop terminates when j equals 1, but the outer loop continues its execution.

The break statement is used to exit a loop or switch statement prematurely. In loops, it can be used to exit the loop based on a specific condition. When used in nested loops, it only exits the innermost loop containing the break statement.

CONTINUE STATEMENT

The continue statement in Java is used to skip the current iteration of a loop and proceed with the next iteration. It is typically used to bypass certain parts of the loop body under specific conditions, without terminating the entire loop.

It can be used in for, while, and do-while loops.

Within a loop:

```
while (condition)
{
        // code
        if (Condition)
        {
                continue;
        }
        // more code
}
```

Example with a for Loop

The continue statement can be used in a for loop to skip the current iteration when a condition is met.

Example:

```
for (int i = 0; i < 10; i++)
{
   if (i % 2 == 0)
   {
         continue; // Skip the rest of the loop body for even numbers
   }
   System.out.println("Odd number: " + i);
}
```

Output:

Odd number: 1
Odd number: 3
Odd number: 5
Odd number: 7
Odd number: 9

In this example, the loop skips the System.out.println statement for even numbers, only printing odd numbers.

Using continue in Nested Loops

When used in nested loops, the continue statement will only skip the current iteration of the innermost loop in which it is placed.

Example:

```
for (int i = 0; i < 3; i++)
{
   for (int j = 0; j < 3; j++)
   {
      if (j == 1)
      {
         continue; // Skip the rest of the loop body when j is 1
      }
      System.out.println("i: " + i + ", j: " + j);
   }
}
```

Output:

i: 0, j: 0

i: 0, j: 2

i: 1, j: 0

i: 1, j: 2

i: 2, j: 0

i: 2, j: 2

In this example, the inner loop skips the **System.out.println statement** when j equals 1, but the outer loop continues its execution.