Object Oriented Programming Using

MCA 2nd Semester

Module-1:
OOPS CONCEPTS AND JAVA PROGRAMMING

PRAMOD NAIK

What is Programming?

What is Programming Language?

PRAMOD NAIK

What is Programming? : Writing a Program

What is Programming Language? : Is a Collection of **Tokens**.

What is a Token:

A Token is a **smallest individual units** of a Program.

In Java, tokens are the smallest **elements** of a program that are **meaningful to the compiler**. Java programs are made up of these tokens.

Java Tokens:

1. **Keywords:** Predefined Words like, class, if, switch etc
2. **Identifiers:** Name given to the programming elements such as class, variables and so on.
3. **Literals:** Constant values that are assigned to variables
4. **Operators:** Symbols that perform operations on variables and values
5. **Separators** (or Delimiters): Characters that separate tokens in the code. Ex: {}, [], (), and so on
6. **Comments:** Text within the code that is not executed but provides explanations or annotations.

Example:

```
public class Example { // keyword: public, class; identifier: Example; separator: {
    public static void main(String[] args) {
// keywords: public, static, void; identifier: main, String, args; separator: (), {}
        int number = 42; // keyword: int; identifier: number; literal: 42; operator: =
        double pi = 3.14; // keyword: double; identifier: pi; literal: 3.14; operator: =
        boolean isJavaFun = true; // keyword: boolean; identifier: isJavaFun; literal: true; operator: =

        if (isJavaFun) { // keyword: if; identifier: isJavaFun; separator: ()
            System.out.println("Java is fun!");
// identifier: System, out, println; literal: "Java is fun!"; separator: ();
        } // separator: }
    } // separator: }
} // separator: }
```

PRAMOD NAIK

What is Java:

Java is a High Level, General Purpose, Object Oriented, Platform Independent, Compiled and Interpreted, Statically Typed, Highly Secured programming language.

Java is Developed By: **James Gosling**, **Patrick Naughton**, and **Mike Sheridan**

History of Java:

Early Beginnings

1991: Project Initiation

Java began as a project called "**Oak**" by Sun Microsystems, led by James Gosling, Mike Sheridan, and Patrick Naughton. They were part of a team at **Sun Microsystems** that initiated the project in the early 1990s. The goal was to develop a language for **embedded systems in consumer electronics,** , and it eventually evolved into the Java programming language we know today.

1992: Green Project

The team, known as the Green Team, worked on creating a platform-independent language. They aimed to create a language that could run on various devices, including **televisions**, **toasters**, and other consumer electronics.

1994: Transition to the Web

The team realized the potential of their new language for the burgeoning **World Wide Web**, where the ability to run the same program on different platforms was highly desirable. They renamed **Oak** to **Java**, inspired by **Java coffee**, which was consumed in large quantities by the developers.

Official Launch

1995: Public Introduction

Java 1.0 was officially released by **Sun Microsystems**. The "Write Once, Run Anywhere" (**WORA**) capability became a significant selling point, allowing Java programs to run on any device with a Java Virtual Machine (**JVM**).

Evolution and Growth

1996: Java Development Kit (JDK) 1.0

Sun Microsystems released JDK 1.0, providing developers with the **tools** **needed to develop Java applications**.

1997-1999: Rapid Evolution

Java 1.1 was released in 1997, introducing new features like inner classes and JavaBeans.

In 1998, Java 2 (formerly known as JDK 1.2) introduced major enhancements to the platform, including the Swing **graphical API** and **the Collections framework.**

Java Community Process

1999: Establishment of the Java Community Process (JCP)

The JCP was established to allow for the participation of the **broader Java community** in the development and evolution of Java standards and specifications.

Open Source and Later Versions

2006: Open Sourcing of Java

Sun Microsystems announced that Java would be released under the GNU General Public License (GPL), making it open-source and allowing the community to **contribute** to its development.

2009: Acquisition by Oracle

Oracle Corporation acquired Sun Microsystems, becoming the steward of Java. This led to concerns about the future direction of Java, but Oracle continued to develop and support the language.

2011-Present: Ongoing Evolution

Java 7 was released in 2011, introducing new language features and performance improvements.

Java 8, released in 2014, brought significant changes, including the introduction of **lambda expressions**, the Stream API, and the new date and time API.

Subsequent versions (Java 9 through Java 20) have continued to add new features, enhance performance, and improve the language's capabilities.

Java 22, officially released on **March 19, 2024**

Flavours of Java:

1. Java Standard Edition (Java SE)

Purpose: Java SE provides the core functionality for **general-purpose programming.** It includes the Java Development Kit (JDK), which contains the Java Runtime Environment (JRE), a compiler (javac), and various development tools.

2. Java Enterprise Edition (Java EE), now Jakarta EE (Web Applications)

Purpose: Java EE is designed for building large-scale, distributed, and component-based applications in the enterprise environment.

3. Java Micro Edition (Java ME) (Android)

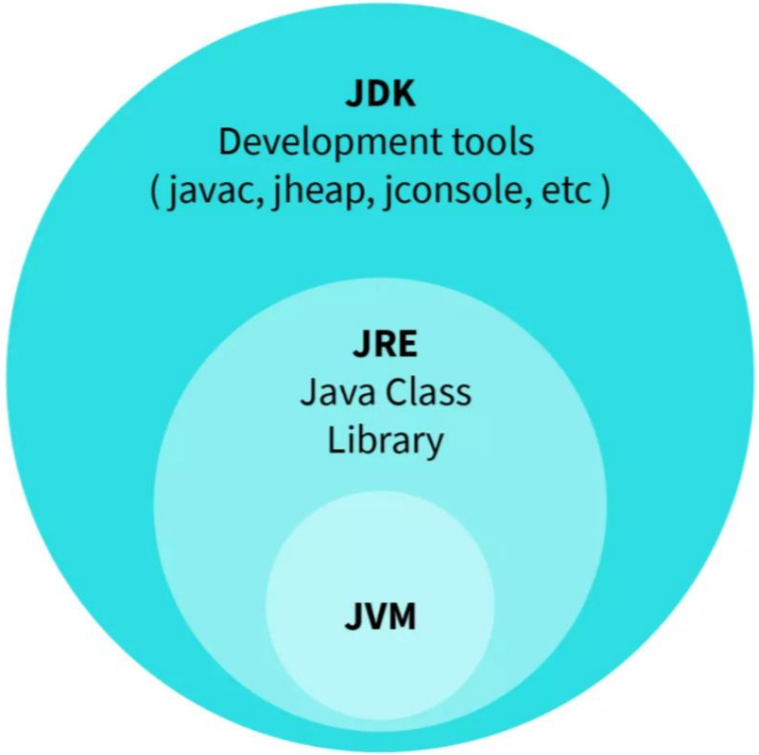
Purpose: Java ME is tailored for resource-constrained devices like embedded systems, mobile phones, and Internet of Things (IoT) devices.

Features of Java:

1. **Object-Oriented:** Java follows an object-oriented programming model, which allows for the creation of modular, reusable code. It supports features like inheritance, polymorphism, encapsulation, and abstraction.
2. **Platform-Independent**: Java's "write once, run anywhere" capability is facilitated by the Java Virtual Machine (JVM). Java code is compiled into bytecode, which can run on any system equipped with a JVM, making Java applications highly portable.
3. **Secure**: Java provides a secure environment through its runtime environment.
4. **High Performance**: Although Java is an interpreted language, the use of Just-In-Time (JIT) compilers helps improve its performance by converting bytecode into native machine code at runtime.
5. **Multi-Threaded:** A thread is the smallest unit of a process that can be scheduled and executed independently by the operating system. It is a **lightweight process** that shares the same memory space and resources of the parent process but can execute code concurrently.
6. **Compiled & Interpreted:** Java has both Compiler and Interpreter.

JDK (Java Development Kit)

The Java Development Kit (JDK) is a **software development environment** used for developing Java applications and applets. It provides the necessary **tools**, **libraries**, and **resources** to **create** and **execute** Java programs.

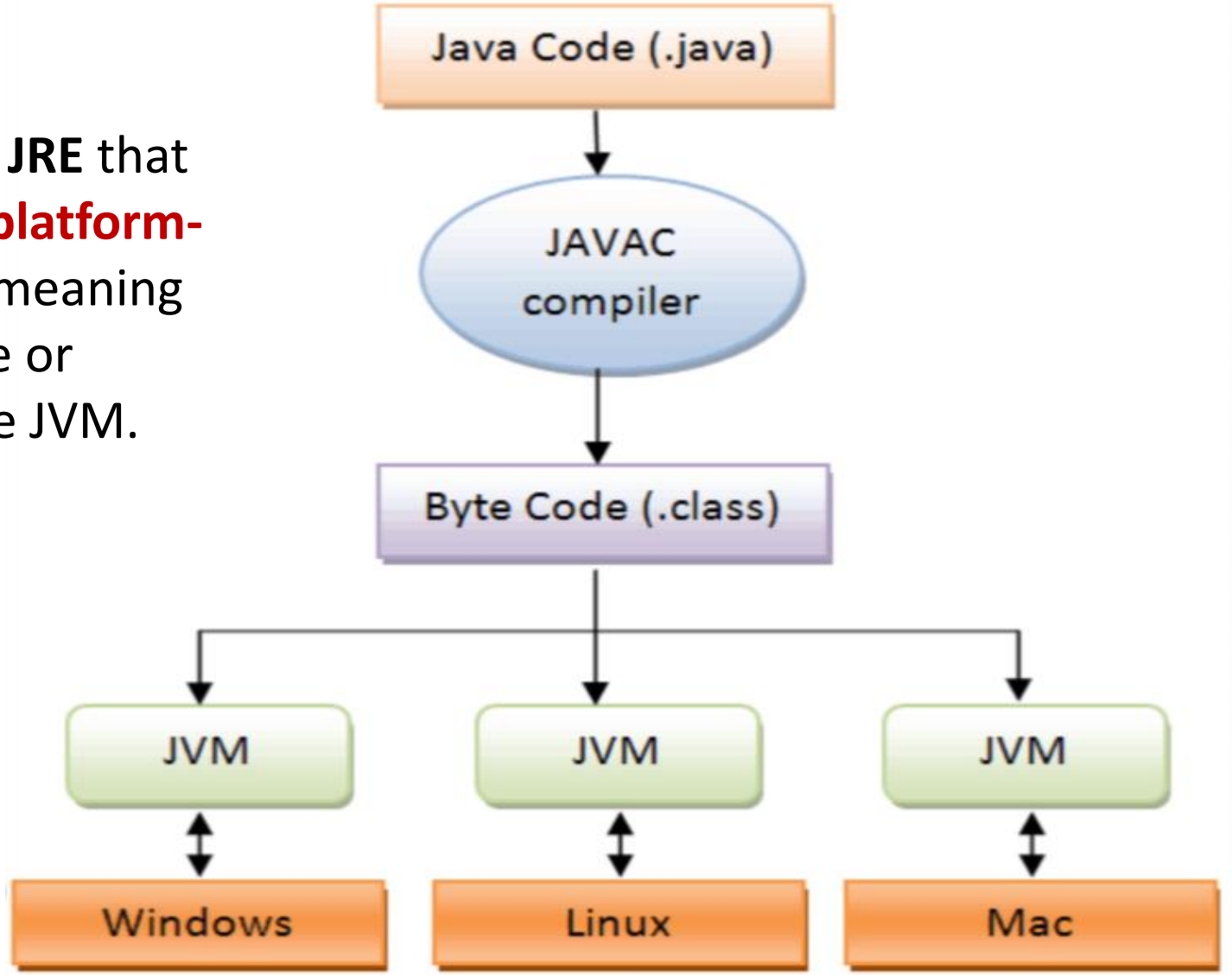


JRE (Java Runtime Environment)

The Java Runtime Environment (JRE) is a **software package** that provides the necessary **libraries**, **Java Virtual Machine** (JVM), and other components to **run applications** written in the Java programming language. It is a part of the Java Development Kit (JDK) but can also be installed separately. The JRE is designed to provide an environment for executing Java applications, not for developing them.

Java Virtual Machine (JVM) :

The JVM is the **core component** of the **JRE** that **executes Java bytecode**. It provides a **platform-independent** execution environment, meaning Java applications can run on any device or operating system that has a compatible JVM.



How to Run the Java Program:

1. Write Your Java Program:

Example: HelloWorld.java

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

2. Compile the Java Program using javac:

```
javac HelloWorld.java
```

3. Run the Compiled Java Program using java:

```
java HelloWorld
```

PRAMOD NAIK

Comments, Data types, Variables

PRAMOD NAIK

Comments:

In Java, comments are annotations in the source code that are ignored by the compiler and the runtime environment. They are intended to make the code easier to understand for humans by explaining the purpose, logic, or structure of the code.

There are 3 types of comments in Java:

1. **Single-Line Comments:** These comments start with **//** and continue to the end of the line.

```
// This is a single-line comment
int x = 5; // This comment explains the variable declaration
```

2. Multi-Line Comments: These comments start with **/*** and end with ***/**. They can span multiple lines.

```
/*
 * This is a multi-line comment.
 * It can span multiple lines.
 */
int y = 10;
```

3. Documentation Comments: These comments start with **/**** and end with ***/**. They are used to generate documentation using the **Javadoc** tool.

Javadoc is a tool provided by the Java Development Kit (JDK) that generates **HTML documentation** from Java source code with special comments called documentation comments.

Run Javadoc: **javadoc** -d doc <source-files>

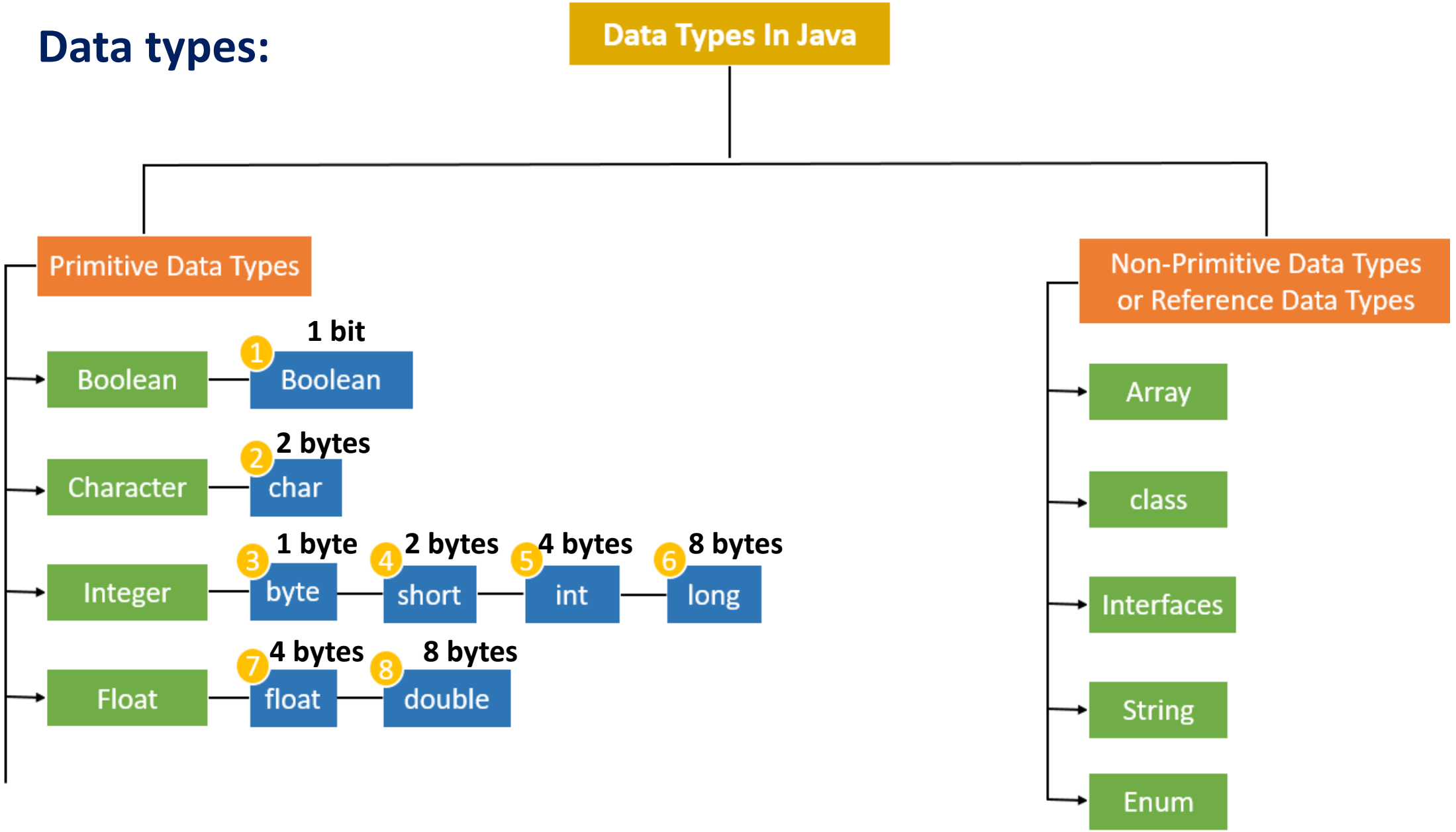
Example:

```java
/**
 * I am writing this program to test the Javadoc.
 * <p>
 * This class currently includes only the main method where the execution
 * starts.
 * </p>
 *
 * @version 1.0
 * @since 2024-07-19
 */
public class Document {
    /**
     * This is the main method which makes use of boolean data types and checks the
     * conditon using if.
     *
     *
     * @param args Unused.
     */
    public static void main(String[] args) {
        boolean isJavaFun = true;

        if (isJavaFun) {
            System.out.println("Java is fun!");
        }
    }

}
```

Run:

javadoc -d doc Document.java

Data types:


Data Types In Java
Primitive Data Types
Boolean
1 bit
1
Boolean
Character
2 bytes
2
char
Integer
1 byte
3
byte
2 bytes
4
short
4 bytes
5
int
8 bytes
6
long
Float
4 bytes
7
float
8 bytes
8
double
Non-Primitive Data Types or Reference Data Types
Array
class
Interfaces
String
Enum

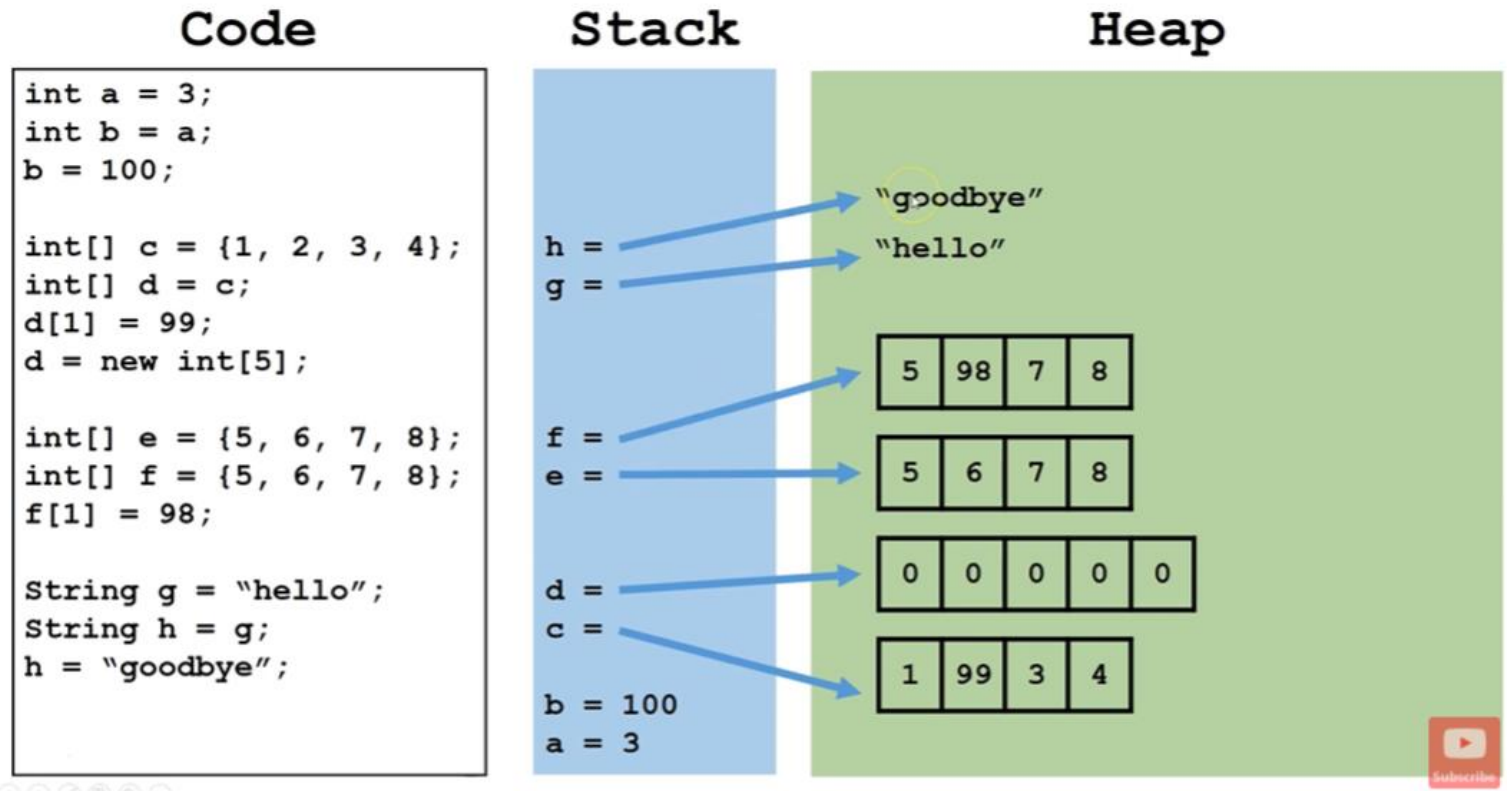
Primitive Types:

Java has **8** primitive data types, which are the most basic data types available within the language. They are **predefined by the language** and **named by a reserved keyword**.

Data Type	Size	Description
`byte`	1 byte	Stores whole numbers from -128 to 127
`short`	2 bytes	Stores whole numbers from -32,768 to 32,767
`int`	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
`long`	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`float`	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
`double`	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal digits
`boolean`	1 bit	Stores true or false values
`char`	2 bytes	Stores a single character/letter or ASCII values

Non-Primitive or Reference Types:

In Java, non-primitive or reference types are more **complex data types** that refer to objects and arrays. Unlike primitive types, which store actual values, reference types **store references to memory locations** where the **objects are stored.**



String:

In Java, a String is an **object** (of String class) that represents a **sequence of characters**. Strings are widely used in Java programming and are a crucial part of the Java language. The String class is **immutable**, **meaning that once a String object is created**, its **value cannot be changed.**

Key Features of Strings in Java

1. **Immutable:** The content of a String object cannot be changed after it is created. Any modification of a string results in the creation of a new String object. So we can reassign new sequence of character to the String Object but we cannot modify the content or char of the String directly by using index.
2. **String Pool:** Java uses a special memory region called the string pool to store string literals. When a new string literal is created, Java checks the string pool first. If the string already exists in the pool, a reference to the existing string is returned, otherwise, the new string is added to the pool.
3. **String Class:** The String class is part of the **java.lang** package and is **implicitly imported.**

Creating Strings

1. Using String Literals:

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

2. Using the new Keyword:

```
String greeting = new String("Hello, World!");
```

Commonly Used Methods: String name = “Shridevi College”

Function	Working	Example
length()	Returns the length of the string.	name.length(); returns 16
charAt(int index)	Returns the character at the specified index.	name.charAt(0); returns ‘S’
substring(int beginIndex, int endIndex)	Returns a substring from beginIndex to endIndex.	name.substring(0,3); returns “Shri”
contains(CharSequence sequence)	Checks if the string contains the specified sequence.	name.contains(“Shri”); returns true
indexOf(String str)	Returns the index of the first occurrence of the specified substring.	name.indexOf(“College”); returns 9;

toUpperCase()	Converts all characters in the string to uppercase.	name.toUpperCase(); return “SHRIDEVI COLLEGE”
toLowerCase()	Converts all characters in the string to lowercase.	name.toLowerCase(); return “Shridevi college”

Array:

An array in Java is a **data structure** that allows you to store multiple values of the **same type** in a single variable. Arrays are useful for managing collections of data that can be accessed using an index.

Key Features of Arrays in Java:

1. **Fixed Size:** Once an array is created, its size cannot be changed. You must specify the number of elements it will hold when you create it.

2. **Indexed Access:** Each element in an array can be accessed using its index, with the first element at index 0 and the last element at index **length-1**.

3. **Homogeneous Elements:** All elements in an array must be of the same type, such as int, String, boolean, etc.

4. **Zero-Based Indexing:** The index of the first element is 0, and the index of the last element is **length-1.**

Declaring and Initializing Arrays:

Declaring-> Syntax:

```
type[] arrayName;
```

Example:

```
int[] numbers; // Declares an array of integers
String[] names; // Declares an array of strings
```

Initializing-> Syntax:

```
arrayName = new type[size];
```

Example:

```
numbers = new int[5]; // Creates an array of 5 integers
names = new String[3]; // Creates an array of 3 strings
```

PRAMOD NAIK

We can combine declaration and initialization in one step:

Syntax:

```
type[] arrayName = new type[size];
```

Example:

```
int[] numbers = new int[5]; // Declares and initializes an array of 5 integers
String[] names = new String[3]; // Declares and initializes an array of 3 strings
```

Directly Assign the Values and the size will be detected by the compiler:

Syntax:

```
type[] arrayName = {value1, value2, ..., valueN};
```

Example:

```
int[] numbers = {1, 2, 3, 4, 5};
// Creates and initializes an array of integers with values
String[] names = {"Alice", "Bob", "Charlie"};
// Creates and initializes an array of strings with values
```

Enum: (Creating our Own Type)

In Java, an **enum** (short for "**enumeration**") is a special data type that enables a variable to be a set of predefined constants. It is used to represent a **fixed set of related constants more efficiently** and **readably**.

Definition: Enums are defined using the **enum keyword**. Each value in an enum is called an **enum constant**.

Example:

```
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}
```

Note:

In Enum we are creating **our own data type** which can hold only the **constant specified** to it while defining it which is called **enum constants**.

Example:

```
public class MyClass {
    public static void main(String[] args) {

        enum Day {
            SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
        };

        Day today = Day.FRIDAY;

        System.out.println(today);
    }
}
```

PRAMOD NAIK

Example: We can have **Functions** and **Constructor** for our Enum

```
public class MyClass {
    public static void main(String[] args) {

        enum Day {
            SUNDAY("Holiday"), MONDAY("Workday");

            private String type;    // type is a Variable of type String

            Day(String type_in) {   // This is a Default Constructor
                this.type = type_in;
            }

            public String getType() {   // This method will return the value of type variable
                return type;
            }
        }

        Day today = Day.MONDAY;          // The constructor will be called automatically.
        System.out.println(today.getType());  // Output: Workday
    }
}
```

PRAN

Class & Interface

PRAMOD NAIK

Evolution of Data Storing Technique:

1. **Variables** (Single Value)
2. **Array** (Multi-Value but Homogeneous)
3. **Structure & Union** (No OOP's and No Methods)
4. **Class**

Procedural and Object Oriented Programming Paradigm:

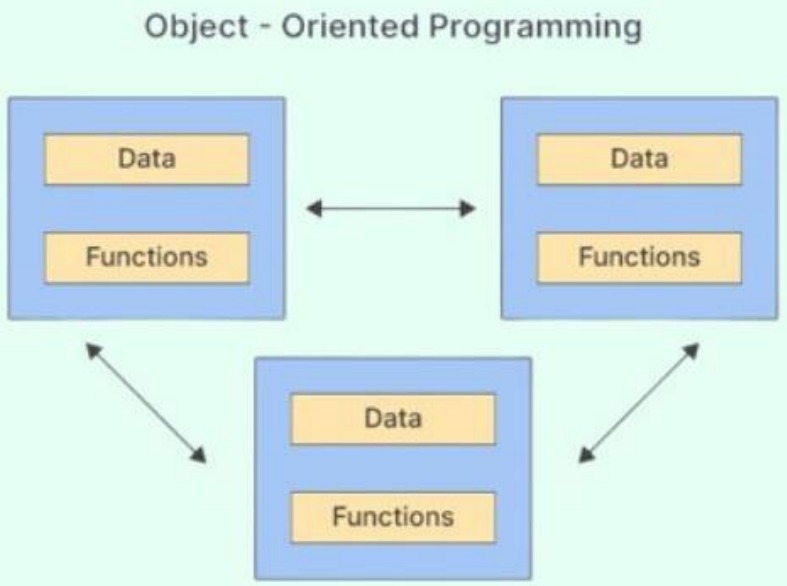
1. Procedural Programming Paradigm:

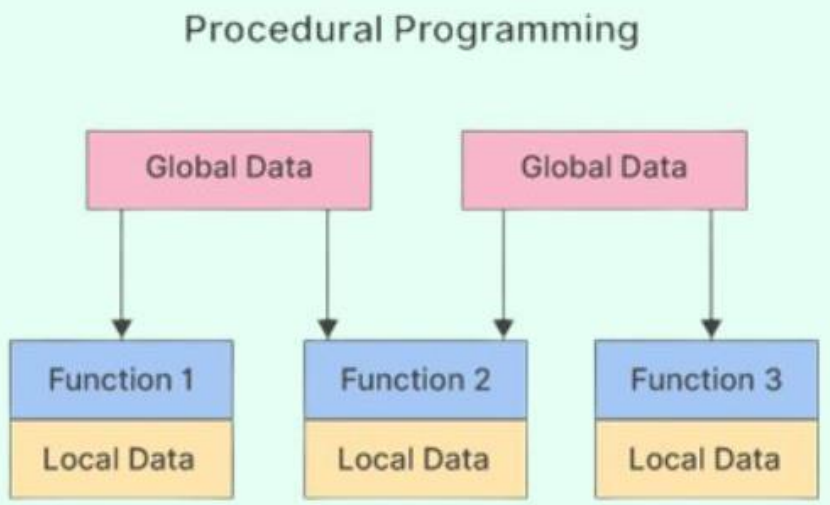
- **Definition:** Procedural programming is a programming paradigm based on the concept of procedure calls, where the program is structured into procedures, also known as functions or routines.
- **Example:** C, Pascal, Fortran.

2. Object-Oriented Programming (OOP) Paradigm:

- **Definition:** Object-oriented programming is a paradigm based on the concept of "objects," which are instances of classes that encapsulate both data and methods.
- **Example:** Java, C++, Python, C#.

OOP and POP


Object - Oriented Programming
Data
Functions
Data
Functions
Data
Functions


Procedural Programming
Global Data
Global Data
Function 1
Local Data
Function 2
Local Data
Function 3
Local Data

Parameter	Object Oriented Programming(OOP)	Procedure Oriented Programming(POP)
Definition	OOP refers to Object Oriented Programming. It deals with **objects** and their **properties**.	POP refers to Procedural Oriented Programming and deals with programs and functions.
Approach	An object-oriented program uses the **bottom-up approach.**	A procedure-oriented program uses the **top-down approach.**
Access Control	Access control is supported by the means of **access modifiers**. The access specifiers such as public, private and protected are used.	No access modifiers are supported.
Data Hiding	Data can be **hidden** using **encapsulation**.	There is no data-hiding mechanism. Data is globally accessible, as there are no access specifiers.
Entity Linkage	Object functions are linked through **message passing**.	Parameter passing is involved in message passing.
Polymorphism	Method **overloading** and method **overriding** are used in OOP to achieve polymorphism.	POP does not support polymorphism.
Virtual Function and Inheritance	OOP supports **inheritance** and virtual functions and virtual classes via it.	There is no concept of inheritance in POP and neither does it support the use of virtual classes or virtual functions.
Code Reuse	OOP supports **code reusability**.	No code reusability is provided by POP.
Operator Overloading	It is allowed in OOP.	Operator overloading is not allowed in POP.

Top-Down:

The top-down approach is a method of designing a system by starting from the highest level of abstraction and progressively breaking it down into more detailed components. In this approach, the overall **structure** and **functionality** of the system are **defined first**, and then specific components and details are developed.

Starts with an overview and breaks it down. **It's like starting with the full picture and filling in the details.**

Bottom-Up:

The bottom-up approach is a method of designing a system by starting with the most basic and fundamental components and gradually integrating them to form higher-level systems. In this approach, small, independent modules or components are developed first, and then they are combined to build the complete system.

Starts with small details and builds up. It's like gathering all the pieces and then assembling the full picture.

OOP's concepts:

Object-Oriented Programming (OOP) is a **programming paradigm** that uses objects and classes to design and develop applications. In Java, OOP principles are fundamental to creating reusable, scalable, and maintainable code.

The main principles of OOP in Java:

1. Class & Objects
2. Encapsulation
3. Inheritance
4. Polymorphism
5. Data Abstraction

Note:

1. Class
2. Data Member OR Variable
3. Member Function OR Function OR Method OR Sub-Routine

1. Class & Objects

In Java, a **class** is a **blueprint** for creating **objects**. It defines a datatype by bundling **data** (attributes) and **methods** (functions) that operate on the data into a single unit.

Class:

Definition: A class is defined using the **class keyword** followed by the class name and a body enclosed in curly braces.

Components: It typically **includes fields** (variables) and **methods** to define the behavior and state of the objects created from the class.

Object

Definition: An object is **an instance of a class**. When a class is **instantiated**, **memory** is allocated for the object, and the **constructor** of the class is called to initialize the object.

Creation: Objects are created using the **new keyword**.

Create and use an object of the Car class:

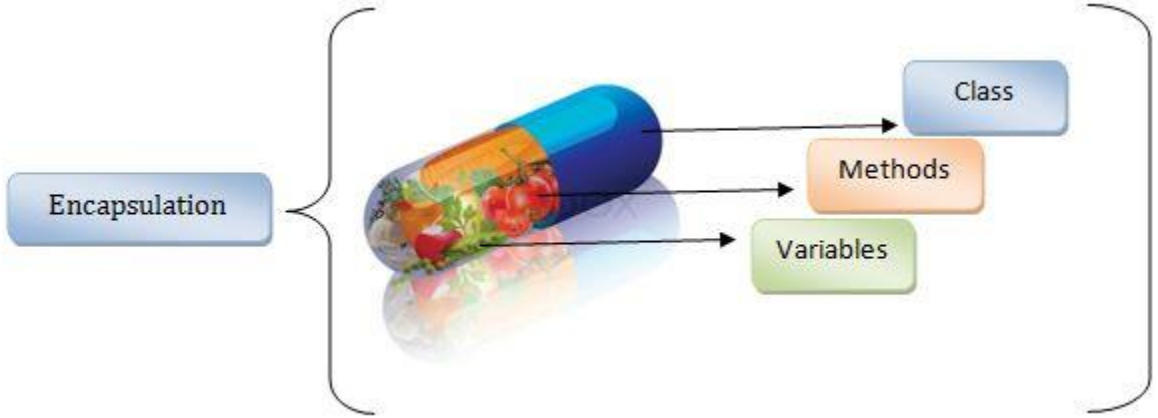
```
// Creating an object of the Car class
Car myCar = new Car("Red", "Toyota", 2021);

// Calling a method on the object
myCar.displayDetails();
```

Example:

```
public class Car {
    // Fields (attributes)
    String color;
    String model;
    int year;

    // Constructor
    public Car(String color, String model, int year) {
        this.color = color;
        this.model = model;
        this.year = year;
    }

    // Method
    public void displayDetails() {
        System.out.println("Car Model: " + model + ", Color: " + color + ", Year: " + year);
    }
}
```

2. Encapsulation

Encapsulation is the mechanism of **bundling** data (fields) and methods (functions) that operate on the data into a **single unit** or **class**, and **restricting access** to some of the object's components. This is achieved using **Access Modifiers** or **Specifiers** such as **private**, **protected**, and **public**.




```java
public class Person {
    private String name;
    private int age;

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

    // Getter for name
    public String getName() {
        return name;
    }

    // Setter for name
    public void setName(String name) {
        this.name = name;
    }

    // Getter for age
    public int getAge() {
        return age;
    }

    // Setter for age
    public void setAge(int age) {
        this.age = age;
    }
}
```

```java
// Superclass
public class Animal {
    public void eat() {
        System.out.println("This animal eats food.");
    }
}

// Subclass
public class Dog extends Animal {
    public void bark() {
        System.out.println("The dog barks.");
    }
}
```

3. Inheritance: (Do not Re-Invent the Wheel)

Inheritance is a mechanism wherein a **new class is derived from** an **existing class**. The new class (child or subclass) inherits the fields and methods of the existing class (parent or superclass), allowing for **code reuse** and the creation of a **hierarchical relationship** between classes.

Benefits of Inheritance:

1. Reusability

- **Code Reusability**: Inheritance promotes code reusability by allowing new classes to use methods and properties of existing classes. This means you can create a new class with minimal additional code.
- **DRY Principle**: It helps adhere to the **"Don't Repeat Yourself"** (DRY) principle, reducing code duplication and making maintenance easier.

2. Maintainability

- **Centralized Changes**: When a method or property in the superclass is modified, all subclasses that inherit from it automatically inherit the changes. This centralizes updates and bug fixes, making the codebase easier to maintain.
- **Consistency**: Changes made in the superclass ensure consistency across all subclasses, reducing the risk of inconsistencies and errors.

3. Extensibility:

- **Adding Functionality**: Inheritance allows for the extension of existing functionality. Subclasses can add new methods or override existing ones to provide specialized behavior while still leveraging the existing code in the superclass.
- **Modular Development**: It promotes modular development by enabling you to build classes in a hierarchical and organized manner.

4. Polymorphism

- **Dynamic Method Binding**: Inheritance is closely related to polymorphism, which allows a subclass to be treated as an instance of its superclass. This enables dynamic method binding (method overriding) where a method call is resolved at runtime, allowing for more flexible and dynamic code.
- **Interface Implementation**: Subclasses can implement methods in different ways, allowing for diverse behavior while maintaining a common interface.

5. Reduced Code Complexity:

- **Simpler Class Definitions**: Subclasses can inherit complex behavior from superclasses, leading to simpler and cleaner class definitions.
- **Focus on Specific Functionality**: Developers can focus on implementing specific functionality in subclasses without worrying about the common functionality already handled by superclasses.

6. Simplified Debugging and Testing

- **Inheritance Hierarchies**: By organizing code into inheritance hierarchies, it becomes easier to understand, debug, and test. Common behavior is isolated in the superclass, which simplifies testing and reduces the likelihood of bugs.
- **Behavioral Consistency**: Testing at the superclass level ensures that all inherited behavior in subclasses is correct, further simplifying the testing process.

7. Improved Organization

- **Logical Structure**: Inheritance provides a logical structure and hierarchy to the codebase. This makes it easier to understand the relationships between different classes and their roles within the application.
- **Clearer Relationships**: It clarifies the relationships between classes, making the system's design and architecture more intuitive.

4. Polymorphism:

In Polymorphism **Poly means many, morphism means form** together it says **More than One Form**.

Polymorphism allows **objects of different classes** to be treated as **objects of a common superclass** (through Method Overriding or Dynamic/ Late Binding). It is mainly achieved through **method overriding** (runtime polymorphism) and **method overloading** (compile-time polymorphism).

Important:

1. **Method** & **Constructor Overloading** & **Operator Overloading** (**compile-time polymorphism**)
2. **Method Overriding** (through Inheritance) (**runtime polymorphism**)

5. Data Abstraction

Abstraction is the concept of **hiding the complex implementation details** and **showing only the essential features of an object**. In Java, abstraction is achieved using **abstract classes** and **interfaces**.

Important:

1. Abstract Class
2. Interface

Control Flow Statements:

Java control flow statements are constructs that **dictate the order in which instructions are executed**.

They can be categorized into 3 main types:

1. **Branching** or **Decision-making** statements
2. Loop statements
3. **Jump** statements

1. Decision-Making Statements:

Decision-Making statements control the flow of execution based on **certain conditions**. These statements allow the program to choose different **paths** of execution based on the evaluation of Boolean expressions.

Main decision-making statements:

1. **if**
2. **if-else**
3. **if-else-if ladder**
4. **nested if**
5. **switch**

1. if statement:

The if statement **evaluates a condition** (a **Boolean expression**). If the condition is **true**, the block of code within the if statement is executed. If the condition is **false**, the block is skipped.

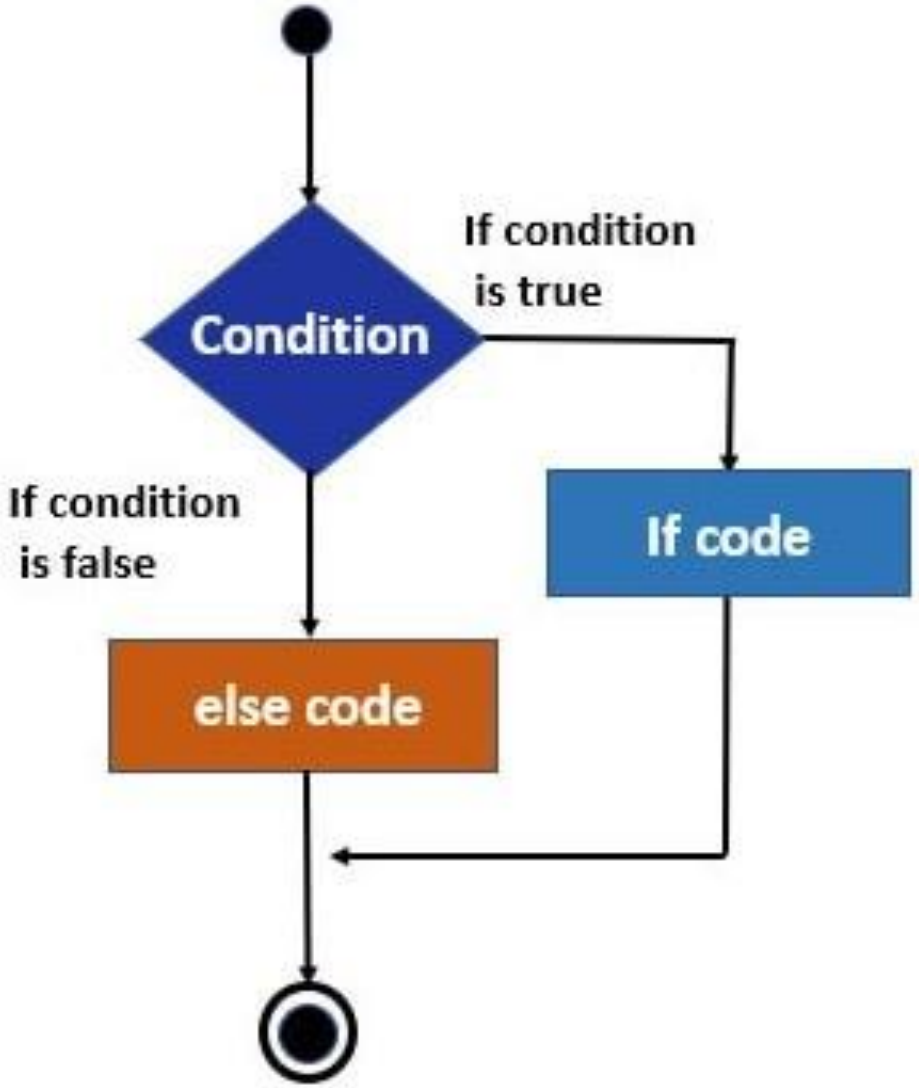
Syntax:

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

Example:

```
int number = 10;
if (number > 0) {
    System.out.println("The number is positive.");
}
```

Simple If
Flow Chart



2. if-else statement:

The if-else statement provides **two blocks of code**: one that executes if the condition is **true** and another that executes if the condition is **false**.

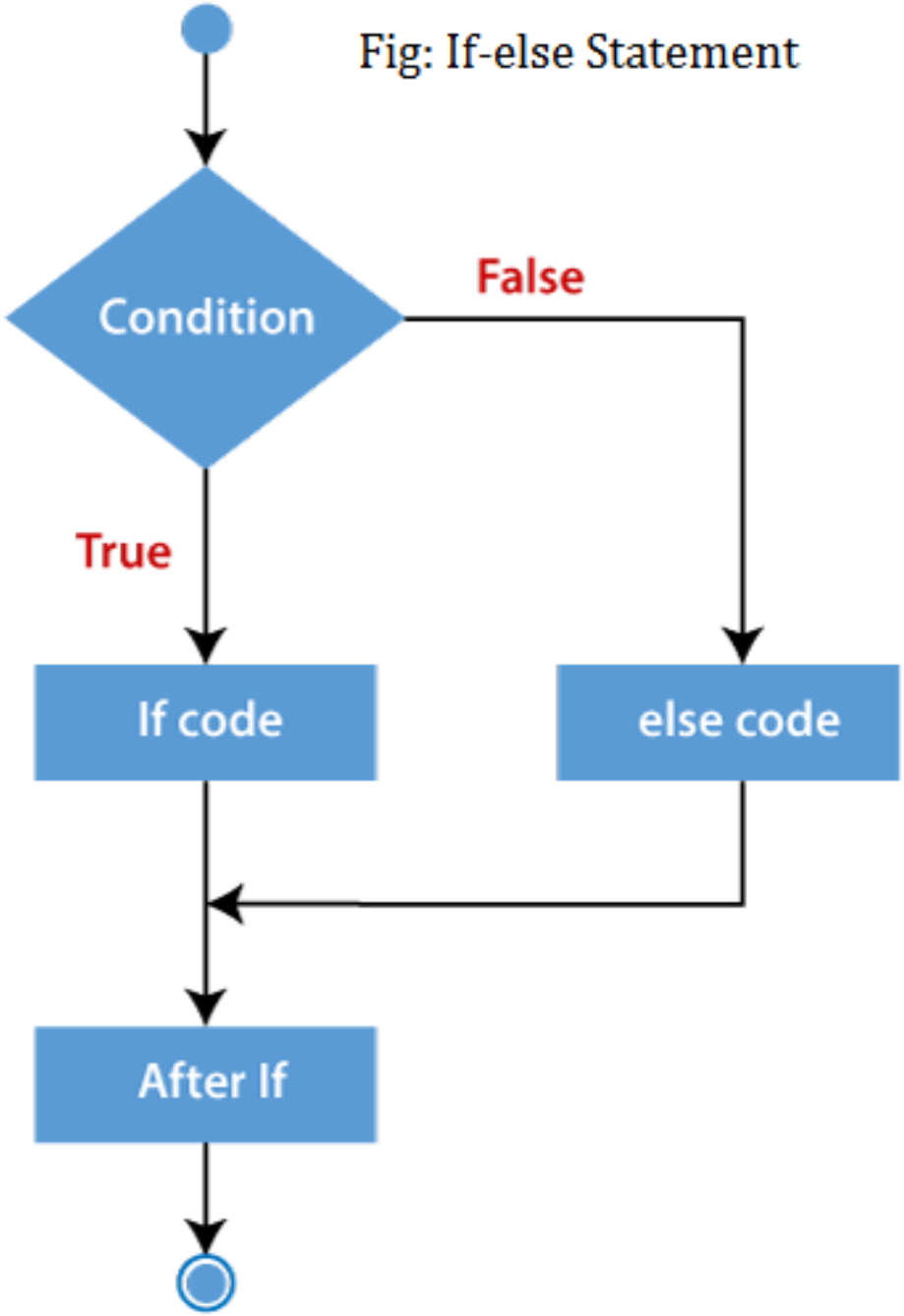
Syntax:

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

Example:

```
int number = -10;
if (number > 0) {
    System.out.println("The number is positive.");
} else {
    System.out.println("The number is negative or zero.");
}
```

if-else Flow Chart



3. if-else-if ladder:

The if-else-if ladder allows for **multiple conditions** to be evaluated **sequentially**. The first condition that evaluates to true will have its corresponding block executed, and the rest of the ladder will be skipped. If none of the conditions are true, the **else** block (if present) will be executed.

```
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 all conditions are false
}
```

```java
int number = 0;
if (number > 0) {
    System.out.println("The number is positive.");
} else if (number < 0) {
    System.out.println("The number is negative.");
} else {
    System.out.println("The number is zero.");
}
```

Example:

4. Nested if statement:

Nested if statements allow an **if statement** to be placed **inside another if statement**. This allows for **more complex decision-making** processes where multiple conditions must be true for a block of code to execute.

Syntax:

```
if (condition1) {
    // code to be executed if condition1 is true
    if (condition2) {
        // code to be executed if condition2 is true
    }
}
```

Example:

```java
int number = 10;
if (number > 0) {
    if (number % 2 == 0) {
        System.out.println("The number is positive and even.");
    }
}
```

PRAMOD NAIK

5. Switch statement:

The switch statement evaluates an expression and compares it to a **list of case** values. When a match is found, the corresponding block of code is executed. The **break** statement is used to exit the switch block after the matched case has been executed. If no match is found, the default block (if present) is executed. The switch statement is often used as an alternative to the **if-else-if ladder** for better readability and performance when dealing with multiple possible values of an expression.

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;
    // you can have any number of case statements
    default:
        // code to be executed if expression doesn't match any case
}
```

Example:

```java
int day = 3;
switch (day) {
    case 1:
        System.out.println("Monday");
        break;
    case 2:
        System.out.println("Tuesday");
        break;
    case 3:
        System.out.println("Wednesday");
        break;
    case 4:
        System.out.println("Thursday");
        break;
    case 5:
        System.out.println("Friday");
        break;
    case 6:
        System.out.println("Saturday");
        break;
    case 7:
        System.out.println("Sunday");
        break;
    default:
        System.out.println("Invalid day");
}
```

2. Loop Statements:

In Java, loop statements **allow** the **execution of a block of code repeatedly based on a condition**. They are essential for tasks that require iteration, such as processing elements in an array or repeatedly performing an operation until a certain condition is met.

Looping Statements are:

1. **for** loop
2. **while** loop **(Entry Controlled Loop)**
3. **do-while** loop **(Exit Controlled Loop)**
4. **for-each** loop (enhanced for loop)

1. for loop:

The for loop provides a **concise way** of writing the loop structure. It is used when the **number of iterations is known beforehand**.

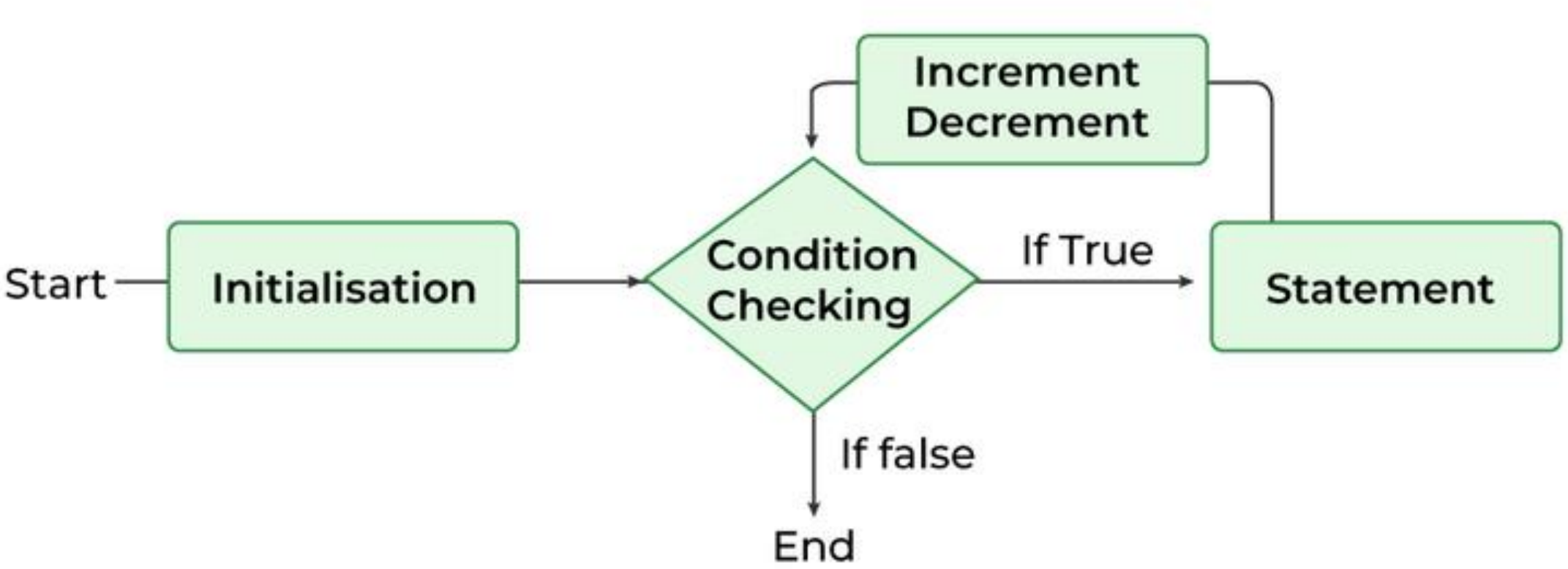
Syntax:

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

Example:

```
for (int i = 0; i < 5; i++) {
    System.out.println(i);
}
```

for Loop Flow Chart:



2. while loop:

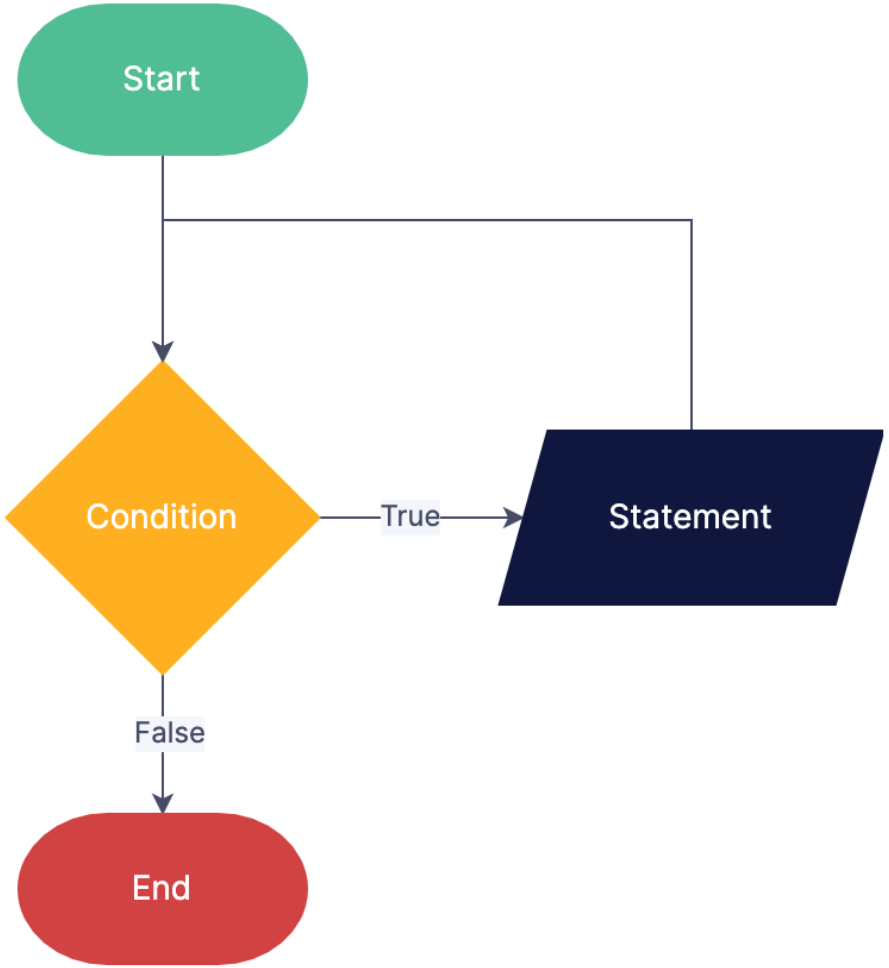
The while loop continually executes a block of code as long as a specified condition is true. The condition is checked before the execution of the loop body, making it a pre-test loop.

Syntax & Example:

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

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

While Loop Flowchart



PRAMOD NAIK

3. do-while loop:

The do-while loop is similar to the while loop, but it guarantees that the loop body will be executed at **least once** because the condition is checked **after the execution of the loop body**.

Syntax & Example:

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

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

do-while Loop Flow Chart

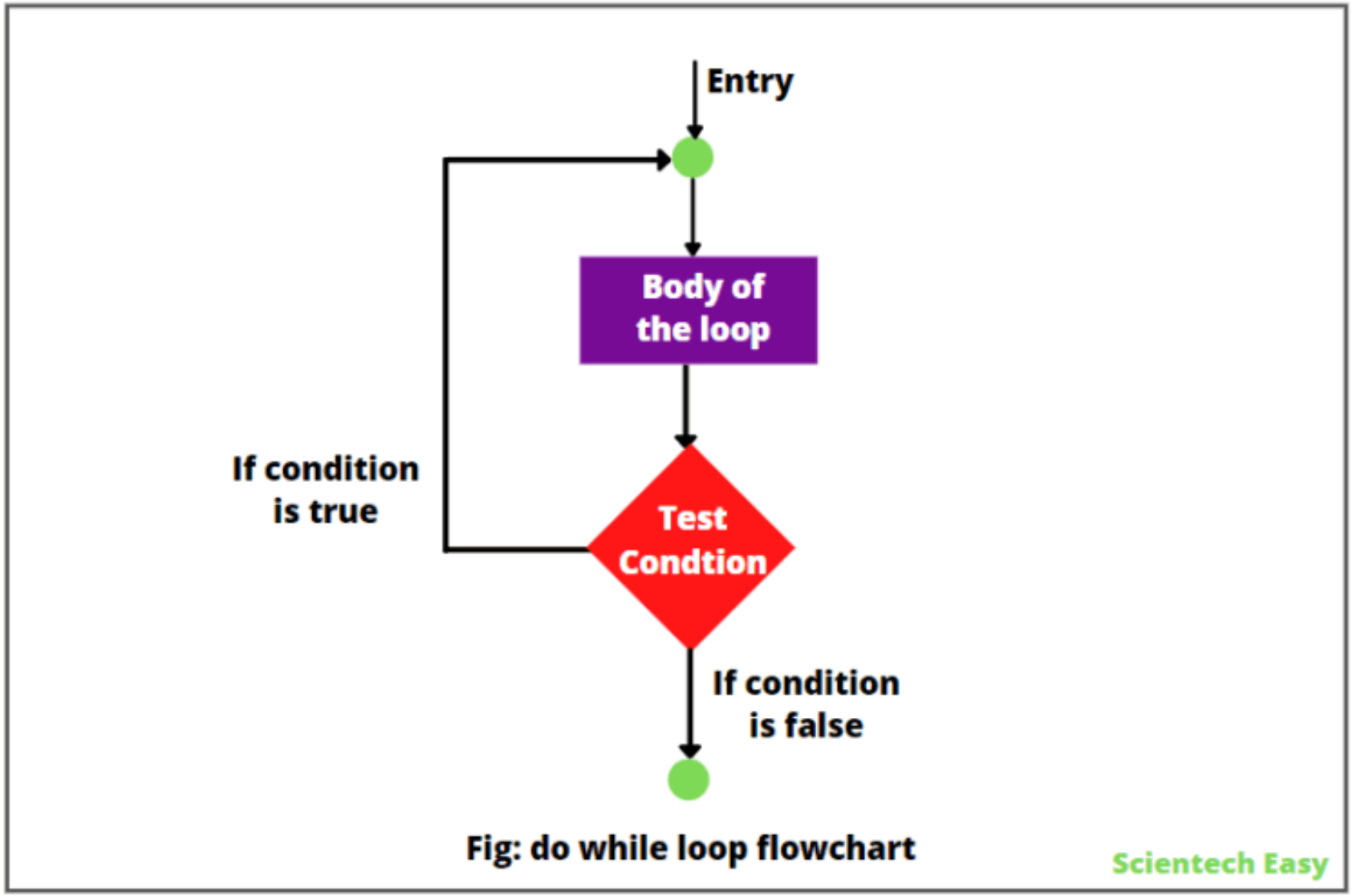


Fig: do while loop flowchart

4. for-each loop (Enhanced for loop)

The for-each loop, introduced in **Java 5,** is used to **iterate over elements in an array or a collection**, making the code more readable and reducing the risk of errors.

Syntax:

```
for (type element : array) {
    // code to be executed
}
```

Example:

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

3. Branching or Jump Statements:

In Java, branching statements are used to **alter the flow of execution** within a program based on certain conditions. They allow for more **complex** and **dynamic** behavior by enabling jumps to different parts of the code.

The main branching statements in Java are:

1. **break**
2. **continue**
3. **return**

1. break statement:

The break statement is used to terminate the execution of a loop or switch statement prematurely. It can be used in for, while, and do-while loops, as well as in switch statements.

Syntax:

```
break;
```

Example: When a break statement is encountered inside a loop or switch, control is transferred to the statement immediately following the loop or switch.

```
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        break;
    }
    System.out.println(i);
}
// Output: 0 1 2 3 4
```

PRAMOD NAIK

2. continue statement:

The continue statement is used to skip the current iteration of a loop and proceed to the next iteration. It can be used in for, while, and do-while loops.

Syntax:

```
continue;
```

Example: When a continue statement is encountered inside a loop, control is immediately transferred to the next iteration of the loop, bypassing the remaining code in the current iteration.

```
for (int i = 0; i < 10; i++) {
    if (i % 2 == 0) {
        continue;
    }
    System.out.println(i);
}
// Output: 1 3 5 7 9
```

3. return statement

The return statement is used to exit from the current method and optionally return a value to the calling method. It can be used in any method to terminate its execution and, if needed, provide a value back to the caller.

Syntax:

```
return;        // for methods that do not return a value (void methods)
return value; // for methods that return a value
```

Example-1: When a return statement is encountered, the method **execution is terminated**, and control is returned to the caller. If the method has a return type other than void, a **value must be returned**.

```
public void checkAge(int age) {
    if (age < 18) {
        System.out.println("You are not eligible.");
        return;
    }
    System.out.println("You are eligible.");
}
```

Example-2: Method that returns a value:

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

Constants:

In Java, a constant is a **variable whose value cannot be changed once it is assigned**. Constants are used when you want to define a value that should remain the same throughout the execution of a program. Constants provide several benefits, including making code more readable, reducing the likelihood of errors, and making it easier to modify values that are used in multiple places.

Example:

```
final int NUM_OF_STUDENTS = 80;
```

Scope and life time of variables

In Java, the scope and lifetime of variables are determined by **where the variables are declared** and **how they are used**.

1. Scope of Variables

The scope of a variable refers to the **region of the program where the variable can be accessed.**

2. Lifetime of Variables

The lifetime of a variable refers to the **duration for which the variable exists in memory** during the execution of a program.

Example:

```
public class Example {
    public int instanceVar = 10;  // Instance variable
    public static int staticVar = 20;  // Static variable

    public void display() {
        System.out.println("Instance Variable: " + instanceVar);
        System.out.println("Static Variable: " + staticVar);
    }

    public static void main(String[] args) {
        Example obj = new Example();
        obj.display();
    }
}
```

1. Local Variables:

Scope:

Local variables are **declared inside** a method, constructor, or block of code (e.g., loops, conditional statements).

They are only accessible within the method, constructor, or block in which they are declared.

Lifetime:

The lifetime of a local variable begins when the **method**, **constructor**, or **block** in which it is defined is invoked, and ends when the method, constructor, or block is **exited**.

Example-1:

```
if (true) {
    int blockVar = 40;  // Block-scoped variable
    System.out.println("Block Variable: " + blockVar);
}
```

Example-2:

```
public static int add(int a, int b) {
        // Here the scope of the variable a, b, and sum will be inside the method add() only.
        int sum = a + b;    // sum is Local Variable
        return sum;
    }
```

PRAMOD NAIK

2. Instance Variables:

Scope: Defined inside a class but outside any method. They belong to an instance of the class.

Lifetime: The lifetime of an instance variable **begins** **when an object is created using the new keyword** and **lasts** **until the object is eligible for garbage collection.**

3. Static Variables:

Scope: Defined inside a class with the static keyword. They belong to the class itself rather than any instance.

Lifetime: The lifetime of a static variable **begins** when the class is loaded into memory and lasts **until** the class is unloaded. Static variables are shared among all instances of the class.

Example: **static** and **instance** variables:

```
class Student{

    String name;
    int rollNum;
    static String college_name;

    public static void print_college(Student obj) {
        System.out.println("This is Static Method!");
    }

}
```

Operators in Java:

In Java, an operator is a **symbol** **that performs operations on one or more operands. Operands** are the values on which the operator **operates**. Operators are used to manipulate data and variables.

Java supports various types of operators, including:

1. **Unary Operators**
2. **Binary Operators**
    1. Arithmetic Operators
    2. Relational Operators
    3. Logical Operators
    4. Assignment Operators
    5. Bitwise Operators
3. **Ternary Operator**

1. **Unary Operators:** Which operates on only one Operand.

Operator	Description	Example
+	Unary plus, indicates a positive value.	+a
-	Unary minus, negates the expression.	-a
++	Increment operator, increases the value by 1.	a++, ++a
--	Decrement operator, decreases the value by 1.	a--, --a
!	Logical NOT, inverts the value of a boolean.	!true

2. **Binary Operators:** Operate on two operands

1. **Arithmetic Operators:**

Operator	Description	Example
+	Addition, adds two operands.	a + b
-	Subtraction, subtracts the second operand from the first.	a - b
*	Multiplication, multiplies two operands.	a * b
/	Division, divides the numerator by the denominator.	a / b
%	Modulus, returns the remainder of the division.	a % b

2. Relational Operators:

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

a=10, b=20

Operator	Description	Example
==	Equal to, checks if two operands are equal.	a == b
!=	Not equal to, checks if two operands are not equal.	a != b
>	Greater than, checks if the left operand is greater than the right.	a > b
<	Less than, checks if the left operand is less than the right.	a < b
>=	Greater than or equal to, checks if the left operand is greater than or equal to the right.	a >= b
<=	Less than or equal to, checks if the left operand is less than or equal to the right.	a <= b

3. Logical Operators:

Logical operators in Java are used to **perform logical operations on boolean expressions.** They are commonly used in conditional statements to **combine multiple conditions.**

Operator	Description	Example
&&	Logical AND, returns true if both operands are true.	cond1 && cond2
||	Logical OR, returns true if any one of the operand is true.	cond1 || cond2
!	Logical NOT, inverts the value of a boolean operand.	!a

4. Assignment Operators:

Operator	Description	Example
=	Simple assignment, assigns the right operand to the left operand.	a = b
+=	Addition assignment, adds right operand to left operand and assigns the result to the left operand.	a += b
-=	Subtraction assignment, subtracts right operand from left operand and assigns the result to the left operand.	a -= b
*=	Multiplication assignment, multiplies right operand with left operand and assigns the result to the left operand.	a *= b
/=	Division assignment, divides left operand by right operand and assigns the result to the left operand.	a /= b
%=	Modulus assignment, calculates modulus using two operands and assigns the result to the left operand.	a %= b

5. Bitwise Operators:

Bitwise operators in Java perform bit-level operations on integer types (int, long, short, char, byte). These operators work directly on the binary representation of numbers.

Operator	Description	Example
&	Bitwise AND, performs a bitwise AND operation.	a & b
|	Bitwise OR, performs a bitwise OR operation.	a|b
^	Bitwise XOR, performs a bitwise XOR operation.	a ^ b
~	Bitwise NOT, inverts all the bits of the operand.	~a
<<	Left shift, shifts the bits of the left operand left by the number of positions specified by the right operand.	a << b
>>	Right shift, shifts the bits of the left operand right by the number of positions specified by the right operand.	a >> b
>>>	Unsigned right shift, shifts zero into the leftmost bits.	a >>> b

Truth Table:

1. &:

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

2. | :

A	B	A | B
0	0	0
0	1	1
1	0	1
1	1	1

3. ~

A	!A
0	1
1	0

4. ^

A	B	A | B
0	0	1
0	1	0
1	0	0
1	1	1

PRAMOD NAIK

Example:

1. &

```
int a = 5;  // Binary: 0101
int b = 3;  // Binary: 0011
int result = a & b; // Binary: 0001
System.out.println("a & b = " + result); // Output: 1
```

2. |

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

3. ~

```
int a = 5;  // Binary:  0101
int result = ~a; // Binary: 1010 (2's complement representation)
System.out.println("~a = " + result); // Output: -6
```

PRAMOD NAIK

3. Ternary Operator

The ternary operator in Java, also known as the conditional operator, is a shorthand for an if-else statement. It has three operands and is used to evaluate a condition and return one of two values, depending on whether the condition is true or false.

Syntax:

```
condition ? expression1 : expression2
```

Where,

condition: This is a boolean expression that evaluates to either true or false.

expression1: This is the value returned if the condition is true.

expression2: This is the value returned if the condition is false.

Example:

```
int a = 5;
int b = 10;

// Using ternary operator to find the maximum of two numbers
int max = (a > b) ? a : b;

// max will be 10 because the condition (a > b) is false
System.out.println("The maximum value is " + max);
```

Expressions:

An expression is a **combination** of **operators**, **constants** and **variables**. An expression may consist of one or more **operands**, and zero or more **operators** to produce a value.

An **expression** in Java is a construct that combines variables, operators, method calls, and literals to produce a single value.

Examples:

1. int number = 5;
2. int sum = 10 + 5;
3. boolean result = 10 > 5;
4. boolean result = (10 > 5) && (3 < 8);
5. int bitwiseAnd = 5 & 3;
6. int max = Math.max(10, 20);
7. boolean isString = "Hello" instanceof String;
8. int result = a + b * (a - 3) / 2;

Operator Hierarchy:

Precedence or Hierarchy refers to the order in which operators are evaluated in an expression when there are multiple operators present. So, it determines the **order in which operators are evaluated in an expression**. Operators with higher precedence are evaluated before operators with lower precedence.

Associativity:

Associativity defines the direction in which operators of the same precedence level are evaluated in an expression. If operators have the same precedence, their **associativity** determines the order of evaluation.

Associativity can be either:

1. **Left-to-Right Associativity:** Operators are evaluated from left to right. This is common for most operators.

   For example, in the expression **a - b - c,** subtraction is evaluated as **(a - b) - c.**

2. Right-to-Left Associativity: Operators are evaluated from right to left. This is less common and is typically seen with assignment (=) and ternary (?:) operators.

For example, in the expression **a = b = c**, the assignment is evaluated as a = (b = c).

Precedence and Associativity:

Precedence Level	Operator Category	Operators	Associativity
1	Postfix	expr++, expr--	Left-to-right
2	Unary	++expr, --expr, +expr, -expr, ~, !	Right-to-left
3	Multiplicative	*, /, %	Left-to-right
4	Additive	+, -	Left-to-right
5	Shift	<<, >>, >>>	Left-to-right
6	Relational	<, >, <=, >=, instanceof	Left-to-right
7	Equality	==, !=	Left-to-right
8	Bitwise AND	&	Left-to-right
9	Bitwise XOR	^	Left-to-right
10	Bitwise OR	`	`
11	Logical AND	&&	Left-to-right
12	Logical OR	`	
13	Ternary	? :	Right-to-left
14	Assignment	=, +=, -=, *=, /=, %=	Right-to-left

Example:

1. int result = **10 + 20 * 2 / 4;** // Result will be **20**

2. int result = **(10 + 20) * (2 / 4);** // Result will be **15**

3. boolean result = **10 > 5 == 5 < 3;**

4. boolean result = **(10 == 10) & (5 < 3);**

5. int a = 10;
   int b = 5;

   int result = a += b * 2;

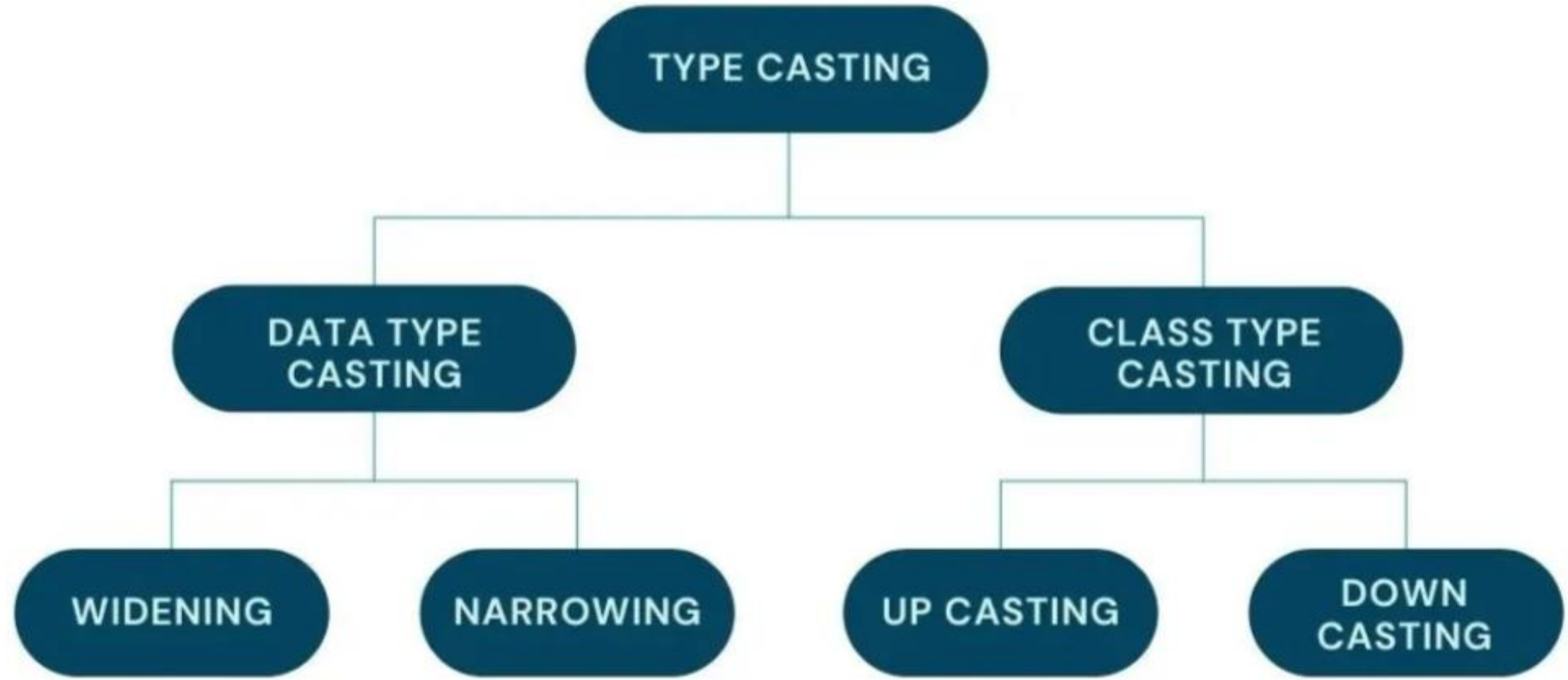
Type conversion and Casting:

Type Conversion and **Casting** in Java are processes that allow you to **convert one data type into another**. These operations are essential when you need to work with **different types of data together.**

PRAMOD NAIK

https://yuvrajscorpio.medium.com/type-casting-in-java-8175fa2875ee

TYPE CASTING IN JAVA



1. Type Conversion or Casting:

Type conversion, also known as **type casting**, is the process of **converting a value from one data type to another.** There are two types of type conversion in Java:

a. **Implicit (Automatic or widening Casting) Type Conversion**

b. **Explicit Type Conversion (Mannual Casting or Narrowing Casting)**

a. Implicit (Automatic) Type Conversion:

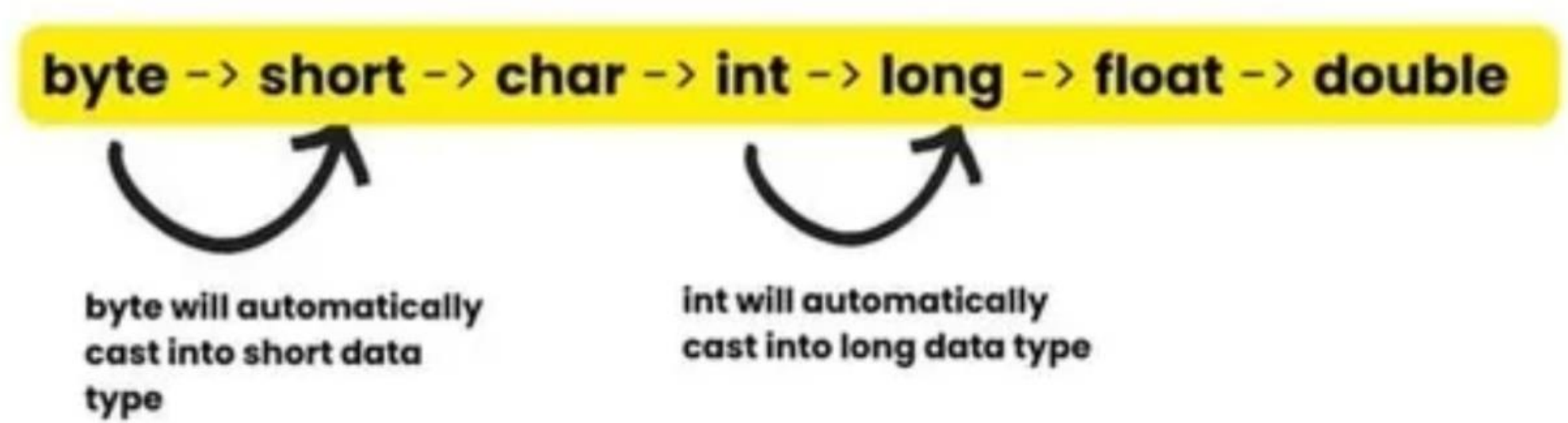
The **Java compiler automatically converts** a **smaller data type into a larger data type** without explicit instructions from the programmer. This is also called **widening** conversion.

Syntax: No explicit syntax is needed as it happens automatically.

This type of casting happens when

- Both data types must be **compatible** with each other.
- The target type must be **larger than** the source type.

For example, if you assign an **integer** value to a variable of type **double**, Java will **automatically convert** the integer to a double. This is because the range of values that can be stored in a double is larger than the range of values that can be stored in an integer.



PRAMOD NAIK

Example:

```
int num = 100;
double doubleNum = num;  // Implicit conversion from int to double
System.out.println("doubleNum: " + doubleNum);  // Outputs 100.0
```

PRAMOD NAIK

b. Explicit Type Conversion (Casting):

When you need to **convert a larger data type into a smaller data type**, or when you need to **explicitly convert between types**, you must perform **explicit casting**. This is also called **narrowing conversion**.

In this process there is high chance that we will **lose our data.**

Converting a **higher data type into a lower one** is called narrowing type casting. It is also known as **explicit casting** or **casting down**. It is done manually by the programmer. If we do not perform casting, then the compiler reports a compile-time error.

This type of casting happens when

- Both data types must be **compatible** with each other.
- The target type must be **smaller than** the source type.

For example, if you assign a **double** value to a variable of type **int**, you will need to **manually convert** the double to an int using an explicit cast. This is because the range of values that can be stored in an int is **smaller** than the range of values that can be stored in a double.

To perform an explicit type cast in Java, **you place the desired data type in parentheses** before the variable that you want to cast.

PRAMOD NAIK

For example:

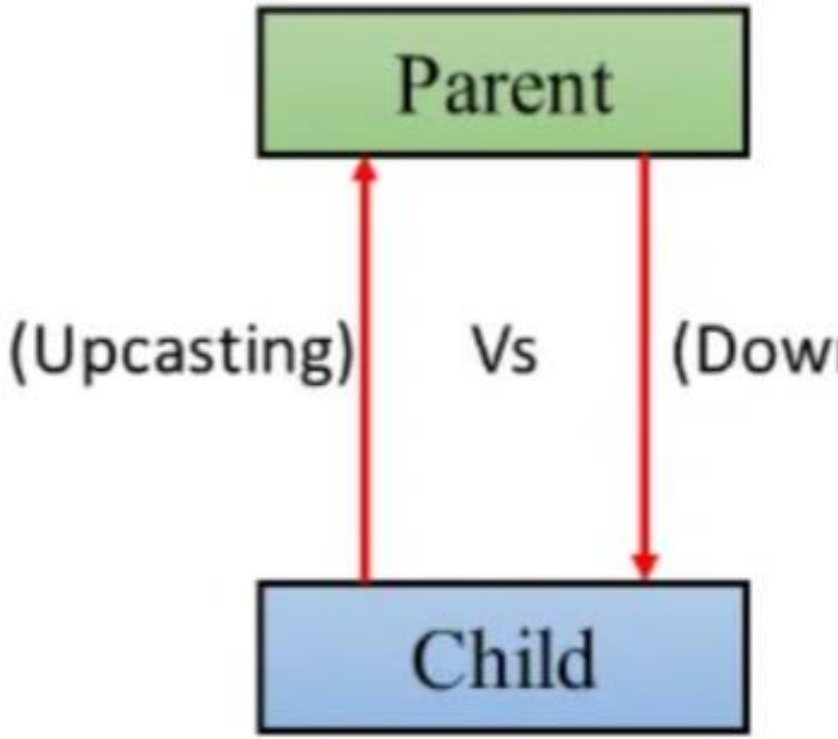
```
double doubleNum = 100.5;
int num = (int) doubleNum;  // Explicit casting from double to int
System.out.println("num: " + num);  // Outputs 100
```

Note:

It is important to note that explicit type casting can result in **data loss** if the value being cast is too large to be stored in the new data type. In such cases, the value will be truncated to fit into the new data type, which may result in unexpected behavior.

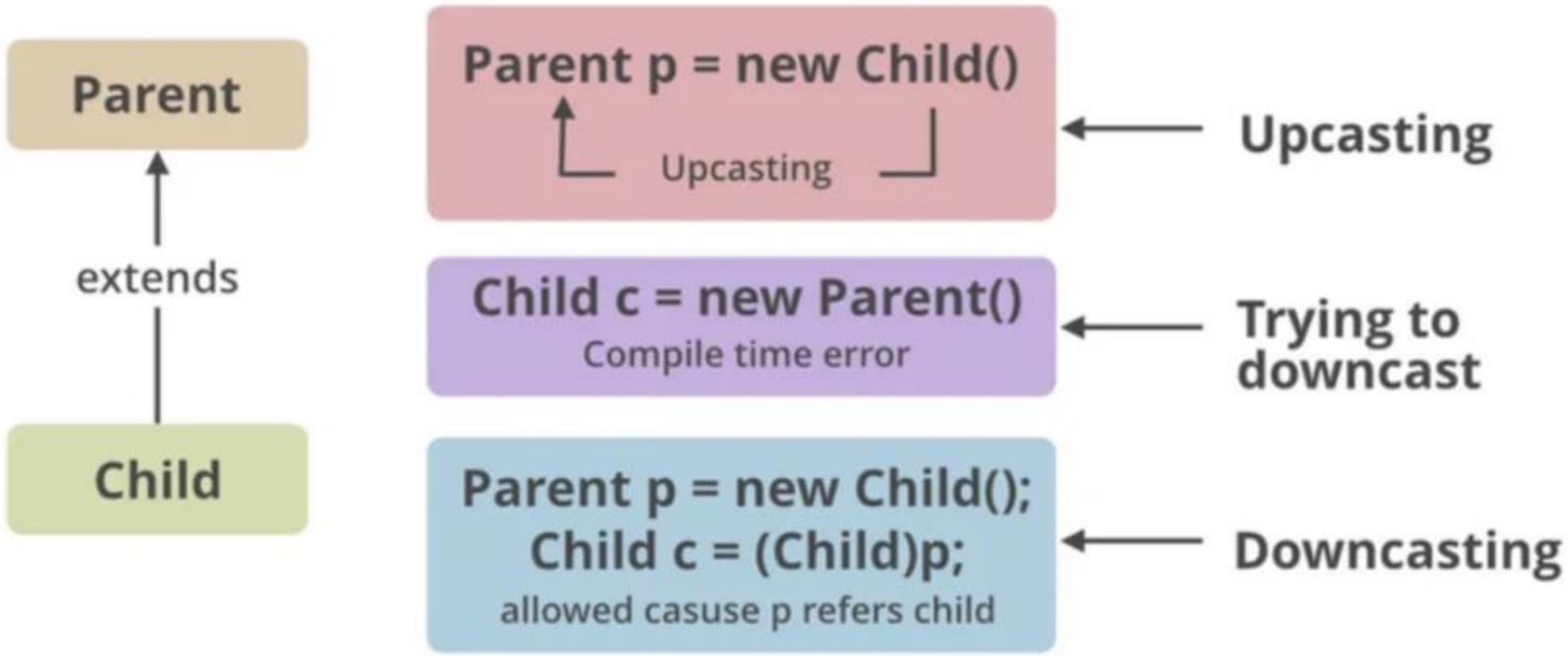
2. Class Type Casting: (Casting between Objects):

Upcasting Vs Downcasting in Java



PRAMOD NAIK

Upcasting vs Downcasting in java programming



Upcasting vs Downcasting

PRAMOD NAIR

a. Upcasting in Java

Upcasting is the process of **converting an object of a subclass to an object of its superclass.** This is generally done **implicitly** by the compiler and is often used to store objects of different subclasses in a collection or array of the superclass type.

For example, if you have a **collection of animals**, which includes dogs, cats, and birds, you can store all these objects in an ArrayList of the Animal class, which is the superclass of the Dog, Cat, and Bird classes.

Example:

```
class Animal {}
class Dog extends Animal {}

Dog dog = new Dog();
Animal animal = dog;   // Implicit upcasting from Dog to Animal
```

b. Downcasting in Java

Downcasting *is the process of converting an* ***object of a superclass to an object of its subclass*****.** This is an **explicit** operation that must be performed using the **cast operator** and can be risky because it may result in a **ClassCastException** if the object is not actually an instance of the subclass. Downcasting is often used to access methods and properties that are **specific to a subclass** after an object has been upcasted to its superclass.

For example, if you have an ArrayList of animals and you know that some of the objects in the list are actually dogs, you can downcast those objects to Dog objects in order to call **methods that are specific to the Dog class.**

Example:

```
Animal animal = new Dog();  // Upcasting
Dog dog = (Dog) animal;  // Explicit downcasting from Animal to Dog
```

Methods and Constructors:

PRAMOD NAIK

Method:

In Java, a **method** is a block of code that performs a specific task and can be executed when called upon. Methods are used to define the behavior of objects, perform operations, and structure the code in a modular and reusable way.

```
accessSpecifier returnType methodName(parameterType1 parameter1, parameterType2 parameter2, ...) {
    // Method body: code to be executed
    // Optionally return a value of type 'returnType'
}
```

Components of a Method

1. **Return Type:** Specifies the type of value the method returns. If the method does not return a value, the return type is void.
2. **Method Name:** The name of the method, which is used to call it. Method names should be meaningful and follow camelCase convention.
3. **Parameters (optional):** A list of variables that the method can accept as input. Each parameter is defined by a type and a name. Parameters are optional; a method can have zero or more parameters.
4. **Method Body:** The block of code enclosed in curly braces {} that defines what the method does
5. **Return Statement (optional):** If the method has a return type other than void, it must include a return statement to return a value.

Calling a Method:

To call a method in Java, you use the method name followed by parentheses, which may include arguments if the method requires parameters. Methods can be called from within other methods or from different classes, depending on their visibility (access specifier).

Syntax:

```
// Calling a method with no arguments
methodName();

// Calling a method with arguments
methodName(argument1, argument2, ...);
```

Parameter Passing :

Java supports two types of parameter passing:

1. **Pass by Value** (the only parameter passing method in Java)
2. **Pass by Reference** (not directly supported in Java for objects, but reference variables are passed by value)

Constructor:

A constructor in Java is a **special method** that is used to **initialize objects.** The constructor is called **automatically** when an object of a class is created. Constructors are important because they allow you to set initial values for object attributes and prepare the object for use.

Key Characteristics of a Constructor:

- **Name:** The constructor's name must be the same as the **class name.**
- **No Return Type:** Constructors **do not have a return type**, not even **void**.
- **Called Automatically:** The constructor is called automatically when an object is created using the new keyword.

Types of Constructors :

1. Default Constructor
2. Parameterized Constructor

1. Default Constructor

A default constructor is a constructor that does not take any arguments. If you do not explicitly define a constructor in your class, the Java compiler provides a default constructor that initializes all member variables to their default values (e.g., 0, null, false).

Example:

```
class Student{
    //Attributes of Student Class
    String name;
    int age;
    String studentId;

    // Default Constructor.
    // This is optional as it will be created automatically.
    Student(){
        this.name = "Unknown";
        this.age = 0;
        this.studentId = "Unknown";
    }
}
```

2. Parameterized Constructor:

A **parameterized constructor** is a constructor that takes one or more arguments. These parameters allow you to pass specific values when creating an object, enabling customized initialization.

Example:

```
class Student{
    //Attributes of Student Class
    String name;
    int age;
    String studentId;

    // Parameterized Constructor
    Student(String name, int age, String Id){
        this.name = name;
        this.age = age;
        this.studentId = Id;
    }
}
```

this Reference:

In Java, **this keyword** is a reference to the **current object**, the object whose method or constructor is being called. It is a powerful and versatile tool that serves several important purposes in object-oriented programming.

Example:

```
class Student{
    //Attributes of Student Class
    String name;
    int age;
    String studentId;

    // Parameterized Constructor
    Student(String name, int age, String Id){
        this.name = name;
        this.age = age;
        this.studentId = Id;
    }
}
```

PRAMOD NAIK

Access control:

Access control in Java is a mechanism that defines the **visibility** and **accessibility** of **classes**, **methods**, and **variables**. It ensures that components of a program are only accessed where it is intended, promoting encapsulation and security in object-oriented programming.

Java provides four access modifiers to control the access levels:

1. **public:** Accessible from any other class.
2. **protected:** Accessible within the same package and by subclasses.
3. **default** (**no modifier**): Accessible only within the same package.
4. **private:** Accessible only within the same class.

1. public Access Modifier:

The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

The public modifier allows the class, method, or variable to be accessed from **any other class in any package**.

Example:

```java
package package1;

public class A {
    public void display() {
        System.out.println("Public method in class A");
    }
}

// In another package
package package2;

import package1.A;

public class B {
    public static void main(String[] args) {
        A obj = new A();
        obj.display();  // Accessible because display() is public
    }
}
```

PRAMOD NAIK

2. protected Access Modifier:

The access level of a protected modifier is **within the package** and **outside the package through child class (Inheritance)**. If you do not make the child class, it cannot be accessed from outside the package.

Example:

```java
package package1;

public class A {
    protected void display() {
        System.out.println("Protected method in class A");
    }
}

// In another package
package package2;

import package1.A;

public class B extends A {
    public static void main(String[] args) {
        B obj = new B();
        obj.display();  // Accessible because B is a subclass of A
    }
}
```

PRAMOD NAIK

3. default (No Modifier) Access Modifier:

The access level of a default modifier is **only within the package**. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.

Example:

```java
package package1;

class A {
    void display() {
        System.out.println("Default method in class A");
    }
}

public class B {
    public static void main(String[] args) {
        A obj = new A();
        obj.display();  // Accessible because B is in the same package as A
    }
}
```

4. Private Access Modifier:

The access level of a private modifier is only within the class. It cannot be accessed from outside the class.

Example:

```
package package1;

public class A {
    private void display() {
        System.out.println("Private method in class A");
    }

    public void show() {
        display();  // Accessible within the same class
    }
}

public class B {
    public static void main(String[] args) {
        A obj = new A();
        // obj.display();  // Compile-time error: display() has private access in A
        obj.show();  // This is fine because show() is public
    }
}
```

Conclusion:

Access Specifier	Same Class	Same Package (Child Class)	Same Package (Non-Child Class)	Other Package (Child Class)	Other Package (Non-Child Class)
Public	Yes	Yes	Yes	Yes	Yes
Protected	Yes	Yes	Yes	Yes	No
Default	Yes	Yes	Yes	No	No
Private	Yes	No	No	No	No

PRAMOD NAIK

static fields and methods:

In Java, the static keyword is used to define fields (variables) and methods that **belong to the class** **rather than to instances** of the class. This means that static members are **shared among all instances** of the class and can be accessed **without creating an instance of the class.**

Key Points About static Fields and Methods:

1. Shared Across All Instances: A static field is shared among all instances of a class. It means that there is only one copy of the static field, regardless of how many objects of the class are created.

2. Accessed Without an Object: Static fields can be accessed using the class name directly (**e.g., ClassName.fieldName**). They can also be accessed using an object reference, but this is not recommended as it can lead to confusion.

3. Initialization: Static fields can be initialized at the point of declaration or in a static block.

4. Memory Allocation: Static fields are allocated memory only once when the class is loaded into memory.

1. Static Fields (Static Variables)

A static field is **shared among all instances of a class**. Unlike instance variables, which are unique to each object, static variables have the same value for all objects of the class. There is only one copy of the static variable in memory, regardless of how many instances of the class are created.

Key Points:

- Declared using the static keyword.
- Belongs to the class, not to any instance.
- Shared among all instances of the class.
- Can be accessed directly using the class name.

Example:

In this example:

- **count** is a static variable, shared across all instances of Counter.
- Every time a Counter object is created, count is incremented.
- The static method **displayCount**() is used to display the value of the count variable.

```
class Counter {
    static int count = 0; // Static variable

    Counter() {
        count++; // Increment the static variable
    }

    public static void displayCount() {
        System.out.println("Count: " + count);
    }
}

public class Main {
    public static void main(String[] args) {
        Counter c1 = new Counter();
        Counter c2 = new Counter();
        Counter c3 = new Counter();

        Counter.displayCount(); // Output: Count: 3
    }
}
```

2. Static Methods

A static method **belongs to the class**, **not to any instance of the class**. It can be called **without creating an object** of the class. Static methods can only access static fields and other static methods. They cannot access instance variables or instance methods directly.

Key Points:

- Declared using the static keyword.
- Can be called without creating an instance of the class.
- Cannot access instance variables or instance methods directly.
- Useful for utility or helper methods (e.g., Math.pow()).

Example:

In this example:

- The methods **add()** and **multiply()** are static, so they can be called **without creating an object** of MathUtils.
- The static methods are designed to perform operations that **don't rely on any instance state.**

```
class MathUtils {
    // Static method
    public static int add(int a, int b) {
        return a + b;
    }

    // Static method
    public static int multiply(int a, int b) {
        return a * b;
    }
}

public class Main {
    public static void main(String[] args) {
        // Access static methods without creating an instance of MathUtils
        int sum = MathUtils.add(10, 20);          // Output: 30
        int product = MathUtils.multiply(10, 20); // Output: 200

        System.out.println("Sum: " + sum);
        System.out.println("Product: " + product);
    }
}
```

PRAMOD NAIK

Garbage Collection:

In Java is an **automatic memory management** process that helps **reclaim memory occupied by objects that are no longer in use**, thus preventing memory leaks and optimizing the application's performance.

PRAMOD NAIK

PRAMOD NAIK

PRAMOD NAIK

PRAMOD NAIK