Unit- I

OBJECT ORIENTED THINKING

Introduction, Need of object-oriented programming, Applications of OOP, history of Java, Java Virtual Machine, Java Features/Buzzwords. Identifiers, Keywords, Datatypes, Variables, Literals, Operators- Unary, Binary, and Ternary, Expressions, Primitive type conversion and casting, flow of control- branching, conditional, loops.

INTRODUCTION TO OOP

Object Oriented Programming (OOP) is a programming paradigm in which programs are designed using **objects and classes**. It organizes **data(variables)**and**methods(functions)**into a single unit called as **class**, making programs easier to develop, maintainand reuse.

Evolution of Programming Languages

C → C++ → Java

- **C** is a procedural-oriented programming language.
- **C++** is an object-oriented programming language.
- **Java** is a pure object-oriented programming language.

Features of OOP(4 foundational pillars):

1. **Abstraction**
2. **Encapsulation**
3. **Inheritance**
4. **Polymorphism**

1.Abstraction

- It is an essential element of Object-OrientedProgramming (OOP).
- Abstraction means hiding the internal implementation and showing only the essential features.
- Abstraction is used to manage complexity.

Example

When you drive a car, you use:

- Steering wheel
- Accelerator
- Brake

You **don't need to know** how the engine, transmission, or fuel injection system works. The car hides those complex details and gives you a simple interface to drive.

- In Java, we achieve this using **Abstract Classes** and **Interfaces**.

2. Encapsulation

Encapsulation is the mechanism of **binding or wrapping data (variables) and the methods (functions)**that operate on that data into a **single unit (class)**, while restricting direct access to the data to protect it from unauthorized access and misuse.

Example

Think of an **ATM**:

- You can **deposit**, **withdraw**, or **check your balance**.
- You **cannot directly change your account balance** by accessing the bank's database.
- The balance is protected, and all changes happen through authorized operations.

In the above example, the data (balance) is hidden, and it can only be accessed or modified through specific methods.

3. Inheritance

Inheritance is the process by which one class acquires the properties of another class.

- Supports hierarchical classification.

Example:

A **Car** is a type of **Vehicle**.

- A **Vehicle** has common properties like start() and stop().
- A **Car** inherits these properties and can also have its own feature like playMusic().

In the above example,**Car**acquires the variables(properties) and methods of the **Vehicle**.

4. Polymorphism

Polymorphism is the ability of an object or method to take many forms. In Java, the same method name can perform different tasks depending on the object or the parameters passed to it. It means "one interface, many forms".

Example

Think of a person.

The same person behaves differently depending on the situation:

- At home, the person is a parent.
- At office, the person is an employee.
- At school, the person is a teacher.

The same person performs different roles. This is an example of polymorphism.

Differences-Procedural-Oriented Programming & Object-Oriented Programming

Procedural-Oriented Programming	Object-Oriented Programming
Program is divided into smaller parts called functions.	Program is divided into smaller parts called classes.
Follows a top-down approach.	Follows a bottom-up approach.
No access specifiers.	Has access specifiers like private,<default>, protected, public.
Adding new data and functions is not easy.	Adding new data and functions is easy.
No proper way of hiding data → less secure.	Provides data hiding → more secure.
Overloading is not possible.	Overloading is possible.
Functionsare more important than data.	Data is more important than functions.
Not suitable for real time applications.	Suitable for real time applications.
Examples: C, FORTRAN, Pascal.	Examples: Java, C++, Python, R.

Need of object-oriented programming

Object-Oriented Programming (OOP) is needed because it helps manage complexity in large software systems, improves code reusability, enhances security, and makes applications easier to maintain and scale. It organizes programs around objects **(data + behaviour)**, reflecting real-world entities.

We Need Object-Oriented Programming as it provides-

1. **Modularity & Structure**
   - Programs are divided into objects rather than functions.
   - Each object encapsulates data and behaviour, making code easier to understand and manage.
   - **Example: A Car class with attributes (colour, speed) and methods (start, stop).**
2. **Code Reusability**
   - Classes can be reused across different projects.
   - Inheritance allows new classes to build on existing ones without rewriting code.
   - **Example: A Vehicle class reused for Car, Bike, and Truck.**
3. **Maintainability & Scalability**
   - Changes in one part of the system don't break the entire application.
   - Easier to expand systems as requirements grow.
   - **Example: Adding new features to an E-commerce system (like discounts) without rewriting core modules.**

4. **Security**
   - Encapsulation hides internal details and exposes only necessary functionality.
   - Access specifiers (private, public, protected) restrict unauthorized access.
   - **Example: A Bank Account class hides balance details, allowing access only through deposit/withdraw methods.**

5. **Real-World Modelling**
   - OOP reflects real-world entities, making software design intuitive.
   - **Example: In a Library Management System, classes like Book, Memberand Librarian represent real entities.**
6. **Team Collaboration**
   - Large projects can be divided into modules (objects/classes).
   - Multiple developers can work in parallel without interfering with each other's code.
7. **Reduced Complexity**
   - Avoids passing too many parameters in procedural functions.
   - Objects carry their own state and behaviour, simplifying interactions.

Applications of OOP

- **Desktop Applications**: Used in GUI-based software like text editors, IDEs, and media players. Example: Microsoft Word, Eclipse IDE.
- **Web Applications**: OOP frameworks (like Java Spring, Django in Python) make web apps modular and scalable. Example: Online banking systems, e-commerce platforms.
- **Mobile Applications**: Android apps (Java/Kotlin) and iOS apps (Swift) rely heavily on OOP concepts.
- **Game Development**: Objects represent characters, weapons, and environments. Example: Unity (C#), Unreal Engine (C++).
- **Enterprise Applications:**Large-scale systems like ERP, CRM, and HR management software use OOP for modularity.
- **Scientific Applications**: Simulations, modelling, and data analysis tools use OOP for complex structures.
- **Embedded Systems**: Devices like smart TVs, washing machines, and automotive systems use OOP for modular control.
- **Cloud & Distributed Applications**: OOP supports distributed computing and cloud-based services. Example: Microservices architecture in Java or Python.

History of Java

- Java was developed by **James Gosling**and his team at **Sun Microsystems**in **1991** as part of the **Green Project**. The main goal of the project was to develop software for consumer electronic devices such as televisions, set top boxes, and home appliances.
- Initially, the programming language was named **Oak**, later in **1995** it was renamed as **Java**.

- Java was designed with the principle **"Write Once, Run Anywhere (WORA)"**. This means that a Java program can be written once and run on any operating system without modification. This is possible because Java programs are compiled into**bytecode**, which is executed by the **Java Virtual Machine (JVM)**.
- The JVM acts as an intermediary between the program and the operating system, making **Java platform independent**.
- Because of its **platform independence, object oriented features, security, simplicity, and reliability**, Java quickly became one of the most popular programming languages for developing desktop, weband mobile applications.
- In **2010**, **Oracle Corporation**acquired**Sun Microsystems**and took over the development and maintenance of Java. Today, **Oracle** continues to release new Java versions with improved features, performance, and security.
- **Different editions (platforms) of Java**, each designed for a specific type of application.

Editions of java:

Edition	Full Form	Purpose
J2SE	Java 2 Standard Edition	Used for desktop applications and general-purpose programming. It provides the basic Java libraries.
J2EE	Java 2 Enterprise Edition	Used for developing large-scale enterprise and web applications. It includes technologies like Servlets, JSP, and EJB.
J2ME	Java 2 Micro Edition	Used for mobile phones, embedded systemsand small electronic devices with limited resources.

Java Virtual Machine (JVM):

The Java Virtual Machine (JVM) is a **software program** that executes Java bytecode andenables Java programs to run on any operating system. It is the main reason why **Java is platform independent.**

JVM working

1. The programmer writes a Java program **(.java file).**
2. The Java Compiler **(javac)**converts the **source code into bytecode(.class file**).
3. The JVM reads the **bytecode.**
4. The JVM converts **bytecode into machine code**that the operating system can understand.
5. The program is then executed.

Features of JVM

- Platform Independent: The same bytecode can run on any operating system that has a JVM.
- Loads and Executes Classes: It loads the required class files and executes them.

- Memory Management: Automatically allocates and deallocates memory.
- Garbage Collection: Removes unused objects from memory automatically.
- Security: Provides a secure environment by checking bytecode before execution.
- Exception Handling: Detects and handles runtime errors.

Advantages of JVM

- Enables Write Once, Run Anywhere (WORA).
- Improves security.
- Provides automatic memory management.
- Makes Java programs portable.
- Optimizes program execution.

Java Features/Buzzwords:

The **features** of Java are known as **Buzzwords of Java**.

List of Buzzwords

1. **Simple**
2. **Object-Oriented**
3. **Portable**
4. **Platform Independent**
5. **Secured**
6. **Robust**
7. **Architecture Neutral**
8. **Interpreted**
9. **High Performance**
10. **Multithreaded**
11. **Distributed**
12. **Dynamic**

1.Simple

Java is very easy to learnwith simple, clean, and easy-to-understand syntax.
According to Sun, Java is simple because:

- Java syntax is based on C++.
- Complicated and rarely used features are removed (e.g., explicit pointers, operator overloading).
- Automatic garbage collection eliminates the need to remove unreferenced objects manually.

2. Object-Oriented

Java is an object-oriented programming language.

- Everything in Java is an object.
- **Basic concepts of OOP in Java:**
  - **Object**
  - **Class**
  - **Inheritance**
  - **Polymorphism**
  - **Abstraction**
  - **Encapsulation**

3. Platform Independent

Java is platform independent. Unlike languages such as C++, which compile into platform-specific machine code, Java is a **write once, run anywhere** language.

4. Secured

Java enables the development of virus-free software systems.

Reasons Java is secure:

- No explicit pointers.
- Programs run inside a **virtual machine sandbox**.

5. Robust

Robust means strong. Java is robust because:

- Strong memory management.
- Lack of pointers avoids security problems.
- Automatic garbage collection runs on the JVM.
- Exception handling and type checking mechanisms strengthen reliability.

6. Architecture Neutral

The size of primitive data types is fixed, ensuring consistency across platforms.

7. Portable

Java bytecode can be carried to any platform.

- No need for specific implementations.

8. High Performance

- Faster than other interpreted languages because Java bytecode is close to native code.
- Still slower than compiled languages like C or C++.
- Java is interpreted, which makes it slower than fully compiled languages.

9. Distributed

Java facilitates the creation of distributed applications.

- Files can be accessed by calling methods from any machine on the internet.

10. Multithreaded

A thread is like a separate program executing concurrently.

Advantages:

- Threads share a common memory area (no separate memory for each thread).
- Useful for multimedia and web applications.

11. Dynamic

- Dynamic loading of classes.
- Integration with native languages (C & C++).
- Dynamic compilation and automatic memory management.

Differences Between C++ and Java

C++	Java
Supports both procedural and object-oriented programming.	Purely object-oriented programming language.
Programs can be written with or without classes.	Programs are written only with classes.
main() function is written outside the class.	main() method is written inside a class.
Platform dependent – compiled into machine code specific to the OS.	Platform independent – compiled into bytecode, runs on JVM (*Write Once, Run Anywhere*).
Manual memory management using pointers.	Automatic garbage collection; no explicit pointers.
Supports multiple inheritance directly.	Supports single inheritance; multiple inheritance achieved via interfaces.
Allows operator overloading.	Does not allow operator overloading (only method overloading).
Provides global scope and supports structures/unions.	No global scope; does not support structures/unions.
Faster execution due to direct compilation.	Slightly slower; bytecode interpreted/compiled by JVM.
Commonly used for system programming, game engines, embedded systems.	Commonly used for enterprise applications, mobile apps (Android), web apps.

1. Identifiers:

An identifier is a name given to a variable, method, array, class, interface, packageetc. for identification purpose.

Rules for writingidentifiers:

1. It can be a combination of alphabets(lowercase and uppercase), digits(0 to 9), underscore (_), and dollar sign ($).
2. It should not start with a digit.
3. Keywords cannot be used as identifiers.
4. Identifiers are strictly case-sensitive.
5. There is no restriction on the length of the identifier.
6. We can use predefined classes and predefined interfaces as identifiers.

Program: Java program on identifiers

```
class IdentifierDemo {
public static void main(String[] args) {
//String is a predefined class-identifier
int String = 10;
System.out.println(String);
// Runnableis a predefined interface-identifier
float Runnable = 20.7f;
System.out.println(Runnable);
}
}
```

Output:

```
c:\nrcm>javac IdentifierDemo.java

c:\nrcm>java IdentifierDemo
10
20.7
```

Valid / Invalid Identifiers:

Identifier	Status	Reason / Rule
Sum	Valid	Starts with a letter.
Sum	Valid	Lowercase is completely valid.
9sum	Invalid	Rule 2 fails (Cannot start with a digit).
sum#details	Invalid	Rule 1 fails (Contains an invalid character #).
sum details	Invalid	Rule 1fails (Contains an invalid characterspace).
$sum	Valid	May begin with a dollar sign ($).
_Sum	Valid	May begin with an underscore (_).
sum123	Valid	Alphanumeric combination is allowed.
Float	Invalid	Rule3 (float is a keyword).

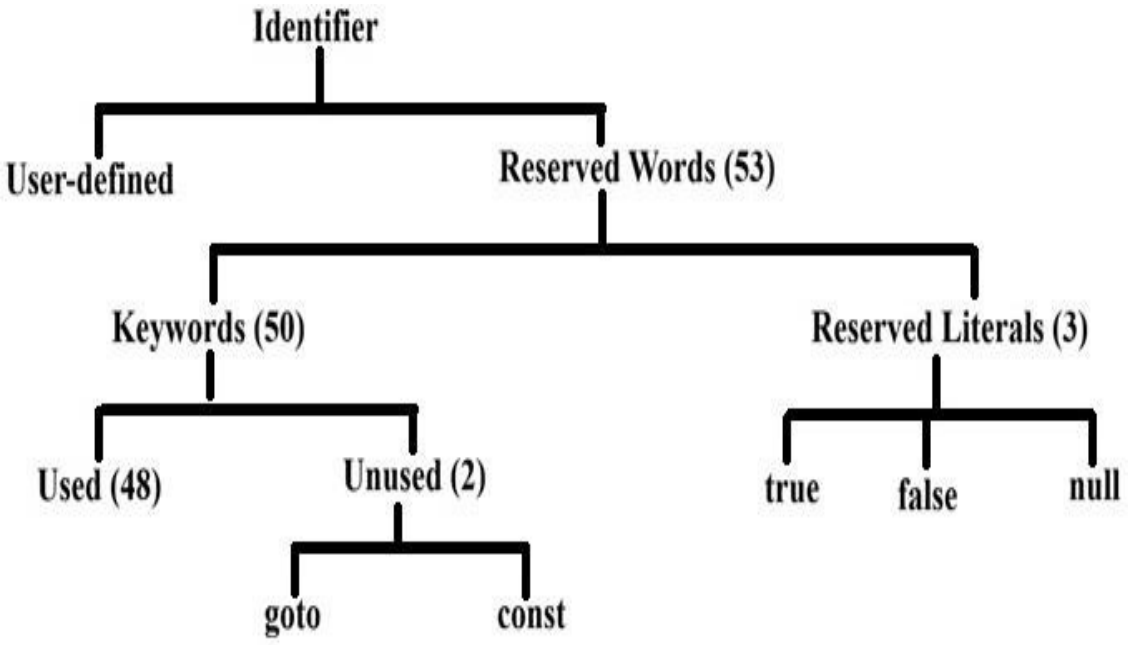
2. Reserved Words / Keywords:

Types of identifiers:

1. User defined identifiers

2. Predefined identifiers

Predefined identifiers are called reserved words.
Reserved word is word which is reserved for a specific purpose by Java developers.
There are 53 reserved words in Java which are given by Java Developers.



Java keywords (48) are categorized based on their specific functionality:

Category	Count (48)	Keywords
Primitive Data Types	8	byte, short, int, long, float, double, char, Boolean
Flow Control	11	if, else, do, while, for, switch, case, default, break, continue, return
Modifiers	11	private, protected, public, static, abstract, final, strictfp, volatile, transient, synchronized, native
Class-Related	6	class, interface, package, extends, implements, import
Object-Related	4	new, instanceof, this, super
Exception Handling	5	try, catch, finally, throw, throws
Others	3	enum, void, assert

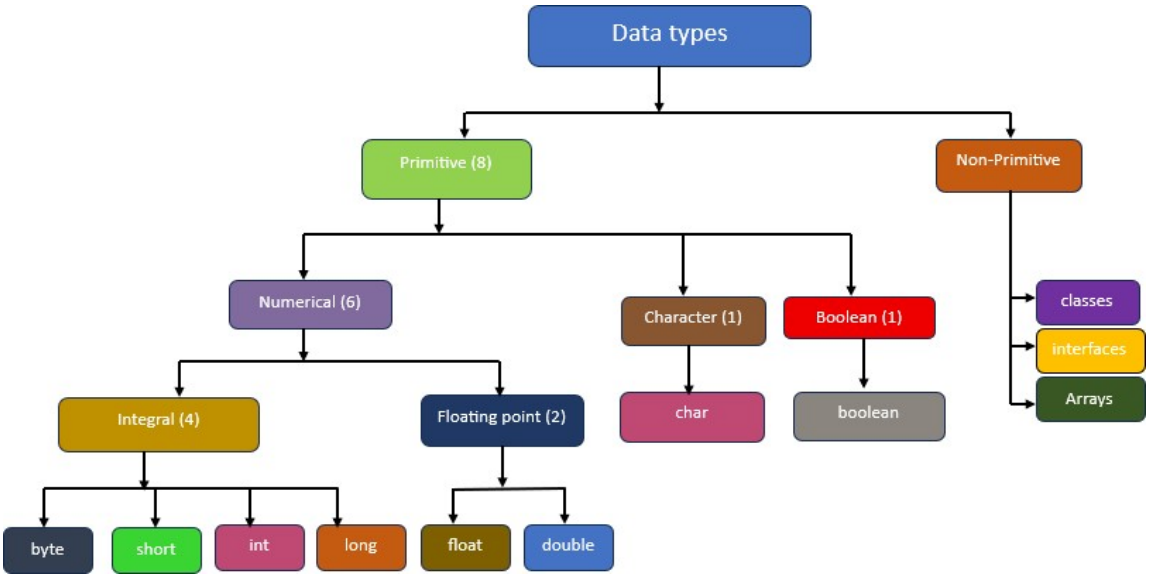
3. Data Types:

A **data type** specifies the type of data a variable can store. It determines the **size** and **range** of values that can be stored. Java is a **strongly typed** language, so every variable must have a data type. Data types help the compiler perform **type checking** and prevent invalid operations. They also determine the amount of **memory** allocated to a variable.

Types of datatypes(2):

I. Primitive Data Types (store actual values)

II. Non-Primitive Data Types (store references to objects)



Default Values and Default Data Types:

- **Integral datatypes (4):**The default value for all four integral data types is**0** and default datatype is **int**.
- **Floating-point datatypes(2):**The default value for float is **0.0f**and the default value for double is **0.0**. The default datatype is **double**.
- **Character datatype (1):** The default value is**'\u0000'**(blank space)
- **Boolean datatype (1):**The default value is**false**.

I. Primitive DataTypes(8):

Java has **8**primitive data types. They are predefined by the language and named by a reserved keyword. They can be divided into **four categories**: **Integers, Floating-Point numbers, Characters and Booleans.**

General formula for the range of an n-bit signed integer:

$$-2^{n-1} \text{ to } 2^{n-1} - 1$$

1. Numerical/Integral Datatypes (4):

Numerical (Integer) datatypes are primitive datatypes used to store whole numbers without decimal values. They can store positive numbers, negative numbers, and zero.

i. byte

- **Size:** 1 byte (8 bits).
- **Range:**-2^7 to $2^7 - 1$ i.e.–128 to 127.
- **Minimum Value:**–128.
- **Maximum Value:** 127.
- **Signed/Unsigned:** signed.

Examples:

- byte b; // Declares a variable b of type byte and stores default value 0
- byte b = 10; /*Even though integer literals defaults to int (4 bytes), 10 fits inside a byte */
- byte b = 128; // Compiler Error: Possible loss of precision,out of range
- byte b = –129; // Compiler Error: Possible loss of precision, out of range
- byte b = 10.5; // Compiler Error:Possible loss of precision
- byte b = 'A'; // Compiler Error: Incompatible types
- byte b = true; // Compiler Error: Incompatible types

ii. short

- **Size:** 2 bytes (16 bits).
- **Range:**– 2^{15} to 2^{15} – 1 i.e.– 32,768 to 32,767.
- **Minimum Value:**– 32,768.
- **Maximum Value:** 32,767.
- **Signed/Unsigned:** signed.

Examples:

- shorts; // Declares a variable s of type short and stores default value 0
- short s = 100; /* Even though integer literals defaults to int (4 bytes), 100 fits inside short */
- short s = 32768; // Compiler Error: Possible loss of precision, out of range
- short s = –32769; // Compiler Error: Possible loss of precision, out of range
- short s = 10.5; // Compiler Error: Possible loss of precision
- short s = 'a'; // Compiler Error: Incompatible types
- short s = false; // Compiler Error: Incompatible types

iii int

- **Size:** 4 bytes (32 bits).
- **Range:**–2^{31} to 2^{31}– 1.
- **Signed/Unsigned:** signed

Examples

- int a; // Declares a variable s of type int and stores default value 0
- int a = 1000; /* Even though integer literals defaults to int (4 bytes), 1000 fits inside int */
- int a = –1000;
- int a = 20.5; // Compiler Error: Possible loss of precision
- int a = 'A'; // Compiler Error:Incompatible types
- int a = true; // Compiler Error:Incompatible types

ivlong

- **Size:** 8 bytes (64 bits).
- **Range:**–2^{63} to 2^{63}–1.
- **Signed/Unsigned:** signed
- **Default Value:** 0L (or 0).
- **Literal Suffix Rules (l or L):**
  - If an integer literal is ***within***the normal **int** range, you do not need to suffix it.
    - long x = 100; // No Compiler Error

- If an integer literal is ***outside*** the int range, you must append an l or L suffix to instruct the compiler to treat the literal as a 64-bit long.
  - long y = 2147483649456; /*Compile-time Error: *Integer number too large*. The compiler reads the literal as an int by default and fails before assigning it */
  - long z= 2147483649456L; // No Compiler Error

2. Numerical / Floating-Point Data Types (Decimals)(2)

Floating point data types are primitive data types in Java used to store numbers with decimal (fractional) values.

i. float

- **Size:** 4 bytes.
- **Signed/Unsigned:** signed.

- **Precision:** Single precision (handles 5 to 6 decimal places). Has less accuracy compared to double.
- **Literal Suffix Rule:** Decimal literals default to double in Java. Therefore, you must suffix a float literal with an f or F (Ex: float f = 9.3f;).

Examples:

- floata; // Declares a variable a of type float and stores default value 0.0f
- float a = 9.8; //Compiler error
- float a = 8.9f;
- float a = - 2.34f;
- float a = 10; //compiler Error
- float a = (float) 10; // Explicit cast, a=10.0
- float a = 'A'; // Compiler Error:Incompatible types
- float a = true; // Compiler Error:Incompatible types

ii. double

- **Size:** 8 bytes.
- **Signed/Unsigned:** signed
- **Precision:** Double precision (handles 14 to 15 decimal places). Offers much higher accuracy.

Examples:

- double a; // Declares a variable a of type double and stores default value 0.0
- double a = 9.8;
- double a = (double) 10; // Explicit cast, a=10.0
- double a = 'A'; // Compiler Error: Incompatible types

3. Character Data Type (1)

char

In Java, character literals are enclosed in single quotes (' ') and represent a single Unicode character.

- **Size:** 2 bytes.
- **Purpose:** Used to store a single 16-bit Unicode character.
- **Range:** 0 to 65535.
- **Signed/Unsigned:**unsigned.
- **Default Value:** '\u0000' (null character literal), which prints out visually as a blank space.
- **Encoding Standard**: Supports Unicode (Universally accepted code), allowing it to represent characters from all global languages. It uses a 4-digit hexadecimal notation prefixed by \u.
- **ASCII Relationship**: The ASCII code system (0 to 255, limited primarily to English alphabets) is a direct subset of Unicode

Examples:

```
char c1 = 'A'; // Alphabet
char c2 = 'z'; // Lowercase letter
char c3 = '7'; // Digit character
char c4 = '$'; // Special character
char c5 = ' '; // Space
```

Escape sequence character literals

```
char c1 = '\n'; // Newline
char c2 = '\t'; // Tab
char c3 = '\''; // Single quote
char c4 = '\"'; // Double quote
char c5 = '\\'; // Backslash
```

In C language char takes 1 byte of memoryand supports ASCII code

In Java language char takes 2 bytes of memoryand supports Unicode

4. Boolean Data Type (1)

boolean

- **Size:** JVM-dependent (effectively 1 bit of information).
- **Values:** Can only hold true or false.
- **Default Value:** false.
- **Case Sensitivity:** Boolean literals must be written in lowercase (true/false). Capitalized variants like True or FALSE will result in errors.

Summary Table - Primitive Data Types

Data Type	Size (Byte)	Size (Bits)	Minimum Value	Maximum Value	Default Value	Default Data Type	Wrapper Class
byte	1	8	-128	127	0	int	Byte
short	2	16	- 32,768	32,767	0	int	Short
int	4	32	-2,147,483,648	2,147,483,647	0	int	Integer
long	8	64	-9,223,372,036, 854,775,808	9,223,372,036, 854,775,807	0/0.0L	int	Long
float	4	32	-3.4 x 10^38	3.4 x 10^38	0.0f	double	Float
double	8	64	-1.7 x 10^308	1.7 x 10^308	0.0/0.0D	double	Double
char	2	16	'\u0000' (0)	'\uffff' (65,535)	'\u0000'	char	Character
boolean	—	—	N/A	N/A	false	boolean	Boolean

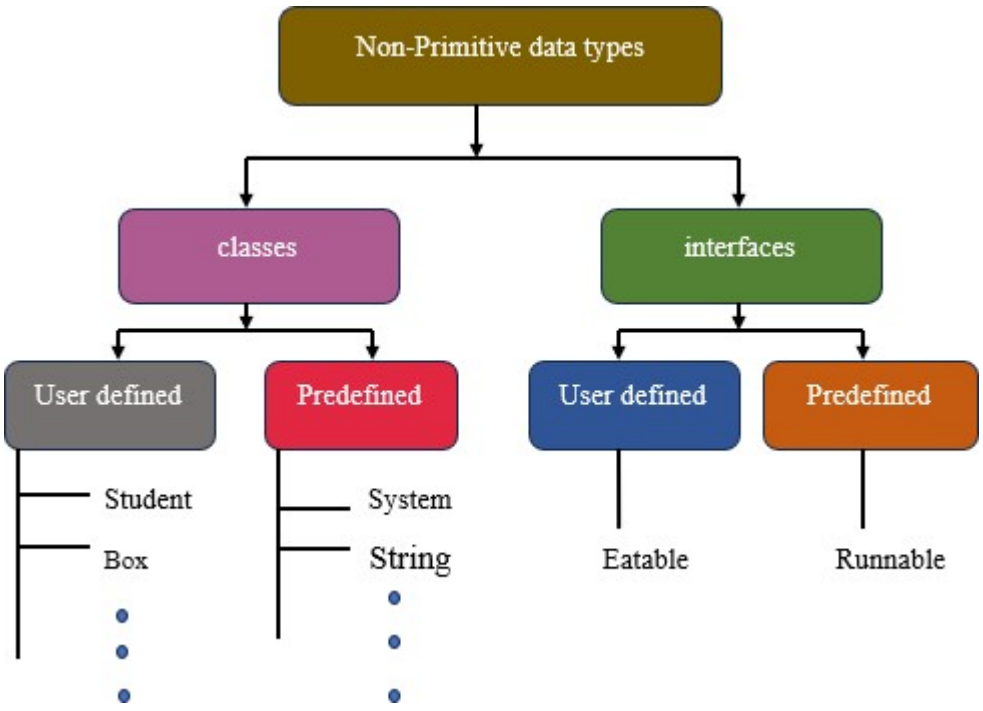
Wrapper Classes: Corresponding to each primitive data type there is a wrapper class associated with it.

1. byte – Byte
2. short – Short
3. int – Integer
4. long –Long
5. float – Float
6. double– Double
7. char – Character
8. boolean– Boolean

II. Non-Primitive Data Types (Reference Types)

Unlike primitive data types which hold their values directly, non-primitive data types hold memory addresses pointing to objects in heap memory.
The **defaultvalue** for any **non-primitive reference variable** is **null**.

Non-primitive types are divided structurally into Classes and Interfaces



1. Classes

- **Declaration:** Allocating a reference variable (e.g., Student s1;) simply places a variable on the stack containing null.
- **Instantiation:** Instantiating the object using the new keyword allocates dynamic heap memory block to the actual object and assigns its reference address back to the variable.

// Predefined Class Example
// String is a non-primitive data type
String s1; // Holds 'null' initially
s1 = new String("ramu");// Points to the address holding "ramu" in memory
// Userdefined Class Example
// Student is a non-primitive data type
Student s1; // Holds 'null' initially
s1 = new Student(); // holds the address of Student object in memory

2. Interfaces

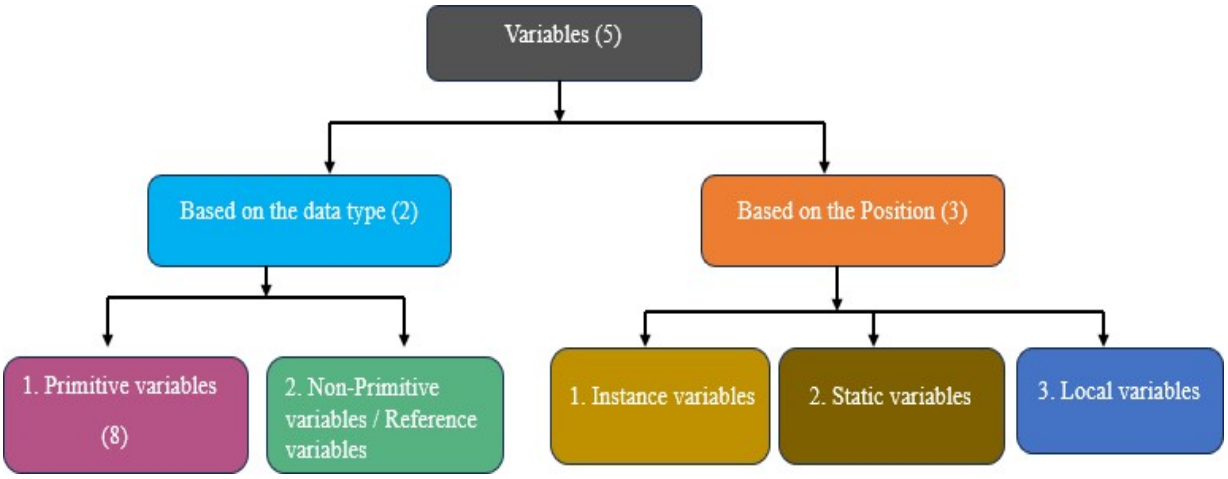
- **Rule:** You cannot instantiate objects directly for an interface. You can only declare reference variables of an interface type.

```
        // Predefined Interface Example
// Runnable is a non-primitive data type
        Runnable r1;
        // User-Defined Interface Example
// Eatable is a non-primitive data type
        Eatable e1;
```

4. Variables

A variable is a **named memory location** that holds data while a Java program is executing. Every variable must be declared with a **specific data type**, which determines its size and layout of the variable's memory.

Variable Declaration Syntax: datatype variable;



Variables: **I. Based on the datatype**

1. **Primitive Variables**
2. **Non- Primitive / Reference Variables**

II. Based on the datatype

1. **Instance Variables**
2. **Static Variables**
3. **Local Variables**

1. Classification of Variables based on Data Type: Primitive vs. Non-Primitive(Reference)

Primitive Variables	Non-Primitive (Reference) Variables
1) A variable which is declared with any one of the 8 primitive data types is called a primitive variable.	1) A variable which is declared with non-primitive datatypes (class or interface) is a reference variable.
2) Primitive variables will hold the actual values.	2) Reference variables will hold the addresses of the objects.
3) The memory is allocated for primitive variables during compilation time.	3) Memory is allocated for reference variables during the run time using new Operator.
4) Static Memory allocation.	4) Dynamic Memory allocation.

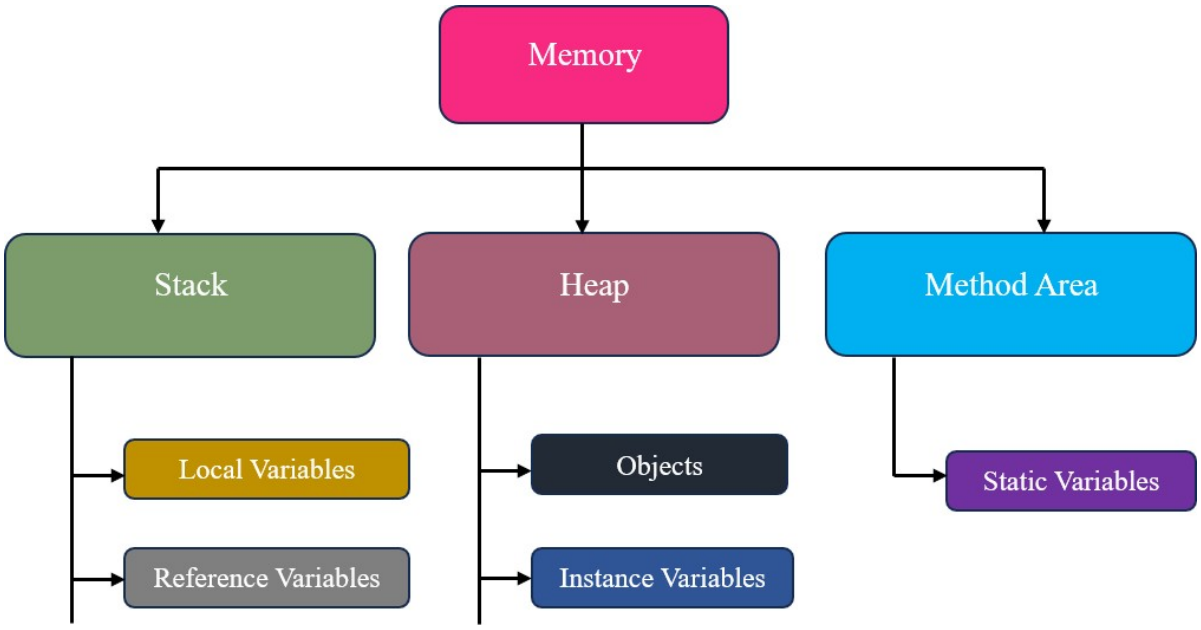
Examples:
int **a** = 30;
float **b**= 5.4f;
boolean**c**= false;
char **ch** = 'A';
a, b, c, chare primitive variables
	Examples:
	Student **s1** = new Student();
	String **s2** = new String("ramu");
	Runnable **r**;
	Eatable **e**;
	s1, s2, r, e are reference variables

2. Classification of Variablesbased on the Position (Scope)

Instance Variables	Static Variables	Local Variables
1.They are placed **inside** the class and **outside** the method or block or constructor and **not declared** with **static** keyword.	1. They are placed **inside** the class and **outside** the method or block or constructor and is **declared** with **static** keyword.	1. They are placed inside the class and **inside** the method or block / constructor.
2. Memory is allocated for instance variables inside the objects. Objects are stored in the heap memory.Therefore memory is allocated for instance variablesin **heap memory.**	2. Memory is allocated for static variables inside the**method area memory**, Sincestatic variables are class variables.	2. Memory is allocated for local variables inside the methods. Methods are stored in the stack memory.Therefore Memory is allocated for local variables in the**stack memory.**
3. Memory is allocated for instance variables inside each and every object.	3. Memory is allocated for the static variables only once for the entire class	

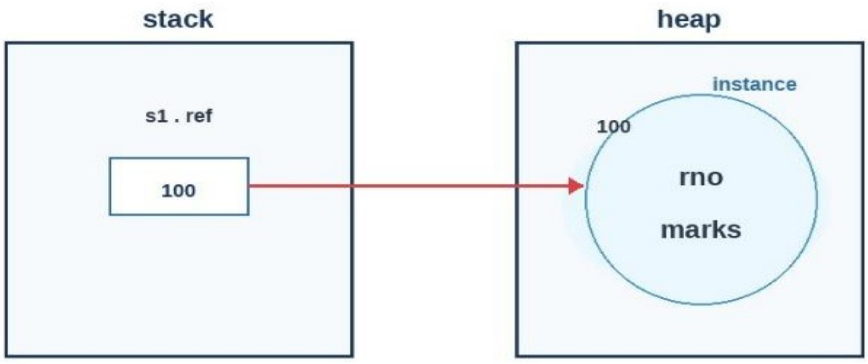
Example:	**Example:**	**Example:**

class Student {
int rno; // P.V and I.V
float marks; // P.V and I.V
String name; //N.P.V and I.V
}
	class Student {
	int rno; // P.V and I.V
	// P.V and S,V
	static int collegecode = 20;
	// N.P.V and I,V
	String name="Ramu";
	// N.P.V and S,V
	static String clgname = "NRCM";
	}
		class Student {
		// P.V and I.V
		int rno;
		void display() {
		// P.V and L.V
		int a;
		}
		}



Ex: Student s1= new Student();

Internally system will assign memory like below

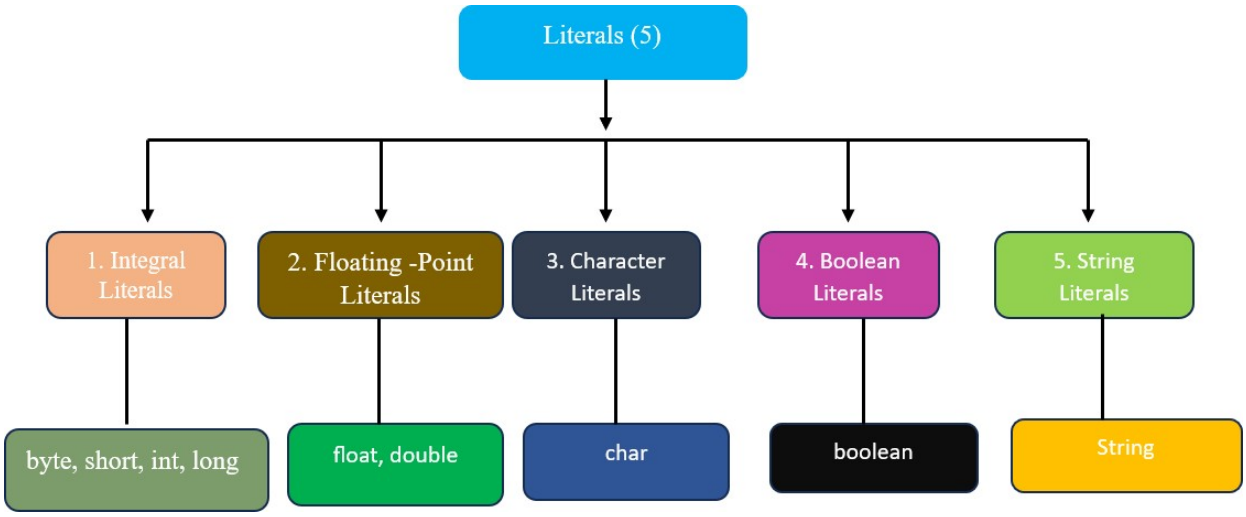


5. Literals:

- **Definition: A literal is a constant value assigned directly to a variable.**

Classification of Literals: Literals are categorized into 5 types

1. **Integral Literals (byte, short, int, long)**
2. **Floating-Point Literals (float, double)**
3. **Character Literals (char)**
4. **Boolean Literals (boolean)**
5. **String Literals (String)**



1. Integral Literals (4):

Integral literals are used for integer values.

They are categorized into:

1. **byte**
2. **short**
3. **int**
4. **long**

Integral literals can be represented in 3 differentnotations

Notation	Base	Allowed Digits / Characters	Prefix	Example
Decimal	Base 10	0 to 9	None	10
Octal	Base 8	0 to 7	0 (Zero)	010
Hexadecimal	Base 16	0 to 9, A to F (or a to f)	0x or 0X	0x10

Program 1: Java Program on Integral Literals

```
// Java program using integral literals
class Test {
public static void main(String[] args)
{ byte b = 10; // Decimal notation
int a = 010; // Octal notation (Evaluates to 8 in decimal)
long l1 = 0x10; // Hexadecimal notation (Evaluates to 16 in decimal)
System.out.println(b);
System.out.println(a);
System.out.println(l1);
}
}
```

Output:

```
c:\nrcm>javac Test.java

c:\nrcm>java Test
10
8
16
```

2. Floating-Point Literals(2):

Floating-point literals are used for fractional or decimal values.

They are categorized into:

- **float**
- **double**

Program 2: Java Program on Floating-Point Literals

```
// Java program using Floating- point literals
class Test {
public static void main(String[] args)
{ float f1 = 2.33f;  // float literal
double d1 = 10.3;     // double literal
System.out.println(f1);
System.out.println(d1);
}
}
```

Output:

```
c:\nrcm>javac Test.java

c:\nrcm>java Test
2.33
10.3
```

3. Character Literal (1):

Character literals represent single characters using char data type.
Java uses the Unicode system to represent characters (range: 0 to 65535).
They can be defined using:

1. **Single Quotes:** Direct character assignment (e.g., 'a').
2. **Integral / ASCII Values:** Decimal value representation (e.g., 97 for 'a').
3. **Unicode Notation:** Represented as \uxxxx where xxxx is a 4-digit hexadecimal number (0-9, A-F).

Program 3: Java Program on Character Literals

```
// Java program using character literals
class Test {
public static void main(String[] args)
{ char c1 = 'a';     // Direct character
char c2 = 97;         // ASCII value for 'a'
// Unicode notation for 'a' (0061 in Hex = 97 in Decimal)
char c3 = '\u0061';
System.out.println(c1);
System.out.println(c2);
System.out.println(c3);
}
}
```

Output:

```
c:\nrcm>javac Test.java

c:\nrcm>java Test
a
a
a
```

4. Boolean Literals

Boolean literals represent logical values and can only have two possible states: true or false.
They are case-sensitive in Java (must be written in lowercase) and are assigned to variables of the boolean data type.

Program 4: Java Program on Boolean Literals

```
// Java program using boolean literals
class Test {
public static void main(String[] args)
{ boolean b = true; // true literal
boolean a = false; // false literal
System.out.println(b + " " + a);
}
}
```

Output:

```
c:\nrcm>javac Test.java

c:\nrcm>java Test
true false
```

5. String Literals:

String literals represent sequences of characters enclosed in double quotes. A String is a non-primitive (reference) data type in Java.

Program 5: Java Program on String Literals

```
class Test {
public static void main(String[] args)
{ String s1 = "ramu";
String s2 = "Krishna";
System.out.println(s1); // Output: ramu
System.out.println(s2); // Output: Krishna
}
}
```

Output:

```
c:\nrcm>javac Test.java

c:\nrcm>java Test
ramu
Krishna
```

Primitive Type Conversion:

Primitive type conversion is the process of converting a value from one primitive data type to another. Java performs this conversion either **automatically (implicit conversion)** or **manually (explicit conversion)**.

There are **two types**of primitive type conversion.

1. **Widening (Implicit/Automatic) Conversion**
2. **Narrowing (Explicit/Manual) Conversion**

1. Widening Type Conversion (Automatic):

Implicit Type Conversion is the automatic conversion of a smaller data type into a larger data type. No explicit cast is required because there is no loss of data.

- Converts a **smaller data type to a larger data type**.
- Done **automatically** by the Java compiler.
- **No data loss** occurs.

Conversion Order

byte → short → int → long → float → double
↘
char → int → long → float → double

Program 1: Java Program on Implicit Type Conversion

```
class ImplicitDemo {
        public static void main(String[] args)
        { int num = 100;
        double d = num;   // Automatic conversion
System.out.println("Integer Value = " + num);
System.out.println("Double Value = " + d);
        }
  }
```

Output:

```
c:\nrcm>javac ImplicitDemo.java

c:\nrcm>java ImplicitDemo
Integer Value = 100
Double Value = 100.0
```

2. Narrowing Type Conversion (Explicit)

Explicit Type Conversion is the conversion of a larger data type into a smaller data type.

Anexplicit cast is required because data may be lost.

- Converts a **larger data type to a smaller data type**.
- Requires **type casting**.
- **Data loss may occur**.

Program 2: Java Program on Explicit Type Conversion

```
class ExplicitDemo {
public static void main(String[] args)
{ double d = 123.75;
int num = (int)d; // Explicit conversion
System.out.println("Double Value = " + d);
System.out.println("Integer Value = " + num);
}
}
```

Output:

```
c:\nrcm>javac ExplicitDemo.java

c:\nrcm>java ExplicitDemo
Double Value = 123.75
Integer Value = 123
```

3. Character to Integer Conversion

- **A char is converted to its Unicode (ASCII) value.**

Program 3: Java Program on Implicit Type Conversion

```
class CharExample {
public static void main(String[] args)
{ char ch = 'A';
int num = ch;
System.out.println(num);
}
}
```

Output:

```
C:\nrcm>javac CharExample.java

C:\nrcm>java CharExample
65
```

4. Integer to Character Conversion

Program 4: Java Program on Explicit Type Conversion

```
class IntToChar {
        public static void main(String[] args)
        { int num = 66;
        char ch = (char) num;
        System.out.println(ch);
        }
}
```

Output:

```
C:\nrcm>javac IntToChar.java

C:\nrcm>java IntToChar
B
```

5. Data Loss in Narrowing

Program 5: Java Program on Data Loss in Narrowing

```
class DataLossExample {
public static void main(String[] args)
{ int num = 130;
byte b = (byte) num;
System.out.println("Integer: " + num);
System.out.println("Byte: " + b);
}
}
```

Output:

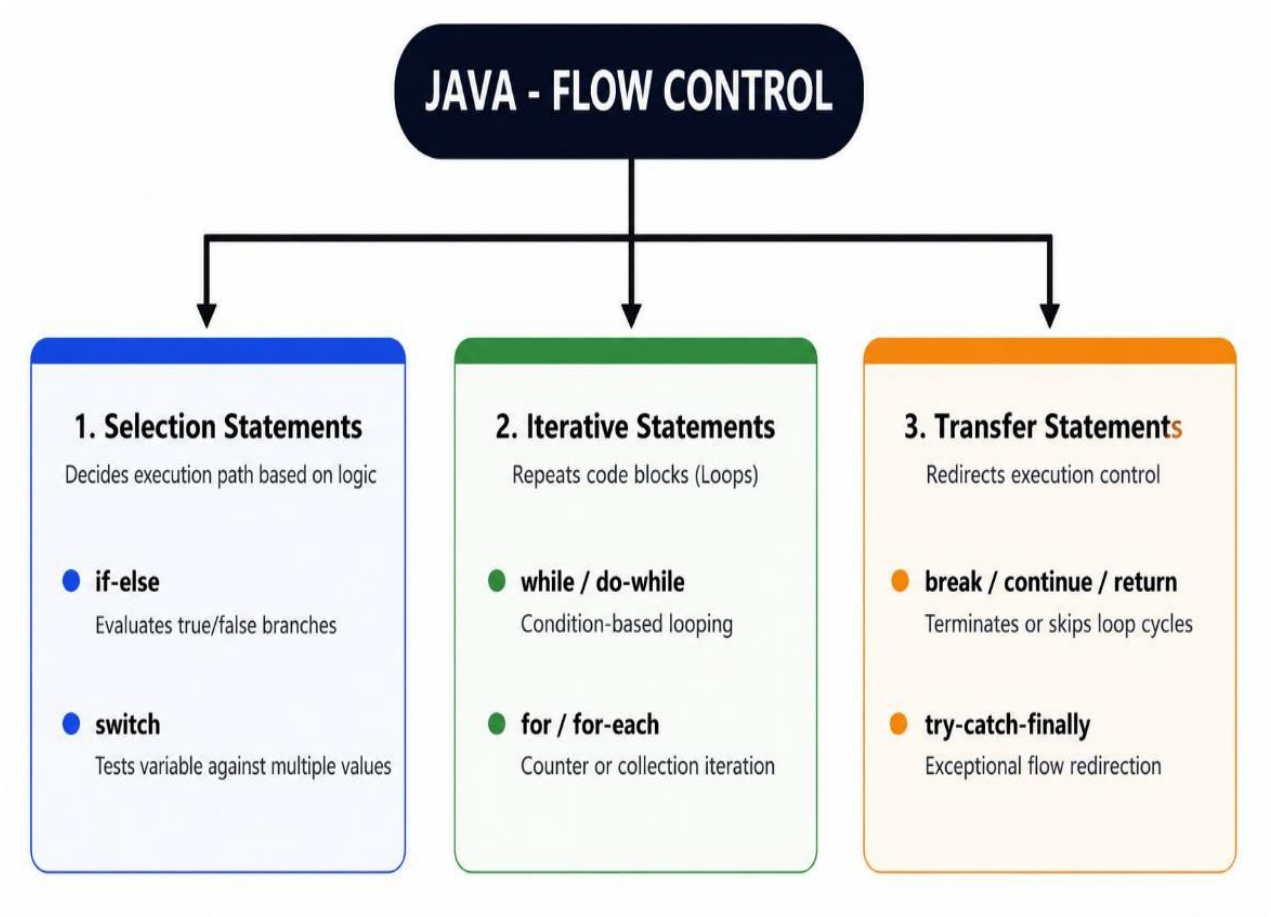
```
C:\nrcm>javac DataLossExample.java

C:\nrcm>java DataLossExample
Integer: 130
Byte: -126
```

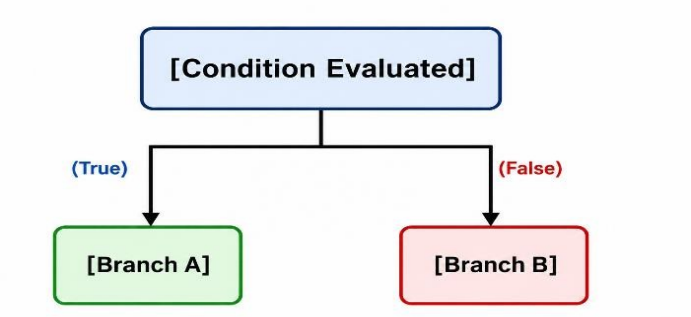
Here, the value changes because a byte can store only values from -128 to 127.

Flow Control:

Flow control describes **the order** in which individual statements inside a program are executed at runtime. **Java flow control is categorized into three main parts:**



1. Selection Statements (Branching): Selection statements, also known as conditional or branching statements, evaluate one or more conditions at runtime to choose which specific path of execution the program should take. They allow a program to make decisions and execute code blocks selectively based on boolean conditions.



• if-else

- **Description:** Evaluates a **boolean** expression. If **true**, the if block executes; if **false**, the optional else block executes. The argument inside if(...) must strictly resolve to a boolean type.
- **Code Example:**

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

• switch

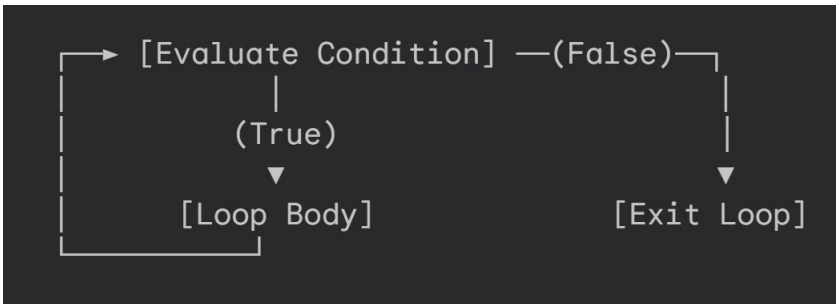
- **Description:** Compares an expression against multiple potential **constant values** (case labels) and executes the matching block. It uses **break** statements to prevent falling through to subsequent cases. Supported types include byte, short, char, int, their corresponding wrapper classes, enum types and String.
- **Code Example:**

```
int day = 2;
switch (day)
{ case 1:
System.out.println("Monday");
break;
case 2:
System.out.println("Tuesday");
break;
default:
System.out.println("Weekend/Other");
}
```

2. Iterative Statements (Loops):

Iterative statements, commonly referred to as **loops**, enable the repeated execution of a block of code as long as a controlling boolean condition remains true. They are divided into:

- **Pre-test loops (while, for):** The condition is checked ***before*** **executing** the loop body.
- **Post-test loops (do-while):** The condition is checked ***after*** **executing** the loop body, guaranteeing at least one execution.



• while

- **Description:** Repeats execution of its block as long as its condition evaluates to true. If the condition starts as false, the loop body never runs.
- **Code Example**:
  int count = 1;
  while(count <= 3) {

System.out.println("Count: " + count);
count++;
}

• do-while

- **Description:** Executes the loop body first, then evaluates the condition. This guarantees that the loop runs at least once, even if the condition is initially false.
  Requires a **trailing semicolon ;**
- **Code Example:**
  int count = 5;
  do {

System.out.println("This prints once even if condition is false.");
} **while (count < 3);**

• for

- **Description:** Combines initialization, conditional testing, and loop-counter updating in a single line. It is highly optimized and best suited when the exact number of iterations is known in advance.
- **Code Example:**

for (int i = 0; i< 3; i++)
{ System.out.println("Iteration: " + i);
}

• for-each (Enhanced For)

- **Description:** Iterates through arrays or Iterable collections sequentially without tracking an index variable. This prevents standard array out-of-bounds errors and makes collection traversal clean.
  Code Example:int[] numbers = {10, 20, 30};
  for (int num : numbers)

{ System.out.println(num);

}

3. Transfer Statements: Transfer statements immediately **redirect execution control** from its current position to another location in the program, **bypassing standard sequential flow**. They are used to **alter loop cycles**, terminate blocks early, return data from method stacks or manage exception handling.

[Sequential Execution]
|
▼
[Transfer Triggered] ——► Jump instantly to a new line
| (skip remaining block)
▼
(Normal flow ends)

Element Breakdown

• break

- **Description:** Instantly terminates the execution of the innermost loop, switch block, or labeled execution block, passing control to the code immediately following that block.
- **Code Example:**

```
for (int i = 1; i<= 5; i++)
{ if (i == 3) {
break; // Stops the loop entirely
}
System.out.println(i); // Outputs: 1, 2
}
```

• continue

- **Description**: Skips the rest of the statements in the current loop iteration and immediately **jumps** execution back to the loop's evaluation check (or increment step) to begin the next iteration.
- **Code Example:**

```
for (int i = 1; i<= 5; i++)
{ if (i == 3) {
continue; // Skips printing 3
}
System.out.println(i); // Outputs: 1, 2, 4, 5
}
```

•return

- **Description:** Immediately exits the current method, terminating its execution frame and returning control (plus an optional computed value) back to the calling method.
- **Code Example:**

```
int sum(int a, int b) {
return a + b; // Exits method and returns the calculation
```

```
}
```

- **try-catch-finally**
  - **Description:** Diverts program flow to intercept runtime exceptions (try and catch) while ensuring critical cleanup code in the finally block runs regardless of whether an exception occurred.
  - **Code Example:**

```
try {
int result = 10 / 0; // Triggers an ArithmeticException
}
catch (ArithmeticException e)
{ System.out.println("Exception caught: Division by zero.");
}
finally {
System.out.println("This block always runs.");
}
```

Operators:

An operator in programming is a special symbol (or keyword) that tells the compiler to perform a specific mathematical, relational, or logical operation on one or more values.

Java Operator Precedence & Evaluation Order

Operator Precedence (Highest to Lowest)

1. **Unary Operators: [], x++, x--, ++x, --x, ~, !, new, <type-cast>**
2. **Arithmetic Operators: *, /, %, +, -**
3. **Shift Operators:>>, >>>, <<**
4. **Comparison Operators: <, <=, >, >=, instanceof**
5. **Equality Operators: ==, !=**
6. **Bitwise Operators: &, ^, |**
7. **Short Circuit Operators: &&, ||**
8. **Conditional Operator (Ternary): ?:**
9. **Assignment Operators: =, +=, -=, *=, /=, %=**

Types of Operators

1. **Unary Operators**
2. **Binary Operators**

3. Ternary Operator

I. Unary Operators

Unary operators require only a single operand

1. Increment (++) & Decrement (--) Operators

- **Pre-increment / Pre-decrement:** Operator is placed before the variable. The value is adjusted *before* assignment/evaluation.
- **Post-increment / Post-decrement**: Operator is placed after the variable. The value is adjusted *after* assignment/evaluation.

Expression	Initial Value of x	Value of y	Final Value of x
y = ++x;	10	11	11
y = x++;	10	10	11
y = --x;	10	9	9
y = x--;	10	10	9

Crucial Rules & Constraints:

- Applicable only to variables: We cannot apply them to constant values.
  int x = 4;
  int y = ++4; // Compile-Time Error (C.E.)
  // C.E: unexpected type; required: variable, found: value
- No Nesting Allowed: We cannot nest increment/decrement operations.
  int x = 4;
  int y = ++(++x); // C.E: unexpected type; required: variable, found: value
- Not Applicable to final variables:
  final int x = 4;
  x++; // C.E: cannot assign a value to final variable 'x'
- Applicable to all primitives except boolean:
  int x = 10; x++; // x is now 11
  char ch = 'a'; ch++; // ch is now 'b'
  double d = 10.5; d++; // d is now 11.5
  boolean b = true; b++; // C.E: operator ++ cannot be applied to Boolean

2. Bitwise Complement Operator (~)

- Applicable only for integral types (not for boolean).
- ~x calculates the bitwise inversion.

- In-memory Representation: Positive numbers are represented directly. Negative numbers are stored in 2's complement form (1's complement + 1). The most significant bit (MSB) acts as the sign bit (0 is positive, 1 is negative).

Example:

System.out.println(~4); // Output: -5

In-memory Analysis:

$$4 \equiv 0000\ldots0100 \quad (\text{Sign bit } 0 \rightarrow \text{positive})$$

$$\sim 4 \equiv 1111\ldots1011 \quad (\text{Sign bit } 1 \rightarrow \text{negative})$$

$$\text{To find value: 2's complement of } 1111\ldots1011 \rightarrow 0000\ldots0100 + 1 = 0000\ldots0101 \equiv 5$$

$$\text{Since sign bit was 1, the result is } -5.$$

3. Boolean Complement Operator (!)

Applicable only for boolean types (not for integral types).
System.out.println(!true); // false
System.out.println(!false); // true
System.out.println(!4); // C.E: operator ! cannot be applied to int

4. Type Cast Operator

Used to convert values from one data type to another.

A. Implicit Type Casting (Widening / Upcasting)

- Performed automatically by the compiler.
- Occurs when assigning a lower data type to a higher data type variable.
- No loss of information.

$$\text{byte} \longrightarrow \text{short} \longrightarrow \text{int} \longrightarrow \text{long} \longrightarrow \text{float} \longrightarrow \text{double}$$

$$\text{char} \longrightarrow \text{int}$$

Example:

int x = 'a'; // Compiler automatically converts char to int -> Output: 97
double d = 10; // Compiler automatically converts int to double -> Output: 10.0

B. Explicit Type Casting (Narrowing / Downcasting)

- Must be explicitly written by the programmer.
- Required when assigning a larger data type to a smaller data type.
- There is a high chance of loss of information (most significant bits are truncated).

$$\text{double} \longrightarrow \text{float} \longrightarrow \text{long} \longrightarrow \text{int} \longrightarrow \text{short/char} \longrightarrow \text{byte}$$

Example:

```
int x = 130;
byte b = (byte)x; // Explicit cast required.
System.out.println(b); // Output: -126
```

In-memory Analysis:

$$130 \equiv 00000000\ldots10000010 \quad \text{(32-bit int)}$$

$$\text{Casting to byte crops it to 8 bits} \rightarrow 10000010 \quad \text{(Sign bit 1} \rightarrow \text{negative)}$$

$$\text{2's complement of } 10000010 \rightarrow 01111101 + 1 = 01111110 \equiv 126$$

$$\text{Result: } -126$$

Example:

```
int x = 150;
short s = (short)x;
byte b = (byte)x;
System.out.println(s); // Output: 150 (fits within 16-bit short)
System.out.println(b); // Output: -106 (truncated to 8-bit byte)
// Assigning fractional floating points to integral types discards decimal digits
double d = 130.456;
int i = (int)d;       // Output: 130
byte b = (byte)d;     // Output: -126
```

5. new and [] Operators

- new: Used to instantiate objects dynamically. There is no delete operator in Java because garbage collection handles memory destruction.
- []: Used to declare and construct arrays.

II. Binary Operators

Binary operators require exactly two operands.

1. Arithmetic Operators (+, -, *, /, %)

If any arithmetic operation is performed between two operands a and b, the resulting type is always:

max(int, type of a, type of b)

- byte + byte = int
- byte + short = int
- short + short = int
- short + long = long
- double + float = double
- char + char = int (e.g., System.out.println('a' + 'b'); \i.e 97 + 98 = 195)

Infinity &NaN (Not a Number) Behavior:

- Integral Arithmetic (byte, short, int, long):
    - No way to represent infinity or undefined results.
    - Attempting to divide by zero causes a Runtime Exception:

      ArithmeticException: / by zero.
- System.out.println(10/0); // Throws RE: ArithmeticException: / by zero
- System.out.println(0/0); // Throws RE: ArithmeticException: / by zero
- Floating-Point Arithmetic (float, double):
    - Contains predefined constants: POSITIVE_INFINITY, NEGATIVE_INFINITY, and NaN.
    - Dividing by 0.0 does not throw an exception.
- System.out.println(10 / 0.0); // infinity
- System.out.println(-10 / 0.0); // -infinity
- System.out.println(0.0 / 0.0); // NaN
- System.out.println(-0.0 / 0.0); // NaN

Float/Double NaN Comparison Rules:

For any value x (including NaN), comparing it with Float.NaN using relational operators always evaluates to false except for != which evaluates to true.

System.out.println(10 <Float.NaN); // false

```
System.out.println(10 <= Float.NaN);       // false
System.out.println(10 >Float.NaN);         // false
System.out.println(10 == Float.NaN);       // false
System.out.println(Float.NaN == Float.NaN); // false
System.out.println(10 != Float.NaN);       // true
System.out.println(Float.NaN != Float.NaN); // true
```

2. String Concatenation Operator (+)

In Java, the + operator is overloaded.

- Acts as String Concatenation if at least one argument is a String.
- Acts as Arithmetic Addition if both arguments are numeric types. Evaluation occurs strictly from left to right.

```
String a = "ashok";
int b = 10, c = 20, d = 30;
System.out.println(a + b + c + d); // ashok102030
System.out.println(b + c + d + a); // 60ashok
System.out.println(b + c + a + d); // 30ashok30
System.out.println(b + a + c + d); // 10ashok2030
```

Compilation Type Validation:

```
String a = "bhaskar";
int b = 10, c = 20, d = 30;
a = b + c + d; // C.E: incompatible types; found: int, required: java.lang.String
a = a + b + c; // Valid
b = a + c + d; // C.E: incompatible types; found: java.lang.String, required: int
b = b + c + d; // Valid
```

3. Relational Operators (<, <=, >, >=)

Relational operators can be applied to all primitive types except boolean.

System.out.println(10 < 10.5); // true

System.out.println('a' > 100.5); // false

System.out.println('b' > 'a'); // true

System.out.println(true > false); // C.E: operator > cannot be applied to boolean

Cannot be applied to Object references:

System.out.println("ashok123" > "ashok"); // C.E: operator > cannot be applied to String, String

Nesting is not allowed:

System.out.println(10 < 20 < 30); // Evaluates as (true < 30) -> C.E: operator < cannot be applied to boolean, int

4. Equality Operators (==, !=)

Can be applied to all primitive types including boolean.

```
System.out.println(10 == 20);      // false
System.out.println('a' == 'b');    // false
System.out.println('a' == 97.0);   // true
System.out.println(false == false); // true
```

Object Reference Comparison: For object references r1 and r2, r1 == r2 returns true if and only if both point to the exact same object in the heap. It compares memory addresses, not content.

```
Thread t1 = new Thread();
Thread t2 = new Thread();
Thread t3 = t1;
System.out.println(t1 == t2); // false
System.out.println(t1 == t3); // true
```

Requirement for Reference Type Equality: Compulsory class relationship (parent-child or same type) must exist between comparing objects, otherwise a compile-time error is thrown.

```
Thread t = new Thread();
Object o = new Object();
String s = new String("durga");
System.out.println(t == o); // false (Thread and Object have relationship)
System.out.println(o == s); // false (String and Object have relationship)
System.out.println(s == t); // C.E: incompatible types: String and Thread
```

Null Comparisons: Comparing an active object reference to null is always false, but null == null is always true.

```
String s = new String("ashok");
System.out.println(s == null);    // false
```

```
        String s2 = null;
        System.out.println(s2 == null);   // true
System.out.println(null == null); // true
```

5.instanceof Operator

Used to check if an object is of a specific class or interface type.
Syntax: r instanceof X (where r is an object reference and X is a Class or Interface)

```
        Thread t = new Thread();
        System.out.println(t instanceof Thread);   // true
        System.out.println(t instanceof Object);   // true
        System.out.println(t instanceof Runnable); // true (Thread implements Runnable)
```

Relationship rule: There must be a parent-child or child-parent relationship between the reference type and the class type, otherwise it fails to compile.

```
    Thread t = new Thread();
    System.out.println(t instanceof String); // C.E: inconvertible types; found: Thread, required:
    String
```

Downcasting Check: Checking if a parent class instance is an instance of a child class returns false.

```
    Object o = new Object();
    System.out.println(o instanceof String); // false
    Object o2 = new String("ashok");
    System.out.println(o2 instanceof String); // true
    Null Check: For any class/interface $X$, checking null instanceof X always returns false.
    System.out.println(null instanceof Object); // false
```

6. Bitwise Operators (&, |, ^)

- & (AND): Evaluates to true if and only if both operands are true.
- | (OR): Evaluates to true if at least one operand is true.
- ^ (X-OR): Evaluates to true if both operands are different.

These can be applied to both boolean types and integral types.

```
// Applied to Booleans
        System.out.println(true & false); // false
        System.out.println(true | false); // true
        System.out.println(true ^ false); // true
        // Applied to Integral types (Bitwise calculation)
        System.out.println(4 & 5); // 4
        System.out.println(4 | 5); // 5
        System.out.println(4 ^ 5); // 1
```

Binary Calculation Representation:

$$\text{Value } 4 \equiv 100 \qquad \text{Value } 5 \equiv 101$$

$$\& \text{ (AND)}: \quad \begin{array}{cccc} & 1 & 0 & 0 \\ \& & 1 & 0 & 1 \\ \hline & 1 & 0 & 0 \end{array} \equiv 4$$

$$| \text{ (OR)}: \quad \begin{array}{cccc} & 1 & 0 & 0 \\ | & 1 & 0 & 1 \\ \hline & 1 & 0 & 1 \end{array} \equiv 5$$

$$\wedge \text{ (XOR)}: \quad \begin{array}{cccc} & 1 & 0 & 0 \\ \wedge & 1 & 0 & 1 \\ \hline & 0 & 0 & 1 \end{array} \equiv 1$$

7. Assignment Operators

- Simple Assignment: int x = 10;
- Chained Assignment: a = b = c = d = 20; (Cannot be used directly at declaration time).
- Example:
  int b, c, d;
  int a = b = c = d = 20; // Valid
  int a = b = c = d = 30; // C.E: cannot find symbol; symbol: variable b
- Compound Assignment: Mixed assignment operators (e.g., +=, -=, *=, /=, %=, &=, |=, ^=, >>=, >>>=, <<=).

Combined Math Assignment Evaluation:

```
int a, b, c, d;
a = b = c = d = 20;
a += b -= c *= d /= 2;
System.out.println(a + "---" + b + "---" + c + "---" + d);
// Output: -160--- -180---200---10
```

Evaluation Trace:

1. $d\ /= 2 \rightarrow d = 20/2 = 10$
2. $c * = d \rightarrow c = 20 \times 10 = 200$
3. $b - = c \rightarrow b = 20 - 200 = -180$
4. $a + = b \rightarrow a = 20 + (-180) = -160$

III. Ternary Operator

Ternary operators require exactly three operands.

1. Conditional Operator (? :)

Syntax: Expression1 ? Expression2 : Expression

Simple Evaluation:

```
int x = (10 > 20) ?30 : 40;
System.out.println(x); // Output: 40

Nested Conditional Operators:
int x = (10 > 20) ?30 : ((40 > 50) ?60 : 70);
System.out.println(x); // Output: 70
int y = (10 > 20) ?30 : ((100 < 20) ?40 : 50);
System.out.println(y); // Output: 50

Precision and Casting rules:
int a = 10, b = 20;
byte c1 = (10 > 20) ?30 : 40; // Valid (evaluates literal constants)
byte c2 = (10 < 20) ?30 : 40; // Valid
byte c3 = (a > b) ?30 :40;  // C.E: possible loss of precision; found: int, required: byte
```

Expressions:

In programming, an expression is a combination of one or more operands (such as variables, constants, literals, or method calls) and operators (like +, -, *, /) that the Java compiler evaluates to produce a single value.

1. The Core Components of an Expression

Every expression is made of two basic parts:

- Operands: The data values or variables being evaluated (e.g., a byte variable, a short variable, or a character literal like 'a').
- Operators: The symbols representing the action being performed on the operands (e.g., +).

2. How it works in your Selected Text

In your selection, the line:

char + char = int

- The Expression: 'a' + 'b' is an expression.
- The Operands: 'a' and 'b' are the operands.
- The Operator: + is the operator.
- The Evaluation: When Java runs this expression, it converts 'a' to its Unicode value (97) and 'b' to its Unicode value (98). It then performs the addition, evaluating the entire expression to a single final value: 195 (of type int).

3. Types of Expressions

Expressions can be categorized by the type of value they produce:

- Arithmetic Expressions: Evaluate to a numeric value (e.g., 10 + 20 evaluates to 30).
- Relational Expressions: Evaluate to a boolean value (e.g., x > 5 evaluates to true or false).
- Conditional Expressions: Use the ternary operator to evaluate to one of two values (e.g., (10 > 20) ?30 : 40 evaluates to 40).