Java notes

Created by: Adipta Basu

Core Java Fundamentals

What is Java? JVM, JDK, JRE

JDK is a software development kit used to build Java applications. it contains the JRE and a set of development tools.

- Includes compiler (javac), debugger, and utilities like jar and javadoc.
- Provides the JRE, so it also allows running Java programs.
- Required by developers to write, compile, and debug code.

JRE provides an environment to run Java programs but does not include development tools. It is intended for end-users who only need to execute applications.

- Contains the JVM and standard class libraries.
- Provides all runtime requirements for Java applications.
- Does not support compilation or debugging.

Working of JRE:

1. Class Loading: Loads compiled .class files into memory.
2. Bytecode Verification: Ensures security and validity of bytecode.
3. Execution: Uses the JVM (interpreter + JIT compiler) to execute the instructions and make system calls.

JVM is the core execution engine of Java. It is responsible for converting bytecode into machine-specific instructions.

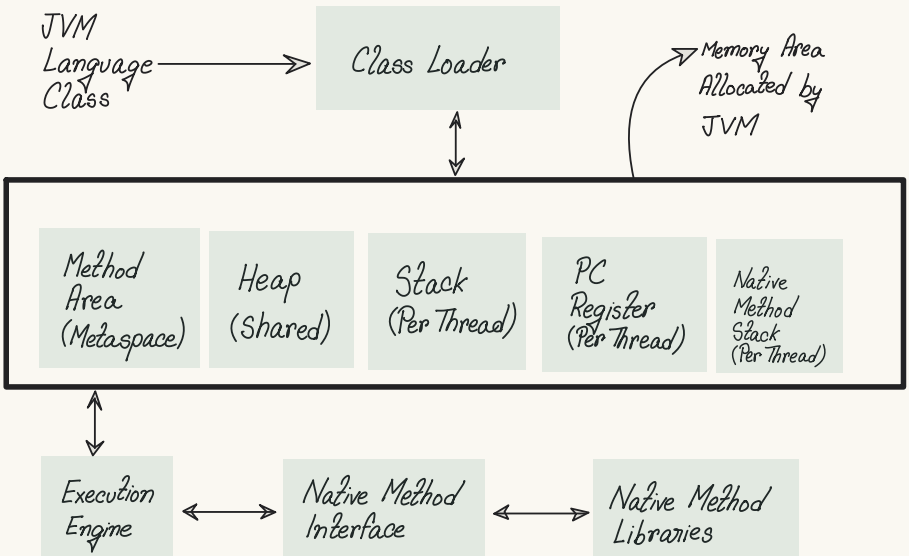
- Part of both JDK and JRE.
- Performs memory management and garbage collection.
- Provides probability by executing the same bytecode on different platforms.

Note:

- JVM implementations are platform-dependent.
- Bytecode is platform-independent and can run on any JVM.
- Modern JVMs rely heavily on Just-In-Time (JIT) compilation for performance.

Working of JVM:

1. Loading: Class loader loads bytecode into memory.
2. Linking: Performs verification, preparation, and resolution.
3. Initialization: Executes class constructors and static initializers.
4. Execution: Interprets or compiles bytecode into native code.



JDK vs JRE vs JVM

Aspect	JDK	JRE	JVM
Purpose	Used to develop Java applications	Used to run Java applications.	Executes Java bytecode
Platform Dependency	Platform-dependent (OS specific)	Platform-dependent (OS specific)	JVM is OS-specific but bytecode is platform-independent
Includes	JRE + Development tools (javac, debugger, etc.)	JVM + Libraries	ClassLoader, JIT Compiler, Garbage Collector

How Java Code Runs

Java follows the model:

Write Code → Compile → Generate Bytecode → Execute on JVM

This is what gives Java its platform independence.

1. **Writing Source Code**: Create a file such as "MyClass.java", this contains human-readable Java code.

2. **Compilation**: The java compiler compiles the ".java" file.
   `javac MyClass.java` It doesn't generate machine code. Instead, it produces a ".class" file. MyClass.class

   This .class file contains bytecode, which is:
   - Platform-independent
   - Verified and safe
   - Executed by the JVM

3. **Class Loading (by ClassLoaders)**: Java uses a hierarchical class loading:
   - Bootstrap ClassLoader
     - Loads core Java classes (java.lang.*, java.util.*).
     - Part of the JVM, written in native code.
   - Platform/Extension ClassLoader
     - Loads extension or platform modules.
     - Acts as a bridge between bootstrap and application loaders.
   - Application/System ClassLoader
     - Loads user classes from the classpath.
     - Responsible for loading the project's .class files.

   Each loader delegates to its parent first, ensuring security and consistency.

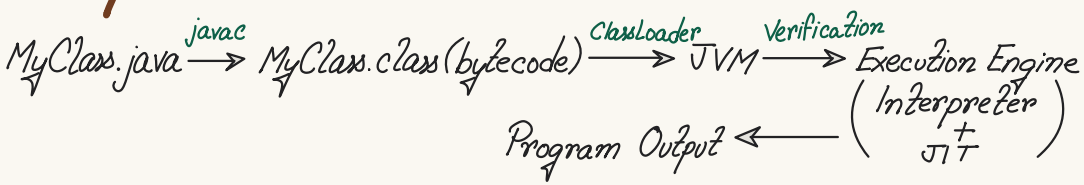
4. **Bytecode Verification**: Before execution, the JVM verifies bytecode to ensure:
   - No illegal bytecode instructions
     No stack corruption
   - No invalid type casts
   - No harmful memory access

5. **Execution by JVM**: The JVM uses two execution strategies:
   - Interpreter
     - Executes bytecode instruction by instruction.
     - Provides fast startup but slower repeated execution.
   - JIT Compiler (Just-In-Time)
     - Converts frequently used bytecode ("hotspots") into native machine code.
     - Stores compiled code in memory for reuse.
     - Greatly boosts performance during runtime.

The JVM dynamically chooses between interpreter and JIT.

6. **Runtime Execution (JVM Internals)**: The JVM uses several internal components:
   - ClassLoader Subsystem
     - Load classes when required.
   - Execution Engine
     - Interpreter
     - JIT Compiler
     - Garbage Collector
   - Runtime Data Areas
     - Heap (Objects)
     - Stack (method calls, local variables)
     - Method Area (class metadata)
     - PC Register (next instruction)
     - Native Method Stack

Compilation & Execution Flow



Important Key Points

- Java is platform-independent because of bytecode + JVM, not the language.
- JIT compiler provides near-native performance after warm-up.
- ClassLoader hierarchy ensures security and prevents malicious class replacement.
- Bytecode verification unsafe or corrupted code from running.

Java Architecture Overview

Java architecture explains how different components work together to run a Java program efficiently and securely.
At a high level, Java architecture consists of:

- Java Source Code
- Compiler
- JVM
- Runtime Environment
- Native System

Main Components of Java

1. Java Compiler (javac)
   - Converts .java source code into .class bytecode.
   - Performs syntax checking and basic validations.
   - Does not generate machine code.
2. Java Virtual Machine (JVM)
   - Core of Java architecture.
   - Executes bytecode generated by the compiler.
   - Platform-dependent (Different JVM for Windows, Linux, macOS).
   - Provides memory management, security, and performance optimizations.
3. Java Runtime Environment (JRE)
   - Provides libraries and runtime support.
   - Required to run Java applications.
   - Includes JVM, Core class libraries, supporting files
4. Java Development Kit (JDK)
   - Used to develop Java applications.
     Includes JRE, Compiler (javac), Debugger, Tools (javadoc, jar, etc.)

Internal Architecture of JVM

The JVM itself is divided into multiple subsystems:

1. **ClassLoader Subsystem**
   - Loads .class files into memory.
   - Follows parent delegation model.
   - Prevents loading of malicious classes.
2. **Runtime Data Areas**

   Memory areas used during program execution:
   - Heap
     - Stores objects and class instances.
     - Shared across threads.

       Managed by Garbage Collector.
   - Stack
     - Stores method calls and local variables.
     - Each thread has its own stack.
   - Method Area

     Stores class-level data:
     - Class metadata
     - Method bytecode
     - Static variables
   - PC Register
     - Stores address of the current instruction.
   - Native Method Stack
     - Use for native (non-Java) methods.
3. **Execution Engine**

   Responsible for executing bytecode. Includes:
   - Interpreter – executes line by line
   - JIT Compiler – converts bytecode to native code
   - Garbage Collector – frees unused memory.

Security in Java Architecture

Java provides built-in security using:

- Bytecode verification
- ClassLoader restrictions
- Security Manager (legacy/optional in modern Java)

This prevents:

- Memory corruption
- Unauthorized access
- Execution of unsafe code

Platform Independence Explained

- Java source code ⟶ bytecode (platform-independent)
- JVM converts bytecode ⟶ machine code (platform-dependent)
- Same bytecode runs on any OS with a compatible JVM.

Points to Remember

- JVM is platform-dependent; Java bytecode is not.
- JRE is required to run Java programs.
- JDK is required to develop Java programs.
- Memory management is automatic via Garbage Collection.
- JVM architecture ensures performance, security, and portability

Java Memory Model

The Java Memory Model (JMM) defines how memory is allocated, used, and shared during program execution.
Each Java program runs inside a JVM, which manages memory automatically.

Main Memory Areas in JVM

Java memory is divided into the following runtime data areas:

1. **Heap Memory**: Used to store objects and class instances.

Key Characteristics:
- Shared across all threads
- Created when JVM starts
- Managed by Garbage Collector
- Can grow or shrink dynamically.

Stores:
- Objects (`new` keyword)
- Instance Variables

Example: `Student s = new Student();`
- `s` → reference → stack
- `new Student()` → object → heap

2. **Stack Memory**: Stores method calls and local variables.

Key Characteristics:
- Each thread has its own stack.
- Memory is allocated in stack frames.
- Fast access (LIFO)
- Automatically freed when method ends.

Stack Frame Contains:
- Local variables
- Method parameters
- Return address
- Operand stack

Example:
```
void add(){
    int x = 10;
}
```
`x` → stack

3. **Method Area**: Stores class-level information.

Stores:

- Class metadata
- Method bytecode
- Static variables
- Constant pools

Key Charecteristics:

- Shared across threads.
- Created when class is loaded
- Part of heap in modern JVMs (Metaspace)

4. **PC (Program Counter) Register**: Keeps track of the current instruction executed.

Key Charecteristics:

- One PC register per thread.
- Stores address of the next bytecode instruction
- Helps in thread switching

5. **Native Method Stack**: Used for native (non-Java) methods.

Key Characteristics:

- Handles calls to C/C++ code
- Used when Java interacts with OS or hardware

Memory Interaction Example:

```
class Test
{
    static int count=0;
    void run()
    {
        int x=5;
    }
}
```

Item	Memory Area
Test class metadata	Method Area
count	Method Area
run() call	Stack
x	Stack
Object of Test	Heap

Common Memory Errors

StackOverflowError
- Deep or infinite recursion
- Stack memory exhausted

OutOfMemoryError
- Heap or Metaspace exhausted
- Too many objects or large objects

Thread Safety & Memory

- Stack memory → thread-safe (each thread has its own)
- Heap memory → shared → requires synchronization

Points to Remember

- Objects live in Heap; references live in Stack
- Static variables live in Method Area
- Stack memory is faster but limited
- Heap memory is slower but larger
- Garbage Collection works on Heap memory

Java Class Structure

A class is the fundamental building block of Java. It represents a blueprint defining:

- State → variables
- Behavior → methods

Objects are created from classes.

Basic Class Structure

```
class ClassName
{
    // 1. Fields (variables)
    // 2. Static Block
    // 3. Instance Block
    // 4. Constructors
    // 5. Methods
}
```

Order is not mandatory (except blocks), but this is the recommended structure.

1. Fields (Variables): Fields represent the state of an object or class.

   Types of Fields:

   - Instance variables → per object
   - Static variables → shared across class

```
class Student
{
    int id;                 // instance variable
    static String school;   // static variable
}
```

   Memory:

   - Instance variables → Heap
   - Static variables → Method Area

2. **Methods**: Methods define the behavior of a class.

```
void study()
{
    System.out.println("Studying");
}
```

Method Components:
- Access modifier
- Return type
- Method name
- Parameters
- Method body

Types of Methods:
- Instance methods → operate on object data
- Static methods → operate on class-level data
- Abstract methods → declared, not implemented
- Final methods → cannot be overridden

3. **Constructors**: Constructors are used to initialize objects.

Key Characteristics:
- Same name as class
- No return type (not even `void`)
- Automatically invoked during object creation

```
class Student
{
    int id;
    Student(int id)
    {
        this.id = id;
    }
}
```

Types of Constructors
- Default constructor (no arguments)
- Parameterized constructor

Constructor Chaining:
- `this()` → calls another constructor in same class.
- `super()` → call parent class constructor

4. **Static Initialization Block**: Executed once, when the class is loaded into memory.

```
static
{
    System.out.println("Class Loaded");
}
```

Used for:
- Initializing static variables
- One-time setup logic

5. Instance Initialization Block: Executed everytime an object is created, before the constructor.

```
{
    System.out.println("Object created");
}
```

Used for:

- Common initialization code shared by all constructors

Execution Order in a Java Class:

1. Static variables
2. Static blocks
3. Instance variables
4. Instance initialization blocks
5. Constructor
6. Methods

Access Modifiers in Class Members

Modifier	Within Class	Same Package	Subclass	Everywhere
private	✓	✗	✗	✗
default	✓	✓	✗	✗
protected	✓	✓	✓	✗
public	✓	✓	✓	✓

Important Rules & Notes

- Only one public class per .java file.
- File name must match the public class name
- Static members belong to the class, not objects
- Constructors are not inherited
- 'this' refers to current object
- 'super' refers to parent class

Data Types

Categories of Data Types in Java

A data type defines:

- What kind of data a variable can hold.
- How much memory it uses.
- What operations can be performed on it.

Java is a strongly typed language, meaning every variable must have a declared type.

Main Categories of Data Types

Java data types are divided into two broad categories:

1. Primitive Data Types: Primitive data types store actual values, not references.

   Key Characteristics:

   - Fixed Memory size
   - Fast access
   - Stored directly in memory
   - Cannot be null
   - Not objects

   Primitive Types:

   - byte
   - short
   - int
   - long
   - float
   - double
   - char
   - boolean

Primitive types are the building blocks of Java

2. Non-Primitive (Reference) Data Types : Reference data types store memory addresses (references), not actual data.

Key Characteristics :

- Can store complex data
- Stored in Heap memory
- Can be null
- Have method and properties
- Size is not fixed

Examples :

- Classes
- Objects
- Arrays
- Interfaces
- Strings
- Enums

`String name = "Java";`

`name` → reference stored in Stack.

`"Java"` → object stored in Heap.

Primitive vs Reference – Difference

Feature	Primitive	Reference
Stores	Actual Value	Memory address
Null Allowed	✗ No	✓ Yes
Size	Fixed	Variable
Methods	✗ No	✓ Yes
Stored in Heap	✗ No	✓ Yes

Memory Perspective (Important)

```
int x=10;
String s="Java";
```

Variables x, s(reference) are stored in Stack Memory.
"Java"(string object) is stored in Heap Memory.

Why Java Seperates Data Types

- Improves performance (primitives are faster)
- Reduces memory overhead
- Allows object-oriented features via references
- Enables better garbage collection

Important Notes to Remember

- Collections work only with reference types.
- Primitives must be wrapped using wrapper class.
- Arrays are reference types even if they store primitives
- Java does not support pointers directly.

Primitive Data Types

Primitive data types are the basic building blocks of data in Java. They store actual values, not references.

Key Characteristics of Primitive Types:

- Fixed size
- Fast performance
- Stored directly in memory
- Cannot be null.
- Not objects (no methods)

Primitive Data Types - Overview

Type	Size	Default	Range	Notes
byte	1 byte	0	-128 to 127	Used in I/O, streams
short	2 bytes	0	-32,768 to 32,767	Rarely Used
int	4 bytes	0	-2^{31} to $2^{31}-1$	Most common integers
long	8 byte	0L	-2^{63} to $2^{63}-1$	Use L suffix
float	4 bytes	0.0f	~6-7 decimal digits	Use f suffix
double	8 bytes	0.0	~15 decimal digits	Default decimal type
char	2 bytes	'\u0000'	0 to 65,535	Unicode
boolean	JVM-dependent	false	true/false	No numeric value

Integer Data Types

byte:
- 8-bit signed integer
- Memory efficient
- Common in file handling and streams.

```
byte b = 100;
```

short:
- 16 bit signed integer
- Limited usage in modern Java.

```
short s = 1000;
```

int:
- 32-bit signed integer
- Default choice for whole numbers

```
int count = 10;
```

long:
- 64-bit signed integer
- Used for very large numbers

```
long population = 8_000_000_000L;
```

Floating-Point Data Types

float:
- Single-precision
- Less accurate
- Faster but rarely used.

```
float price = 10.5f;
```

double:
- Double-precision
- More accurate
- Default for decimal values.

```
double pi = 3.14159;
```

Character Data Type(char)

- Stores a single Unicode character
- 16-bit unsigned
- Supports international characters

```
char c = 'A';
char symbol = '\u20B9'; // ₹
```

Boolean Data Type

- Represents logical values
- Only true or false
- Not convertible to/from numbers

```
boolean isValid=true;
```

Memory Behavior of Primitives

```
int x=10;
```

- x is stored directly in stack memory
- No object creation
- Very fast access

Important Rules & Notes

- Size of primitives is fixed across platforms
- Arithmetic on smaller types (byte, short) promotes to int.
- Floating-point numbers may have precision issues
- Use BigDecimal for financial calculations

Interview Key Points

- int is preferred over byte/short
- double is preferred over float
- char uses Unicode, not ASCII
- boolean size is JVM-dependent

Type Conversion in Java

Type conversion is the process of converting one data type into another.
Java supports two kinds of type conversion:

1. Widening (Implicit)
2. Narrowing (Explicit/Casting)

Widening Conversion (Implicit)

Widening conversion happens automatically.
Characteristics:

- No data loss
- Safe conversion
- Performed by compiler
- Smaller → large data type

Widening Order:

byte → short → int → long → float → double

Example:

```
int x = 10;
double d = x;
```

Narrowing Conversion (Explicit)

Narrowing conversion must be done manually.
Characteristics:

- Possible data loss
- Explicit cast required
- Larger → smaller data type

Example:

```
double d = 10.75;
int x = (int) d; // x = 10
```

Decimal part is truncated, not rounded.

Primitive Casting Examples

```
int a = 130;
byte b = (byte) a;
```

Result → b = -126

This happens due to overflow

Why Overflow Happens
byte range: -128 to 127
It will just keep the last byte, or 8 bits

Java doesn't throw an error – it wraps the value

char and numeric conversion

```
char c = 'A';
int x = c; // x = 65
int y = 66;
char d = (char) y; // d = 'B';
```

char is internally treated as an unsigned integer.

Type Promotion in Expressions

Java promotes similar types during arithmetic.
Example:

```
byte a = 10;
byte b = 20;
byte c = a + b; // ✗ compile error
```

Correct way:

```
byte c = (byte)(a + b);
```

Reason: a + b is promoted to int

Integer vs Floating Division

```
int a = 5/2;    // 2
double b = 5/2; // 2.0
double c = 5/2.0; // 2.5
```

At least one operand must be floating point

Important Casting Rules

- boolean cannot be cast to/from any other type.
- Object casting works only with inheritance
- Widening is always safe; narrowing is risky
- Casting doesn't change the underlying value, only representation

Common Pitfalls

- Data loss during narrowing
- Overflow and underflow
- Unexpected integer division
- Forgetting explicit cast

Interview Key Points

- Widening is implicit, narrowing is explicit
- Smaller integer types are promoted to int.
- Casting floating integers truncates value
- boolean has no numeric casting

Wrapper Class

Wrapper classes are object representation of primitive data types.
They allow primitives to be:

- Used in Collections
- Used with Generics
- Treated as objects

Why Wrapper Classes Exists

Java is object-oriented, but primitives are not objects.
Wrapper classes solve this by:

- Wrapping primitive values inside objects
- Providing utility methods.
- Allowing null values

Primitive ↔ Wrapper Mapping

Primitive	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Key Characteristics:

- Stored as objects in Heap memory
- Can be null
- Immutable (value cannot change)
- Belong to java.lang package

Creating Wrapper Objects

1. Using Constructor (Depricated)

```
Integer a=new Integer(10);
```

Depricated → avoid using

2. Using valueOf() (Recommended)

```
Integer b=Integer.valueOf(10);
```

- Uses caching
- Better performance

3. From String

```
Integer c=Integer.valueOf("123");
```

- Throws NumformatExcception if invalid.

4. Extracting Primitive Value

```
Integer x=10;
int y=x.intValue();
```

5. Utility Methods (Very Important)

```
Integer.parseInt("100"); // int
Integer.valueOf("100"); // Integer
```

→ returns primitive
→ wrapper

Why Collections Require Wrappers

```
List<int> list; // ✗ Not allowed
List<Integer> list; // ✓ Allowed
```

Generics work only with reference types, not primitives.

Immutability of Wrapper Classes

```
Integer a=10;
a=a+5;
```

A new object is created; original value is unchanged.

Comparison Pitfall

```
Integer a = 100;
Integer b = 100;
System.out.println(a == b); // true
Integer x = 200;
Integer y = 200;
System.out.println(x == y) // false
```

Reason:

- Integer cache range: -128 to 127
- Use .equals() for value comparison.

Primitive variable / Wrapper references are stored in Stack. Wrapper object in Heap.

Important Rules to Remember

- Wrapper classes are immutable
- Use equals(), not ==
- Avoid constructors
- Required for Collection and Streams
- Caching improves performance

Interview One-Liners

- Wrapper classes convert primitives into objects
- Need for generics and collections
- parseXxx() returns primitive, valueOf() returns wrapper
- Wrapper classes are immutable.

Autoboxing & Unboxing

Autoboxing and unboxing are automatic conversions between:

- Primitive types
- Their corresponding wrapper classes

Introduced in Java 5.

Autoboxing Primitive → Wrapper Obj

```
int a = 10;
Integer x = a; //autoboxing
```

What the compiler actually does:

```
Integer x = Integer.valueOf(a);
```

Unboxing Wrapper Obj → Primitive

```
Integer x = 20;
int b = x;
```

What the compiler actually does:

```
int b = x.intValue();
```

Why Autoboxing Exists

- To simplify code
- To allow primitives to work with:
  - Collections
  - Generics
  - Streams
- To remove boilerplate conversions

Autoboxing in Collections

```
List<Integer> list = new ArrayList<>();
list.add(10); //autoboxing
int x=list.get(0); //unboxing
```

Behind the scenes:

- 10 → Integer.valueOf(10)
- Integer → intValue()

Performance Consideration

```
for(int i=0;i<1000;i++){
    Integer x=i; //repeated boxing
}
```

- Creates many objects
- Slower than primitives
- Avoid in tight loops

Autoboxing with Operators

```
Integer a=10;
Integer b=20;
Integer c=a+b;
```

a, b are unboxed to int, then the addition happens. After which the results are boxed back into Integer.

Best Practices

- Prefer primitives for calculations.
- Avoid unnecessary boxing/unboxing
- Always check for null before unboxing
- Use .equals() for wrapper comparison

Integer Cache & Common Pitfalls

Java caches certain wrapper objects to improve performance. This mainly affects Integer (and some other wrappers).

What is Integer Cache?

Java maintains a cache of Integer object for values: -128 to 127 (inclusive)
These cached objects are reused, not recreated.

Why Integer Cache Exists

- These values are commonly used
- Reduces memory usage
- Improves performance
- Implemented inside Integer.valueOf()

How Caching Works

```
Integer a = 100;
Integer b = 100;
System.out.println(a==b); //true
```

Why?
- Both a and b point to the same cached object.

```
Integer a = 200;
Integer b = 200;
System.out.println(a==b); //false
```

Why?
- 200 is outside cached range
- Two seperate objects created.

Other Wrapper Caches

Wrapper	Cached Range
Byte	Entire Range
Short	-128 to 127
Integer	-128 to 127
Long	-128 to 127
Character	0 to 127
Boolean	true/false

Memory Perspective

Scenario	Result
Cached Integer	Same object reused
Non-cached Integer	New object created
== on wrappers	Reference comparison
.equals()	Value comparison

Best Practices

- Never use == for wrapper comparison.
- Use primitives when possible
- Be cautious with autoboxing in loops
- Check for null before unboxing.

Literals in Java

A literal is a fixed value written directly in the source code.

```
int x = 10; // 10 is a literal
```

Why Literals Matter

- Define the type and value at compile time
- Affect memory usage and type inference
- Incorrect literals cause compile-time errors

1. Integer Literals

Int Literals

By default, integer literals are of type int.

```
int a = 100;
```

Long Literals

- Must end with 'L' or 'l' (prefer 'L')

```
long population = 8_000_000_000L;
```

Without L → compile-time error (out of int range).

Numeric Separators (_)

Improves readability.

```
int salary = 1_00_000;
long distance = 9_223_372_036_854_775_807L;
```

Rules:

- Cannot be at start or end
- Cannot be next to decimal point

2. Floating-Point Literals

Double Literals

- Default type for decimals

```
double pi = 3.14;
```

Float Literals

- Must end with "f" or "F"

```
float price = 10.5f;
```

Without f → compile-time error.

Scientific Notation

```
double d = 1.2e3; // 1200.0
```

3. Character Literals (char)

Can be represented in multiple ways:

All represent "A".

```
char a = 'A';
char b = 65;          // ASCII value
char c = '\u0041'; // Unicode
```

4. Boolean Literals

Only two values:

```
boolean isValid = true;
boolean isDone = false;
```

- No numeric equivalents
- Cannot cast from/to numbers

5. Binary Literals (Java 7+)

Prefix: 0b or 0B

```
int x = 0b1010; // 10
```

6. Octal Literals

Prefix: 0

```
int x=010;//8
```

Rarely used, but important to know.

7. Hexadecimal Literals

Prefix: "0x" or "0X"

```
int x=0x1A;//26
```

Common use:

- Bit masks
- Colors
- Low-level programming

Default Types of Literals

Literal	Default Type
10	int
10L	long
10.5	double
10.5f	float
'A'	char
true	boolean

Important Notes

- Integer literals default to int
- Decimal literals default to double
- Use suffixes to change type
- Literals are resolved at compile time.

String & Text Processing Basics

In Java, text is represented using String-related classes from java.lang.

What is a String?

A String is a sequence of characters.

```
String s = "Java";
```

Key Properties:

- String is a class
- Strings are immutable
- Stored in String Pool (special heap area)

String Immutability

Once a String object is created, it cannot be changed.

```
String s = "Java";
s.concat(" World");
```

Result:

- "Java" remains unchanged
- "Java World" is created as a new object.

Why Strings are Immutable

- Security (class loaders, file paths, URLs)
- Thread safety
- Caching in String Pool
- Better memory management

String Pool

```
String a = "Java";
String b = "Java";
```

- Both a and b point to the same object
- Improves memory efficiency

```
String c = new String("Java");
```

- Forces creation of a new object outside pool.

String Creation Summary

Code	Objects Created
"Java"	1 (pool)
new String("Java")	2 (pool + heap)

StringBuilder

Mutable version of String (non-thread-safe).

```
StringBuilder sb = new StringBuilder("Java");
sb.append(" World");
```

✔ Same object modified
✔ Fast
✘ Not thread-safe

Use when:

- Single-threaded environment
- Frequent string modifications

StringBuffer

Mutable and thread-safe version of String.

```
StringBuilder sb = new StringBuilder("Java");
sb.append(" World");
```

✔ Thread-safe
✘ Slower than StringBuffer

Use when:

- Multi-threaded environment

String vs StringBuilder vs StringBuffer

Feature	String	StringBuilder	StringBuffer
Mutability	Immutable	Mutable	Mutable
Thread-safe	✓	✗	✓
Performance	Slow (modification)	Fast	Slower
Memory	Efficient (pool)	Heap	Heap

Performance Pitfall

```
String s="";
for(int i=0;i<1000;i++)
{
    s=s+i;
}
```

✗ Creates many objects → slow

```
StringBuilder sb=new StringBuilder();
for(int i=0;i<1000;i++)
{
    sb.append(i);
}
```

✓ Only one objected created

Important Notes

- Always use .equals() for String comparison
- Prefer StringBuilder for concatenation in loops
- Avoid new String() unless necessary
- Strings are immutable but references are not