Core JAVA Notes

Java Introductions:

---

1. What is Java?

Java is a high-level, object-oriented, platform-independent programming language developed by **Sun Microsystems** (now owned by Oracle) in 1995. It follows the **"Write Once, Run Anywhere" (WORA)** principle, meaning compiled Java code can run on any device with a **Java Virtual Machine (JVM)**.

Key Features of Java:

- **Platform Independent** (JVM-based)
- **Object-Oriented** (Supports classes, inheritance, polymorphism)
- **Robust & Secure** (Automatic memory management, exception handling)
- **Multithreading Support** (Concurrent execution of tasks)
- **Rich Standard Library** (Collections, I/O, networking, etc.)

Example: Hello World in Java

java

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

---

2. Where is Java Used?

Java is widely used in various domains due to its versatility, performance, and security.

Major Areas of Java Usage:

Domain	Usage
Web Development	Backend services (Spring, Jakarta EE)
Mobile Apps	Android app development (Kotlin also uses JVM)

Enterprise Apps	Banking, ERP, CRM systems (e.g., SAP, Oracle)
Big Data	Hadoop, Spark, and data processing tools
Cloud Computing	AWS, Google Cloud, and microservices
Embedded Systems	IoT devices, smart cards
Game Development	LibGDX, Minecraft (partially written in Java)
Scientific Apps	MATLAB, research simulations

---

3. Real-Life Applications of Java

Java powers many real-world applications across industries.

Examples of Java in Real Life:

1. Android Applications

- Most Android apps are built using **Java/Kotlin** (both JVM-based).
- **Example:** Twitter, Spotify, and Uber use Java for backend services.

2. Web Applications

- Java frameworks like **Spring Boot** and **Jakarta EE** are used for scalable web apps.
- **Example:** LinkedIn, Amazon (backend services).

3. Banking & Financial Systems

- Java's security makes it ideal for banking software.
- **Example:** Goldman Sachs, Citibank (transaction systems).

4. Big Data Technologies

- **Hadoop, Apache Spark**, and **Flink** use Java for distributed computing.
- **Example:** Data processing in Netflix, eBay.

5. Enterprise Software

- ERP and CRM systems rely on Java for reliability.
- **Example:** SAP, Oracle Financials.

6. Embedded & IoT Systems

- Java runs on smart cards, sensors, and IoT devices.
- **Example:** Blu-ray players, SIM cards.

7. Scientific & Research Applications

- Used in simulations and mathematical modeling.
- **Example:** NASA World Wind (geospatial data visualization).

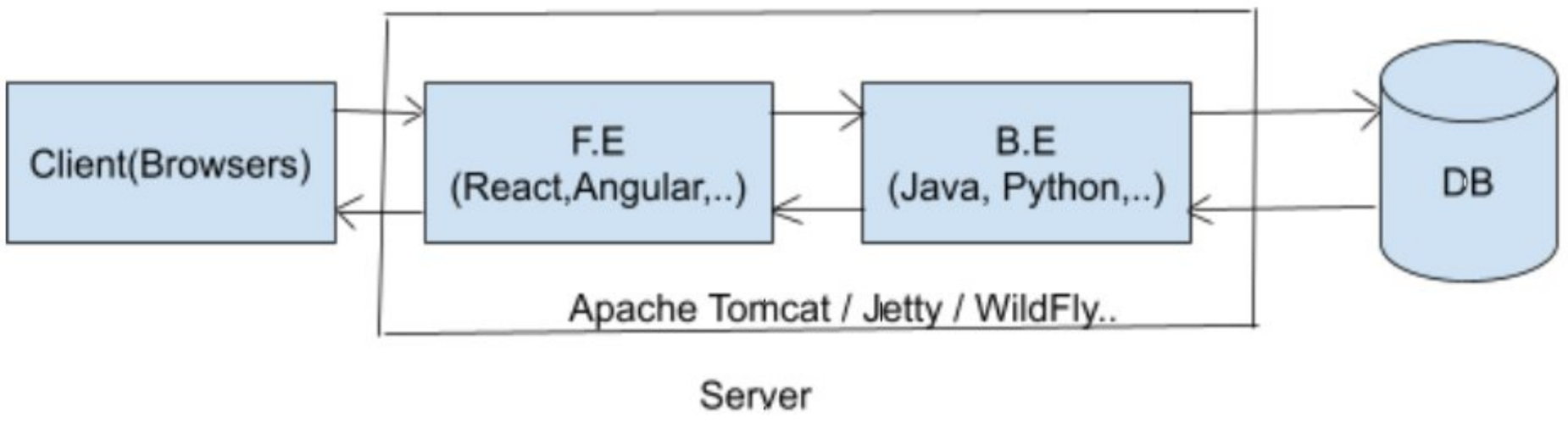
8. Gaming

- Java is used in game engines like **LibGDX**.
- **Example:** Minecraft (originally written in Java).

Conclusion

Java remains one of the most popular programming languages due to its **portability, security, and scalability**. It is used in **web, mobile, enterprise, and embedded systems**, making it a versatile choice for developers.

4. Application Flow Diagram



Topics Summary:

- **1 Data Types and Wrapper Classes**
- **1.1 Primitive Data Types**
- **1.2. Wrapper Classes**
- **1.3. Differences Between Primitive and Wrapper Classes**
- **1.4. Default Initial Values**
- **1.5. Memory Size and Locations**
- **1.6. Autoboxing and Unboxing**
- **1.7. Types of Conversions**
- **1.8. Introduction Operators**
- **1.9 Types of statements**
-

- 2 Objects, Classes and POJO Class
- 2.1. Object in Java
- 2.2. Class in Java
- 2.3. What is POJO Class and Why do we use POJO Class?
-
- 3 Java Methods/Functions: Types and Invocation Techniques
- 3.1. What is Method/Function and Why do we use methods?
- 3.2. Types of methods?
- 3.3. How many ways to call those methods?
- 3.4. Difference between Instance Methods and Static Methods?
- 3.5. What is a Constructor and Why do we use Constructors?
- 3.6. Types of Constructors?
- 3.7. Difference between this and super keywords?
- 3.8 Difference between this() and super() keywords?
- 3.9 Difference between special operators, Pre/Post increment, Pre/Post decrement, shortcuts for sum, multi,..?
-
- 4 Java OOPs Concepts
- 4.1. Inheritance and Its Types
- 4.2. Polymorphism and Its Types
- 4.3. Abstraction
- 4.4. Encapsulation
- 4.5 Interfaces
- 4.6. Upcasting and Downcasting
-
- 5 Access Modifiers
- 5.1. Access Modifiers in Java
- 5.2 Types of Access Modifiers
-
- 6 String Handling in Java: String, StringBuilder, and StringBuffer
- 6.1. String Class
- 6.2. StringBuilder Class
- 6.3. StringBuffer Class
-
- 7 Java Arrays
- 7.1 1D Array
- 7.2. 2D Arrays (Matrix) (Optional)
- 7.3. 3D Arrays (Optional)
-
- 8 Java Exceptions
- 8.1. Introduction to Exceptions
- 8.2. Types of Exceptions
- 8.3. Ways to Handle Exceptions
- 8.4. Best Practices for Exception Handling
-
- 9 Java Threads and Multithreading (Optional)
- 9.1. Introduction to Threads
- 9.2. Multithreading in Java
- 9.3. Thread Lifecycle
- 9.4. Process vs. Thread
- 9.5. Thread Synchronization
-

- 10 Collections Framework
- 10.1. List Interface
- 10.2. Set Interface
- 10.3. Map Interface
- 10.4. Queue Interface (Optional)
- 10.5. Difference Between Array and ArrayList
- 10.6. Difference Between Collection and Collections
- 10.7. File I/O Handling in Java
- 10.8. Generics in Java (Optional)
-
- 11 Java 8 Features (Optional)
- 11.1. Lambda Expressions
- 11.2. Functional Interfaces
- 11.3. Stream API
- 11.4. Method References
- 11.5. Default Methods in Interfaces
- 11.6. Optional Class
- 11.7. Date and Time API (java.time)
- 11.8. Java Version History
-
- 12 Java Memory Management & Memory Leaks
- 12.1. Types of Memory Management in Java
- 12.2. What is a Memory Leak?
- 12.3. Why Does Memory Leaking Happen?
- 12.4. How to Handle Memory Leaks?
- 12.5. Example: Fixing a Memory Leak
- 12.6. Best Practices to Avoid Memory Leaks

1. Data Types and Wrapper Classes

1.1 Primitive Data Types

Java has eight primitive data types that represent simple values:

Numeric Types

- **byte**
  - Size: 8-bit (1 byte)
  - Range: -128 to 127
  - Default: 0
  - Example: `byte fileSize = 120;`
- **short**
  - Size: 16-bit (2 bytes)
  - Range: -32,768 to 32,767
  - Default: 0

    - Example: `short temperature = -200;`
- **int**
    - Size: 32-bit (4 bytes)
    - Range: -2^{31} to $2^{31}-1$
    - Default: 0
    - Example: `int population = 2147483647;`
- **long**
    - Size: 64-bit (8 bytes)
    - Range: -2^{63} to $2^{63}-1$
    - Default: 0L
    - Example: `long globalPopulation = 7900000000L;`
- **float**
    - Size: 32-bit (4 bytes)
    - Range: ±1.4E-45 to ±3.4E+38
    - Default: 0.0f
    - Example: `float piApprox = 3.14159f;`
- **double**
    - Size: 64-bit (8 bytes)
    - Range: ±4.9E-324 to ±1.7E+308
    - Default: 0.0d
    - Example: `double precisePi = 3.141592653589793;`

Non-Numeric Types

- **char**
    - Size: 16-bit (2 bytes)
    - Range: '\u0000' to '\uffff' (0 to 65,535)
    - Default: '\u0000'
    - Example: `char grade = 'A';`
- **boolean**
    - Size: Not precisely defined (typically 1 bit)
    - Range: true or false
    - Default: false
    - Example: `boolean isJavaFun = true;`

1.2. Wrapper Classes

Wrapper classes provide object representations for primitive types:

Primitive	Wrapper Class	Example
byte	Byte	`Byte b = Byte.valueOf((byte)10);`

short	Short	`Short s = Short.valueOf((short)100);`
int	Integer	`Integer i = Integer.valueOf(1000);`
long	Long	`Long l = Long.valueOf(1000000L);`
float	Float	`Float f = Float.valueOf(3.14f);`
double	Double	`Double d = Double.valueOf(3.14159);`
char	Character	`Character c = Character.valueOf('A');`
boolean	Boolean	`Boolean b = Boolean.valueOf(true);`

Autoboxing/Unboxing Examples:

java

```
// Autoboxing (primitive → wrapper)
Integer autoBoxed = 42;  // Compiler does Integer.valueOf(42)

// Unboxing (wrapper → primitive)
int unboxed = autoBoxed; // Compiler does autoBoxed.intValue()
```

1.3. Differences Between Primitive and Wrapper Classes

Feature	Primitive Types	Wrapper Classes
Type	Simple values	Objects
Memory	Fixed size (1-8 bytes)	Additional object overhead
Storage Location	Stack memory	Heap memory
Default Value	Defined (0, false, etc.)	null
Performance	Faster	Slower (object overhead)
Usage	Local variables, arrays	Collections, generics

Methods	No methods	Utility methods available
Nullability	Cannot be null	Can be null

1.4. Default Initial Values

When declared as class fields (instance variables):

Data Type	Default Value
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
boolean	false
All object types	null

Note: Local variables must be explicitly initialized before use.

1.5. Memory Size and Locations

Memory Sizes:

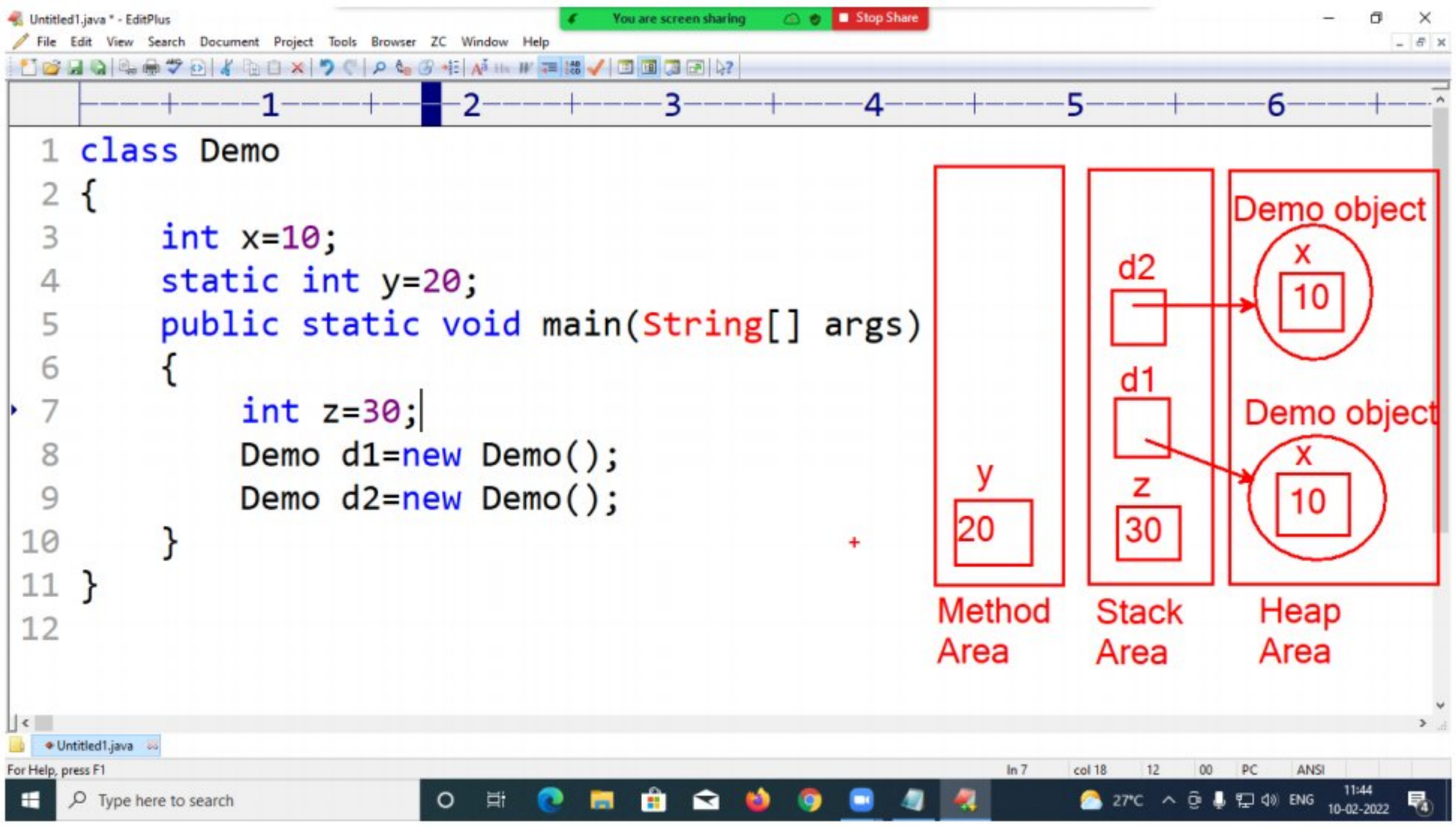
- **Primitive Types**:
  - byte, boolean: 1 byte (boolean size JVM-dependent)
  - short, char: 2 bytes
  - int, float: 4 bytes
  - long, double: 8 bytes
- **Wrapper Objects**:
  - Typically consume more memory due to object overhead (header + fields)
  - Example: Integer object requires ~16 bytes (12 object header + 4 for int value)

Memory Locations:

- **Primitive Types**:
  - Local variables: Stack memory
  - Instance variables: Inside object on heap
  - Array elements: Contiguous memory (heap for object arrays)
- **Wrapper Objects**:
  - Always allocated on heap
  - References stored in stack (for local vars) or heap (for instance vars)

Example Memory Layout:

```
class Example {
    int primitiveField;       // 4 bytes in object's heap space
    Integer wrapperField;     // Reference (4/8 bytes) + Integer object on heap

    void method() {
        int localPrimitive = 10;  // 4 bytes on stack
        Integer localWrapper = 20; // Reference on stack, object on heap
    }
}
```



1.6. Autoboxing and Unboxing

Definition:

- **Autoboxing**: Automatic conversion of primitive types to their corresponding wrapper class objects.
- **Unboxing**: Automatic conversion of wrapper class objects back to primitive types.

Examples:

Autoboxing Example:

java
```
int primitiveInt = 42;
Integer wrapperInt = primitiveInt; // Autoboxing (int → Integer)
System.out.println(wrapperInt); // Output: 42
```

Unboxing Example:

java
```
Integer wrapperInt = 100;
int primitiveInt = wrapperInt; // Unboxing (Integer → int)
System.out.println(primitiveInt); // Output: 100
```

Use in Collections:

java
```
List<Integer> numbers = new ArrayList<>();
numbers.add(5); // Autoboxing (int → Integer)
int num = numbers.get(0); // Unboxing (Integer → int)
```

1.7. Types of Conversions

Definition:

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

1. **Implicit (Widening) Casting** – Automatically done by the compiler (smaller → larger type).
2. **Explicit (Narrowing) Casting** – Requires manual intervention (larger → smaller type).

Data Type	Size (bits)	Minimum Value	Maximum Value
`byte`	8	-128	127
`short`	16	-32,768	32,767
`int`	32	-2,147,483,648	2,147,483,647
`long`	64	-9,223,372,036,854,775,808	9,223,372,036,854,775,807

`float`	32	~1.4E-45 (1.401298464324817E-45)	~3.4028235E38
`double`	64	~4.9E-324 (4.9E-324)	~1.79769313486231 57E308
`char`	16	'\u0000' (0)	'\uffff' (65,535)
`boolean`	1 (bit)	false	true

ASCII codes for uppercase (A-Z) and lowercase (a-z) letters in Java:

Uppercase Letters (A-Z)

Char	ASCII Code	Hex	Java Representation
A	65	0x41	`'\u0041'` or `'A'`
B	66	0x42	`'\u0042'` or `'B'`
C	67	0x43	`'\u0043'` or `'C'`
...	...	...	...
Z	90	0x5A	`'\u005A'` or `'Z'`

Lowercase Letters (a-z)

Char	ASCII Code	Hex	Java Representation
a	97	0x61	`'\u0061'` or `'a'`
b	98	0x62	`'\u0062'` or `'b'`
c	99	0x63	`'\u0063'` or `'c'`
...	...	...	...
z	122	0x7A	`'\u007A'` or `'z'`

```java
package Demo;
public class Main {
	public static void main(String[] args) {

		byte b = 65;
		System.out.println(b);
```

```
            b = (byte) 129;

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

boolean type cannot be converted to other types & other types cannot be converted to boolean type.

The following 19 conversions are done by system implicitly. These conversions are called widening conversions:

byte to short, int, long, float, double => 5

short to int, long, float, double => 4

int to long, float, double => 3

long to float, double => 2

float to double => 1

char to int, long, float, double => 4

==========

Total =>19

==========

```
package Demo;
public class Main {
      public static void main(String[] args) {

            byte b = 65;
            short s = b;
            System.out.println(s);

            long l = b;
            System.out.println(l);
```

```
            char c = 'A';
            System.out.println(c);

        }
}
```

The following 23 conversions are must be done by programmer explicitly otherwise compile time error occurs. These conversions are called narrowing conversions.

byte to char => 1

short to byte, char => 2

int to byte, short, char => 3

long to byte, short, int, char => 4

float to byte, short, int, long, char => 5

double to byte, short, int, long, float, char => 6

char to byte, short => 2

==========

Total =>23

==========

```
package Demo;
public class Main {
      public static void main(String[] args) {

            int b = 65;

            char c = (char) b;
            System.out.println(c);

            short s = (short) b;
            System.out.println(s);

      }
}
```

Operations on data:

1) (byte, short, int, char) + (byte, short, int, char) => int

2) (byte, short, int, long, char) + long => long

3) (byte, short, int, long, float, char) + float => float

4) (byte, short, int, long, float, double, char) + double => double

Note: Operations cannot be performed on boolean types.

```java
package Demo;
public class Main {
	public static void main(String[] args) {

		byte a = 5;
		byte b = 10;
		int sum = a+b;
		System.out.println(sum);


		char c = 'A';
		int d = 10;
		System.out.println(c+d); //65+10
	}
}
```

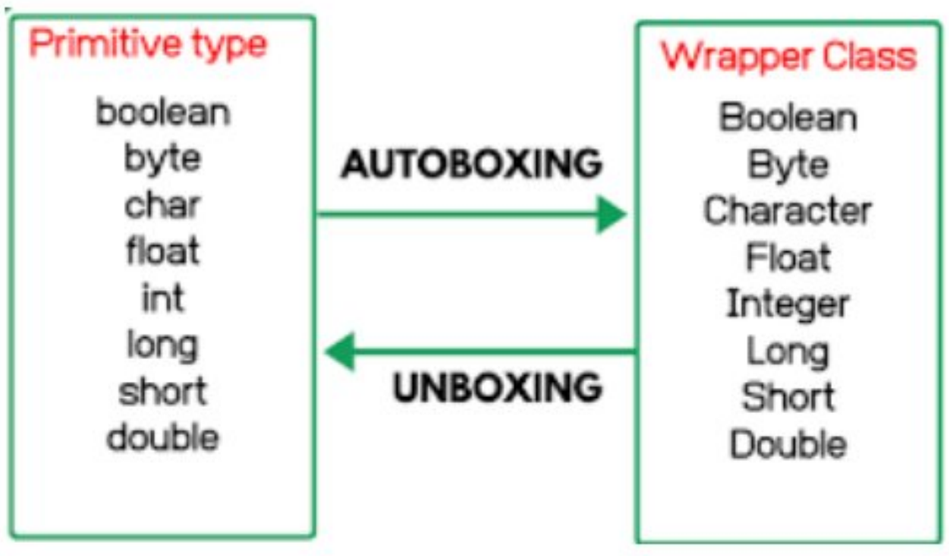
Autoboxing & Unboxing in Simple Terms

```java
package Demo;
public class Main {
	public static void main(String[] args) {

		int a = 10;
		Integer b = a; // Auto converts int to Integer (autoboxing)
		System.out.println(b);
		double d = 9.99;
		Double e = d; // Auto converts double to Double
		System.out.println(e);

		boolean flag = true;
		Boolean f = flag; // Auto converts boolean to Boolean
		System.out.println(f);
```

```
	}
}
```

```
package Demo;
public class Main {
	public static void main(String[] args) {

		Integer b = 10; // Integer object
	 int a = b; // Auto converts Integer to int (auto-unboxing)
	 System.out.println(a);
	 Double e = 9.99; // Double object
	 double d = e; // Auto converts Double to double
	 System.out.println(d);

	 Boolean f = true; // Boolean object
	 boolean flag = f; // Auto converts Boolean to boolean
	 System.out.println(flag);

	}
}
```

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



List of Java Keywords

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

Feature	Primitive Types	Wrapper Classes
Type	Basic data types (`int`, `double`, `boolean`, etc.)	Objects (`Integer`, `Double`, `Boolean`, etc.)
Memory Usage	Stored directly in stack memory (more efficient)	Stored in heap memory (require object overhead)
Default Value	`int` → `0`, `boolean` → `false`, etc.	`null` (since they are objects)
Performance	Faster (no object overhead)	Slower (due to object creation & garbage collection)

Usage	**Best for simple values and performance-critical code**	**Required for collections (`ArrayList`, `HashMap`), generics, and nullable values**
Nullability	**Cannot be `null`**	**Can be `null`**
Functionality	**Limited (just holds value)**	**Provides utility methods (e.g., `Integer.parseInt()`, `Double.compare()`)**
Auto Conversion	**Autoboxing (primitive → wrapper)**	**Auto-unboxing (wrapper → primitiv**

Main Differences Between Primitive Types and Wrapper Classes in Java

1) byte a=65; char b=a; System.out.println(b);	5) float a=3.2; int b=a; System.out.println(b);	9) short a=99; char b=(char)a; System.out.println(b);
2) byte a=10; int b=(int)a; System.out.println(b);	6) double a=4.3; int b=(int)a; System.out.println(b);	10) boolean a=true; boolean b=a; System.out.println(b);
3) char a='C'; int b=a; System.out.println(b);	7) float a=9.3f; int b=a; System.out.println(b);	11) int x=98; char y=(char)x; System.out.println(y);
4) int a=1; boolean b=a; System.out.println(b);	8) int a=10; long b=a; System.out.println(b);	12) char a='a'; int b=(int)a; System.out.println(b);

1.8 Java Operators:

Operators in Java are symbols that perform operations on variables and values. Java provides a variety of operators categorized into different types. Below is a detailed breakdown of each type with examples.

---

1. Arithmetic Operators

Used for performing mathematical operations.

Operator	Description	Example
`+`	Addition	`int sum = a + b;`
`-`	Subtraction	`int diff = a - b;`
`*`	Multiplication	`int product = a * b;`
`/`	Division	`int quotient = a / b;`
`%`	Modulus (Remainder)	`int remainder = a % b;`
`++`	Increment	`a++;` or `++a;`
`--`	Decrement	`a--;` or `--a;`

Example:

java
```
int a = 10, b = 3;
System.out.println(a + b); // 13
System.out.println(a % b); // 1 (remainder)
System.out.println(++a);   // 11 (pre-increment)
```

---

2. Relational Operators

Used for comparisons between two values.

Operator	Description	Example
`==`	Equal to	`if (a == b)`

`!=`	Not equal to	`if (a != b)`
`>`	Greater than	`if (a > b)`
`<`	Less than	`if (a < b)`
`>=`	Greater than or equal to	`if (a >= b)`
`<=`	Less than or equal to	`if (a <= b)`

Example:

java
int x = 5, y = 10;
System.out.println(x == y); // false
System.out.println(x < y); // true

---

3. Logical Operators

Used for combining multiple conditions.

Operator	Description	Example
`&&`	Logical AND	`if (a > 0 && b > 0)`
`||`	Logical OR	`if (a > 0 || b > 0)`
`!`	Logical NOT	`if (!(a == b))`

Example:

java
boolean isTrue = true, isFalse = false;
System.out.println(isTrue && isFalse); // false
System.out.println(isTrue || isFalse); // true
System.out.println(!isTrue);   // false

---

4. Assignment Operators

Used to assign values to variables.

Operator	Description	Example
`=`	Simple assignment	`int a = 5;`
`+=`	Add and assign	`a += 3;` (a = a + 3)
`-=`	Subtract and assign	`a -= 2;` (a = a - 2)
`*=`	Multiply and assign	`a *= 4;` (a = a * 4)
`/=`	Divide and assign	`a /= 2;` (a = a / 2)
`%=`	Modulus and assign	`a %= 3;` (a = a % 3)

Example:

java
int num = 10;
num += 5; // num = 15
num %= 3; // num = 0

---

5. Bitwise Operators

Used for performing bit-level operations.

Operator	Description	Example
`&`	Bitwise AND	`a & b`
`|`	Bitwise OR	`a | b`
`^`	Bitwise XOR	`a ^ b`
`~`	Bitwise NOT (Complement)	`~a`
`<<`	Left Shift	`a << 2`
`>>`	Right Shift	`a >> 2`
`>>>`	Unsigned Right Shift	`a >>> 2`

Example:

java
int x = 5; // Binary: 0101
int y = 3; // Binary: 0011

```
System.out.println(x & y); // 1 (0001)
System.out.println(x << 1); // 10 (1010)
```

---

6. Ternary Operator

A shorthand for `if-else`.

Operator	Description	Example
`?:`	Ternary (Conditional) Operator	`result = (a > b) ? a : b;`

Example:

java
```
int a = 10, b = 20;
int max = (a > b) ? a : b; // max = 20
```

---

7. instanceof Operator

Checks if an object is an instance of a class.

Operator	Description	Example
`instanceof`	Checks object type	`if (obj instanceof String)`

Example:

java
```
String str = "Hello";
System.out.println(str instanceof String); // true
```

*Untitled - Notepad
File Edit Format View Help

You are screen sharing | Stop Share

Logical Operators:

==========

Operator	Meaning
&&	Logical AND
||	Logical OR
!	Logical NOT

&&	true	false
true	true	false
false	false	false

||	true	false
true	true	true
false	true	false

! true	false
! false	true

Ln 20, Col 8 | 100% | Windows (CRLF) | UTF-8

Type here to search | 28°C | ENG | 11:15 | 31-01-2022

Pre increment Examples

```
1) int a=5;
   ++a;
   System.out.println(a); => 6
```

a: ~~5~~ 6

```
2) int a=5;
   int b=++a;
   System.out.println(a);=> 6
   System.out.println(b);=> 6
```

a: ~~5~~ 6 | b: 6

```
3) int a=5;
   System.out.println(++a);
```

a: ~~5~~ 6

6

Post increment Examples

```
1) int a=5;
   a++;
   System.out.println(a); => 6
```

a: ~~5~~ 6

```
2) int a=5;
   int b=a++;
   System.out.println(a); => 6
   System.out.println(b); => 5
```

a: ~~5~~ 6 | b: 5

```
3) int a=5;
   System.out.println(a++);
```

a: ~~5~~ 6

5

.

1.9 Java Statements

Statements in Java are complete instructions that perform actions. They can include declarations, expressions, and control flow structures. Below is a detailed breakdown of different types of statements in Java with examples.

---

1. Expression Statements

Expressions that end with a semicolon (`;`) and perform an action.

Examples:

```
java
int a = 10;          // Assignment statement
a++;                 // Increment statement
System.out.println(a); // Method call statement
```

---

2. Declaration Statements

Used to declare variables.

Examples:

```
java
int age = 25;           // Variable declaration & initialization
String name = "Alice";   // String variable
final double PI = 3.14;  // Constant declaration
```

---

3. Control Flow Statements

Used to control the execution flow of a program.

A. Conditional Statements

1. `if` Statement

Executes a block if a condition is `true`.

```
java
int num = 10;
if (num > 0) {
    System.out.println("Positive number");
}
```

2. `if-else` Statement

Executes one block if `true`, another if `false`.

```
java
int age = 17;
if (age >= 18) {
    System.out.println("Adult");
```

```
} else {
    System.out.println("Minor");
}
```

3. `if-else-if` Ladder

Checks multiple conditions sequentially.

```
java
int marks = 85;
if (marks >= 90) {
    System.out.println("Grade A");
} else if (marks >= 80) {
    System.out.println("Grade B");
} else {
    System.out.println("Grade C");
}
```

4. `switch` Statement

Selects one of many code blocks to execute.

```
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;
    default: System.out.println("Invalid day");
}
```

B. Looping Statements

1. `for` Loop

Repeats a block for a specified number of times.

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

2. `while` Loop

Repeats a block while a condition is `true`.

```
java
int i = 0;
while (i < 5) {
```

```
    System.out.println(i);
    i++;
}
```

3. `do-while` Loop

Executes at least once before checking the condition.

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

4. Enhanced `for` Loop (for-each)

Iterates over arrays and collections.

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

C. Jump Statements

1. `break` Statement

Exits a loop or `switch` block.

```
java
for (int i = 0; i < 10; i++) {
    if (i == 5) break; // Stops at 5
    System.out.println(i);
}
```

2. `continue` Statement

Skips the current iteration of a loop.

```
java
for (int i = 0; i < 5; i++) {
    if (i == 2) continue; // Skips 2
    System.out.println(i);
}
```

3. `return` Statement

Exits a method and optionally returns a value.

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

---

4. Exception Handling Statements

Used to handle runtime errors.

A. `try-catch` Block

Catches exceptions to prevent program crashes.

```
java
try {
    int result = 10 / 0;
} catch (ArithmeticException e) {
    System.out.println("Cannot divide by zero!");
}
```

B. `try-catch-finally` Block

Ensures `finally` block executes regardless of exceptions.

```
java
try {
    int[] arr = new int[3];
    arr[5] = 10; // Throws ArrayIndexOutOfBoundsException
} catch (Exception e) {
    System.out.println("Error: " + e.getMessage());
} finally {
    System.out.println("This always executes.");
}
```

C. `throw` Statement

Manually throws an exception.

```
java
if (age < 0) {
    throw new IllegalArgumentException("Age cannot be negative!");
}
```

D. `throws` Keyword

Declares exceptions a method might throw.

java
```
void readFile() throws IOException {
    // Code that may throw IOException
}
```

---

5. Synchronization Statements

Used in multithreading to control access to shared resources.

`synchronized` Block

Ensures only one thread executes a block at a time.

java
```
synchronized (this) {
    // Thread-safe code
}
```

---

6. Assertion Statements

Used for debugging to test assumptions in code.

java
```
int value = 15;
assert value > 10 : "Value must be greater than 10";
```

2. Objects, Classes, POJO Class, and OOPs Concepts

2.1. Object in Java

Definition

An **object** is a real-world entity that has a **state** (attributes/fields) and **behavior** (methods/functions).
In Java, objects are instances of classes.

Example

```
public class Car {
    String color;
    int speed;

    void drive() {
        System.out.println("Car is driving");
    }

    void stop() {
        System.out.println("Car is stopped");
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();  // Object creation
        myCar.color = "Red";
        myCar.speed = 120;

        myCar.drive();
        myCar.stop();
    }
}
```

2.2. Class in Java

Definition

A **class** is a blueprint or template that defines the structure and behavior (via fields and methods) that the objects created from the class will have.

Syntax

```
class ClassName {
    // fields
    // methods
}
```

Example

```
public class Employee {
    int id;
    String name;

    void displayInfo() {
        System.out.println("ID: " + id + ", Name: " + name);
```

```
    }
}
```

2.3. POJO (Plain Old Java Object)

Definition

A **POJO** is a simple Java object that doesn't depend on any framework.
It only contains **private fields**, **constructors**, **getters/setters**, and **toString/hashCode/equals** if needed.
It is used to represent data without any business logic.

Example

```java
package Demo;
// POJO Class Example
public class Person {
	int id;
	String name;
	public int getId() {
		return id;
	}
	public void setId(int id) {
		this.id = id;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
}
```

```java
package Demo;
public class Test {
	public void display(Person p) {
		System.out.println(p.getId()+" "+p.getName());

		p.setId(2);
		p.setName("Ramesh");
	}
}
```

```java
package Demo;
public class Main {

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

```
        Person p = new Person();
        p.setId(1);
        p.setName("Maqbool");

        Test t = new Test();
        t.display(p);

        System.out.println(p.getId()+" "+p.getName());

    }
}
```

```
public class Product {
    private int id;
    private String name;
    private double price;

    public Product() {}  // Default constructor

    public Product(int id, String name, double price) {  // Parameterized constructor
        this.id = id;
        this.name = name;
        this.price = price;
    }

    // Getters and Setters
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

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

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
```

```
        this.price = price;
    }
}
```

---

3. Java Methods/Functions: Types and Invocation Techniques

3.1. What is a Method (Function) and Why Do We Use Methods in Java?

🔹 What is a Method?

A **method** (also called a **function**) in Java is a **block of code** that performs a specific task and can be **called whenever needed**.

It is used to:

- Group related code together.
- Perform operations like calculations, data manipulation, or logic execution.
- Reuse code and improve modularity.

🔧 **Syntax of a Method:**

```
returnType methodName(parameter1, parameter2, ...) {
        // method body
        return value; // (if not void)
}
```

3.2. Types of Methods in Java

Java supports several types of methods, each serving different purposes in program structure and behavior. There are mainly 4 types of methods:

1 Without Parameter Without Return Type

Example:

```
 public void display() {
```

```
        System.out.println("Without Parameter Without Return Type called..");
    }
```

2 Without Parameter With Return Type

Example

```
public String  test(){
  String str = "Without Parameter With Return Type Called..";
   return str;
}
```

3 With Parameter Without Return Type

Example

```
public void show(String s) {
   String str = "With Parameter Without Return Type Called..";
   System.out.println(s);
 }
```

4 With Parameter With Return Type

Example

```
  public String  getShow(String s) {
     String str = "With Parameter With Return Type Called..";
     System.out.println(s);
     return str;
   }
```

1 Instance Methods

- Belong to an object of a class
- Require object creation to be called
- Can access instance variables and other instance methods

```
package Demo;

public class Demo {

public void display() {

    System.out.println("Without Parameter Without Return Type called..");

}
```

```
public String  test() {
    String str = "Without Parameter With Return Type Called..";
    return str;
}

public void show(String s) {
    String str = "With Parameter Without Return Type Called..";
    System.out.println(str);
    System.out.println(s);
}

public String  getShow(String s) {
    String str = "With Parameter With Return Type Called..";
    System.out.println(s);
        return str;
    }
}
```

```
package Demo;
public class Test {
    public static void main(String[] args) {
        Demo d = new Demo();
d.display();
String msg = d.test();
 System.out.println(msg);
d.show("Hello World");
```

```
        String showMsg = d.getShow("Hello World");

         System.out.println(showMsg);

    }

}
```

```
java
class Calculator {
    // Instance method
    public int add(int a, int b) {
        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator(); // Object creation
        int result = calc.add(5, 3);      // Calling instance method
        System.out.println(result);       // Output: 8
    }
}
```

2 Static Methods

- Belong to the class rather than any object
- Can be called without object instantiation
- Can only access static variables and other static methods

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

```
public class Main {
    public static void main(String[] args) {
        int product = MathOperations.multiply(4, 5); // Calling static method
        System.out.println(product);              // Output: 20
    }
}
```

```
package Demo;
public class Demo {

    public static void display() {
        System.out.println("Without Parameter Without Return Type called..");
    }

    public static String  test() {
        String str = "Without Parameter With Return Type Called..";
        return str;
    }

    public static void show(String s) {
        String str = "With Parameter Without Return Type Called..";
        System.out.println(str);
        System.out.println(s);
    }

    public static String  getShow(String s) {
        String str = "With Parameter With Return Type Called..";
        System.out.println(s);
        return str;
    }
}
```

```
package Demo;
public class Test {
    public static void main(String[] args) {

        Demo.display();
        String msg = Demo.test();
        System.out.println(msg);
        Demo.show("Hello World");
        String showMsg = Demo.getShow("Hello World");
        System.out.println(showMsg);

    }
}
```

3 Abstract Methods

- Declared without implementation
- Must be implemented by subclasses
- Exist in abstract classes and interfaces

```
java
abstract class Shape {
    // Abstract method
    public abstract double calculateArea();
}

class Circle extends Shape {
    private double radius;

    public Circle(double r) {
        this.radius = r;
    }

    @Override
    public double calculateArea() {
        return Math.PI * radius * radius;
    }
}
```

4 Factory Methods

- Static methods that return an instance of a class
- Often used in place of constructors
- Provide more flexible object creation

```
java
class Logger {
    private Logger() {} // Private constructor

    // Factory method
    public static Logger getLogger() {
        return new Logger();
    }

    public void log(String message) {
        System.out.println("LOG: " + message);
    }
}
```

5 Synchronized Methods

- Used in multithreading to prevent thread interference

- **Only one thread can execute a synchronized method at a time**

```
java
class Counter {
    private int count = 0;

    // Synchronized method
    public synchronized void increment() {
        count++;
    }

    public int getCount() {
        return count;
    }
}
```

3.3. Ways to Call Methods in Java

```
package Demo;
public class Demo {

	public void display() {
		System.out.println("Without Parameter Without Return Type
called..");
	}

	public String  test() {
		String str = "Without Parameter With Return Type Called..";
		return str;
	}

	public void show(String s) {
		String str = "With Parameter With Return Type Called..";
		System.out.println(str);
		System.out.println(s);
	}

	public String  getShow(String s) {
		String str = "With Parameter With Return Type Called..";
		System.out.println(str);
		System.out.println(s);
		return str;
	}
}
```

```
package Demo;
public class Test {
	public static void main(String[] args) {
```

```
		Demo d = new Demo();
		d.display();
		String msg = d.test();
		d.show("Hello World");
		String showMsg = d.getShow("Hello World");

	}
}
```

```
package Demo;
public class Demo {

	public static void display() {
		System.out.println("Without Parameter Without Return Type
called..");
	}

	public static String  test() {
		String str = "Without Parameter With Return Type Called..";
		return str;
	}

	public static void show(String s) {
		String str = "With Parameter Without Return Type Called..";
		System.out.println(str);
		System.out.println(s);
	}

	public static String  getShow(String s) {
		String str = "With Parameter With Return Type Called..";
		System.out.println(str);
		System.out.println(s);
		return str;
	}
}
```

```
package Demo;
public class Test {
	public static void main(String[] args) {

		Demo.display();
		String msg = Demo.test();
		Demo.show("Hello World");
		String showMsg = Demo.getShow("Hello World");

	}
```

```
}
```

1 Direct Invocation

The most common way to call a method by its name with appropriate arguments.

```
java
class Greeter {
    public void greet(String name) {
        System.out.println("Hello, " + name);
    }
}

public class Main {
    public static void main(String[] args) {
        Greeter greeter = new Greeter();
        greeter.greet("Alice"); // Direct method call
    }
}
```

2 Method Chaining

Multiple method calls on the same object in sequence.

```
java
class StringBuilderExample {
    private StringBuilder sb = new StringBuilder();

    public StringBuilderExample append(String str) {
        sb.append(str);
        return this;
    }

    public String toString() {
        return sb.toString();
    }
}

public class Main {
    public static void main(String[] args) {
        String result = new StringBuilderExample()
            .append("Hello")
            .append(" ")
            .append("World")
            .toString();
        System.out.println(result); // Output: Hello World
    }
```

```
}
```

3 Recursive Calls

A method that calls itself, either directly or indirectly.

```
java
class Factorial {
    public static int calculate(int n) {
        if (n <= 1) return 1;
        return n * calculate(n - 1); // Recursive call
    }
}

public class Main {
    public static void main(String[] args) {
        System.out.println(Factorial.calculate(5)); // Output: 120
    }
}
```

4 Callback Methods

Methods passed as arguments to be executed later.

```
java
interface Callback {
    void execute();
}

class Task {
    public void performTask(Callback callback) {
        System.out.println("Performing task...");
        callback.execute();
    }
}

public class Main {
    public static void main(String[] args) {
        new Task().performTask(() -> {
            System.out.println("Callback executed!");
        });
    }
}
```

5 Method References

Compact way to refer to methods using the `::` operator.

```
java
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<String> names = List.of("Alice", "Bob", "Charlie");
        names.forEach(System.out::println); // Method reference
    }
}
```

6 Reflection API

Dynamically invoking methods using Java Reflection.

```
java
import java.lang.reflect.Method;

class ReflectionExample {
    public void showMessage(String msg) {
        System.out.println("Message: " + msg);
    }
}

public class Main {
    public static void main(String[] args) throws Exception {
        Class<?> clazz = ReflectionExample.class;
        Object obj = clazz.newInstance();
        Method method = clazz.getMethod("showMessage", String.class);
        method.invoke(obj, "Hello via Reflection!");
    }
}
```

Summary Table: Method Types and Invocation

Method Type	Description	Invocation Example
Instance	Requires object creation	`obj.method()`
Static	Called on class	`Class.method()`
Abstract	Must be implemented	Overridden in subclass
Factory	Returns new instances	`Class.createInstance()`
Synchronized	Thread-safe	`synchronized method()`

Invocation Type	Description	Example
Direct	Standard method call	`obj.method()`
Chaining	Multiple calls in sequence	`obj.a().b().c()`
Recursive	Method calls itself	`method(n-1)`
Callback	Passed as parameter	`callback.execute()`
Method Reference	Compact lambda alternative	`System.out::println`
Reflection	Dynamic invocation	

3.4. Difference between Instance Methods and Static Methods in Java:

Feature	Instance Method	Static Method
Belongs to	An object (instance) of a class	The class itself, not to an object
Called using	Object reference (`obj.method()`)	Class name (`ClassName.method()`)
Can access instance variables	✅ Yes	❌ No (unless they get an object reference)
Can access static variables	✅ Yes	✅ Yes
Can use `this` keyword	✅ Yes	❌ No (no object context)
Memory Usage	Stored in object memory	Stored in class memory (once per class)
Use case	When behavior depends on individual object state	When behavior is common to all objects

✅ Example:

```
class Example {
        int x = 5;          // instance variable
```

```
static int y = 10;        // static variable

// Instance Method
void showInstance() {
System.out.println("Instance x: " + x);
System.out.println("Static y: " + y);
}

// Static Method
static void showStatic() {
        // System.out.println(x); ❌ Error: Cannot access instance variable directly
        System.out.println("Static y: " + y);
        }
}

public class Main {
        public static void main(String[] args) {
        Example obj = new Example();
        obj.showInstance();          // ✅ Instance method
        Example.showStatic();        // ✅ Static method
        }
}
```

3.5. What is a Constructor and Why Do We Use Constructors?

A constructor in Java is a special method used to initialize objects. It has the same name as the class and does not have a return type (not even `void`).

🔧 Why do we use Constructors?

- **To initialize objects at the time of creation.**
- **To assign default or user-defined values to object fields.**
- **Ensures that every object starts in a valid state.**

✅ Syntax:

```
class Student {
        String name;
        int age;
```

```
        // Constructor
        Student(String n, int a) {
        name = n;
        age = a;
        }
}
```

3.6. Types of Constructors in Java

Java supports 3 types of constructors:

1️⃣ Default Constructor

- **Created automatically by Java if no constructor is defined.**
- **Takes no parameters.**
- **Used to create objects with default values.**

```
class Student {
        String name;
        int age;

        Student() {  // default constructor
        name = "Unknown";
        age = 0;
        }
}
```

2️⃣ Parameterized Constructor

- **Takes parameters to initialize fields with custom values.**

```
class Student {
        String name;
        int age;

        Student(String n, int a) {  // parameterized constructor
        name = n;
        age = a;
        }
}
```

3️⃣ Copy Constructor

- **Used to create a copy of an object.**

- **Java doesn't provide this by default; we have to define it manually.**

```
class Student {
        String name;
        int age;

        Student(String n, int a) {
        name = n;
        age = a;
        }

        // Copy constructor
        Student(Student s) {
        name = s.name;
        age = s.age;
        }
}
```

3.7. Difference between this and super keywords?

`this` keyword:

- **Refers to the current class object.**

- **Used to access current class variables, methods, and constructors.**

- **Commonly used to resolve variable name conflicts or call another constructor in the same class.**

```java
package Demo;
public class Main {
      public static void main(String[] args) {

            Car c = new Car();
            c.setId(1);
            c.setName("BMW");

//          Car c = new Car();
//          c.setId(11);
//          c.setName("Car Engine");

            c.getCarDetails();

            c.getEngineDetails();
```

```
	}
}
```

```
package Demo;
public class Car extends Engine{

	public int id;
	public String name;

	public int getId() {
		return id;
	}
	public void setId(int id) {
		this.id = id;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}

	public void getCarDetails() {
		System.out.println("getCarDetails: "+this.id+" "+this.name);
	}

}
```

`super` keyword:

- **Refers to the parent class object.**
- **Used to access parent class variables, methods, and constructors.**
- **Mainly used in inheritance to call the superclass constructor or overridden methods.**

```
package Demo;
public class Car extends Engine{

	public int id;
	public String name;
```

```
	public void setId(int id) {
		super.id = id;
	}
	public void setName(String name) {
		super.name = name;
	}

	public void getCarDetails() {
		System.out.println("getCarDetails: "+this.id+" "+this.name);
	}

}
```

3.8 Difference between this() and super() keywords?

`this()`

- Calls a constructor in the **same class**.
- Must be the **first statement** in the constructor.
- Used for **constructor chaining** within the same class.

```
package Demo;
public class Main {
	public static void main(String[] args) {

		Bike b = new Bike(2,"Kawasaki");

//		Bike b = new Bike(22,"Bike Engine");

		b.getBikeDetails();

		b.getEngineDetails();


	}
}
```

```java
package Demo;
public class Bike extends Engine{

	public int id;
	public String name;

	public Bike() {
		System.out.println(" Bike() costructor called..");
	}

	public Bike(int id, String name) {
		this();
		this.id = id;
		this.name = name;

	}

	public void getBikeDetails() {
		System.out.println("getBikeDetails: "+this.id+" "+this.name);
	}

}
```

```java
package Demo;
public class Engine {

	public int id;
	public String name;

	public Engine() {

	}

	public Engine(int id, String name) {
		this.id=id;
		this.name = name;
	}

	public void getEngineDetails(){
		System.out.println("getEngineDetails: "+this.id+" "+this.name);
	}
}
```

---

super()

- Calls the **constructor of the parent class**.
- Must also be the **first statement** in the constructor.
- Used to **inherit and initialize** parent class members.

```java
package Demo;
public class Main {
	public static void main(String[] args) {

//		Bike b = new Bike(2,"Kawasaki");

		Bike b = new Bike(22,"Bike Engine");

		b.getBikeDetails();

		b.getEngineDetails();


	}
}
```

```java
package Demo;
public class Bike extends Engine{

	public int id;
	public String name;

	public Bike() {
		System.out.println(" Bike() costructor called..");
	}

	public Bike(int id, String name) {
		super(id,name);
//		this.id = id;
//		this.name = name;

	}

	public void getBikeDetails() {
		System.out.println("getBikeDetails: "+this.id+" "+this.name);
	}

}
```

```java
package Demo;
public class Engine {
```

```
    public int id;
    public String name;

    public Engine() {

    }

    public Engine(int id, String name) {
          this.id=id;
          this.name = name;
    }

          public void getEngineDetails(){
                System.out.println("getEngineDetails: "+this.id+" "+this.name);
          }
}
```

3.9. Difference between special operators, Pre/Post increment, Pre/Post decrement, shortcuts for sum, multi,..?

1. **Pre/Post**

```
package Demo;
public class Main {
      public static void main(String[] args) {

            int a = 5;
            //Pre Increment
//          System.out.println(++a);

            //Post Increment
//          System.out.println(a++);
//          System.out.println(a);

            //Pre deccrement
//          System.out.println(--a);

            // Post deccrement
            System.out.println(a--);
            System.out.println(a);


      }
}
```

2. **Difference between special operators shortcuts for sum, multi,..?**

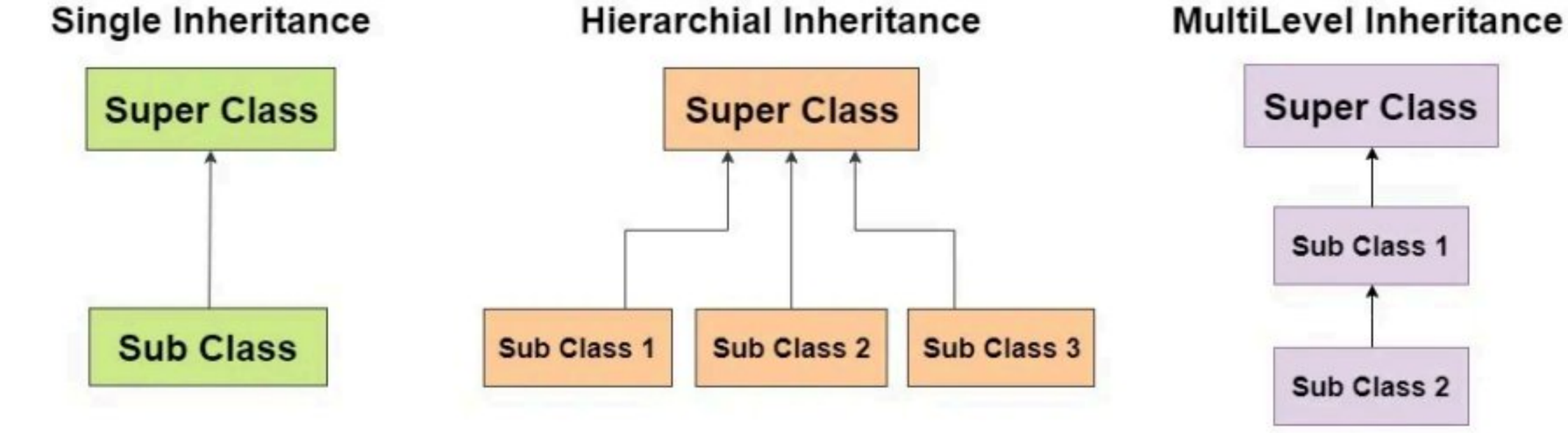
```java
package Demo;
public class Main {
	public static void main(String[] args) {

		int a =5;
		int b =10;

//		a = a+b; //a=5+10
		a+=b;

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

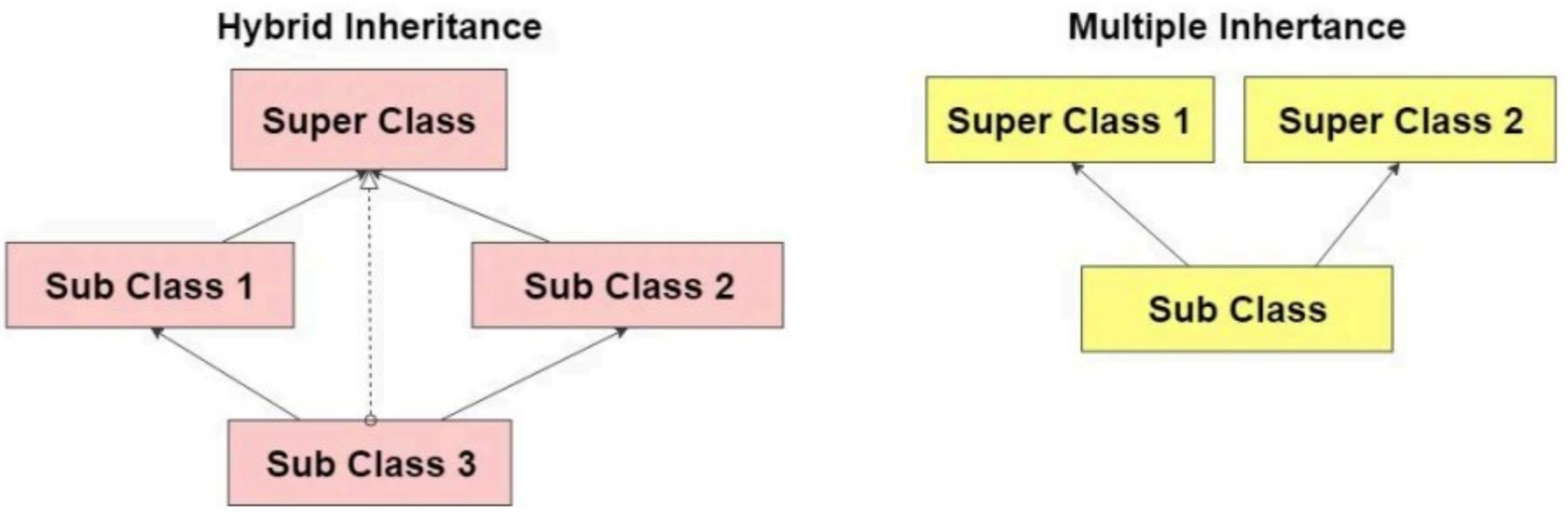
3. **Division and modular devision**

```java
package Demo;
public class Main {
	public static void main(String[] args) {

		int a =5;
		int b =11;

		int c = b/a; // 11/5 = 2
//		int c = b%a; // 11%5 = 1

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

4 Java OOPs Concepts

Java is an object-oriented programming language that follows four main pillars of OOPs: Inheritance, Polymorphism, Abstraction, and Encapsulation. These principles enable modular, scalable, and reusable code.





4.1. Inheritance

Definition:

Inheritance is a mechanism where a child class acquires the properties (fields) and behaviors (methods) of a parent class. It promotes code reusability.

Types of Inheritance in Java:

1. **Single Inheritance**
2. **Multilevel Inheritance**

3. **Hierarchical Inheritance**
   (***Note: Java doesn't support multiple inheritance with classes due to the Diamond Problem. Interfaces are used instead.***)
4. **Multiple Inheritance**
5. **Hybrid Inheritance**

Types of Inheritance:

1. **Single Inheritance – A subclass inherits from one superclass.**

```
class Animal {
    void eat() {
        System.out.println("Eating...");
    }
}
class Dog extends Animal {
    void bark() {
        System.out.println("Barking...");
    }
}
```

2 Multilevel Inheritance – A class inherits from another derived class.

```
class Animal {
    void eat() { System.out.println("Eating..."); }
}
class Dog extends Animal {
    void bark() { System.out.println("Barking..."); }
}
class Puppy extends Dog {
    void weep() { System.out.println("Weeping..."); }
}
```

3 Hierarchical Inheritance – Multiple subclasses inherit from a single superclass.

```
java

class Animal {
    void eat() { System.out.println("Eating..."); }
}
class Dog extends Animal {
    void bark() { System.out.println("Barking..."); }
}
class Cat extends Animal {
    void meow() { System.out.println("Meowing..."); }
}
```

4 Multiple Inheritance (Achieved via Interfaces) – A class implements multiple interfaces.
java
```
interface A { void showA(); }
interface B { void showB(); }
class C implements A, B {
    public void showA() { System.out.println("A"); }
    public void showB() { System.out.println("B"); }
}
```
5 Hybrid Inheritance – Combination of two or more inheritance types (Java uses interfaces to achieve this).

4.2. Polymorphism and Its Types

Polymorphism allows methods to perform different tasks based on the object's class.

Types of Polymorphism:

1. **Compile-Time Polymorphism (Method Overloading)**
    - Multiple methods with the same name but different parameters.

```
class Animal {
    void sound() { System.out.println("Animal sound"); }
}
class Dog extends Animal {
    @Override
    void sound() { System.out.println("Bark"); }
```

1. }

---

1 Method Overloading:

```java
package Demo;
public class Calculator {
        // Method to add two integers
    int add(int a, int b) {
        return a + b;
    }

    // Overloaded method to add three integers
    int add(int a, int b, int c) {
        return a + b + c;
    }

    // Overloaded method to add two doubles
    double add(double a, double b) {
        return a + b;
    }
}
```

```java
package Demo;
public class Main {
      public static void main(String[] args) {

        Calculator c = new Calculator();

        System.out.println(c.add(5, 10));         // Output: 15
        System.out.println(c.add(5, 10, 15));     // Output: 30
        System.out.println(c.add(3.5, 2.7));      // Output: 6.2
       }
}
```

```
class Calculator {
    int add(int a, int b) { return a + b; }
    double add(double a, double b) { return a + b; }
}
```

2 Run-Time Polymorphism (Method Overriding)

- A subclass provides a specific implementation of a method already defined in its superclass.

```
java
class Animal {
    void sound() { System.out.println("Animal sound"); }
```

```
}
class Dog extends Animal {
    @Override
    void sound() { System.out.println("Bark"); }
}
```

2 Method Overriding:

```java
package Demo;
public class Main {
	public static void main(String[] args) {

		// Same type (Engine) but different behaviors

			Engine engine1 = new Car(); // Car engine  - Up casting
			Engine engine2 = new Bike(); // Bike engine - Up casting
			engine1.start(); // Output: "Vroom! Car engine starts"
			engine2.start(); // Output: "Put-put! Bike engine starts"
	}
}
```

```java
package Demo;
public class Bike extends Engine{

	public void start() {
		System.out.println("Put-put! Bike engine starts");
	}
}
```

```java
package Demo;
public class Car extends Engine{

	public void start() {
		System.out.println("Vroom! Car engine starts");
	}
}
```

```java
package Demo;
public class Engine {
	public void start() {
		System.out.println("Engine start called...");
	}
```

```
}
```

4.3. Abstraction

Abstraction hides complex implementation details and shows only essential features.

Ways to Achieve Abstraction:

1. **Abstract Classes**

```
abstract class Animal {
    abstract void sound();
    void sleep() { System.out.println("Sleeping..."); }
}
class Dog extends Animal {
    @Override
    void sound() { System.out.println("Bark"); }
}
```

4.4. Encapsulation

Encapsulation binds data (variables) and methods into a single unit and restricts direct access.

Example:

java

```
class Student {
    private String name;  // Private field

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

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

public class Main {
```

```
    public static void main(String[] args) {
        Student s = new Student();
        s.setName("John");
        System.out.println(s.getName());  // Output: John
    }
}
```

4.5. Interfaces in Java

An interface is a blueprint of a class that defines abstract methods (by default) and constants (`public static final`). It supports multiple inheritance and is used to achieve abstraction and loose coupling.

Key Features:

✔ All methods are public abstract by default (before Java 8).
✔ All variables are public static final (constants).
✔ Java 8+ supports default and static methods.
✔ Java 9+ supports private methods.

Example: Basic Interface

java

```
interface Vehicle {

    void start();  // Abstract method (public abstract by default)

    void stop();   // Abstract method

}



class Car implements Vehicle {

    public void start() {

        System.out.println("Car started.");

    }

    public void stop() {

        System.out.println("Car stopped.");

    }
```

```
}


public class Main {

    public static void main(String[] args) {

        Vehicle car = new Car();

        car.start();  // Output: Car started.

        car.stop();   // Output: Car stopped.

    }

}
```

Example: Multiple Inheritance Using Interfaces

```
java

interface A {

    void showA();

}


interface B {

    void showB();

}


class C implements A, B {

    public void showA() {

        System.out.println("A");

    }

    public void showB() {
```

```
        System.out.println("B");
    }
}

public class Main {
    public static void main(String[] args) {
        C obj = new C();
        obj.showA();  // Output: A
        obj.showB();   // Output: B
    }
}
```

Example: Default and Static Methods (Java 8+)

```
java

interface Calculator {
    default void display() {  // Default method (can be overridden)
        System.out.println("Default method in interface.");
    }
    static void print() {     // Static method (cannot be overridden)
        System.out.println("Static method in interface.");
    }
}

class MyCalculator implements Calculator {
    // Optional: Override default method
```

```
    @Override
    public void display() {
        System.out.println("Overridden default method.");
    }
}

public class Main {
    public static void main(String[] args) {
        MyCalculator calc = new MyCalculator();
        calc.display();  // Output: Overridden default method.
        Calculator.print();  // Output: Static method in interface.
    }
}
```

4.6. Upcasting and Downcasting:

1. **Upcasting:**
    - `ChildClass obj = new ParentClass()`
    - **Automatic (implicit)**
    - **Lose access to child-specific methods**
2. **Downcasting:**
    - `ParentClass obj = new ChildClass()`
    - **Needs explicit casting** `(ChildClass)`
    - **Must check with** `instanceof` **first**
    - **Gives access to child-specific methods**

```java
package Demo;
public class Main {
	public static void main(String[] args) {
		Engine engine = new Bike(); // Upcasting
//		engine.start();

//		engine.kickStart();
```

```
            // Downcasting - must check first
            if (engine instanceof Bike) {
                  Bike bike = (Bike) engine;  //Down casting
                  bike.kickStart(); // Now we can use BikeEngine methods
            }
      }
}
```

```
package Demo;
public class Bike extends Engine {
      public void start() {
            System.out.println("Put-put! Bike engine starts");
      }
      public void kickStart() {
            System.out.println("Kick starting bike!");
      }
}
```

```
package Demo;
public class Engine {
      public void start() {
            System.out.println("Engine start called...");
      }
}
```

Difference between Abstract class and Interface?

```
package Demo;
public class Main {

      public static void main(String[] args) {

            Car car = new Car();
            car.engineType(); // non-abstract method
            car.fuelType(); // non-abstract method
            // You cannot create object of Bike because it's abstract
//             Bike bike = new Bike();
      }
}
```

```java
package Demo;
public abstract class Bike implements Engine{

    // Abstract method: must be implemented by subclass
  public abstract void fuelType(); // abstract method
   // Non-abstract method: has a body
  public void start() {
        System.out.println("Bike is starting...");
    }
    // Implementing abstract method from interface
    public void engineType() {
        System.out.println("Bike has a 2-stroke engine.");
    }


}
```

```java
package Demo;
public class Car implements Engine {
        // Non-abstract method: has a body
    public void fuelType() {
        System.out.println("Car uses diesel.");
    }
    // Implementing interface method
    public void engineType() {
        System.out.println("Car has a 4-stroke engine.");
    }
}
```

```java
package Demo;
public interface Engine {

      int maxSpeed = 120;
//    public static final int maxSpeed = 120;

       void engineType();
//     public abstract void engineType();

//     String getEngineName(int engineId);
//     public abstract  String getEngineName(int engineId);
}
```

Difference between String and StringBuilder?

```
String s = "Hello";
        s = s + " World";
        s = s + "!";
        System.out.println(s);
```

🔍 What happens in memory:

Total Objects = 6

- `"Hello"` is stored in String Pool. // 1 object
- `" World"` and `"!"` are also stored in String Pool. // 2 object
- Each `+` operation creates a new `String` object. // 3 object
  So total:
  - `"Hello"`
  - `"Hello World"`
  - `"Hello World!"`

    → 3 different String objects created in heap memory.

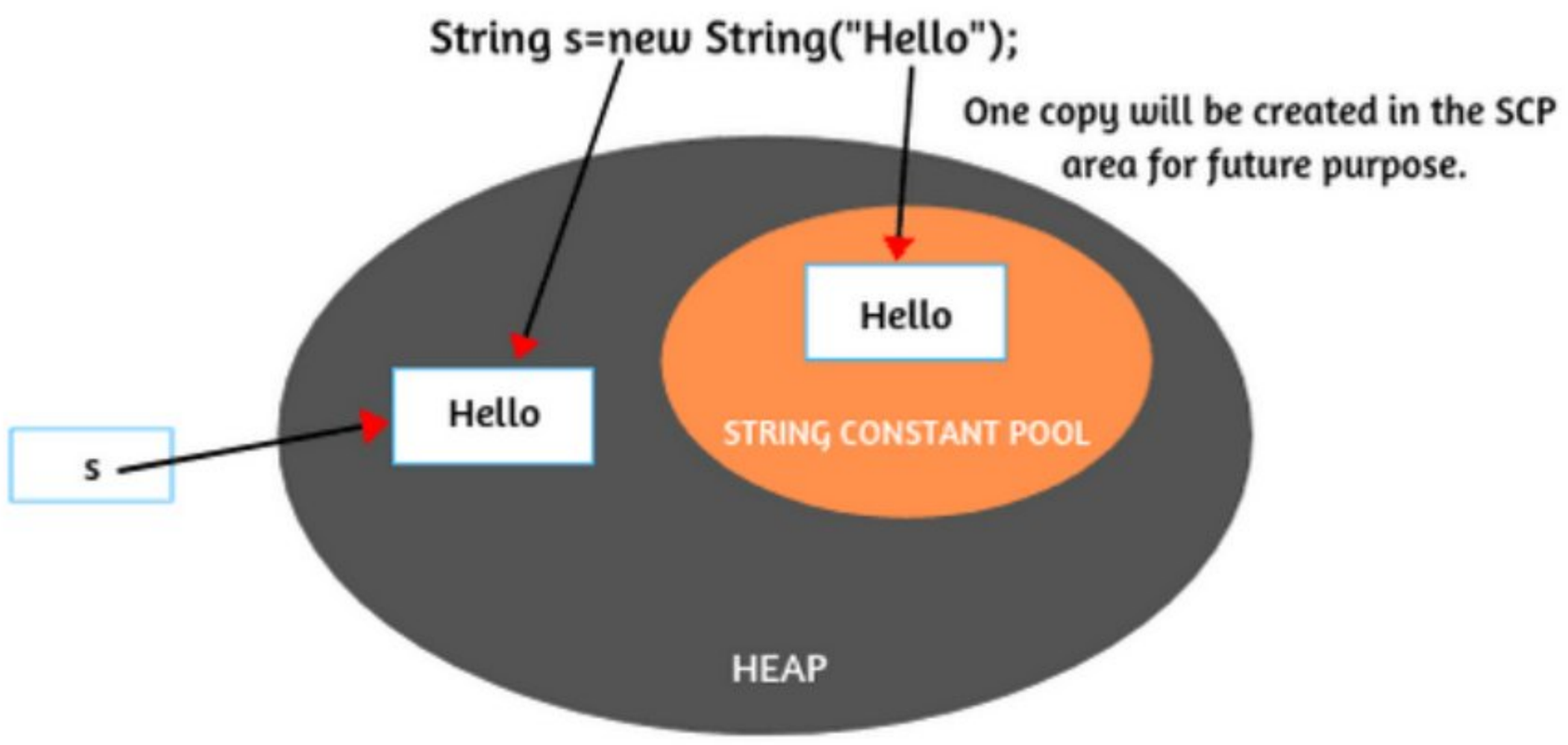
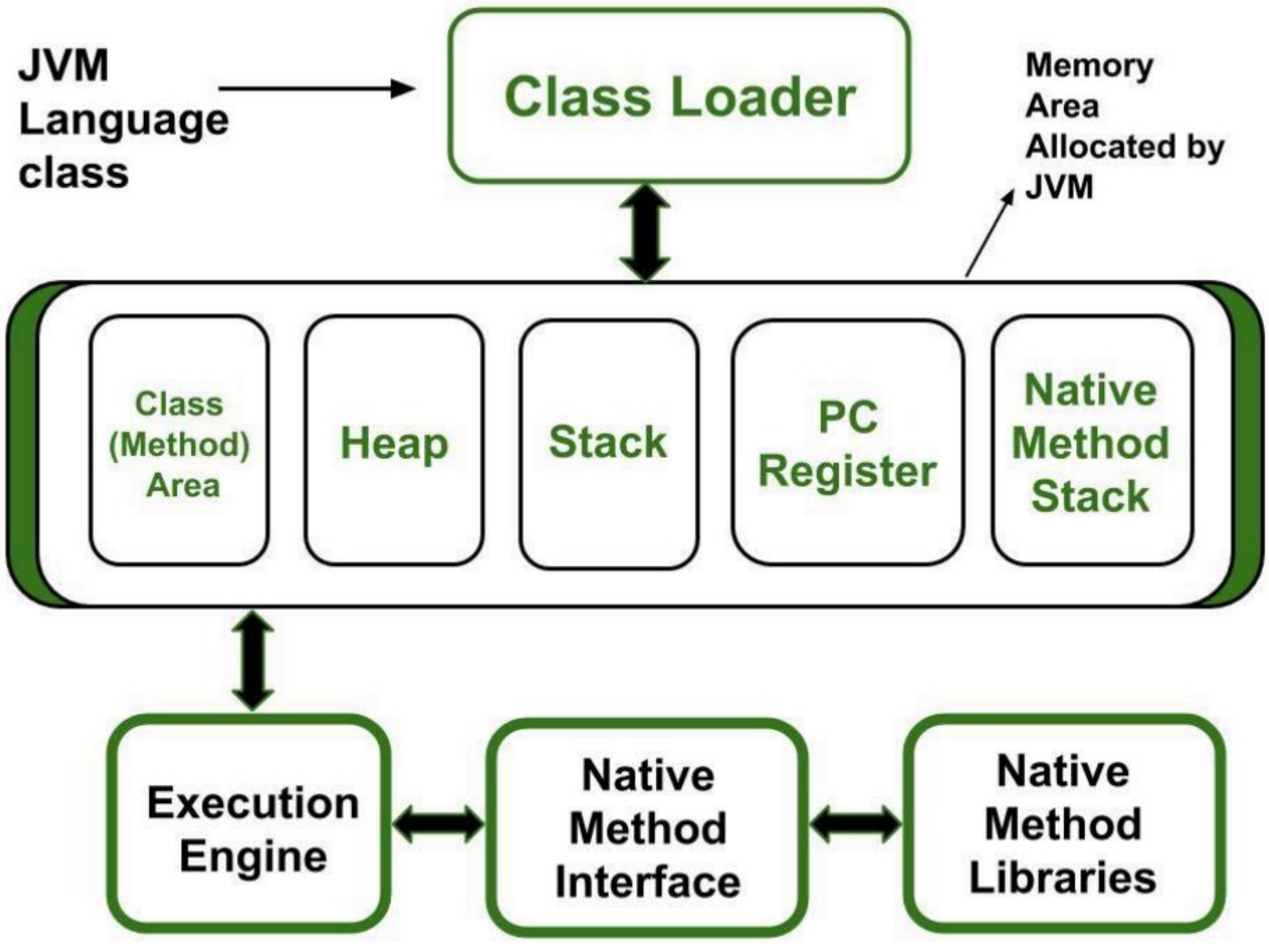


Fig: Allotting memory for storing object.




```
StringBuilder sb = new StringBuilder("Hello");
        sb.append(" World");
        sb.append("!");
        System.out.println(sb.toString());
```

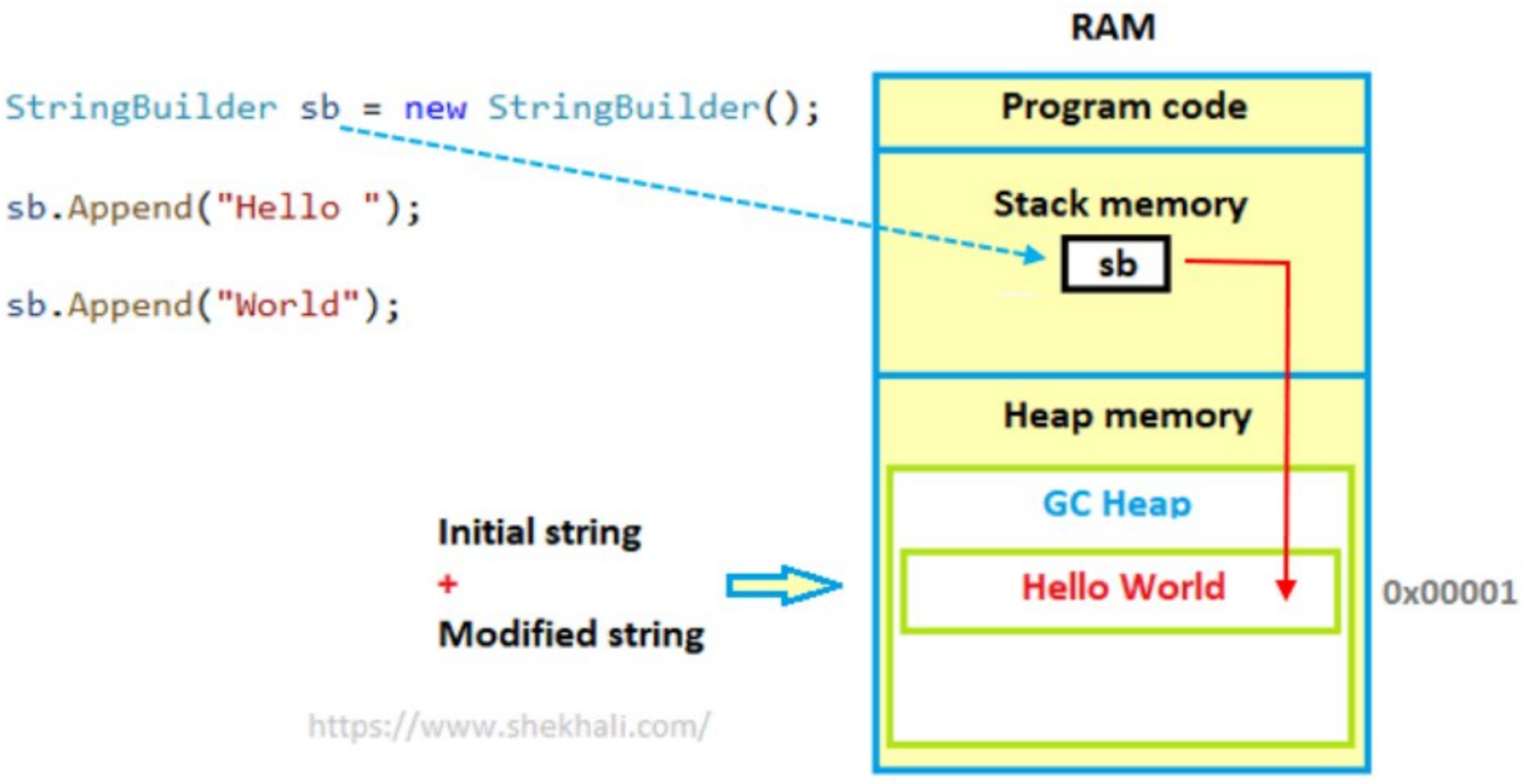
🔍 **What happens in memory:**

Total objects = `4`

- `"Hello"`, `" World"`, and `"!"` are String literals in the pool. // 3 objects
- Only one `StringBuilder` object is created. // 1 object
- Modifications happen in-place (same object), using an internal char[] array.

✅ **Advantage:**

- More memory-efficient and faster for repeated changes.



5 Java Access Modifiers

5.1. Access Modifiers in Java

Access modifiers control the visibility of classes, methods, and variables.

Modifier	Class	Package	Subclass	World
`public`	✅	✅	✅	✅
`protected`	✅	✅	✅	❌
`default` (no modifier)	✅	✅	❌	❌
`private`	✅	❌	❌	❌

Examples:

1. Public Access Modifier (Accessible Everywhere)

```
java
public class Student {
    public String name = "John";  // Public variable
```

```
    public void display() {  // Public method
        System.out.println("Name: " + name);
    }
}

public class Main {
    public static void main(String[] args) {
        Student s = new Student();
        System.out.println(s.name);  // Accessible
        s.display();  // Accessible
    }
}
```

2. Protected Access Modifier (Within Package + Subclasses)

java

```
package com.example;

public class Person {
    protected int age = 25;
}

class Employee extends Person {
    void showAge() {
        System.out.println("Age: " + age);  // Accessible (subclass)
    }
}
```

3. Default (Package-Private) Access Modifier (Within Same Package Only)

java

```
class Student {
    String name = "Alice";  // Default access

    void display() {  // Default access
        System.out.println("Name: " + name);
    }
}

public class Main {
    public static void main(String[] args) {
        Student s = new Student();
        System.out.println(s.name);  // Accessible (same package)
        s.display();  // Accessible (same package)
    }
}
```

4. Private Access Modifier (Within Class Only)

java

```
class BankAccount {
    private double balance = 1000.0;

    private void showBalance() {
        System.out.println("Balance: " + balance);
    }

    public void display() {
        showBalance();  // Accessible (same class)
    }
}

public class Main {
    public static void main(String[] args) {
        BankAccount acc = new BankAccount();
        // System.out.println(acc.balance);  ❌ Error (private)
        // acc.showBalance();  ❌ Error (private)
        acc.display();  // ✅ Accessible via public method
    }
}
```

---

6. String Handling in Java: String, StringBuilder, and StringBuffer

6.1. String Class

The `String` class represents immutable sequences of characters. Once created, String objects cannot be modified.

Key Characteristics:

✔ Immutable (thread-safe by nature)
✔ Stored in String Pool for memory efficiency
✔ Rich set of built-in methods for string manipulation

Example: String Creation and Methods

```
java
public class StringExample {
    public static void main(String[] args) {
        // String creation
        String str1 = "Hello";                    // String literal (uses pool)
        String str2 = new String("World");        // String object (heap)
        String str3 = str1.concat(" ").concat(str2);    // Concatenation
```

```
        System.out.println(str3);                  // Output: Hello World
        System.out.println(str3.length());         // Output: 11
        System.out.println(str3.substring(6));     // Output: World
        System.out.println(str3.toUpperCase());    // Output: HELLO WORLD
    }
}
```

Important String Methods

Method	Description	Example
`length()`	Returns string length	`"Java".length()` → 4
`charAt()`	Gets character at index	`"Java".charAt(2)` → 'v'
`substring()`	Extracts substring	`"Hello".substring(1,4)` → "ell"
`equals()`	Compares content	`"Java".equals("java")` → false
`compareTo()`	Lexicographical comparison	`"A".compareTo("B")` → -1
`replace()`	Replaces characters	`"Java".replace('a', 'o')` → "Jovo"

6.2. StringBuilder Class

A mutable sequence of characters designed for single-threaded environments where frequent modifications are needed.

Key Characteristics:

✔ Mutable (can be modified after creation)
✔ Not thread-safe (faster than StringBuffer)
✔ Better performance for repeated modifications

Example: StringBuilder Usage

java

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

```
StringBuilder sb = new StringBuilder("Java");

// Append operations
sb.append(" is");
sb.append(" awesome!");
System.out.println(sb);  // Output: Java is awesome!

// Insert and delete
sb.insert(5, "really ");
System.out.println(sb);  // Output: Java really is awesome!

        sb.delete(5, 12);
        System.out.println(sb);  // Output: Java is awesome!

        // Reverse
        sb.reverse();
        System.out.println(sb);  // Output: !emosewa si avaJ
    }
}
```

Important StringBuilder Methods

Method	Description	Example
`append()`	Adds to the end	`sb.append("!")`
`insert()`	Inserts at position	`sb.insert(0, "Hi ")`
`delete()`	Removes characters	`sb.delete(0, 3)`
`reverse()`	Reverses content	`sb.reverse()`
`capacity()`	Returns current capacity	`sb.capacity()`
`setLength()`	Sets character length	`sb.setLength(10)`

6.3. StringBuffer Class

A thread-safe, mutable sequence of characters with synchronized methods.

Key Characteristics:

✔ Mutable like StringBuilder
✔ Thread-safe (synchronized methods)
✔ Slightly slower than StringBuilder due to synchronization

Example: StringBuffer Usage

```
java
public class StringBufferExample {
    public static void main(String[] args) {
        StringBuffer sbf = new StringBuffer("Thread");

        // Append operations
        sbf.append("-Safe");
        System.out.println(sbf);  // Output: Thread-Safe

        // Thread-safe modification
        Runnable task = () -> {
            sbf.append("!");
        };

        new Thread(task).start();
        new Thread(task).start();

        // Wait for threads to complete
        try { Thread.sleep(100); } catch (InterruptedException e) {}

        System.out.println(sbf);  // Output: Thread-Safe!! (or similar)
    }
}
```

Important StringBuffer Methods

Method	Description	Example
`append()`	Adds to the end	`sbf.append("!")`
`insert()`	Inserts at position	`sbf.insert(0, "Hi ")`
`delete()`	Removes characters	`sbf.delete(0, 3)`
`reverse()`	Reverses content	`sbf.reverse()`
`ensureCapacity()`	Ensures minimum capacity	`sbf.ensureCapacity(100)`
`setCharAt()`	Sets character at index	`sbf.setCharAt(0, 'J')`

Comparison Table: String vs StringBuilder vs StringBuffer

Feature	String	StringBuilder	StringBuffer
Mutability	Immutable	Mutable	Mutable
Thread Safety	Yes (immutable)	No	Yes
Performance	Slow for modifications	Fast	Moderate
Storage	String Pool	Heap	Heap
Synchronization	N/A	Not synchronized	Synchronized
Use Case	When immutability needed	Single-threaded modifications	Multi-threaded modifications

---

When to Use Each?

1. Use String when:
   - You need immutability
   - String values won't change frequently
   - Thread safety is required (via immutability)
2. Use StringBuilder when:
   - You need to make many modifications
   - Working in a single-threaded environment
   - Performance is critical
3. Use StringBuffer when:
   - You need to make many modifications
   - Working in a multi-threaded environment
   - Thread safety is required

Performance Benchmark Example

java

```
public class PerformanceTest {
    public static void main(String[] args) {
        final int ITERATIONS = 100000;
```

```
// String concatenation
long start = System.currentTimeMillis();
String s = "";
for (int i = 0; i < ITERATIONS; i++) {
    s += "a";
}
System.out.println("String time: " + (System.currentTimeMillis() - start) + "ms");

// StringBuilder
start = System.currentTimeMillis();
StringBuilder sb = new StringBuilder();
for (int i = 0; i < ITERATIONS; i++) {
    sb.append("a");
}
System.out.println("StringBuilder time: " + (System.currentTimeMillis() - start) +
"ms");

// StringBuffer
start = System.currentTimeMillis();
StringBuffer sbf = new StringBuffer();
for (int i = 0; i < ITERATIONS; i++) {
    sbf.append("a");
}
System.out.println("StringBuffer time: " + (System.currentTimeMillis() - start) +
"ms");
    }
}
```

Typical Output:

```
text
String time: 4235ms
StringBuilder time: 5ms
StringBuffer time: 8ms
```

7. Java Arrays

An array is a container object that holds a fixed number of values of a single type. Arrays are indexed, starting from `0`.

7.1 1D Array

java

```
public class Main {
    public static void main(String[] args) {
        // Declaration & Initialization
        int[] numbers = {10, 20, 30, 40, 50};

        // Accessing elements
        System.out.println("First Element: " + numbers[0]); // Output: 10
        System.out.println("Third Element: " + numbers[2]); // Output: 30

        // Modifying an element
        numbers[1] = 25;
        System.out.println("Modified Second Element: " + numbers[1]); // Output: 25

        // Length of the array
        System.out.println("Array Length: " + numbers.length); // Output: 5

        // Iterating using for loop
        for (int i = 0; i < numbers.length; i++) {
            System.out.println("Element at index " + i + ": " + numbers[i]);
        }

        // Iterating using for-each loop
        for (int num : numbers) {
            System.out.println("Number: " + num);
        }
    }
}
```

7.2. 2D Arrays (Matrix)

A 2D array is an array of arrays, often used to represent matrices or tables.

Example: 2D Array

```
java
public class Main {
    public static void main(String[] args) {
        // Declaration & Initialization
        int[][] matrix = {
            {1, 2, 3},
            {4, 5, 6},
            {7, 8, 9}
        };

        // Accessing elements
        System.out.println("Element at [0][1]: " + matrix[0][1]); // Output: 2
        System.out.println("Element at [2][2]: " + matrix[2][2]); // Output: 9
```

```
        // Modifying an element
        matrix[1][1] = 50;
        System.out.println("Modified Element at [1][1]: " + matrix[1][1]); // Output: 50

        // Length of rows and columns
        System.out.println("Number of Rows: " + matrix.length); // Output: 3
        System.out.println("Number of Columns in Row 0: " + matrix[0].length); // Output:
3

        // Iterating using nested for loop
        for (int i = 0; i < matrix.length; i++) {
            for (int j = 0; j < matrix[i].length; j++) {
                System.out.print(matrix[i][j] + " ");
            }
            System.out.println();
        }

        // Iterating using for-each loop
        for (int[] row : matrix) {
            for (int num : row) {
                System.out.print(num + " ");
            }
            System.out.println();
        }
    }
}
```

7.3. 3D Arrays

A 3D array is an array of 2D arrays, useful for representing 3D structures like cubes.

Example: 3D Array

```
java
public class Main {
    public static void main(String[] args) {
        // Declaration & Initialization
        int[][][] cube = {
            {
                {1, 2, 3},
                {4, 5, 6}
            },
            {
                {7, 8, 9},
                {10, 11, 12}
            }
```

```
};

// Accessing elements
System.out.println("Element at [0][1][2]: " + cube[0][1][2]); // Output: 6
System.out.println("Element at [1][0][1]: " + cube[1][0][1]); // Output: 8

// Modifying an element
cube[1][1][0] = 100;
System.out.println("Modified Element at [1][1][0]: " + cube[1][1][0]); // Output: 100

// Length of dimensions
System.out.println("Depth (Z-axis): " + cube.length); // Output: 2
System.out.println("Rows (Y-axis): " + cube[0].length); // Output: 2
System.out.println("Columns (X-axis): " + cube[0][0].length); // Output: 3

// Iterating using nested loops
for (int i = 0; i < cube.length; i++) {
    System.out.println("Layer " + (i + 1) + ":");
    for (int j = 0; j < cube[i].length; j++) {
        for (int k = 0; k < cube[i][j].length; k++) {
            System.out.print(cube[i][j][k] + " ");
        }
        System.out.println();
    }
    System.out.println();
  }
}
```

Key Takeaways

Feature	1D Array	2D Array	3D Array
Definition	Single row of elements	Table-like structure	Cube-like structure
Declaration	`int[] arr;`	`int[][] matrix;`	`int[][][] cube;`
Initialization	`int[] arr = {1, 2, 3};`	`int[][] matrix = {{1,2}, {3,4}};`	`int[][][] cube = {{{1,2}, {3,4}}};`
Accessing Elements	`arr[0]`	`matrix[0][1]`	`cube[0][1][2]`

Iteration	Single loop	Nested loop (2 levels)	Triple loop (3 levels)

8. Java Exceptions

8.1. Introduction to Exceptions

An exception is an event that disrupts the normal flow of a program. Java provides a robust mechanism to handle runtime errors using Exception Handling.

Example: Basic Exception

java

```
public class Main {
    public static void main(String[] args) {
        int a = 10;
        int b = 0;

        // This will throw ArithmeticException
        int result = a / b;
        System.out.println("Result: " + result);
    }
}
```

Output:

text

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

8.2. Types of Exceptions

Java exceptions are categorized into three types:

1. Checked Exceptions (Compile-Time Exceptions)

- **Must be handled at compile time.**
- **Examples: `IOException`, `SQLException`, `ClassNotFoundException`.**

Example: Checked Exception (File Handling)

java

```
import java.io.File;
```

```
import java.io.FileReader;

public class Main {
    public static void main(String[] args) {
        File file = new File("nonexistent.txt");

        // This will throw FileNotFoundException (Checked Exception)
        FileReader fr = new FileReader(file);
    }
}
```

Output (Compile-Time Error):

```
text
Unhandled exception: java.io.FileNotFoundException
```

---

2. Unchecked Exceptions (Runtime Exceptions)

- **Occur during program execution.**
- **Examples: `ArithmeticException`, `NullPointerException`, `ArrayIndexOutOfBoundsException`.**

Example: Unchecked Exception (NullPointerException)

java

```
public class Main {
    public static void main(String[] args) {
        String str = null;

        // This will throw NullPointerException
        System.out.println(str.length());
    }
}
```

Output:

```
text
Exception in thread "main" java.lang.NullPointerException
```

---

3. Errors

- **Serious issues that are not meant to be caught (e.g., `OutOfMemoryError`, `StackOverflowError`).**
- **Usually caused by JVM failures.**

Example: Error (StackOverflowError)

java

```
public class Main {
    static void recursiveMethod() {
        recursiveMethod(); // Infinite recursion
    }

    public static void main(String[] args) {
        recursiveMethod();
    }
}
```

Output:

text
Exception in thread "main" java.lang.StackOverflowError

8.3. Ways to Handle Exceptions

There are five ways to handle exceptions in Java:

1. Using `try-catch` Block

Catches and handles exceptions gracefully.

Example: Handling ArithmeticException

java

```
public class Main {
    public static void main(String[] args) {
        try {
            int a = 10;
            int b = 0;
            int result = a / b; // Throws ArithmeticException
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Error: Division by zero!");
        }
    }
}
```

Output:

text
Error: Division by zero!

---

2. Using `try-catch-finally`

- `finally` block executes always, whether an exception occurs or not.
- Used for cleanup (e.g., closing files, database connections).

Example: try-catch-finally

java

```
public class Main {
    public static void main(String[] args) {
        try {
            int[] arr = {1, 2, 3};
            System.out.println(arr[5]); // Throws ArrayIndexOutOfBoundsException
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Error: Array index out of bounds!");
        } finally {
            System.out.println("This will always execute.");
        }
    }
}
```

Output:

text

```
Error: Array index out of bounds!
This will always execute.
```

---

3. Using `throws` Keyword

- Declares that a method may throw an exception.
- The caller must handle it.

Example: throws Keyword

java

```
import java.io.*;

public class Main {
    public static void readFile() throws FileNotFoundException {
        File file = new File("nonexistent.txt");
        FileReader fr = new FileReader(file); // May throw FileNotFoundException
    }

    public static void main(String[] args) {
        try {
            readFile();
        } catch (FileNotFoundException e) {
            System.out.println("Error: File not found!");
        }
    }
}
```

Output:

text
Error: File not found!

---

4. Using `throw` Keyword

- **Explicitly throws an exception.**

Example: Custom Exception with `throw`

java

```
public class Main {
    static void checkAge(int age) {
        if (age < 18) {
            throw new ArithmeticException("Access denied - You must be at least 18!");
        } else {
            System.out.println("Access granted!");
        }
    }

    public static void main(String[] args) {
        checkAge(15); // Throws ArithmeticException
    }
}
```

Output:

text
Exception in thread "main" java.lang.ArithmeticException: Access denied - You must be at least 18!

---

5. Using `try-with-resources` (AutoCloseable)

- **Automatically closes resources (e.g., `FileReader`, `BufferedReader`).**

Example: try-with-resources

java

```
import java.io.*;

public class Main {
    public static void main(String[] args) {
        try (FileReader fr = new FileReader("example.txt")) {
            int data;
            while ((data = fr.read()) != -1) {
                System.out.print((char) data);
            }
```

```
        } catch (IOException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

Output (if file not found):

```
text
Error: example.txt (No such file or directory)
```

8.4. Best Practices for Exception Handling

✅ **Use Specific Exceptions** (e.g., `catch (FileNotFoundException e)` instead of `catch (Exception e)`).
✅ **Log Exceptions** (Use `e.printStackTrace()` or logging frameworks like Log4j).
✅ **Avoid Empty Catch Blocks** (Never ignore exceptions silently).
✅ **Use `finally`** for Cleanup (Close resources like files, DB connections).
✅ **Throw Meaningful Exceptions** (Use custom exceptions for better debugging).

---

5. Summary Table

Exception Type	Handling Mechanism	Example
Checked	Must be handled at compile time (`try-catch` or `throws`)	`FileNotFoundException`
Unchecked	Handled at runtime (`try-catch`)	`NullPointerException`
Errors	Not meant to be caught	`OutOfMemoryError`
Custom Exceptions	Extend `Exception` or `RuntimeException`	`throw new InvalidUserException()`

---

9. Java Threads and Multithreading

9.1. Introduction to Threads

A **thread** is the smallest unit of execution within a process. Java provides built-in support for multithreaded programming.

Example: Creating a Thread (Extending `Thread` Class)

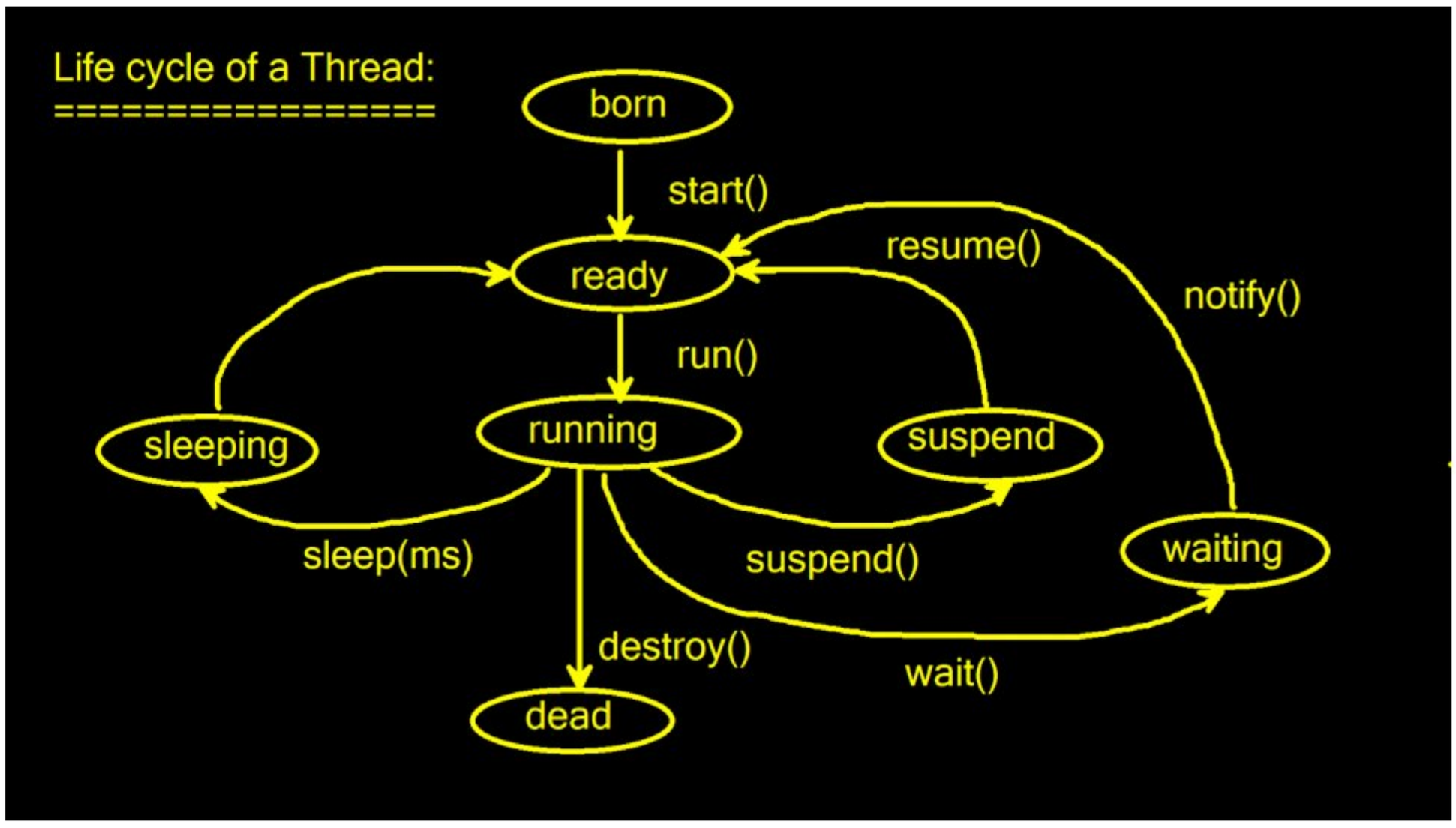
java

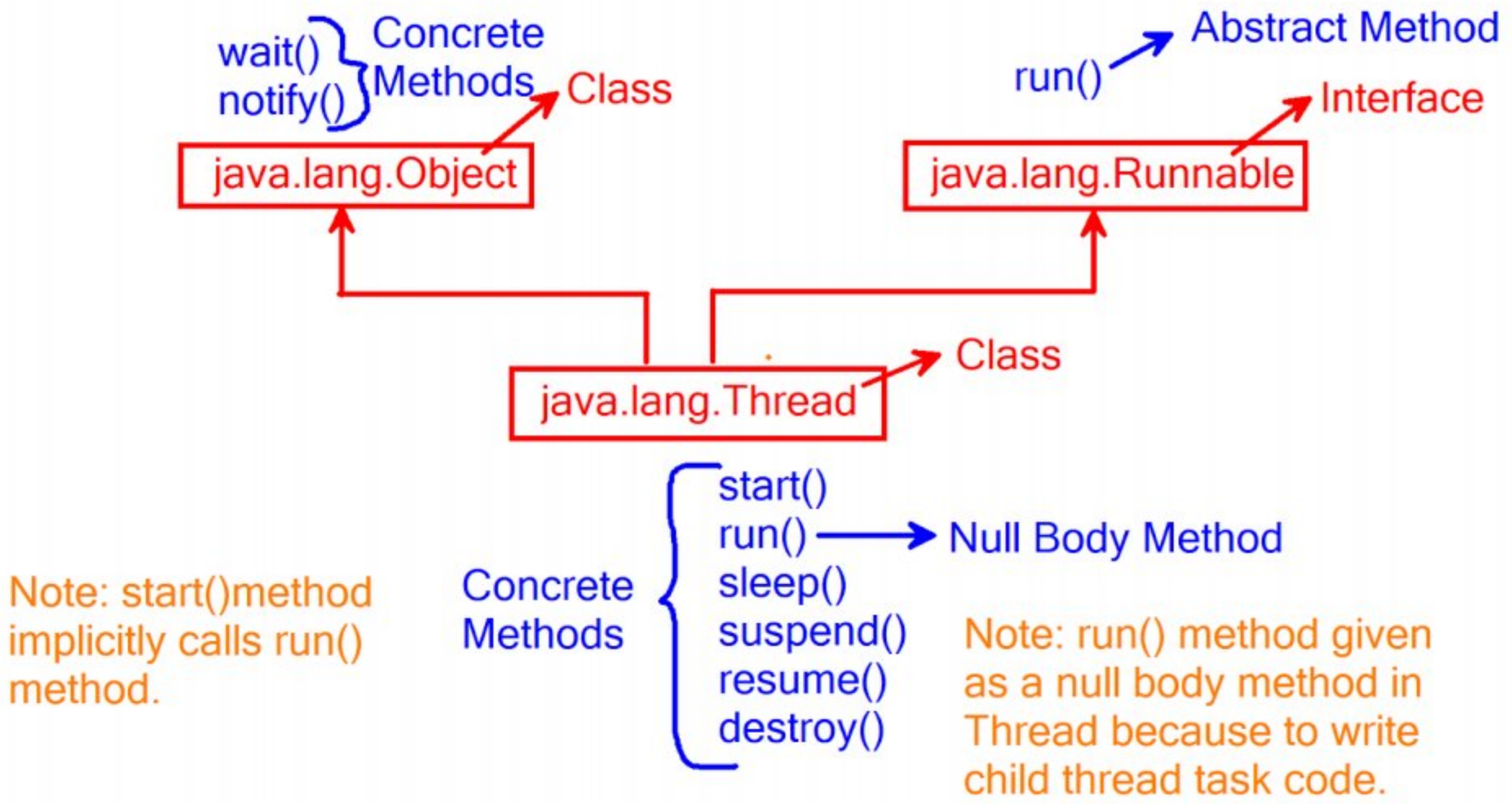
```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread is running...");
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start(); // Starts the thread
    }
}
```

Output:

text

```
Thread is running...
```





9.2. Multithreading in Java

Multithreading allows concurrent execution of two or more threads for maximum CPU utilization.

Example: Implementing `Runnable` Interface

java
```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Thread (Runnable) is running...");
    }
}

public class Main {
    public static void main(String[] args) {
        Thread t1 = new Thread(new MyRunnable());
        t1.start();
    }
}
```

Output:

text
```
Thread (Runnable) is running...
```

9.3. Thread Lifecycle

A thread can be in one of these states:

State	Description
NEW	Thread created but not started
RUNNABLE	Thread is executing
BLOCKED	Waiting for a monitor lock
WAITING	Waiting indefinitely
TIMED_WAITING	Waiting for a specified time
TERMINATED	Thread has finished execution

Example: Thread States

java

```
public class Main {
    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(() -> {
            try {
                Thread.sleep(1000); // TIMED_WAITING
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });

        System.out.println("Before start(): " + t1.getState()); // NEW
        t1.start();
        System.out.println("After start(): " + t1.getState()); // RUNNABLE
        Thread.sleep(500);
        System.out.println("During sleep(): " + t1.getState()); // TIMED_WAITING
        t1.join();
        System.out.println("After completion: " + t1.getState()); // TERMINATED
    }
}
```

Output:

text

```
Before start(): NEW
After start(): RUNNABLE
During sleep(): TIMED_WAITING
After completion: TERMINATED
```

9.4. Process vs. Thread

Feature	Process	Thread
Definition	Independent program in execution	Lightweight subunit of a process
Memory	Separate memory space	Shares memory with other threads
Creation	Heavyweight (Slower)	Lightweight (Faster)
Communication	IPC (Pipes, Sockets)	Shared memory
Context Switching	Expensive	Less expensive
Examples	Running multiple apps (Chrome, Word)	Multiple tabs in Chrome

Example: Process vs. Thread

java

```
public class Main {
    public static void main(String[] args) {
        // Process: Runs a separate Java program
        Runtime runtime = Runtime.getRuntime();
        try {
            Process process = runtime.exec("notepad.exe"); // Opens Notepad (separate
process)
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Thread: Runs within the same process
        new Thread(() -> System.out.println("Thread inside Java process")).start();
    }
}
```

9.5. Thread Synchronization

When multiple threads access shared resources, synchronization prevents race conditions.

Example: Synchronized Method

java

```
class Counter {
    private int count = 0;
```

```
    public synchronized void increment() {
        count++;
    }

    public int getCount() {
        return count;
    }
}

public class Main {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();

        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });

        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });

        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println("Final Count: " + counter.getCount()); // Expected: 2000
    }
}
```

Output:

text
Final Count: 2000

9.6. Thread Pools (Executor Framework)

Efficiently manages multiple threads using a fixed pool.

Example: ExecutorService

java

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class Main {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(3);

        for (int i = 0; i < 5; i++) {
            executor.execute(() -> {
                System.out.println("Thread: " + Thread.currentThread().getName());
            });
        }

        executor.shutdown();
    }
}
```

Output (Sample):

```
text
Thread: pool-1-thread-1
Thread: pool-1-thread-2
Thread: pool-1-thread-3
Thread: pool-1-thread-1
Thread: pool-1-thread-2
```

9.7. Key Takeaways

Concept	Description
Thread	Lightweight, shares memory
Process	Independent, has separate memory
Multithreading	Concurrent thread execution
Synchronization	Prevents race conditions
Thread Pool	Efficient thread management

---

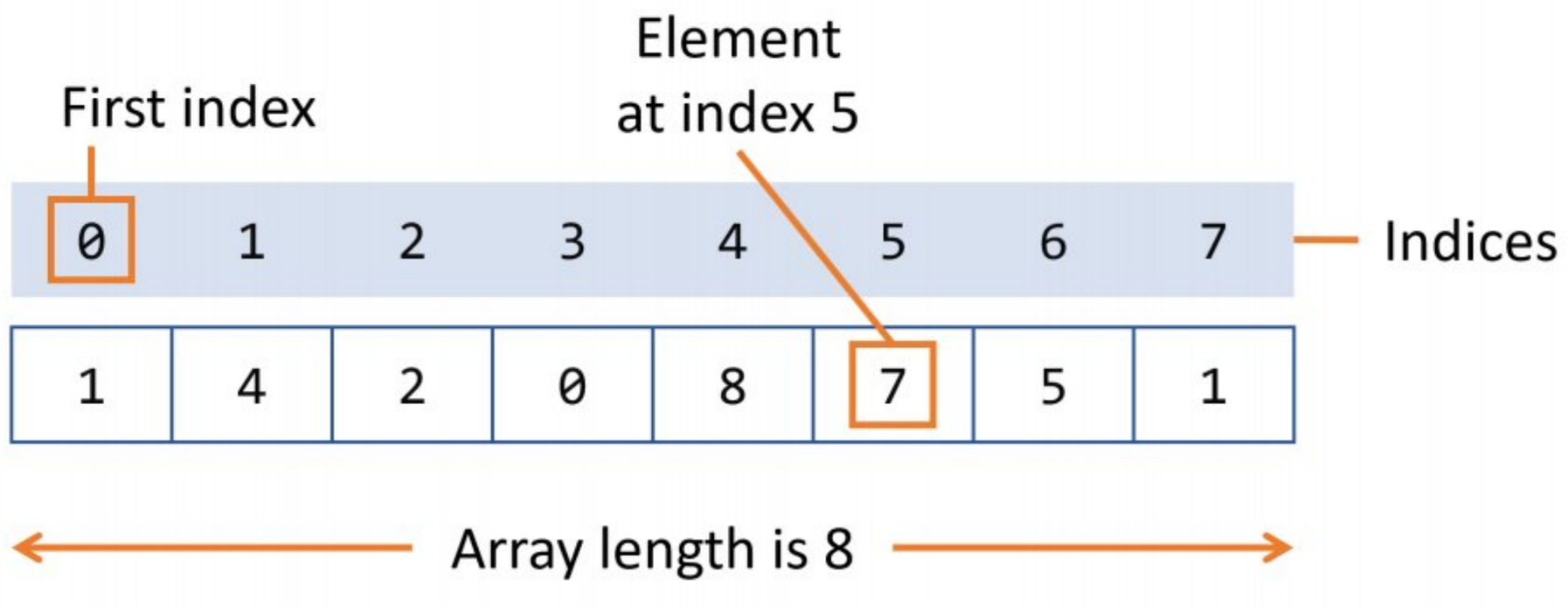
10. Collections Framework

The Java Collections Framework provides a set of interfaces and classes to store and manipulate groups of objects efficiently.

Difference between ArrayList and Array?

```
package Demo;
public class Main {

      public static void main(String[] args) {

            String[] fruits = new String[3];
            fruits[0] = "Apple";
            fruits[1] = "Banana";
            fruits[2] = "Mango";
            fruits[3] = "watermelon";

            // Print
            for (int i = 0; i < fruits.length; i++) {
                System.out.println(fruits[i]);
            }
      }
}
package Demo;
import java.util.ArrayList;
public class Main {

      public static void main(String[] args) {

            ArrayList<String> fruits = new ArrayList<>();
            fruits.add("Apple");
            fruits.add("Banana");
            fruits.add("Mango");
            fruits.add("watermelon");
            // Print
            for (String fruit : fruits) {
                System.out.println(fruit);
            }
      }
}
```

Index	Value
0	Apple
1	Banana
2	Mango
3	Watermelon



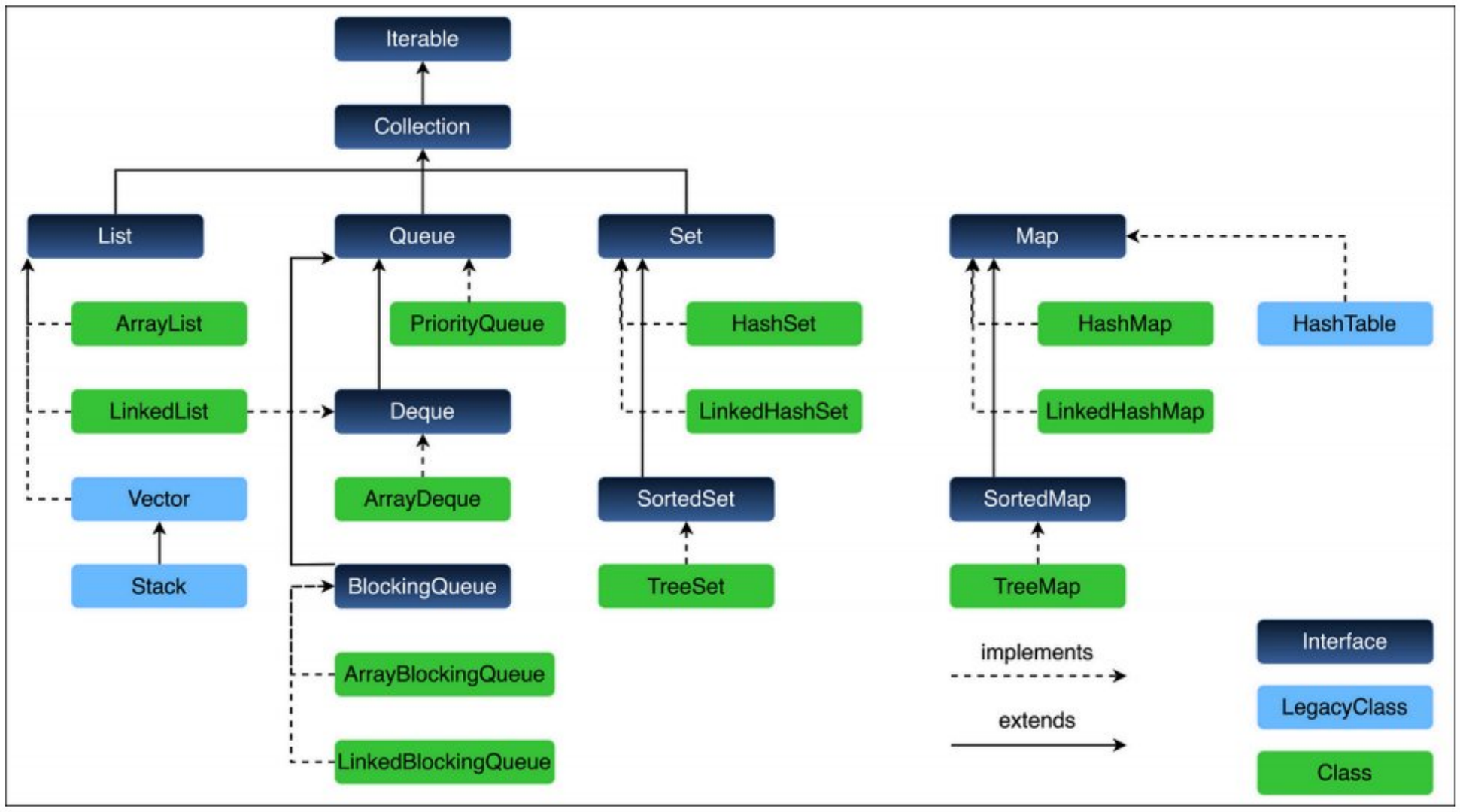
Key Interfaces:

- `List` (Ordered, allows duplicates)
- `Set` (No duplicates)
- `Queue` (FIFO order)
- `Map` (Key-value pairs)

Example:

java

```
import java.util.*;

public class CollectionsExample {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("Java");
        list.add("Python");
        System.out.println("List: " + list); // Output: [Java, Python]
    }
}
```



10.1. List Interface

A `List` is an ordered collection that allows duplicates.

Implementations:

- **`ArrayList` (Resizable array)**
- **`LinkedList` (Doubly-linked list)**
- **`Vector` (Thread-safe, legacy)**

Example:

java

```
import java.util.*;

public class ListExample {
    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>();
        numbers.add(10);
        numbers.add(20);
        System.out.println("List: " + numbers); // Output: [10, 20]
    }
}
```

Differences Between ArrayList and LinkedList in Java

Underlying Data Structure

- **ArrayList: Uses a dynamic array internally**
- **LinkedList: Uses a doubly linked list internally**

Memory Usage

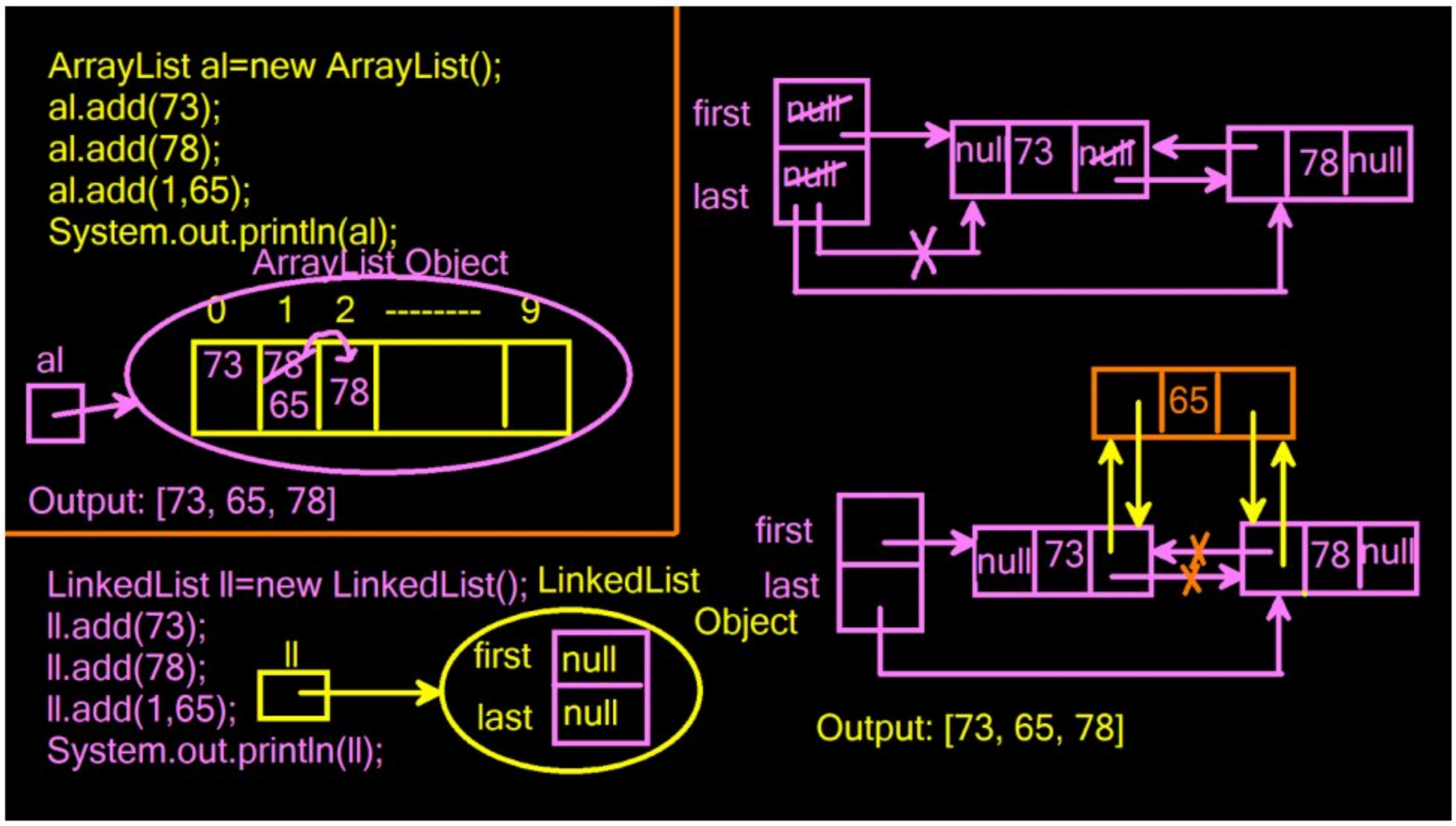
- **ArrayList: Generally uses less memory as it only stores elements**
- **LinkedList: Uses more memory due to storing both elements and node references (next/previous pointers)**

2. Key Differences

Feature	ArrayList	LinkedList
Data Structure	Dynamic array	Doubly linked list
Random Access (get)	O(1) ✅ (Fast)	O(n) ❌ (Slow)
Insertion at End	O(1) ✅ (Amortized)	O(1) ✅
Insertion at Start	O(n) ❌ (Requires shifting)	O(1) ✅
Deletion in Middle	O(n) ❌ (Shifts elements)	O(n) (Traversal) but O(1) node removal ✅
Memory Overhead	Lower ✅ (Stores only data)	Higher ❌ (Stores node pointers)
Cache Locality	Better ✅ (Contiguous memory)	Poor ❌ (Scattered memory)
Best Use Case	Frequent reads, less modifications	Frequent insertions/deletions

```java
package Demo;
import java.util.ArrayList;
public class Demo extends Object {
	public static void main(String[] args) {
		ArrayList<String> fruits = new ArrayList<>();
		// Adding elements (O(1) at end)
		fruits.add("Apple");
		fruits.add("Banana");
		fruits.add("Cherry");
		// Fast random access (O(1))
		System.out.println(fruits.get(1)); // "Banana"
		// Slow insertion in middle (O(n))
		fruits.add(1, "Mango"); // Shifts elements right
		// Fast iteration (better cache performance)
		for (String fruit : fruits) {
			System.out.println(fruit);
		}
	}
}
```

```java
package Demo;
import java.util.LinkedList;
public class Demo extends Object {
	public static void main(String[] args) {
		LinkedList<String> fruits = new LinkedList<>();
		// Adding elements (O(1) at end)
		fruits.add("Apple");
		fruits.add("Banana");
		fruits.add("Cherry");
		// Fast insertion at start (O(1))
		fruits.addFirst("Mango");
		// Fast removal at end (O(1))
		fruits.removeLast();
		// Slow random access (O(n))
		System.out.println(fruits.get(1)); // Traverses nodes
		// Slower iteration (due to pointer chasing)
		for (String fruit : fruits) {
			System.out.println(fruit);
		}
	}
}
```



10.2. Set Interface

A `Set` is a collection that does not allow duplicate elements.

Implementations:

- `HashSet` (Unordered, uses hashing)
- `LinkedHashSet` (Maintains insertion order)
- `TreeSet` (Sorted in natural order)

Example:

```
java
import java.util.*;

public class SetExample {
    public static void main(String[] args) {
        Set<String> fruits = new HashSet<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Apple"); // Duplicate ignored
        System.out.println("Set: " + fruits); // Output: [Apple, Banana]
    }
}
```

HashSet vs. TreeSet in Java

---

1. Introduction

In Java, `HashSet` and `TreeSet` are two implementations of the `Set` interface under `java.util`. Both store unique elements, but they differ in ordering, performance, and underlying implementation.

Feature	HashSet	TreeSet
Ordering	Unordered (based on `hashCode()`)	Sorted (natural or custom order)
Implementation	Hash table	Red-Black Tree
Null Values	Allows one `null`	No `null` (if natural ordering)
Performance	O(1) avg. for `add()`, `remove()`	O(log n) for `add()`, `remove()`
Best Use Case	Fast lookups, no ordering needed	Sorted unique elements

2.1 HashSet Example (Unordered, Fast Operations)

```
import java.util.HashSet;

public class HashSetExample {
    public static void main(String[] args) {
        HashSet<String> cities = new HashSet<>();

        // Add elements (order not preserved)
        cities.add("Delhi");
        cities.add("Mumbai");
        cities.add("Bangalore");
        cities.add("Delhi"); // Duplicate (ignored)

        System.out.println(cities); // [Delhi, Mumbai, Bangalore] (order varies)
```

```
        // Fast contains-check (O(1))
        System.out.println(cities.contains("Mumbai")); // true

        // Allows one null
        cities.add(null);
        System.out.println(cities); // [null, Delhi, Mumbai, Bangalore]
    }
}
```

2.2 TreeSet Example (Sorted, Slower but Ordered)

```
import java.util.TreeSet;

public class TreeSetExample {
    public static void main(String[] args) {
        TreeSet<String> cities = new TreeSet<>();

        // Add elements (automatically sorted)
        cities.add("Delhi");
        cities.add("Mumbai");
        cities.add("Bangalore");
        cities.add("Delhi"); // Duplicate (ignored)

        System.out.println(cities); // [Bangalore, Delhi, Mumbai] (sorted)

        // Slower contains-check (O(log n))
        System.out.println(cities.contains("Mumbai")); // true

        // No null allowed (throws NullPointerException)
        // cities.add(null); // ❌ Runtime error
    }
}
```

10.3. Map Interface

A `Map` stores key-value pairs and does not allow duplicate keys.

Implementations:

- `HashMap` (Unordered, allows one `null` key)
- `LinkedHashMap` (Maintains insertion order)

- `TreeMap` (Sorted by keys)

Example:

```
java
import java.util.*;

public class MapExample {
    public static void main(String[] args) {
        Map<Integer, String> employees = new HashMap<>();
        employees.put(101, "John");
        employees.put(102, "Alice");
        System.out.println("Map: " + employees); // Output: {101=John, 102=Alice}
    }
}
```

HashMap vs. TreeMap in Java

1. Introduction

In Java, `HashMap` and `TreeMap` are two implementations of the `Map` interface under `java.util`. Both store key-value pairs, but they differ in ordering, performance, and use cases.

Feature	HashMap	TreeMap
Ordering	Unordered (based on `hashCode()`)	Sorted by keys (natural/custom order)
Implementation	Hash table	Red-Black Tree
Null Keys	Allows one `null` key	No `null` keys (if natural ordering)
Null Values	Allows multiple `null` values	Allows multiple `null` values
Performance	O(1) avg. for `get()`, `put()`	O(log n) for `get()`, `put()`

Best Use Case	Fast lookups, no ordering needed	Sorted key-value pairs

2.1 HashMap Example (Unordered, Fast Operations)

```
import java.util.HashMap;

public class HashMapExample {
    public static void main(String[] args) {
        HashMap<String, Integer> countryPopulations = new
HashMap<>();

        // Insert key-value pairs (order not preserved)
        countryPopulations.put("India", 1_417_000_000);
        countryPopulations.put("China", 1_412_000_000);
        countryPopulations.put("USA", 333_000_000);
        countryPopulations.put("India", 1_418_000_000); //
Updates existing key

        System.out.println(countryPopulations);
        // Output varies: {USA=333000000, China=1412000000,
India=1418000000}

        // Fast lookup (O(1))
        System.out.println("Population of India: " +
countryPopulations.get("India"));

        // Allows one null key and multiple null values
        countryPopulations.put(null, 0);
        countryPopulations.put("Unknown", null);
        System.out.println(countryPopulations); // {null=0,
USA=333000000, Unknown=null, ...}
    }
}
```

2.2 TreeMap Example (Sorted by Keys, Slower but Ordered)

```
import java.util.TreeMap;

public class TreeMapExample {
    public static void main(String[] args) {
        TreeMap<String, Integer> countryPopulations = new TreeMap<>();

        // Insert key-value pairs (automatically sorted by keys)
        countryPopulations.put("India", 1_417_000_000);
        countryPopulations.put("China", 1_412_000_000);
        countryPopulations.put("USA", 333_000_000);

        System.out.println(countryPopulations);
        // Output: {China=1412000000, India=1417000000, USA=333000000} (alphabetical
order)

        // Slower lookup (O(log n))
        System.out.println("Population of USA: " + countryPopulations.get("USA"));

        // No null keys allowed (throws NullPointerException)
        // countryPopulations.put(null, 0); // ❌ Runtime error

        // Null values allowed
        countryPopulations.put("Unknown", null);
        System.out.println(countryPopulations);
    }
}
```

10.4. Queue Interface

A `Queue` follows the FIFO (First-In-First-Out) principle.

Implementations:

- `LinkedList` (Can be used as a Queue)
- `PriorityQueue` (Orders elements based on priority)

Example:

java

```
import java.util.*;

public class QueueExample {
```

```
    public static void main(String[] args) {
        Queue<String> queue = new LinkedList<>();
        queue.add("Task1");
        queue.add("Task2");
        System.out.println("Queue: " + queue.poll()); // Output: Task1 (removes & returns
head)
    }
}
```

10.5. Difference Between Array and ArrayList

Feature	Array	ArrayList
Size	Fixed size	Dynamic (resizable)
Type Safety	Can be primitive or object	Only objects (uses generics)
Performance	Faster (fixed memory allocation)	Slightly slower (dynamic resizing)
Methods	No built-in methods	Rich API (`add()`, `remove()`, etc.)

Example:

```
java
// Array Example
int[] arr = new int[3];
arr[0] = 10;

// ArrayList Example
ArrayList<Integer> list = new ArrayList<>();
list.add(10);
```

10.6. Difference Between `Collection` and `Collections`

Feature	Collection (Interface)	Collections (Class)
Type	Root interface of collections	Utility class with static methods
Purpose	Defines basic operations (`add`, `remove`)	Provides helper methods (`sort()`, `reverse()`)
Usage	Implemented by `List`, `Set`, etc.	Used to operate on collections

Example:

```
java
import java.util.*;

public class CollectionVsCollections {
    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>(Arrays.asList(5, 2, 8));

        // Collections utility class
        Collections.sort(numbers); // Sorts the list
        System.out.println("Sorted List: " + numbers); // [2, 5, 8]
    }
}
```

10.7. File I/O Handling in Java

Java provides several APIs for file operations through the `java.io` and `java.nio` packages.

1 Core Classes for File I/O

Class	Purpose
`File`	Represents file/directory paths
`FileInputStream`/`FileOutputStream`	Byte stream I/O
`FileReader`/`FileWriter`	Character stream I/O
`BufferedReader`/`BufferedWriter`	Buffered character I/O
`Files` (NIO)	Modern file operations

2 Reading a File (Traditional Approach)

```
java
import java.io.*;

public class FileReadExample {
    public static void main(String[] args) {
        try (BufferedReader reader = new BufferedReader(new FileReader("input.txt"))) {
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
```

```
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

3 Writing to a File (NIO Approach)

```
java
import java.nio.file.*;
import java.util.*;

public class FileWriteExample {
    public static void main(String[] args) throws IOException {
        List<String> lines = Arrays.asList("Line 1", "Line 2", "Line 3");
        Files.write(Paths.get("output.txt"), lines, StandardOpenOption.CREATE);
    }
}
```

4 File Operations Cheat Sheet

```
java
// Check if file exists
boolean exists = Files.exists(Paths.get("file.txt"));

// Copy file
Files.copy(Paths.get("source.txt"), Paths.get("dest.txt"));

// Delete file
Files.deleteIfExists(Paths.get("file.txt"));
```

Tight Coupling And Loose Coupling:

🔒 Tight Coupling Example

In this version, `Bike` and `Car` are directly creating and using the `Engine` class. That means they're tightly coupled to it.

❌ Problem:

If you change how `Engine` works (e.g. rename method, add parameters), both `Bike` and `Car` must be changed.

```
package Demo;
public class Main {
```

```
        public static void main(String[] args) {
                Bike bike = new Bike();
                bike.startBike();

                Car car = new Car();
                car.startCar();
        }
}
```

```
package Demo;
public class Bike {
        Engine engine = new Engine(); // Tightly coupled
        void startBike() {
                engine.start();
                System.out.println("Bike started");
        }
}
```

```
package Demo;
public class Car {

        Engine engine = new Engine(); // Tightly coupled
        void startCar() {
                engine.start();
                System.out.println("Car started");
        }
}
```

```
package Demo;
public class Engine {

        void start() {
                System.out.println("Engine started");
        }
}
```

🔓 Loose Coupling Example

In this version, `Bike` and `Car` receive the `Engine` from outside (via constructor). That means they're loosely coupled — they don't care how `Engine` is created or what kind it is.

✅ Benefit:

- **You can reuse the same `Engine` object.**
- **`Bike` and `Car` don't depend on how `Engine` is created — this makes testing and changes easier.**

```
package Demo;
public class Main {

	public static void main(String[] args) {
		Bike bike = new Bike(new Engine());
		bike.startBike();

		Car car = new Car(new Engine());
		car.startCar();
	}
}
```

```
package Demo;
public class Car {

//	Engine engine = new Engine(); // Tightly coupled
//	void startCar() {
//		engine.start();
//		System.out.println("Car started");
//	}

	Engine engine;

   // Constructor Injection
   Car(Engine engine) {
	this.engine = engine;
   }
   void startCar() {
	engine.start();
	System.out.println("Car started");
   }
}
```

```
package Demo;
public class Bike {
//	Engine engine = new Engine(); // Tightly coupled
//
//	void startBike() {
//		engine.start();
//		System.out.println("Bike started");
//	}

	Engine engine;
   // Constructor Injection
```

```
    Bike(Engine engine) {
        this.engine = engine;
    }
    void startBike() {
        engine.start();
        System.out.println("Bike started");
    }
}
```

```
package Demo;
public class Engine {

    void start() {
        System.out.println("Engine started");
    }
}
```

10.8. Generics in Java

✅ Definition:

Generics allow you to write code that works with any type (class) while maintaining type safety.

🔍 Why use Generics?

- **Avoid `ClassCastException`**
- **Enable compile-time type checking**
- **Reusable and flexible code**

✅ Example without Generics:

```
package Demo;
import java.util.ArrayList;
public class Main {
    public static void main(String[] args) {
        ArrayList list = new ArrayList(); // raw type
        list.add("Hello");
        list.add(123); // No error here
        String text = (String) list.get(0); // OK
        String num = (String) list.get(1); // Runtime error:
ClassCastException
```

```
        }
}
```

```
package Demo;
import java.util.ArrayList;
public class Main {
      public static void main(String[] args) {
            ArrayList<String> list = new ArrayList<>();
            list.add("Hello");
            // list.add(123); // Compile-time error
            String text = list.get(0); // No need to cast
            System.out.println(text);
      }
}
```

✅ Generic Class Example:

```
package Demo;
public class Main {
      public static void main(String[] args) {
            Box<String> stringBox = new Box<>();
             stringBox.set("Hello");
             System.out.println(stringBox.get());
             Box<Integer> intBox = new Box<>();
             intBox.set(123);
             System.out.println(intBox.get());
      }
}
```

```
package Demo;
public class Box<T> {
   T value;
   void set(T value) {
       this.value = value;
   }
   T get() {
       return value;
   }
}
```

Lambda:
1.1

```java
package Demo;
public class Test {

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

```java
package Demo;
public class Demo{
	public static void main(String[] args) {

		Demo d = new Demo();
		d.display(new Test());
	}



	public void display(Test t) {
		int sum = t.add(5, 10);
		System.out.println(sum);
	}


}
```

1.2

```java
package Demo;
public interface Test {

	int add(int a,int b);
}
```

```java
package Demo;
```

```java
public class Demo implements Test{
	public static void main(String[] args) {

		Demo d = new Demo();
		int sum = d.add(5, 10);
		System.out.println(sum);
	}
	@Override
	public int add(int a, int b) {
		// TODO Auto-generated method stub
		return a+b;
	}



//	public void display(Test t) {
//		int sum = t.add(5, 10);
//		System.out.println(sum);
//	}


}
```

1.3 🔁 Functional Interface in Java

✅ Definition:

A functional interface is an interface that contains only one abstract method (but can have default/static methods).
It is used mainly with lambda expressions and method references.

✅ Declaring a Functional Interface:

```java
package Demo;
@FunctionalInterface
public interface Test {

	int add(int a,int b);
}
```

Why Do We Need Interfaces in Java?

1. ✅ Achieve Abstraction

2. 🔁 Multiple Inheritance and
3. 🔌 Loose Coupling

Anonymous Inner Class

✅ Definition:

An anonymous inner class is a one-time use class without a name that:

- **Implements an interface or extends a class**
- **Is defined inside a method or block**
- **Is used for quick implementation**

🔍 Why "anonymous"?

Because you are not giving the class a name — you're creating an object and defining the class body at the same time.

```
package Demo;
public class Demo {
	public static void main(String[] args) {
		Demo d = new Demo();
//		Test t = new Test() {
//			@Override
//			public int add(int a,int b) {
//				return a+b;
//			}
//		};
		d.display((a,b)->a+b);
	}
	public void display(Test t) {
		int sum = t.add(5, 10);
		System.out.println(sum);
```

```
        }
}
```

```
package Demo;
@FunctionalInterface
public interface Test {

      int add(int a,int b);
}
```

Lambda:

```
package Demo;
public class Demo {
      public static void main(String[] args) {

            Demo d =new Demo();
//          d.display(new Test());
            d.display((a,b)-> a+b);
      }



      public void display(Test t) {
            int sum = t.add(5, 10);
            System.out.println(sum);
      }
}
```

(a, b) -> a + b

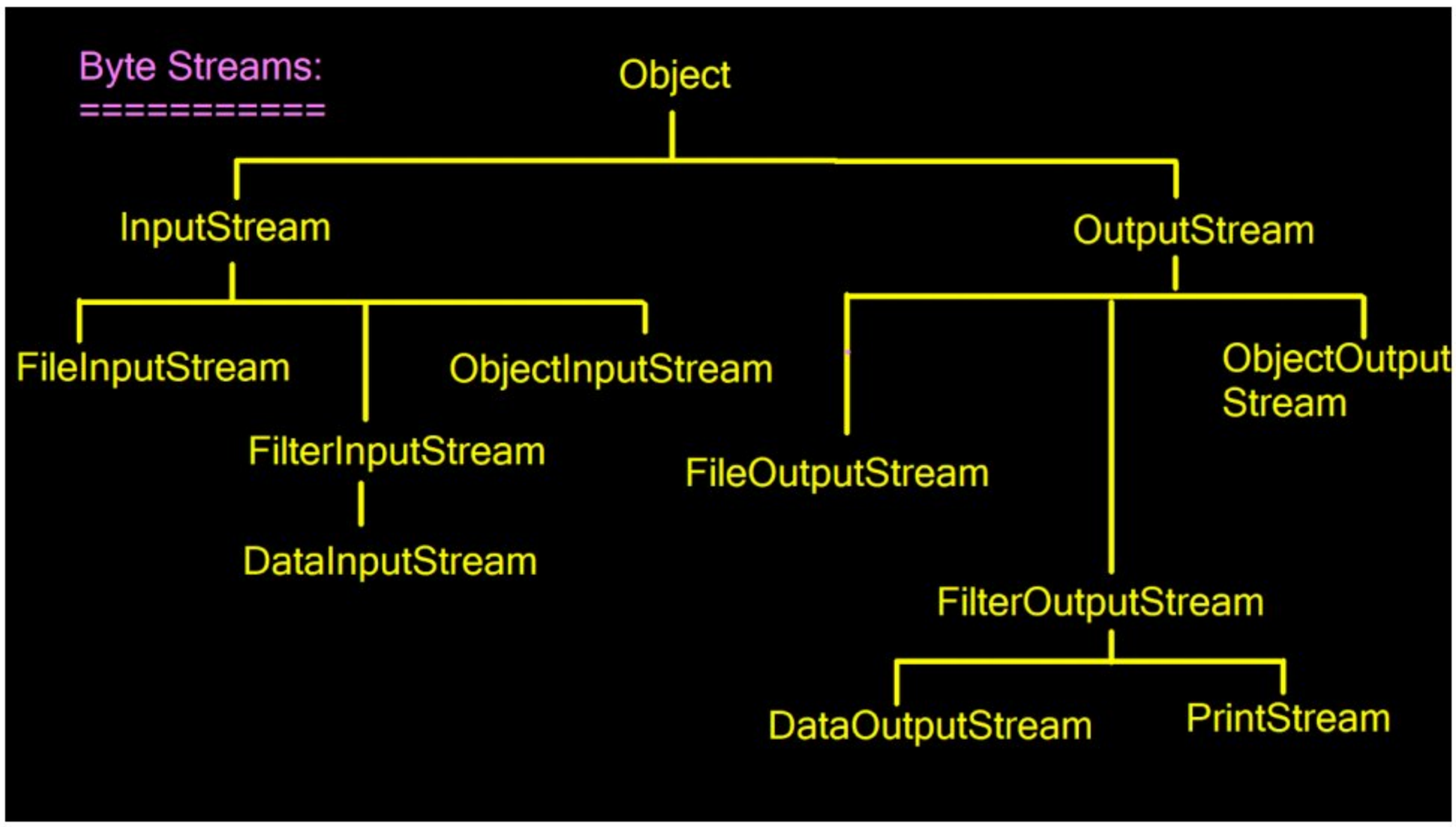
This is a lambda expression that implements the `add(int a, int b)` method of the `Test` interface.

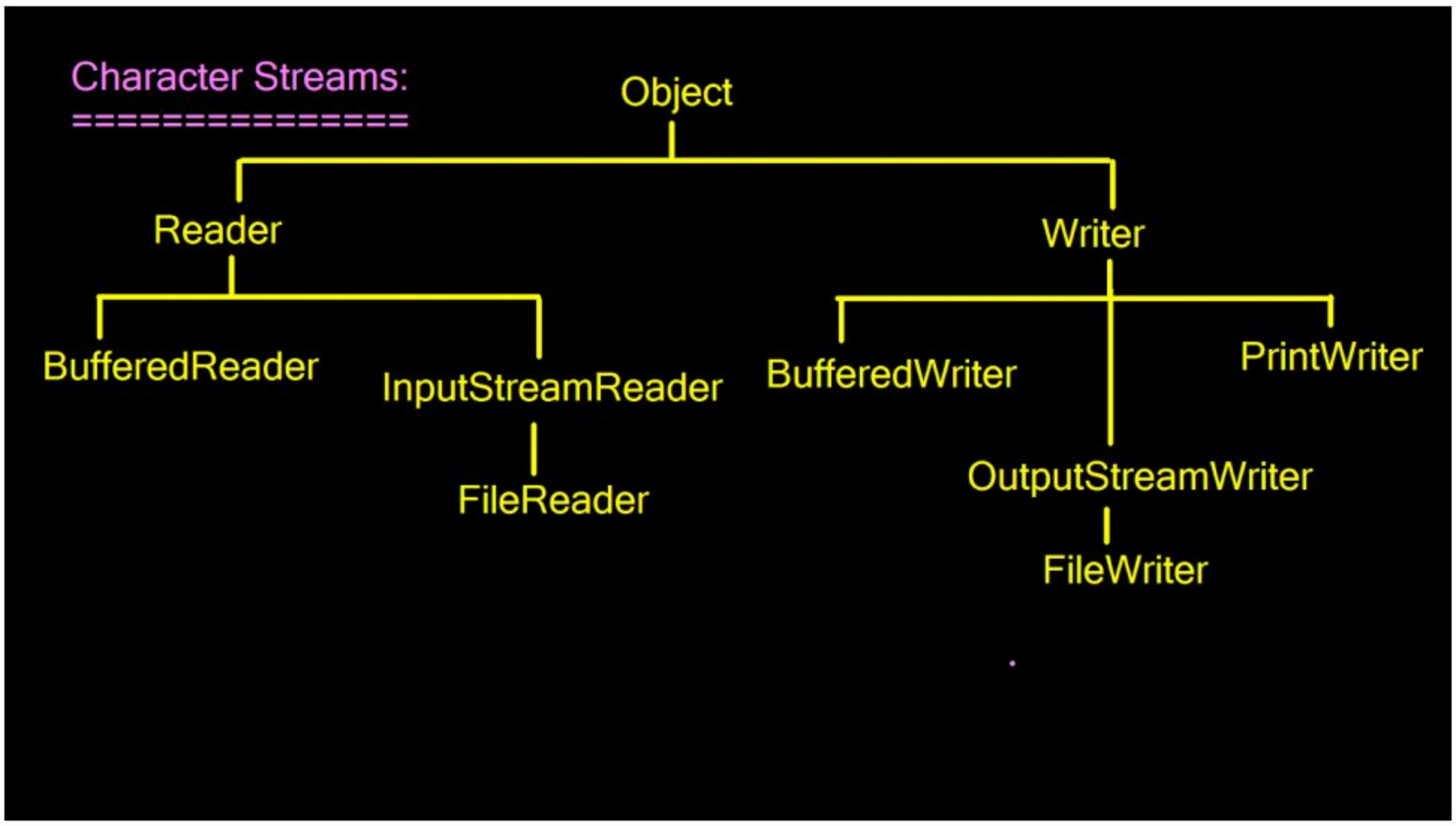
- `(a, b)` are the parameters.
- `a + b` is the return value.
- It matches the signature of `Test.add(int, int)`.

lambda expressions can only be used with functional interfaces.

This is equivalent to writing:

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

            Demo d =new Demo();
//          d.display(new Test());
//          d.display((a,b)-> a+b);
            Test t = new Test() {
                @Override
                public int add(int a, int b) {
                    return a + b;
                }
            };
            d.display(t);
      }
```





10.9. Lambda Expressions

Lambda expressions provide a concise way to represent functional interfaces.

1 Lambda Syntax

java
(parameters) -> expression
(parameters) -> { statements; }

2 Common Functional Interfaces

Interface	Method	Example
`Predicate<T>`	`boolean test(T t)`	`x -> x > 5`
`Function<T,R>`	`R apply(T t)`	`s -> s.length()`
`Consumer<T>`	`void accept(T t)`	`x -> System.out.println(x)`
`Supplier<T>`	`T get()`	`() -> Math.random()`

3 Practical Examples

```
java
import java.util.*;
import java.util.function.*;

public class LambdaExamples {
    public static void main(String[] args) {
        // Predicate example
        Predicate<Integer> isEven = n -> n % 2 == 0;
        System.out.println(isEven.test(4)); // true

        // Function example
        Function<String, Integer> lengthFinder = s -> s.length();
        System.out.println(lengthFinder.apply("Java")); // 4

        // Consumer example
        Consumer<String> printer = s -> System.out.println(">> " + s);
        printer.accept("Hello"); // >> Hello

        // Runnable example
        Runnable task = () -> System.out.println("Task executed");
        new Thread(task).start();
    }
}
```

4 Method References

```
java
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

// Equivalent to: names.forEach(s -> System.out.println(s));
names.forEach(System.out::println);
```

5 Comparator with Lambda

```
java
List<String> words = Arrays.asList("apple", "banana", "cherry");
words.sort((s1, s2) -> s1.length() - s2.length());
```

10.10. Combining Concepts: File Processing with Lambdas

```
java
import java.io.*;
import java.nio.file.*;
import java.util.stream.*;

public class AdvancedFileProcessing {
```

```
public static void main(String[] args) throws IOException {
    // Read all lines and filter
    Files.lines(Paths.get("data.txt"))
        .filter(line -> line.startsWith("A"))
        .map(String::toUpperCase)
        .forEach(System.out::println);

    // Write processed data
    List<String> processed = Files.lines(Paths.get("input.txt"))
                            .map(s -> s.replace("old", "new"))
                            .collect(Collectors.toList());
        Files.write(Paths.get("output.txt"), processed);
    }
}
```

Key Takeaways

1. File I/O: Prefer `java.nio.Files` for modern operations
2. Generics: Provide type safety and reduce casting
3. Lambdas: Enable functional programming with concise syntax
4. Best Practice: Always use try-with-resources for file operations

---

11. Java 8 Features

Java 8 introduced major enhancements to the Java programming language, including functional programming features, improved APIs, and performance optimizations. Below is a structured breakdown of key Java 8 features with professional examples.

11.1. Lambda Expressions

Lambda expressions enable functional programming by allowing concise implementation of single-method interfaces (Functional Interfaces).

Syntax:

java
(parameters) -> { body }

Example:

```
java
// Traditional approach (Anonymous class)
Runnable runnable1 = new Runnable() {
    @Override
    public void run() {
        System.out.println("Running without Lambda");
    }
};

// Using Lambda
Runnable runnable2 = () -> System.out.println("Running with Lambda");

runnable1.run(); // Output: Running without Lambda
runnable2.run(); // Output: Running with Lambda
```

11.2. Functional Interfaces

A Functional Interface has exactly one abstract method and can be annotated with `@FunctionalInterface`.

Common Built-in Functional Interfaces:

Interface	Method	Description
`Predicate<T>`	`boolean test(T t)`	Checks a condition (returns boolean)
`Function<T,R>`	`R apply(T t)`	Takes input `T`, returns output `R`
`Consumer<T>`	`void accept(T t)`	Performs an operation on `T` (no return)
`Supplier<T>`	`T get()`	Supplies a value (no input)

Example:

```
java
import java.util.function.*;

public class FunctionalInterfaceExample {
    public static void main(String[] args) {
        // Predicate: Check if number is even
        Predicate<Integer> isEven = num -> num % 2 == 0;
```

```
System.out.println(isEven.test(10)); // true

// Function: Convert String to its length
Function<String, Integer> strLength = s -> s.length();
System.out.println(strLength.apply("Java")); // 4

// Consumer: Print a message
Consumer<String> printer = msg -> System.out.println("Message: " + msg);
printer.accept("Hello Java 8!");

    // Supplier: Generate a random number
    Supplier<Double> randomSupplier = () -> Math.random();
    System.out.println(randomSupplier.get()); // e.g., 0.742913
  }
}
```

11.3. Stream API

The Stream API allows functional-style operations on collections (e.g., filtering, mapping, reducing).

Key Stream Operations:

Operation	Description	Example
`filter()`	Filters elements based on condition	`list.stream().filter(x -> x > 5)`
`map()`	Transforms elements	`list.stream().map(x -> x * 2)`
`sorted()`	Sorts elements	`list.stream().sorted()`
`forEach()`	Performs action on each element	`list.stream().forEach(System.out::println)`
`collect()`	Converts stream back to collection	`list.stream().collect(Collectors.toList())`

Example:

```
java
import java.util.*;
import java.util.stream.*;

public class StreamExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(3, 1, 4, 1, 5, 9);
```

```
    // Filter even numbers, square them, and collect to list
    List<Integer> result = numbers.stream()
        .filter(n -> n % 2 == 1)  // Keep odd numbers
        .map(n -> n * n)        // Square them
        .sorted()               // Sort
        .collect(Collectors.toList());

    System.out.println(result); // [1, 1, 9, 25, 81]
}
```

11.4. Method References

Method references provide a shorthand for lambda expressions calling existing methods.

Types of Method References:

Type	Syntax	Example
Static Method Ref	`ClassName::method`	`Math::max`
Instance Method Ref	`object::method`	`System.out::println`
Constructor Ref	`ClassName::new`	`ArrayList::new`

Example:

```
java
import java.util.*;

public class MethodReferenceExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

        // Using Lambda
        names.forEach(name -> System.out.println(name));

        // Using Method Reference (equivalent)
        names.forEach(System.out::println);
    }
}
```

11.5. Default Methods in Interfaces

Interfaces can now have default methods (with implementation) to ensure backward compatibility.

Example:

```
java
interface Vehicle {
    void start();  // Abstract method

    // Default method (Java 8+)
    default void honk() {
        System.out.println("Beep beep!");
    }
}

class Car implements Vehicle {
    @Override
    public void start() {
        System.out.println("Car started");
    }
}

public class DefaultMethodExample {
    public static void main(String[] args) {
        Car car = new Car();
        car.start(); // Output: Car started
        car.honk();  // Output: Beep beep! (default method)
    }
}
```

11.6. Optional Class

`Optional<T>` **helps avoid NullPointerException by explicitly handling null values.**

Common Methods:

Method	Description
`ofNullable()`	Wraps a nullable value
`isPresent()`	Checks if value exists

`orElse()`	Provides a default if value is null

Example:

```
java
import java.util.Optional;

public class OptionalExample {
    public static void main(String[] args) {
        Optional<String> name = Optional.ofNullable(getName());

        // Traditional null check
        if (name.isPresent()) {
            System.out.println(name.get());
        } else {
            System.out.println("Name not found");
        }

        // Functional style
        System.out.println(name.orElse("Name not found"));
    }

    static String getName() {
        return Math.random() > 0.5 ? "Alice" : null;
    }
}
```

11.7. Date and Time API (java.time)

Java 8 introduced a new immutable and thread-safe Date-Time API under `java.time`.

Key Classes:

Class	Description
`LocalDate`	Date without time (e.g., `2025-06-09`)
`LocalTime`	Time without date (e.g., `14:30:00`)
`LocalDateTime`	Date + Time (e.g., `2025-06-09T14:30:00`)
`ZonedDateTime`	Date + Time + Timezone

Example:

```
java
import java.time.*;

public class DateTimeExample {
    public static void main(String[] args) {
        LocalDate today = LocalDate.now();
        System.out.println("Today: " + today); // e.g., 2025-06-09

        LocalTime now = LocalTime.now();
        System.out.println("Now: " + now); // e.g., 14:30:00.123

        LocalDateTime current = LocalDateTime.now();
        System.out.println("Current: " + current); // e.g., 2025-06-09T14:30:00.123
    }
}
```

✅ Top Java 8 Features (with Examples)

1. 🔁 Lambda Expressions
2. 🔗 Functional Interfaces
3. 📦 Default & Static Methods in Interfaces

```java
package Demo;
public interface Engine {
      public abstract void start() ;

      abstract void stop();

       default void show() {
              System.out.println("Default method in interface");
          }

       static int square(int x) {
              return x * x;
          }

       default void display() {
              System.out.println("Default method in interface");
          }

      static int add(int x,int y) {
              return x + y;
          }
}
```

```java
package Demo;
public class Main {

	public static void main(String[] args) {

		Engine engine = new Engine() {

			@Override
			public void stop() {
				// TODO Auto-generated method stub

			}

			@Override
			public void start() {
				// TODO Auto-generated method stub

			}
		};

		engine.show();
		System.out.println(Engine.square(3));

	}
}
```

4. 🌊 Stream API

```
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6);

List<Integer> even = list.stream()
                         .filter(n -> n % 2 == 0)
                         .collect(Collectors.toList());

System.out.println(even);  // Output: [2, 4, 6]
```

5. 🎁 Optional Class

✅ What is `Optional<T>`?

`Optional` is a container object that may or may not contain a non-null value.

```
Optional<String> name = Optional.of("John");

Optional<String> empty = Optional.empty();
```

🔧 Common `Optional` Methods

Method	Description
`of(value)`	Returns Optional with non-null value
`ofNullable(value)`	Returns Optional, allows `null` value
`empty()`	Returns empty Optional
`isPresent()`	Checks if value is present
`ifPresent()`	Runs action if value is present
`orElse(default)`	Returns value or default if absent
`orElseGet(Supplier)`	Like `orElse` but takes a lambda
`orElseThrow()`	Throws exception if value not present
`map()`	Transforms the value if present
`flatMap()`	Like map, but avoids nested Optionals

6. 📅 New Date and Time API (`java.time`)

```
import java.time.LocalDate;

LocalDate today = LocalDate.now();
LocalDate birthday = LocalDate.of(1995, 6, 25);

System.out.println(today);       // e.g. 2025-06-11
System.out.println(birthday);    // 1995-06-25
```

✅ Key Classes in `java.time` Package

Class	Description	Example
`LocalDate`	Date without time (YYYY-MM-DD)	`2025-06-11`
`LocalTime`	Time without date	`10:15:30`
`LocalDateTime`	Date and time without timezone	`2025-06-11T10:15:30`
`ZonedDateTime`	Date and time with timezone	`2025-06-11T10:15:30+05:30[Asia/Kolkata]`
`Instant`	Timestamp (epoch time)	`2025-06-11T04:45:30Z`
`Duration`	Amount of time (in seconds/nanos)	`PT20M` (20 minutes)
`Period`	Amount of time (in days, months, years)	`P1Y2M3D`
`DateTimeFormatter`	Format and parse dates	`dd-MM-yyyy`

7. 🔍 Method References

```
List<String> list = Arrays.asList("a", "b", "c");
list.forEach(System.out::println);
OR
 list.forEach(s -> System.out.println(s));
OR
  Consumer<String> printConsumer = new Consumer<String>() {
        @Override
        public void accept(String s) {
                System.out.println(s);
        }
        };

        // Pass it to forEach
        list.forEach(printConsumer);

OR
List<String> list = Arrays.asList("a", "b", "c");
```

```
for (String s : list) {
        System.out.println(s);
}
```

System.out.println();

🧠 Summary Table:

Part	Type	Description
`System`	Class	Gives access to system-level resources
`out`	Static field	A `PrintStream` object (console output)
`println()`	Method	Prints data + newline

8. 🧩 Collectors (Part of Stream API)

```
List<String> result = names.stream()
                .filter(n -> n.startsWith("A"))
                .collect(Collectors.toList());
```

Java Version History:
https://en.wikipedia.org/wiki/Java_version_history

11.8. Java Version History

https://en.wikipedia.org/wiki/Java_version_history

Conclusion

Java 8 revolutionized Java with lambda expressions, streams, functional interfaces, and modern APIs like `Optional` and `java.time`. These features improve code readability, maintainability, and performance.

12. Java Memory Management & Memory Leaks - Professional Documentation

12.1. Types of Memory Management in Java

Java uses automatic memory management via Garbage Collection (GC). The JVM divides memory into different regions:

1.1 Heap Memory

- **Stores objects and runtime data.**
- **Divided into:**
  - **Young Generation (Eden, S0, S1) – Short-lived objects.**
  - **Old Generation (Tenured Space) – Long-lived objects.**
  - **Metaspace (Java 8+) – Stores class metadata (replaced PermGen).**

1.2 Stack Memory

- **Stores method calls, local variables, and references.**
- **Thread-specific (each thread has its own stack).**
- **Fixed size (can throw `StackOverflowError` if exceeded).**

1.3 Non-Heap Memory (Native Memory)

- **Includes Metaspace, JVM internal structures, and native libraries.**
- **Not managed by GC.**

Example: Memory Allocation

java

```
public class MemoryExample {
    public static void main(String[] args) {
        // Heap Memory - Object allocation
        String str = new String("Hello, Heap!"); // Stored in Heap

        // Stack Memory - Local variables & method calls
        int num = 10; // Stored in Stack
    }
}
```

12.2. What is a Memory Leak?

A memory leak occurs when:

- **Objects are no longer needed but still referenced.**
- **Garbage Collector (GC) cannot reclaim their memory.**
- **Leads to `OutOfMemoryError` over time.**

Common Causes of Memory Leaks

Cause	Description	Example
Static Collections	Static `Map`/`List` holding objects indefinitely	`static Map<Integer, Object> cache = new HashMap<>();`
Unclosed Resources	Not closing streams, DB connections, or files	`BufferedReader reader = new BufferedReader(new FileReader("file.txt"));`
Listeners & Callbacks	Failing to deregister event listeners	`button.addActionListener(...)` but never removed
ThreadLocal Variables	Thread-local variables not cleaned up	`ThreadLocal<Object> threadData = new ThreadLocal<>();`

12.3. Why Does Memory Leaking Happen?

Root Causes

1. **Unintentional Object Retention**
   - **Objects are referenced longer than needed.**
   - **Example: Caching without eviction policy.**
2. **Long-lived References**
   - **Static fields, singletons, or thread pools holding objects.**
3. **Poor Resource Management**
   - **Not closing `InputStream`, `Connection`, or `Session`.**

Example: Static Map Leak

java

```
import java.util.*;

public class MemoryLeakExample {
    private static final Map<Integer, String> CACHE = new HashMap<>();
```

```
public static void addToCache(Integer id, String value) {
    CACHE.put(id, value); // Objects never removed → Leak!
}

public static void main(String[] args) {
    for (int i = 0; i < 1_000_000; i++) {
        addToCache(i, "Data " + i); // Eventually causes OutOfMemoryError
    }
}
```

12.4. How to Handle Memory Leaks?

4.1 Detection Tools

Tool	Purpose
VisualVM	Monitor heap usage, analyze GC behavior
Eclipse MAT (Memory Analyzer)	Identify leak suspects in heap dumps
JProfiler	Real-time memory profiling
`-XX:+HeapDumpOnOutOfMemoryError`	Automatically dump heap on `OOM`

4.2 Prevention & Fixes

✅ **Solution 1: Use Weak References**

- Allows GC to collect objects when memory is low.

```
java
private static final Map<Integer, WeakReference<String>> CACHE = new
HashMap<>();
```

✅ **Solution 2: Close Resources Properly**

- Use try-with-resources (AutoCloseable).

```
java
try (BufferedReader reader = new BufferedReader(new FileReader("file.txt"))) {
    // Automatically closes reader
}
```

✅ **Solution 3: Evict Unused Objects**

- Use `WeakHashMap` or `LinkedHashMap` with `removeEldestEntry()`.

```
java
private static final Map<Integer, String> CACHE = new LinkedHashMap<>() {
    @Override
    protected boolean removeEldestEntry(Map.Entry<Integer, String> eldest) {
        return size() > 100; // Keep only 100 entries
    }
};
```

✅ **Solution 4: Deregister Listeners**

```
java
button.removeActionListener(listener); // Prevent leaks
```

✅ **Solution 5: Avoid `ThreadLocal` Misuse**

- Always call `remove()` after use.

```
java
threadLocal.set(data);
try {
    // Use data
} finally {
    threadLocal.remove(); // Clean up
}
```

12.5. Example: Fixing a Memory Leak

Before (Leaking Code)

```
java
public class LeakyClass {
    private static List<Object> LIST = new ArrayList<>();

    public void addToGlobalList(Object obj) {
        LIST.add(obj); // Never removed → Leak!
    }
}
```

After (Fixed Code)

```
java
public class FixedClass {
    private static final int MAX_SIZE = 1000;
    private static List<Object> LIST = new ArrayList<>();
```

```
    public void addToGlobalList(Object obj) {
        if (LIST.size() >= MAX_SIZE) {
            LIST.remove(0); // Evict oldest entry
        }
        LIST.add(obj);
    }
}
```

12.6. Best Practices to Avoid Memory Leaks

✔ Avoid long-lived static collections (use soft/weak references).
✔ Always close resources (`try-with-resources`).
✔ Profile applications (VisualVM, MAT).
✔ Limit cache sizes (use `Caffeine`, `Guava Cache`).
✔ Monitor `OutOfMemoryError` (enable heap dumps).

Conclusion

- Memory leaks occur due to unintentional object retention.
- Detection tools (VisualVM, MAT) help identify leaks.
- Solutions: Weak references, proper resource closing, eviction policies.
