

# The history of Java

is a fascinating journey of innovation and adaptation, tracing the evolution of one of the most widely used programming languages in the world. Here's an overview:

## 1. Genesis and Early Development (1991–1995)

- **Creators:** Java was developed by James Gosling, Mike Sheridan, and Patrick Naughton at Sun Microsystems, which is now owned by Oracle Corporation.
- **Project Name:** Initially, the project was called "**Green Project**", and the language was known as "**Oak**" (named after an oak tree outside Gosling's office).
- **Purpose:** It was originally designed for interactive television systems, but it turned out to be too advanced for that industry at the time.
- **Rebranding:** In 1995, Oak was renamed **Java** due to trademark issues. The name was inspired by Java coffee, reflecting the creators' preference for coffee while coding.

## 2. The Key Principles of Java

The creators aimed to develop a language that adhered to these principles:

- **Platform Independence:** Write Once, Run Anywhere (WORA)
- **Object-Oriented:** Based on classes and objects
- **Robust and Secure:** Designed for reliability and secure operations
- **Multithreaded:** Support for concurrent programming
- **Dynamic and Portable:** Adaptable to evolving environments

## 3. Major Milestones

### Java 1.0 (1996)

- Officially released by Sun Microsystems.
- Core features introduced:
  - Platform independence with the **Java Virtual Machine (JVM)**.
  - Applets for web browsers.
- Widely adopted by web developers.

### Java 2 (1998-1999)

- Marked the division of Java into different editions:
  - **J2SE (Java 2 Standard Edition)**: Core Java features for desktop applications.
  - **J2EE (Java 2 Enterprise Edition)**: For enterprise-level applications.
  - **J2ME (Java 2 Micro Edition)**: For mobile and embedded devices.
- Enhanced performance and introduced features like Swing for GUI development.

### Java SE 5.0 (2004)

- Introduced major enhancements:
  - **Generics**: Type safety for collections.
  - **Enhanced for-loop**
  - **Annotations**
  - **Autoboxing and Unboxing**

### Java SE 6 (2006)

- Focused on improvements in performance and web services.
- Enhanced tools like the compiler API and scripting.

### Java SE 7 (2011)

- Added features such as:
  - **Try-with-resources statement** for better exception handling.
  - **Diamond operator** for simpler generics.

- NIO (New Input/Output) enhancements.

### Java SE 8 (2014)

- Revolutionary update with:
  - **Lambda Expressions:** Functional programming support.
  - **Streams API:** Simplified data processing.
  - **Date and Time API:** Modern date/time handling.
  - **Default Methods:** Allowed interfaces to have default implementations.

### Java SE 9 (2017)

- Introduced the **Java Platform Module System (JPMS)**.
- REPL tool (**JShell**) for interactive Java coding.

### Java SE 11 (2018)

- Part of Oracle's shift to a faster release cycle (every 6 months).
- Introduced **LTS (Long-Term Support)** releases.

### Recent Updates (Java SE 12–21)

- Continued enhancements in performance, garbage collection, pattern matching, and language features.
- Regular introduction of modern features to stay competitive in modern development ecosystems.

---

## 4. Java Today

- **Ownership:** Oracle Corporation acquired Sun Microsystems in 2010 and continues to develop Java.
- **Popularity:** Java remains one of the most popular programming languages globally, widely used in web development, enterprise applications, Android app development, and more.

- **Ecosystem:** It has a vibrant ecosystem with frameworks like Spring, Hibernate, and tools for almost every development need.

Java's emphasis on platform independence, reliability, and continual evolution has made it a cornerstone of modern computing for over three decades.

# Core Java Notes By AJAY RAZZ

Java is a widely-used programming language known for its robust, secure, and platform-independent features. Here's a detailed explanation of

# Java's key features

---

## 1. Platform Independence

- "**Write Once, Run Anywhere**" (**WORA**): Java programs can run on any platform with a compatible Java Virtual Machine (JVM).
- Java source code is compiled into bytecode, which is platform-independent and executed by the JVM on any device or operating system.

## 2. Object-Oriented

- Everything in Java revolves around objects and classes.
- Core Object-Oriented Programming (OOP) principles:
  - **Encapsulation:** Data hiding using classes and access modifiers.
  - **Inheritance:** Code reuse by inheriting features from a parent class.
  - **Polymorphism:** One interface, multiple implementations.
  - **Abstraction:** Hiding implementation details from the user.

## 3. Simple and Familiar

- Java syntax is simple and easy to understand, especially for developers familiar with C or C++.
- Features like pointers and operator overloading, which complicate C/C++, are excluded in Java.

## 4. Secure

- Java emphasizes security at multiple levels:
    - **No explicit pointers:** Prevents unauthorized memory access.
    - **Bytecode verification:** Ensures safe code execution by the JVM.
    - **Security APIs:** For encryption, decryption, and secure communication.
    - **Sandboxing:** Limits applets from accessing system resources.
- 

## 5. Robust

- Designed to handle runtime errors and prevent system crashes.
  - Key aspects of robustness:
    - **Automatic garbage collection:** Manages memory and prevents memory leaks.
    - **Exception handling:** Provides mechanisms to handle runtime exceptions gracefully.
    - **Strong type checking:** Reduces runtime errors by catching many issues during compilation.
- 

## 6. Multithreaded

- Java has built-in support for multithreading, allowing multiple threads to run simultaneously.
  - Features:
    - **Thread class and Runnable interface** for thread creation.
    - Simplifies performing multitasking within a program.
    - Offers synchronization mechanisms to handle thread interference.
- 

## 7. High Performance

- Java is faster than traditional interpreted languages due to its:
    - **Just-In-Time (JIT) Compiler:** Compiles frequently executed code into native machine code for better performance.
    - Efficient garbage collection and memory management.
- 

## 8. Distributed

- Java is designed for building distributed systems:
    - **Remote Method Invocation (RMI)** and **CORBA:** Allow Java programs to interact over a network.
    - Supports internet protocols like HTTP and FTP.
- 

## 9. Dynamic

- Java is capable of adapting to evolving environments:
    - **Dynamic loading:** Classes are loaded into memory as needed during runtime.
    - **Reflection API:** Enables inspection and modification of classes, methods, and fields at runtime.
- 

## 10. Portable

- Java bytecode is platform-independent, and the Java API ensures consistency across platforms.
  - Java programs can run on any machine with minimal adjustments, provided a JVM is available.
- 

## 11. Architecture-Neutral

- Java code does not depend on any specific architecture or system.

- The size of data types (e.g., `int`, `long`) is consistent across platforms, ensuring predictable behavior.
- 

## 12. Scalable

- Java is suitable for developing both small-scale applications and large-scale enterprise systems.
  - Frameworks like Spring and Hibernate extend its capabilities for building scalable, high-performance applications.
- 

## 13. Rich Standard Library

- Java offers a vast library (Java Standard API) for various functionalities:
    - **Data structures:** Collections framework, arrays, and more.
    - **I/O operations:** File handling, streams, readers, and writers.
    - **Networking:** Handling HTTP requests, sockets, and more.
    - **Database access:** Java Database Connectivity (JDBC).
    - **GUI Development:** Swing, JavaFX, and AWT.
- 

## 14. Open-Source

- The Java Development Kit (JDK) is open-source, enabling a wide range of developers to contribute and use it freely.
- 

These features collectively make Java a versatile, reliable, and widely-adopted programming language for web development, enterprise applications, Android apps, and more.

The **Java Virtual Machine (JVM)** is a core component of the Java platform, responsible for running Java applications by converting **Java bytecode** into machine code for execution. The JVM architecture ensures platform independence, memory management, and runtime execution.

Here's a detailed explanation of the JVM architecture:

---

## Key Components of the JVM Architecture

### 1. Classloader

- The **Classloader** is a subsystem responsible for loading class files into memory.
  - Java uses dynamic class loading, meaning classes are loaded as needed during runtime.
  - Types of Classloaders:
    - **Bootstrap Classloader**: Loads core Java classes (`java.lang`, `java.util`).
    - **Extension Classloader**: Loads classes from the `ext` directory.
    - **Application Classloader**: Loads classes from the application's classpath.
- 

### 2. Method Area

- Stores **class-level metadata**, such as:
    - Class names
    - Fully qualified names of methods and fields
    - Constant pool (literals and references)
  - Shared among all threads.
  - Also referred to as the **Permanent Generation (PermGen)** in older JVM versions or **Metaspace** in newer versions (from Java 8).
-

### 3. Heap

- The heap is the region of memory used for allocating **objects** and **class instances**.
  - It is shared among all threads.
  - Garbage collection occurs in the heap to reclaim memory that is no longer in use.
  - Divided into:
    - **Young Generation**: Newly created objects (further divided into Eden and Survivor spaces).
    - **Old Generation (Tenured)**: Long-lived objects.
    - **Metaspace**: Stores class metadata.
- 

### 4. Java Stack

- Each thread has its own stack.
  - Stores **method call frames**, which include:
    - Local variables
    - Method return values
    - Operand stack (used for intermediate calculations).
  - Operates in a **Last In, First Out (LIFO)** manner.
- 

### 5. Program Counter (PC) Register

- Each thread has its own PC register.
  - Holds the **address of the currently executing instruction** for the thread.
  - Updated as the thread progresses through instructions.
- 

### 6. Native Method Stack

- Used for executing **native methods** (methods written in languages like C/C++).
  - Stores the state and execution context for native method invocations.
- 

## 7. Execution Engine

- The execution engine is responsible for executing the bytecode loaded into memory. It has three main parts:
  1. **Interpreter:**
    - Executes bytecode line by line.
    - Slower due to the repetitive process of interpreting instructions.
  2. **Just-In-Time (JIT) Compiler:**
    - Converts frequently used bytecode into native machine code.
    - Improves performance by avoiding repetitive interpretation.
  3. **Garbage Collector (GC):**
    - Automatically manages memory.
    - Reclaims memory occupied by objects no longer in use.
    - Uses algorithms like Mark-Sweep, Copying, and Generational GC.

## 8. Java Native Interface (JNI)

- Provides an interface for Java to interact with native applications written in other languages (e.g., C/C++).
  - Allows integration of platform-specific libraries and resources.
- 

## 9. Native Method Libraries

- Contain platform-specific code required by the JVM to execute native methods.
- Typically implemented as shared libraries (.dll, .so).

---

## JVM Workflow

1. **Compilation:** Java source code is compiled into bytecode by the **Java Compiler (javac)**.
2. **Class Loading:** The Classloader loads the compiled bytecode into the JVM.
3. **Bytecode Execution:**
  - o The Execution Engine interprets or compiles the bytecode.
  - o The PC register points to the next instruction.
  - o Objects and variables are stored in the heap and stack as needed.
4. **Memory Management:**
  - o The Garbage Collector reclaims memory from unused objects.

---

## Diagram of JVM Architecture

(If you want, I can generate a detailed visual representation of the JVM architecture to make it clearer.)

---

## Key Features of the JVM

- **Platform Independence:** Executes bytecode on any platform with a compatible JVM.
- **Automatic Memory Management:** Handles allocation and deallocation of memory.
- **Thread Management:** Ensures concurrent execution through thread handling.

The JVM's architecture is at the heart of Java's **robustness**, **security**, and **platform independence**, making it a powerful and versatile runtime environment.

In Java, **data types** specify the kind of data a variable can store. They are broadly categorized into **primitive** and **non-primitive** (or reference) data types.

---

## 1. Primitive Data Types

Primitive data types are predefined by the language and directly represent values. There are **eight** primitive data types in Java:

### A. Numeric Data Types

#### 1. Integer Types

These store whole numbers.

- **byte:**
  - Size: 1 byte (8 bits)
  - Range: -128 to 127
  - Example: `byte b = 10;`
- **short:**
  - Size: 2 bytes (16 bits)
  - Range: -32,768 to 32,767
  - Example: `short s = 1000;`
- **int:**
  - Size: 4 bytes (32 bits)
  - Range: -2,147,483,648 to 2,147,483,647
  - Example: `int i = 12345;`
- **long:**
  - Size: 8 bytes (64 bits)
  - Range: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
  - Example: `long l = 123456789L;`

#### 2. Floating-Point Types

These store decimal numbers.

- **float:**
  - Size: 4 bytes (32 bits)
  - Precision: Up to 7 decimal digits
  - Example: `float f = 3.14f;`
- **double:**
  - Size: 8 bytes (64 bits)
  - Precision: Up to 15 decimal digits
  - Example: `double d = 3.14159265359;`

## B. Non-Numeric Data Types

3. **char**
  - Size: 2 bytes (16 bits)
  - Represents a single Unicode character (e.g., letters, digits, symbols).
  - Range: 0 to 65,535 (Unicode values)
  - Example: `char c = 'A';`
4. **boolean**
  - Size: 1 bit (logical representation, not precisely defined in memory).
  - Represents logical values `true` or `false`.
  - Example: `boolean flag = true;`

---

## 2. Non-Primitive (Reference) Data Types

Non-primitive data types refer to objects or arrays that store references to memory locations.

### Examples:

1. **Strings**
  - A sequence of characters.
  - Example: `String name = "Java";`

## 2. Arrays

- o A collection of elements of the same type.
- o Example: `int[] numbers = {1, 2, 3};`

## 3. Classes and Objects

- o Custom-defined data types using the `class` keyword.
- o Example:

```
java
CopyEdit
class Person {
    String name;
    int age;
}
Person person = new Person();
```

## 4. Interfaces

- o Define abstract methods that classes can implement.
- o Example: `List<String> list = new ArrayList<>();`

## Comparison: Primitive vs. Non-Primitive

Feature	Primitive	Non-Primitive
Defined by	Language	Programmer (custom classes)
Memory storage	Store actual values	Store references to objects
Examples	<code>int, boolean, char</code>	<code>String, Array, Object</code>
Null allowed	No	Yes

## 3. Type Casting

Java supports converting one data type into another.

### A. Implicit (Widening) Casting

- Automatically converts smaller to larger types.
- Example:

```
java
CopyEdit
int num = 100;
double d = num; // int to double
```

### B. Explicit (Narrowing) Casting

- Requires manual conversion from larger to smaller types.
- Example:

```
java
CopyEdit
double d = 3.14;
int num = (int) d; // double to int
```

---

## 4. Default Values

If variables are not explicitly initialized, they have default values:

- Numeric types: 0
  - char: '\u0000'
  - boolean: false
  - Reference types: null
-

By understanding Java data types, developers can efficiently handle and manipulate data in programs, ensuring memory optimization and type safety.

# Core Java Notes By AJAY RAZZ

# Variables in Java

A **variable** in Java is a container for storing data values. It provides a name to a memory location that holds a value.

---

## 1. Types of Variables

Variables in Java are classified into three categories:

### A. Local Variables

- Declared inside a method, constructor, or block.
- Scope is limited to the block in which they are declared.
- Must be initialized before use.
- Example:

```
java
CopyEdit
public void display() {
    int num = 10; // Local variable
    System.out.println(num);
}
```

### B. Instance Variables

- Declared inside a class but outside methods, constructors, or blocks.
- Belong to an object and are not declared as `static`.
- Default values are assigned if not explicitly initialized.
- Example:

```
java
CopyEdit
class Person {
    String name; // Instance variable
    int age;     // Instance variable
}
```

### C. Static Variables (Class Variables)

- Declared with the `static` keyword inside a class but outside methods, constructors, or blocks.
- Shared among all instances of the class.
- Memory is allocated only once.
- Example:

```
java
CopyEdit
class Employee {
    static int count = 0; // Static variable
}
```

## 2. Declaration and Initialization

- Syntax:

```
java
CopyEdit
data_type variable_name = value;
```

- Example:

```
java
CopyEdit
int age = 25;      // Integer variable
double salary = 5000.75; // Floating-point variable
```

---

### 3. Variable Scope

- **Local Scope:** Limited to the method or block where declared.
  - **Instance Scope:** Exists as long as the object exists.
  - **Class Scope:** Exists as long as the program runs.
- 

# Operators in Java

Operators in Java are symbols used to perform operations on variables and values. They are classified into several types.

---

## 1. Arithmetic Operators

Used for performing basic mathematical operations.

Operator	Description	Example
+	Addition	$a + b$
-	Subtraction	$a - b$
*	Multiplication	$a * b$
/	Division	$a / b$
%	Modulus (Remainder)	$a \% b$

---

## 2. Relational (Comparison) Operators

Used to compare two values.

Operator	Description	Example
<code>==</code>	Equal to	<code>a == b</code>
<code>!=</code>	Not equal to	<code>a != b</code>
<code>&gt;</code>	Greater than	<code>a &gt; b</code>
<code>&lt;</code>	Less than	<code>a &lt; b</code>
<code>&gt;=</code>	Greater than or equal to	<code>a &gt;= b</code>
<code>&lt;=</code>	Less than or equal to	<code>a &lt;= b</code>

## 3. Logical Operators

Used to perform logical operations.

Operator	Description	Example
<code>&amp;&amp;</code>	Logical AND	<code>a &gt; 0 &amp;&amp; b &gt; 0</code>
<code>~</code>		<code>~</code>
<code>!</code>	Logical NOT	<code>!(a &gt; 0)</code>

## 4. Assignment Operators

Used to assign values to variables.

Operator	Description	Example
<code>=</code>	Assign	<code>a = 10</code>

Operator	Description	Example
<code>+=</code>	Add and assign	<code>a += 5</code>
<code>-=</code>	Subtract and assign	<code>a -= 5</code>
<code>*=</code>	Multiply and assign	<code>a *= 5</code>
<code>/=</code>	Divide and assign	<code>a /= 5</code>
<code>%=</code>	Modulus and assign	<code>a %= 5</code>

---

## 5. Bitwise Operators

Operate at the bit level.

Operator	Description	Example
<code>&amp;</code>	Bitwise AND	<code>a &amp; b</code>
<code>^</code>	Bitwise OR	
<code>^</code>	Bitwise XOR	<code>a ^ b</code>
<code>~</code>	Bitwise Complement	<code>~a</code>
<code>&lt;&lt;</code>	Left shift	<code>a &lt;&lt; 2</code>
<code>&gt;&gt;</code>	Right shift	<code>a &gt;&gt; 2</code>

---

## 6. Unary Operators

Operate on a single operand.

Operator	Description	Example
<code>+</code>	Unary plus	<code>+a</code>
<code>-</code>	Unary minus	<code>-a</code>

Operator	Description	Example
++	Increment (pre/post)	++a or a++
--	Decrement (pre/post)	--a or a--
!	Logical NOT	!a

---

## 7. Ternary Operator

- Shorthand for if-else conditions.
- Syntax: condition ? value\_if\_true : value\_if\_false
- Example:

```
java
CopyEdit
int result = (a > b) ? a : b;
```

---

## 8. instanceof Operator

- Tests whether an object is an instance of a specific class or subclass.
- Example:

```
java
CopyEdit
if (obj instanceof String) {
    System.out.println("It's a String!");
}
```

---

## 9. Type Cast Operator

- Converts one data type into another.

- Example:

```
java
CopyEdit
double d = 10.5;
int i = (int) d; // Explicit casting
```

---

## Summary

- **Variables** store data, and their type determines the kind of data they can hold.
- **Operators** perform computations, comparisons, and logical decisions in Java programs. Mastering both variables and operators is fundamental for effective Java programming.

Core Java Notes By AJAY RAZZ

# Control Flow in Java

Control flow statements in Java direct the order in which statements execute based on conditions or specific criteria. They include **if**, **else**, and **switch** statements, among others.

---

## 1. if Statement

The **if** statement is used to execute a block of code only if a specified condition evaluates to `true`.

### Syntax

```
java
CopyEdit
if (condition) {
    // Code to execute if the condition is true
}
```

### Example

```
java
CopyEdit
int age = 18;
if (age >= 18) {
    System.out.println("You are eligible to vote.");
}
```

---

## 2. if-else Statement

The **if-else** statement allows you to execute one block of code if the condition is `true` and another block if it is `false`.

## Syntax

```
java
CopyEdit
if (condition) {
    // Code to execute if the condition is true
} else {
    // Code to execute if the condition is false
}
```

## Example

```
java
CopyEdit
int number = 10;
if (number % 2 == 0) {
    System.out.println("The number is even.");
} else {
    System.out.println("The number is odd.");
}
```

## 3. if-else-if Ladder

The `if-else-if` ladder is used to check multiple conditions. Once a condition evaluates to `true`, the corresponding block is executed, and the rest are skipped.

## Syntax

```
java
CopyEdit
if (condition1) {
    // Code to execute if condition1 is true
} else if (condition2) {
    // Code to execute if condition2 is true
} else {
```

```
// Code to execute if none of the above conditions are true  
}
```

## Example

```
java  
CopyEdit  
int marks = 75;  
if (marks >= 90) {  
    System.out.println("Grade: A");  
} else if (marks >= 75) {  
    System.out.println("Grade: B");  
} else if (marks >= 50) {  
    System.out.println("Grade: C");  
} else {  
    System.out.println("Grade: F");  
}
```

## 4. Nested if Statement

A nested if is an if statement inside another if statement. It allows testing multiple conditions hierarchically.

### Syntax

```
java  
CopyEdit  
if (condition1) {  
    if (condition2) {  
        // Code to execute if both condition1 and condition2 are true  
    }  
}
```

## Example

```
java
```

```
CopyEdit
int age = 20;
int income = 50000;
if (age > 18) {
    if (income > 30000) {
        System.out.println("You are eligible for the loan.");
    }
}
```

---

## 5. switch Statement

The `switch` statement is used to execute one block of code out of many based on a variable's value. It is an alternative to a long `if-else-if` ladder.

### Syntax

```
java
CopyEdit
switch (expression) {
    case value1:
        // Code to execute if expression == value1
        break;
    case value2:
        // Code to execute if expression == value2
        break;
    // Add more cases as needed
    default:
        // Code to execute if none of the cases match
}
```

### Example

```
java
CopyEdit
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");  
}
```

---

## Important Notes

1. **Break Statement in switch:**
  - o The `break` statement prevents the execution from falling through to the next case.
  - o Without `break`, all subsequent cases after a match are executed.
2. **Default Case:**
  - o The `default` case is optional but is executed if no other cases match.
  - o It should usually be the last case.
3. **Expression Type for switch:**
  - o Supported data types for the `switch` expression:
    - **Primitive types:** `byte`, `short`, `int`, `char`
    - **Wrapper types:** `Byte`, `Short`, `Integer`, `Character`
    - **String** (from Java 7 onwards)
    - **Enums**

---

## Comparison: if-else vs. switch

Feature	if-else	switch
Use Case	For evaluating complex conditions.	For checking a single variable's value.
Data Types	Supports all data types and conditions.	Limited to specific data types.
Readability	Becomes harder to read for many conditions.	More readable for a large set of values.
Execution Speed	Slightly slower for many conditions.	Faster for large, discrete value checks.

---

## Summary

Control flow statements like `if`, `else`, and `switch` allow Java programs to make decisions and execute different blocks of code based on conditions. They provide flexibility to create dynamic and responsive programs.

Core Java Notes By AJAY RAZZ

# Loops in Java

Loops are used in Java to execute a block of code repeatedly as long as a specific condition is true. Java provides three types of loops: **for**, **while**, and **do-while**.

---

## 1. for Loop

The **for** loop is used when the number of iterations is known beforehand. It consists of three parts: initialization, condition, and increment/decrement.

### Syntax

```
java
CopyEdit
for (initialization; condition; update) {
    // Code to execute
}
```

### Example

```
java
CopyEdit
for (int i = 1; i <= 5; i++) {
    System.out.println("Iteration: " + i);
}
```

### How It Works

1. **Initialization:** Executes once, before the loop starts (e.g., `int i = 1`).
2. **Condition:** Evaluated before each iteration. If `true`, the loop body executes; if `false`, the loop exits.

- 
3. **Update:** Executes after each iteration (e.g., `i++`).

---

## 2. while Loop

The `while` loop is used when the number of iterations is not known beforehand, and the condition is evaluated before entering the loop.

### Syntax

```
java
CopyEdit
while (condition) {
    // Code to execute
}
```

### Example

```
java
CopyEdit
int i = 1;
while (i <= 5) {
    System.out.println("Iteration: " + i);
    i++;
}
```

### How It Works

1. The condition is checked before entering the loop.
  2. If the condition is `true`, the loop body executes.
  3. The condition is re-evaluated after each iteration.
-

### 3. do-while Loop

The `do-while` loop is similar to the `while` loop but guarantees that the loop body will execute at least once because the condition is evaluated after the loop body.

#### Syntax

```
java
CopyEdit
do {
    // Code to execute
} while (condition);
```

#### Example

```
java
CopyEdit
int i = 1;
do {
    System.out.println("Iteration: " + i);
    i++;
} while (i <= 5);
```

#### How It Works

1. The loop body executes once, even if the condition is `false`.
2. After the body executes, the condition is checked.
3. If the condition is `true`, the loop repeats; otherwise, it exits.

---

#### Comparison of Loops

<b>Feature</b>	<b>for Loop</b>	<b>while Loop</b>	<b>do-while Loop</b>
<b>When to Use</b>	When the number of iterations is known.	When the condition must be checked before execution.	When the loop body must execute at least once.
<b>Condition Check</b>	Before each iteration.	Before each iteration.	After the loop body executes.
<b>Guaranteed Execution</b>	No	No	Yes, at least once.

## Special Control Statements in Loops

### 1. **break Statement**

- Exits the loop immediately.
- Example:

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

### 2. **continue Statement**

- Skips the rest of the current iteration and proceeds to the next one.
- Example:

```
java
CopyEdit
for (int i = 1; i <= 5; i++) {
    if (i == 3) {
        continue;
    }
}
```

```
        System.out.println(i);
    }
// Output: 1 2 4 5
```

### 3. return Statement

- o Exits the loop and the method containing it.
- o Example:

```
java
CopyEdit
for (int i = 1; i <= 5; i++) {
    if (i == 3) {
        return;
    }
    System.out.println(i);
}
```

## Enhanced for Loop (for-each Loop)

Used to iterate over arrays or collections. It's simpler and avoids using an index.

### Syntax

```
java
CopyEdit
for (data_type element : array) {
    // Code to execute
}
```

### Example

```
java
CopyEdit
int[] numbers = {1, 2, 3, 4, 5};
for (int num : numbers) {
```

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

---

## Examples of Nested Loops

Loops can be nested within each other.

### Example

```
java
CopyEdit
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
        System.out.println("i = " + i + ", j = " + j);
    }
}
```

### Output:

```
css
CopyEdit
i = 1, j = 1
i = 1, j = 2
i = 1, j = 3
...
...
```

---

## Summary

- **for Loop:** Best for situations where the number of iterations is known.
- **while Loop:** Ideal for indefinite loops where the condition is checked before each iteration.
- **do-while Loop:** Ensures the loop body executes at least once.

By mastering these loops, you can effectively control the flow of repeated execution in your Java programs.

# Object-Oriented Programming (OOP) in Java

Object-Oriented Programming (OOP) is a programming paradigm that organizes code around **objects** and their interactions. Java is a **fully object-oriented language**, meaning everything revolves around objects, classes, and their relationships.

---

## Key Principles of OOP

### 1. Encapsulation

Encapsulation is the bundling of data (fields) and methods (functions) that operate on the data into a single unit, called a **class**. It restricts direct access to some components, enforcing controlled access through **getters** and **setters**.

- **Benefits:**
  - Protects the data from unintended interference.
  - Improves modularity and maintainability.
- **Example:**

```
java
CopyEdit
class Person {
    private String name; // Private field
    private int age;

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

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

```
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        if (age > 0) {
            this.age = age;
        }
    }
}
```

---

## 2. Inheritance

Inheritance allows one class (child or subclass) to inherit the properties and methods of another class (parent or superclass). This promotes **code reuse** and establishes a relationship between classes.

- **Syntax:**

```
java
CopyEdit
class Parent {
    void display() {
        System.out.println("This is the parent class.");
    }
}

class Child extends Parent {
    void show() {
        System.out.println("This is the child class.");
    }
}
```

- **Example:**

```
java
CopyEdit
Child obj = new Child();
obj.display(); // Access parent method
obj.show(); // Access child method
```

---

### 3. Polymorphism

Polymorphism allows methods to take on different forms based on the context, making programs more flexible and scalable. It has two types:

- **Compile-time Polymorphism (Method Overloading):**

- Same method name, different parameter lists.
- Example:

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

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

MathOperations math = new MathOperations();
System.out.println(math.add(5, 10)); // Output: 15
System.out.println(math.add(5, 10, 15)); // Output: 30
```

- **Runtime Polymorphism (Method Overriding):**

- A subclass provides a specific implementation for a method declared in the superclass.
- Example:

```
java
```

```

CopyEdit
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

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

Animal obj = new Dog();
obj.sound(); // Output: Dog barks

```

## 4. Abstraction

Abstraction focuses on exposing only essential details while hiding unnecessary complexity. It is achieved using **abstract classes** or **interfaces**.

- **Abstract Class:**

- Cannot be instantiated.
- May contain abstract methods (methods without a body) and concrete methods.
- Example:

```

java
CopyEdit
abstract class Shape {
    abstract void draw(); // Abstract method
}

class Circle extends Shape {
    @Override
    void draw() {
        System.out.println("Drawing Circle");
}

```

```
        }
    }

Shape obj = new Circle();
obj.draw(); // Output: Drawing Circle
```

- **Interface:**

- A blueprint for a class that enforces the implementation of its methods.
- Example:

```
java
CopyEdit
interface Animal {
    void sound();
}

class Cat implements Animal {
    @Override
    public void sound() {
        System.out.println("Cat meows");
    }
}

Animal obj = new Cat();
obj.sound(); // Output: Cat meows
```

---

## Key Concepts in OOP

### 1. Class

- A blueprint for objects.
- Contains fields (variables) and methods.
- Example:

```
java
```

```
CopyEdit
class Car {
    String model;
    int year;

    void display() {
        System.out.println(model + " - " + year);
    }
}
```

## 2. Object

- An instance of a class.
- Created using the `new` keyword.
- Example:

```
java
CopyEdit
Car car = new Car();
car.model = "Tesla";
car.year = 2022;
car.display(); // Output: Tesla - 2022
```

## 3. Constructor

- A special method used to initialize objects.
- Has the same name as the class and no return type.
- Example:

```
java
CopyEdit
class Car {
    String model;

    // Constructor
    Car(String model) {
```

```
        this.model = model;
    }

    void display() {
        System.out.println("Model: " + model);
    }
}

Car car = new Car("Ford");
car.display(); // Output: Model: Ford
```

## 4. Access Modifiers

- Define the visibility of fields and methods:
  - **private**: Accessible only within the class.
  - **default**: Accessible within the same package.
  - **protected**: Accessible within the same package and subclasses.
  - **public**: Accessible from anywhere.

## Advantages of OOP

1. **Code Reusability**: Inheritance allows the reuse of existing code.
2. **Modularity**: Encapsulation ensures components are independent and modular.
3. **Scalability**: Polymorphism makes it easier to scale and modify programs.
4. **Data Security**: Encapsulation restricts unauthorized access.
5. **Ease of Maintenance**: Abstraction and modularity simplify maintenance.

---

## Summary

OOP in Java revolves around four core principles: **Encapsulation**, **Inheritance**, **Polymorphism**, and **Abstraction**. These principles enable developers to write reusable, modular, and maintainable code, making Java one of the most robust and widely used programming languages.

# Core Java Notes By AJAY RAZZ

# Classes and Objects in Java

**Classes** and **Objects** are the fundamental building blocks of Object-Oriented Programming (OOP) in Java. They help to organize code in a modular, reusable, and logical way.

---

## What is a Class?

A **class** is a blueprint or template for creating objects. It defines:

- **Attributes** (fields or properties): Characteristics or data of the object.
- **Methods**: Actions or behaviors that the object can perform.

## Syntax of a Class

```
java
CopyEdit
class ClassName {
    // Fields (attributes)
    data_type fieldName;

    // Methods
    return_type methodName(parameters) {
        // Method body
    }
}
```

## Example

```
java
CopyEdit
```

```
class Car {  
    // Attributes (fields)  
    String brand;  
    int year;  
  
    // Method  
    void displayDetails() {  
        System.out.println("Brand: " + brand);  
        System.out.println("Year: " + year);  
    }  
}
```

---

## What is an Object?

An **object** is an instance of a class. It is created using the `new` keyword, and it represents a real-world entity that has:

- **State** (fields or attributes)
- **Behavior** (methods)

## Creating an Object

```
java  
CopyEdit  
ClassName objectName = new ClassName();
```

## Example

```
java  
CopyEdit  
public class Main {  
    public static void main(String[] args) {  
        Car car1 = new Car(); // Creating an object  
        car1.brand = "Tesla"; // Assigning values to fields  
        car1.year = 2022;
```

```
        car1.displayDetails(); // Calling a method
    }
}
```

## Output:

```
yaml
CopyEdit
Brand: Tesla
Year: 2022
```

---

## Components of a Class

### 1. Fields (Attributes):

- o Variables that hold data about the object.
- o Example:

```
java
CopyEdit
String name;
int age;
```

### 2. Methods:

- o Functions defined in the class that perform actions or behaviors.
- o Example:

```
java
CopyEdit
void display() {
    System.out.println("Displaying information.");
}
```

### 3. Constructors:

- o Special methods used to initialize objects.

- Example:

```
java
CopyEdit
class Person {
    String name;

    // Constructor
    Person(String name) {
        this.name = name;
    }
}
```

#### 4. Access Modifiers:

- Define the visibility of class members.
- **Private**: Accessible only within the class.
- **Public**: Accessible from anywhere.
- **Protected**: Accessible within the same package and subclasses.
- **Default**: Accessible within the same package.

## Key Features of Classes and Objects

### 1. Encapsulation

- Hiding implementation details and exposing only necessary functionality.
- Achieved by making fields `private` and providing `public` getter and setter methods.
- Example:

```
java
CopyEdit
class Student {
    private String name;
```

```
public String getName() {
    return name;
}

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

## 2. Methods with Parameters

- Classes can define methods that take parameters for dynamic functionality.
- Example:

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

public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println(calc.add(5, 10)); // Output: 15
    }
}
```

## 3. Multiple Objects

- A class can be used to create multiple objects with different states.
- Example:

```
java
CopyEdit
Car carl = new Car();
```

```
car1.brand = "Toyota";
car1.year = 2021;

Car car2 = new Car();
car2.brand = "Honda";
car2.year = 2022;

car1.displayDetails();
car2.displayDetails();
```

#### 4. Static Members

- **Static fields and methods** belong to the class rather than any specific object.
- Example:

```
java
CopyEdit
class Counter {
    static int count = 0;

    Counter() {
        count++;
    }
}

public class Main {
    public static void main(String[] args) {
        new Counter();
        new Counter();
        System.out.println(Counter.count); // Output: 2
    }
}
```

---

#### Constructors in Classes

##### What is a Constructor?

- A constructor is a special method used to initialize objects.
- It has the same name as the class and no return type.
- It is called automatically when an object is created.

## Types of Constructors

### 1. Default Constructor:

- A no-argument constructor provided by Java if no constructor is defined.
- Example:

```
java
CopyEdit
class Person {
    Person() {
        System.out.println("Default Constructor Called");
    }
}

public class Main {
    public static void main(String[] args) {
        Person p = new Person(); // Output: Default Constructor Called
    }
}
```

### 2. Parameterized Constructor:

- Used to pass arguments during object creation.
- Example:

```
java
CopyEdit
class Person {
    String name;

    Person(String name) {
        this.name = name;
    }
}
```

```

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

public class Main {
    public static void main(String[] args) {
        Person p = new Person("John");
        p.display(); // Output: Name: John
    }
}

```

---

## Difference Between Class and Object

Aspect	Class	Object
<b>Definition</b>	A blueprint or template for objects.	An instance of a class.
<b>Existence</b>	Logical entity (does not occupy memory).	Physical entity (occupies memory).
<b>Creation</b>	Defined using the <code>class</code> keyword.	Created using the <code>new</code> keyword.
<b>Example</b>	<code>class Car { ... }</code>	<code>Car car1 = new Car();</code>

## Advantages of Using Classes and Objects

- Code Reusability:** Once a class is defined, it can be reused to create multiple objects.
- Modularity:** Classes organize code into logical units, improving readability and maintenance.
- Abstraction:** Hides unnecessary details and exposes only the essential features.
- Encapsulation:** Protects data and restricts unauthorized access.
- Scalability:** Makes it easier to add new features or modify existing ones.

## Summary

- A **class** is a blueprint for creating **objects**.
- An **object** is an instance of a class, representing a real-world entity.
- Classes define attributes and behaviors, while objects hold specific values for those attributes and perform actions. By mastering classes and objects, you can design modular, reusable, and efficient programs in Java.

Core Java Notes By AJAY RAZZ

# Encapsulation in Java

**Encapsulation** is one of the four fundamental principles of Object-Oriented Programming (OOP). It is the process of bundling the **data** (fields) and **methods** (functions) that operate on the data into a single unit (class) and restricting direct access to them. Encapsulation ensures that an object's internal state is hidden from the outside and is accessible only through controlled mechanisms.

---

## Key Features of Encapsulation

1. **Data Hiding:**
    - The internal state of an object is not directly accessible from outside the class. Only specific methods (getters and setters) are exposed for accessing and modifying the data.
  2. **Controlled Access:**
    - Access to class variables is controlled through access modifiers like `private`, `public`, and `protected`.
  3. **Improved Security:**
    - Encapsulation prevents unauthorized or unintended changes to an object's state.
  4. **Ease of Maintenance:**
    - The implementation of a class can be changed without affecting the code that uses it, as long as the public methods remain the same.
- 

## How Encapsulation is Achieved in Java

Encapsulation is implemented in Java using the following steps:

1. Declare the fields of a class as `private` (or restrict access using modifiers).
2. Provide **public getter and setter methods** to access and update the values of private fields.
3. Enforce rules in setter methods to validate or control changes to the fields.

---

## Example of Encapsulation

```
java
CopyEdit
class BankAccount {
    // Private fields
    private String accountNumber;
    private double balance;

    // Constructor to initialize fields
    public BankAccount(String accountNumber, double initialBalance) {
        this.accountNumber = accountNumber;
        this.balance = initialBalance;
    }

    // Getter method for accountNumber
    public String getAccountNumber() {
        return accountNumber;
    }

    // Getter method for balance
    public double getBalance() {
        return balance;
    }

    // Setter method for balance with validation
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        } else {
            System.out.println("Deposit amount must be positive.");
        }
    }

    // Method to withdraw money with validation
    public void withdraw(double amount) {
```

```
        if (amount > 0 && amount <= balance) {
            balance -= amount;
        } else {
            System.out.println("Invalid withdrawal amount.");
        }
    }

public class Main {
    public static void main(String[] args) {
        // Creating an object of BankAccount
        BankAccount account = new BankAccount("123456", 5000);

        // Accessing balance through getter
        System.out.println("Account Number: " + account.getAccountNumber());
        System.out.println("Initial Balance: " + account.getBalance());

        // Depositing money
        account.deposit(2000);
        System.out.println("Balance after deposit: " + account.getBalance());

        // Withdrawing money
        account.withdraw(1500);
        System.out.println("Balance after withdrawal: " + account.getBalance());
    }
}
```

---

## Output of the Example

```
yaml
CopyEdit
Account Number: 123456
Initial Balance: 5000.0
Balance after deposit: 7000.0
Balance after withdrawal: 5500.0
```

---

## Advantages of Encapsulation

1. **Data Security:**
  - By making fields private, the data is protected from unauthorized or accidental changes.
2. **Improved Code Maintainability:**
  - Changes to the internal implementation of a class do not affect code that uses the class, as long as the public interface remains consistent.
3. **Validation Control:**
  - You can add validation logic in setter methods to enforce constraints on data.
  - Example: Ensuring a deposit amount is positive.
4. **Modularity:**
  - Encapsulation ensures that different parts of a program are independent, making it easier to test and debug.
5. **Readability:**
  - Encapsulation makes the code more organized and understandable.

## Access Modifiers and Encapsulation

In Java, access modifiers play a key role in achieving encapsulation:

Modifier	Scope
private	Accessible only within the same class.
default	Accessible within the same package.
protected	Accessible within the same package and subclasses in other packages.
public	Accessible from anywhere in the program.

By using these access modifiers, you can control the visibility and accessibility of a class's fields and methods.

---

## Example with Validation

```
java
CopyEdit
class Employee {
    private String name;
    private int age;

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

    // Setter for name with validation
    public void setName(String name) {
        if (name != null && !name.isEmpty()) {
            this.name = name;
        } else {
            System.out.println("Invalid name.");
        }
    }

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

    // Setter for age with validation
    public void setAge(int age) {
        if (age > 0) {
            this.age = age;
        } else {
            System.out.println("Age must be greater than 0.");
        }
    }
}

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

```
Employee emp = new Employee();

// Setting values using setters
emp.setName("John");
emp.setAge(30);

// Getting values using getters
System.out.println("Employee Name: " + emp.getName());
System.out.println("Employee Age: " + emp.getAge());
}

}
```

---

## Summary

1. **Encapsulation** ensures that the internal state of an object is hidden and can only be accessed or modified through controlled mechanisms (getters and setters).
2. It improves **data security**, **code maintainability**, and **validation control**.
3. Encapsulation is achieved by using **private fields**, **public methods**, and **access modifiers** to control visibility.

This approach makes the code modular, secure, and easier to maintain, aligning with the principles of Object-Oriented Programming.

# Inheritance in Java

**Inheritance** is a key feature of Object-Oriented Programming (OOP) that allows one class (child class) to inherit the properties (fields) and behaviors (methods) of another class (parent class). This promotes **code reuse** and helps to establish a hierarchical relationship between classes.

---

## Key Concepts of Inheritance

1. **Parent Class (Superclass):**
  - The class whose properties and methods are inherited by another class.
2. **Child Class (Subclass):**
  - The class that inherits the properties and methods of the parent class. It can also have additional properties and methods.
3. **extends Keyword:**
  - In Java, inheritance is achieved using the `extends` keyword.

## Types of Inheritance in Java

1. **Single Inheritance:**
  - A class inherits from one parent class.
  - Example: `Class B extends Class A.`
2. **Multilevel Inheritance:**
  - A class inherits from a class, and another class inherits from it.
  - Example: `Class C extends Class B extends Class A.`
3. **Hierarchical Inheritance:**
  - Multiple child classes inherit from a single parent class.
  - Example: `Class B extends Class A, Class C extends Class A.`
4. **Hybrid Inheritance (Not Supported Directly in Java):**

- A combination of multiple and other inheritance types. It is managed through **interfaces** in Java.

#### 5. Multiple Inheritance via Interfaces:

- A class can implement multiple interfaces to achieve multiple inheritance.
- 

## How to Implement Inheritance in Java

### Syntax

```
java
CopyEdit
class ParentClass {
    // Parent class properties and methods
}

class ChildClass extends ParentClass {
    // Additional properties and methods
}
```

### Example

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

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

```
public class Main {  
    public static void main(String[] args) {  
        Dog dog = new Dog();  
        dog.eat(); // Method from parent class  
        dog.bark(); // Method from child class  
    }  
}
```

## Output:

CopyEdit  
This animal eats food.  
The dog barks.

---

## Features of Inheritance

1. **Code Reusability:**
    - o Inheritance allows a class to reuse the fields and methods of an existing class.
  2. **Method Overriding:**
    - o A child class can provide its own implementation of a method defined in the parent class.
  3. **Transitive Nature:**
    - o If Class C inherits from Class B and Class B inherits from Class A, then Class C indirectly inherits from Class A.
  4. **Hierarchical Representation:**
    - o Classes can be arranged in a hierarchical structure based on inheritance relationships.
- 

## Access Control in Inheritance

Modifier	Same Class	Same Package	Subclass (Other Package)	Other Classes
private	Yes	No	No	No
default	Yes	Yes	No	No

### **Modifier Same Class Same Package Subclass (Other Package) Other Classes**

protected	Yes	Yes	Yes	No
public	Yes	Yes	Yes	Yes

## **Method Overriding**

**Method overriding** allows a child class to provide a specific implementation of a method that is already defined in its parent class.

### **Key Points about Overriding:**

1. The method in the child class must have the **same name, return type, and parameters** as the method in the parent class.
2. The overridden method in the child class cannot have a stricter access modifier than the parent class method.
3. The `@Override` annotation is used for better readability and error checking.

### **Example**

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

class Cat extends Animal {
    @Override
    void sound() {
        System.out.println("Cat meows.");
    }
}

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

```
        Animal animal = new Cat(); // Upcasting
        animal.sound(); // Output: Cat meows.
    }
}
```

---

## super Keyword

The `super` keyword is used to refer to the immediate parent class of the current object. It can be used to:

1. Access parent class methods or variables.
2. Call the parent class constructor.

### Example of `super` Keyword

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

class Dog extends Animal {
    @Override
    void sound() {
        super.sound(); // Call parent class method
        System.out.println("Dog barks.");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.sound();
    }
}
```

## **Output:**

```
go
CopyEdit
Animals make sounds.
Dog barks.
```

---

## **Constructors and Inheritance**

- Constructors are not inherited, but the parent class constructor can be invoked using `super()`.
- If the parent class does not have a default (no-argument) constructor, the child class must explicitly call a parameterized constructor of the parent.

### **Example**

```
java
CopyEdit
class Animal {
    Animal(String name) {
        System.out.println("Animal: " + name);
    }
}

class Dog extends Animal {
    Dog(String name) {
        super(name); // Calling parent class constructor
        System.out.println("Dog: " + name);
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog("Buddy");
    }
}
```

## **Output:**

```
makefile
CopyEdit
Animal: Buddy
Dog: Buddy
```

---

## **Advantages of Inheritance**

1. **Code Reusability:**
  - o Reduces duplication of code by reusing existing functionality.
2. **Extensibility:**
  - o Allows adding new features to an existing class without modifying it.
3. **Maintainability:**
  - o Easier to update code since changes in the parent class automatically propagate to the child classes.
4. **Polymorphism:**
  - o Inheritance enables runtime polymorphism, which enhances flexibility and scalability.

## **Disadvantages of Inheritance**

1. **Tight Coupling:**
  - o Child classes are tightly dependent on the parent class, making changes in the parent class potentially impact all child classes.
2. **Increased Complexity:**
  - o Deep inheritance hierarchies can make code difficult to understand and maintain.
3. **Reduced Flexibility:**
  - o Over-reliance on inheritance can lead to inflexible designs.

## Summary

- **Inheritance** allows classes to inherit fields and methods from other classes, enabling code reuse and extensibility.
- It is implemented using the `extends` keyword.
- **Method overriding** and the `super` keyword enhance inheritance functionality.
- While inheritance is powerful, it should be used judiciously to avoid overly complex hierarchies and tight coupling.

Core Java Notes By AJAY RAZZ

# Polymorphism in Java

**Polymorphism** is one of the four pillars of Object-Oriented Programming (OOP). The term means "many forms" and refers to the ability of an object to take on different forms or behaviors depending on the context. In Java, polymorphism allows the same method or operation to behave differently based on the object on which it is invoked.

---

## Types of Polymorphism in Java

1. **Compile-Time Polymorphism (Static Polymorphism):**
    - Achieved using **method overloading**.
    - The method to be executed is determined at compile time.
    - Example: Methods with the same name but different parameter lists.
  2. **Run-Time Polymorphism (Dynamic Polymorphism):**
    - Achieved using **method overriding**.
    - The method to be executed is determined at runtime based on the actual object.
    - Example: Subclass methods overriding superclass methods.
- 

### 1. Compile-Time Polymorphism (Method Overloading)

In method overloading, multiple methods in the same class share the same name but differ in:

- Number of parameters.
- Type of parameters.
- Order of parameters.

#### Example of Method Overloading

```
java
CopyEdit
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;
    }
}

public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println(calc.add(10, 20));           // Calls first method
        System.out.println(calc.add(10, 20, 30));       // Calls second method
        System.out.println(calc.add(10.5, 20.5));       // Calls third method
    }
}
```

### Output:

```
CopyEdit
30
60
31.0
```

---

## 2. Run-Time Polymorphism (Method Overriding)

In method overriding, a subclass provides a specific implementation of a method already defined in its superclass. The method to execute is determined dynamically at runtime based on the object being referred to.

### Key Points about Method Overriding:

1. The method in the child class must have the same **name**, **return type**, and **parameters** as the method in the parent class.
  2. The child class cannot override a method declared as `final` or `static`.
  3. The `@Override` annotation is used to explicitly indicate that a method is being overridden.
- 

### Example of Method Overriding

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

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

class Cat extends Animal {
    @Override
    void sound() {
        System.out.println("Cats meow.");
    }
}

public class Main {
```

```
public static void main(String[] args) {
    Animal myAnimal; // Reference of superclass

    myAnimal = new Dog(); // Object of Dog class
    myAnimal.sound(); // Calls Dog's sound method

    myAnimal = new Cat(); // Object of Cat class
    myAnimal.sound(); // Calls Cat's sound method
}
```

### Output:

```
CopyEdit
Dogs bark.
Cats meow.
```

## Polymorphism with Interfaces and Abstract Classes

Polymorphism is often used with **interfaces** and **abstract classes**. These allow defining general behaviors in a parent type while subclasses or implementing classes provide specific behavior.

### Example with an Interface

```
java
CopyEdit
interface Shape {
    void draw(); // Abstract method
}

class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a Circle.");
    }
}
```

```
}

class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a Rectangle.");
    }
}

public class Main {
    public static void main(String[] args) {
        Shape shape;

        shape = new Circle(); // Object of Circle
        shape.draw(); // Calls Circle's draw method

        shape = new Rectangle(); // Object of Rectangle
        shape.draw(); // Calls Rectangle's draw method
    }
}
```

### Output:

```
css
CopyEdit
Drawing a Circle.
Drawing a Rectangle.
```

---

## Advantages of Polymorphism

1. **Code Reusability:**
  - o A single method or interface can be used for a variety of implementations.
2. **Flexibility:**
  - o The behavior of a method can be altered dynamically depending on the object being used.
3. **Extensibility:**

- New classes can easily integrate into existing systems by overriding existing methods.
4. **Improved Readability and Maintainability:**
- Polymorphism reduces code duplication and improves code clarity.
- 

## Practical Use of Polymorphism

### Real-World Example

Consider a payment system where different payment methods (e.g., Credit Card, PayPal, or Cash) are handled using polymorphism.

```
java
CopyEdit
abstract class Payment {
    abstract void pay(double amount);
}

class CreditCardPayment extends Payment {
    @Override
    void pay(double amount) {
        System.out.println("Paid $" + amount + " using Credit Card.");
    }
}

class PayPalPayment extends Payment {
    @Override
    void pay(double amount) {
        System.out.println("Paid $" + amount + " using PayPal.");
    }
}

public class Main {
    public static void main(String[] args) {
        Payment payment;
```

```

        payment = new CreditCardPayment();
        payment.pay(100.0);

        payment = new PayPalPayment();
        payment.pay(150.0);
    }
}

```

### **Output:**

```

arduino
CopyEdit
Paid $100.0 using Credit Card.
Paid $150.0 using PayPal.

```

---

## **Key Differences Between Method Overloading and Method Overriding**

<b>Feature</b>	<b>Method Overloading</b>	<b>Method Overriding</b>
<b>Polymorphism Type</b>	Compile-time	Runtime
<b>Scope</b>	Same class	Subclass and superclass
<b>Method Signature</b>	Must differ (parameters or type)	Must be the same
<b>Access Modifier</b>	No restrictions	Cannot reduce visibility
<b>Static/Final Methods</b>	Allowed	Not allowed

---

## **Summary**

- **Polymorphism** enables objects to take on multiple forms, providing flexibility and scalability in OOP.
- **Compile-Time Polymorphism** is achieved through method overloading.
- **Run-Time Polymorphism** is achieved through method overriding.

- Polymorphism plays a crucial role in designing reusable and maintainable systems by leveraging abstract classes, interfaces, and inheritance.

4o

Core Java Notes By AJAY RAZZ

# Abstraction in Java

**Abstraction** is one of the four pillars of Object-Oriented Programming (OOP) and refers to the process of hiding implementation details while exposing only the essential features of an object. It focuses on "what an object does" rather than "how it does it." In Java, abstraction is achieved using **abstract classes** and **interfaces**.

---

## Why Abstraction?

1. **Simplifies Code:** Provides a clear separation between abstract concepts and their implementations.
2. **Enhances Flexibility:** Allows changes in the implementation without affecting the code that uses the abstraction.
3. **Promotes Reusability:** Encourages code reuse by defining standard behavior in a parent class or interface.
4. **Improves Security:** Hides internal implementation details and reduces external interference.

## How to Achieve Abstraction in Java

1. **Abstract Classes:**
  - A class declared using the `abstract` keyword.
  - Can have both **abstract methods** (methods without implementation) and **concrete methods** (methods with implementation).
  - Cannot be instantiated directly; it must be extended by a subclass.
2. **Interfaces:**
  - A reference type in Java, declared using the `interface` keyword.
  - Can only contain **abstract methods** (before Java 8), default methods, static methods, and constants.
  - A class must implement an interface to provide concrete definitions for its methods.

## Abstract Classes

### Syntax

```
java
CopyEdit
abstract class AbstractClass {
    // Abstract method
    abstract void abstractMethod();

    // Concrete method
    void concreteMethod() {
        System.out.println("This is a concrete method.");
    }
}
```

### Example

```
java
CopyEdit
abstract class Animal {
    abstract void sound(); // Abstract method

    void sleep() { // Concrete method
        System.out.println("This animal sleeps.");
    }
}

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

```
public class Main {  
    public static void main(String[] args) {  
        Animal myDog = new Dog(); // Upcasting  
        myDog.sound();  
        myDog.sleep();  
    }  
}
```

## Output:

```
CopyEdit  
Dog barks.  
This animal sleeps.
```

## Key Points about Abstract Classes

1. Abstract methods must be implemented by the subclass.
2. A class containing at least one abstract method must be declared abstract.
3. Abstract classes can have constructors, static methods, and member variables.

---

## Interfaces

### Syntax

```
java  
CopyEdit  
interface InterfaceName {  
    // Abstract method  
    void abstractMethod();  
}
```

### Example

```
java
CopyEdit
interface Shape {
    void draw(); // Abstract method
}

class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a Circle.");
    }
}

class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a Rectangle.");
    }
}

public class Main {
    public static void main(String[] args) {
        Shape circle = new Circle();
        Shape rectangle = new Rectangle();

        circle.draw();
        rectangle.draw();
    }
}
```

### Output:

```
css
CopyEdit
Drawing a Circle.
Drawing a Rectangle.
```

### Key Points about Interfaces

1. Methods in an interface are implicitly `public` and `abstract` (prior to Java 8).
  2. Variables in an interface are implicitly `public`, `static`, and `final`.
  3. A class can implement multiple interfaces, enabling **multiple inheritance**.
  4. Default and static methods were introduced in Java 8.
- 

## Abstract Classes vs Interfaces

Feature	Abstract Class	Interface
<b>Purpose</b>	Partial abstraction	Complete abstraction
<b>Methods</b>	Can have abstract and concrete methods	Only abstract methods (default and static methods allowed since Java 8)
<b>Variables</b>	Can have instance variables	Only constants ( <code>public static final</code> )
<b>Inheritance</b>	Can extend one abstract class	Can implement multiple interfaces
<b>Constructors</b>	Can have constructors	Cannot have constructors
<b>Default Implementation</b>	Supported for concrete methods	Supported via default methods (Java 8+)

---

## Example Using Both Abstract Class and Interface

```
java
CopyEdit
abstract class Animal {
    abstract void sound();

    void eat() {
        System.out.println("This animal eats food.");
    }
}

interface Pet {
    void play();
```

```
}

class Dog extends Animal implements Pet {
    @Override
    void sound() {
        System.out.println("Dog barks.");
    }

    @Override
    public void play() {
        System.out.println("Dog loves to play fetch.");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.sound();
        dog.eat();
        dog.play();
    }
}
```

### Output:

```
css
CopyEdit
Dog barks.
This animal eats food.
Dog loves to play fetch.
```

---

## Advantages of Abstraction

- Reduces Complexity:** Hides unnecessary details and shows only essential features.
- Improves Code Maintenance:** Implementation details can be changed without affecting the code using the abstraction.
- Facilitates Code Reuse:** Common behavior is defined in a base class or interface.

- 
- 4. **Encourages Loose Coupling:** Decouples the definition and implementation of functionality.
- 

## Disadvantages of Abstraction

- 1. **Design Complexity:** Designing abstract classes or interfaces may require additional effort.
  - 2. **Performance Overhead:** Abstract layers can introduce slight runtime overhead due to dynamic method calls.
  - 3. **Limited by Constraints:** Abstract classes can only be extended by one subclass (Java does not support multiple inheritance with classes).
- 

## Real-World Example of Abstraction

Imagine a payment system with multiple payment methods (Credit Card, PayPal, Cash). Abstraction can define a common `Payment` interface while hiding the specific details of each payment method.

```
java
CopyEdit
interface Payment {
    void pay(double amount);
}

class CreditCardPayment implements Payment {
    @Override
    public void pay(double amount) {
        System.out.println("Paid $" + amount + " using Credit Card.");
    }
}

class PayPalPayment implements Payment {
    @Override
    public void pay(double amount) {
        System.out.println("Paid $" + amount + " using PayPal.");
    }
}
```

```
        }

    }

public class Main {
    public static void main(String[] args) {
        Payment payment;

        payment = new CreditCardPayment();
        payment.pay(100.0);

        payment = new PayPalPayment();
        payment.pay(150.0);
    }
}
```

### Output:

```
arduino
CopyEdit
Paid $100.0 using Credit Card.
Paid $150.0 using PayPal.
```

### Conclusion

- Abstraction in Java helps to focus on what an object does, not how it does it.
- It can be achieved through **abstract classes** and **interfaces**.
- Abstract classes provide partial abstraction, whereas interfaces provide complete abstraction.
- Proper use of abstraction results in more modular, reusable, and maintainable code.

# Packages and Interfaces in Java

---

## Packages in Java

**Packages** in Java are used to group related classes, interfaces, and sub-packages. They provide a namespace management mechanism and help in organizing code systematically.

### Why Use Packages?

1. **Organize Code:** Makes the structure more manageable by grouping similar classes together.
  2. **Avoid Name Conflicts:** Classes in different packages can have the same name.
  3. **Access Control:** Provides access levels to classes and methods within the package.
  4. **Reusable Code:** Classes in a package can be reused across multiple projects.
- 

## Types of Packages

1. **Built-in Packages:** Provided by Java, such as `java.util`, `java.io`, `java.net`, etc.
  2. **User-defined Packages:** Created by the programmer to organize their custom classes and interfaces.
- 

## Creating a Package

A package is declared at the beginning of a Java file using the `package` keyword.

### Syntax:

```
java
CopyEdit
package packageName;
```

### **Example:**

```
java
CopyEdit
package mypackage;

public class MyClass {
    public void display() {
        System.out.println("This is a class in a package.");
    }
}
```

## **Using a Package**

To use a class from a package, you can:

### **1. Import the Package:**

```
java
CopyEdit
import mypackage.MyClass;
```

### **2. Use the Fully Qualified Name:**

```
java
CopyEdit
mypackage.MyClass obj = new mypackage.MyClass();
```

### **Example:**

```
java
CopyEdit
```

```
import mypackage.MyClass;

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

---

## Access Modifiers and Packages

1. **Public:** Accessible from any package.
2. **Protected:** Accessible within the same package and subclasses in other packages.
3. **Default:** Accessible only within the same package.
4. **Private:** Not accessible outside the class.

## Interfaces in Java

An **interface** in Java is a reference type, similar to a class, that can contain only **abstract methods** (before Java 8) or a combination of abstract methods, default methods, and static methods (from Java 8 onward). It provides a blueprint for classes.

---

## Why Use Interfaces?

1. **Achieve Abstraction:** Provides complete abstraction as it hides implementation details.
  2. **Multiple Inheritance:** A class can implement multiple interfaces.
  3. **Standardization:** Defines a contract that implementing classes must follow.
-

## Defining an Interface

An interface is declared using the `interface` keyword.

### Syntax:

```
java
CopyEdit
interface InterfaceName {
    void method1(); // Abstract method
    void method2();
}
```

## Implementing an Interface

A class implements an interface using the `implements` keyword.

### Example:

```
java
CopyEdit
interface Animal {
    void sound(); // Abstract method
}

class Dog implements Animal {
    @Override
    public void sound() {
        System.out.println("Dog barks.");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog();
        myDog.sound();
```

```
    }  
}
```

### Output:

```
CopyEdit  
Dog barks.
```

---

### Key Features of Interfaces

1. **Abstract Methods:** Before Java 8, all methods were implicitly abstract.
2. **Default Methods:** Introduced in Java 8, allows methods with implementation in interfaces.

```
java  
CopyEdit  
default void greet() {  
    System.out.println("Hello!");  
}
```

3. **Static Methods:** Also introduced in Java 8, can be called without creating an object.

```
java  
CopyEdit  
static void info() {  
    System.out.println("Static method in an interface.");  
}
```

4. **Constants:** Variables in an interface are implicitly public static final.
- 

### Differences Between Abstract Classes and Interfaces

Feature	Abstract Class	Interface
<b>Abstraction Level</b>	Partial abstraction	Complete abstraction
<b>Methods</b>	Can have abstract and concrete methods	Can have abstract, default, and static methods
<b>Variables</b>	Instance and static variables allowed	Only constants ( <code>public static final</code> )
<b>Inheritance</b>	Can extend one abstract class	Can implement multiple interfaces
<b>Constructors</b>	Can have constructors	Cannot have constructors

---

## Packages with Interfaces

Interfaces are often used in packages to define reusable contracts that multiple classes in the package or other packages can implement.

### Example:

```
java
CopyEdit
// File: mypackage/Shape.java
package mypackage;

public interface Shape {
    void draw();
}

// File: mypackage/Circle.java
package mypackage;

public class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a Circle.");
    }
}

// File: mypackage/Rectangle.java
```

```
package mypackage;

public class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a Rectangle.");
    }
}

// File: Main.java
import mypackage.*;

public class Main {
    public static void main(String[] args) {
        Shape circle = new Circle();
        Shape rectangle = new Rectangle();

        circle.draw();
        rectangle.draw();
    }
}
```

## Output:

```
css
CopyEdit
Drawing a Circle.
Drawing a Rectangle.
```

---

## Conclusion

- **Packages** group related classes, interfaces, and sub-packages to organize code and prevent naming conflicts.
- **Interfaces** define contracts for classes, enabling abstraction and multiple inheritance.
- Together, packages and interfaces enhance modularity, maintainability, and reusability in Java programs.

# Packages in Java: Creating and Using

A **package** in Java is a namespace that organizes classes and interfaces. It is used to group related classes and interfaces to avoid name conflicts and to provide easier access control. Additionally, packages help improve the maintainability of the code.

---

## Why Use Packages?

1. **Organize Classes:** Group related classes and interfaces.
2. **Avoid Name Conflicts:** Two classes with the same name can exist in different packages.
3. **Access Protection:** Control access levels using access modifiers.
4. **Reusable Code:** Classes in a package can be reused in other programs.
5. **Ease of Maintenance:** Modularization of code makes it easier to maintain.

## Types of Packages

1. **Built-in Packages:** Java provides predefined packages like `java.util`, `java.io`, `java.net`, etc.
2. **User-defined Packages:** Developers can create their own packages.

## Creating a Package

1. **Declare the Package:** Use the `package` keyword at the very beginning of the Java file.

```
java  
CopyEdit
```

```
package mypackage;
```

2. **Define Classes:** Add your classes and interfaces after the package declaration.

```
java
CopyEdit
package mypackage;

public class MyClass {
    public void displayMessage() {
        System.out.println("This is my package!");
    }
}
```

3. **Compile the Class:** When compiling, the package structure is reflected in the file system.

```
bash
CopyEdit
javac -d . MyClass.java
```

This creates a directory structure (`mypackage`) that contains the compiled `.class` file.

---

## Using a Package

1. **Import the Package:** Use the `import` statement to include the package in other programs.

```
java
CopyEdit
import mypackage.MyClass;

public class Test {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        obj.displayMessage();
    }
}
```

```
    }  
}
```

## 2. Compile and Run:

- o Compile:

```
bash  
CopyEdit  
javac Test.java
```

- o Run:

```
bash  
CopyEdit  
java Test
```

## 3. Use Fully Qualified Name (Optional): Instead of importing, you can use the full name of the class.

```
java  
CopyEdit  
public class Test {  
    public static void main(String[] args) {  
        mypackage.MyClass obj = new mypackage.MyClass();  
        obj.displayMessage();  
    }  
}
```

---

## Key Notes

- **Access Control:**
  - o Classes in the same package can access each other's **protected** and **default** members.
  - o Public members are accessible from anywhere.
- **Nested Packages:** Packages can have subpackages, e.g., `mypackage.subpackage`.

---

## Example

### Create a package:

```
java
CopyEdit
package utilities;

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

### Use the package:

```
java
CopyEdit
import utilities.Calculator;

public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println("Sum: " + calc.add(5, 10));
    }
}
```

### Compile and run:

```
bash
CopyEdit
javac -d . Calculator.java
javac Main.java
java Main
```

# Access Modifiers in Java

Access modifiers in Java define the **visibility** or **accessibility** of classes, methods, and variables. They control where a member (field, method, or constructor) can be accessed from, and they are essential for implementing encapsulation.

---

## Types of Access Modifiers

Java provides four levels of access control:

Modifier	Same Class	Same Package	Subclass (Different Package)	Everywhere
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
(default)	✓	✓	✗	✗
private	✓	✗	✗	✗

---

### 1. Public

- Description:** Members declared as `public` are accessible from **anywhere**.
- Use Case:** Use `public` when you want a member to be accessible globally.
- Example:**

```
java
CopyEdit
public class MyClass {
    public void display() {
        System.out.println("Public method can be accessed anywhere.");
```

```
    }

}

public class Test {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        obj.display(); // Accessible
    }
}
```

---

## 2. Private

- **Description:** Members declared as `private` are accessible **only within the same class**.
- **Use Case:** Use `private` to implement encapsulation and hide details from other classes.
- **Example:**

```
java
CopyEdit
public class MyClass {
    private String message = "Private message";

    private void display() {
        System.out.println(message);
    }

    public void accessPrivate() {
        display(); // Accessing private method within the same class
    }
}

public class Test {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        // obj.display(); // Error: display() is private
        obj.accessPrivate(); // Works because access is through a public method
    }
}
```

```
}
```

---

### 3. Protected

- **Description:** Members declared as `protected` are accessible:
  - Within the **same package**.
  - In subclasses (even if the subclass is in a different package).
- **Use Case:** Use `protected` to allow controlled access to subclasses while keeping it hidden from other classes outside the package.
- **Example:**

```
java
CopyEdit
package mypackage;

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

// Subclass in the same package
package mypackage;

public class Child extends Parent {
    public void show() {
        display(); // Accessible in subclass
    }
}

// Subclass in a different package
package anotherpackage;

import mypackage.Parent;

public class SubChild extends Parent {
    public void show() {
```

```
        display(); // Accessible in subclass (different package)
    }
}

// Non-subclass
package anotherpackage;

import mypackage.Parent;

public class Test {
    public static void main(String[] args) {
        Parent p = new Parent();
        // p.display(); // Error: Protected is not accessible in a non-subclass
    }
}
```

---

#### 4. Default (No Modifier)

- **Description:** Members with no modifier are accessible **only within the same package**.
- **Use Case:** Use default access when the class or member should only be accessible to other classes in the same package.
- **Example:**

```
java
CopyEdit
class MyClass {
    void display() {
        System.out.println("Default access method");
    }
}

public class Test {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        obj.display(); // Accessible because Test and MyClass are in the same package
    }
}
```

---

## Key Points

### 1. Class Access:

- o Classes can only be `public` or `default`.
- o A `public` class is accessible everywhere; a `default` class is accessible only within the same package.

### 2. Method and Variable Access:

- o Use `private` for encapsulation.
- o Use `public` for global access.
- o Use `protected` to allow access to subclasses.

### 3. Best Practices:

- o Keep fields `private` and provide access via getter/setter methods for encapsulation.
- o Use the least permissive modifier needed for better security and maintainability.

By understanding and using access modifiers effectively, you can enforce access restrictions and ensure your code is secure and modular.

# Interfaces in Java: Defining and Implementing

An **interface** in Java is a blueprint of a class. It contains abstract methods (methods without a body) and constants. Interfaces are used to achieve abstraction and multiple inheritance in Java.

---

## Why Use Interfaces?

1. **Achieve Abstraction:** Interfaces define what a class should do without specifying how.
2. **Multiple Inheritance:** A class can implement multiple interfaces, overcoming Java's single inheritance limitation.
3. **Standardization:** Interfaces can define a contract that multiple classes can follow.

## Defining an Interface

An interface is defined using the `interface` keyword. All methods in an interface are **implicitly abstract and public**, and all fields are **implicitly public, static, and final**.

### Syntax:

```
java
CopyEdit
interface InterfaceName {
    // Abstract methods
    void method1();
    void method2();

    // Default methods (since Java 8)
    default void method3() {
        System.out.println("Default method in an interface");
    }
}
```

```
}

// Static methods (since Java 8)
static void method4() {
    System.out.println("Static method in an interface");
}
}
```

---

## Implementing an Interface

A class uses the `implements` keyword to implement an interface. The class must provide concrete implementations for all abstract methods in the interface.

### Example:

```
java
CopyEdit
interface Animal {
    void eat();
    void sleep();
}

class Dog implements Animal {
    @Override
    public void eat() {
        System.out.println("Dog eats food.");
    }

    @Override
    public void sleep() {
        System.out.println("Dog sleeps.");
    }
}

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

```
Dog dog = new Dog();
dog.eat();
dog.sleep();
}
}
```

### Output:

```
CopyEdit
Dog eats food.
Dog sleeps.
```

## Key Features of Interfaces

1. **All methods are implicitly public and abstract:**
  - o You can omit the `public` and `abstract` keywords in method declarations.
2. **Constants:**
  - o All variables in an interface are implicitly `public`, `static`, and `final`.
3. **Default and Static Methods** (Introduced in Java 8):
  - o **Default Methods:** Provide method implementation in an interface.

```
java
CopyEdit
interface Vehicle {
    void move();

    default void start() {
        System.out.println("Vehicle is starting");
    }
}
```

- o **Static Methods:** Belong to the interface and can be called without an object.

```
java
```

```
CopyEdit
interface Vehicle {
    static void maintenance() {
        System.out.println("Vehicle maintenance");
    }
}
Vehicle.maintenance();
```

4. **Multiple Interfaces:** A class can implement multiple interfaces, which allows for multiple inheritance.

```
java
CopyEdit
interface InterfaceA {
    void methodA();
}

interface InterfaceB {
    void methodB();
}

class MyClass implements InterfaceA, InterfaceB {
    @Override
    public void methodA() {
        System.out.println("MethodA implemented");
    }

    @Override
    public void methodB() {
        System.out.println("MethodB implemented");
    }
}
```

---

## Extending Interfaces

Interfaces can extend other interfaces, allowing for hierarchical design.

```

java
CopyEdit
interface ParentInterface {
    void method1();
}

interface ChildInterface extends ParentInterface {
    void method2();
}

class Implementation implements ChildInterface {
    @Override
    public void method1() {
        System.out.println("Method1 implemented");
    }

    @Override
    public void method2() {
        System.out.println("Method2 implemented");
    }
}

```

## Key Differences Between Interfaces and Abstract Classes

Feature	Interface	Abstract Class
<b>Methods</b>	All methods are abstract (except default/static). Can have both abstract and concrete methods.	
<b>Fields</b>	Only <code>public, static, final</code> .	Can have instance variables.
<b>Inheritance</b>	A class can implement multiple interfaces.	A class can extend only one abstract class.
<b>Constructors</b>	Interfaces cannot have constructors.	Abstract classes can have constructors.

---

## Practical Example

**Scenario:** A payment system with multiple payment methods.

```
java
CopyEdit
interface Payment {
    void pay(double amount);
}

class CreditCardPayment implements Payment {
    @Override
    public void pay(double amount) {
        System.out.println("Paid $" + amount + " using Credit Card.");
    }
}

class PayPalPayment implements Payment {
    @Override
    public void pay(double amount) {
        System.out.println("Paid $" + amount + " using PayPal.");
    }
}

public class PaymentTest {
    public static void main(String[] args) {
        Payment payment1 = new CreditCardPayment();
        Payment payment2 = new PayPalPayment();

        payment1.pay(100.50);
        payment2.pay(75.25);
    }
}
```

### **Output:**

```
arduino
CopyEdit
Paid $100.5 using Credit Card.
Paid $75.25 using PayPal.
```

---

## **Conclusion**

Interfaces in Java are a powerful way to enforce a contract for classes to implement specific methods. They are essential for abstraction, standardization, and achieving multiple inheritance. By understanding and leveraging interfaces, you can create flexible, reusable, and maintainable code.

Core Java Notes By AJAY RAZZ

# Exception Handling in Java

**Exception Handling** in Java is a mechanism to handle runtime errors so that normal application flow is maintained. It helps in gracefully recovering from unexpected scenarios like invalid input, file not found, database errors, etc.

---

## What is an Exception?

An **exception** is an unwanted or unexpected event that occurs during program execution and disrupts its normal flow.

## Types of Exceptions:

1. **Checked Exceptions:**
  - o Exceptions checked at compile-time.
  - o Example: `IOException`, `SQLException`.
2. **Unchecked Exceptions:**
  - o Exceptions that occur at runtime and are not checked at compile time.
  - o Example: `NullPointerException`, `ArrayIndexOutOfBoundsException`.
3. **Errors:**
  - o Serious issues that the application should not try to catch (e.g., `OutOfMemoryError`, `StackOverflowError`).

---

## Hierarchy of Exception Classes

In Java, all exceptions are subclasses of the `Throwable` class:

php  
CopyEdit

```
Throwable
└─ Error
   └─ Exception
      └─ RuntimeException
         └─ Other Checked Exceptions
```

---

## Exception Handling Keywords

Java provides five keywords for exception handling:

1. **try**: Defines a block of code to test for exceptions.
2. **catch**: Defines a block of code to handle the exception.
3. **finally**: Defines a block that executes after try-catch, regardless of the exception being handled or not.
4. **throw**: Used to explicitly throw an exception.
5. **throws**: Declares exceptions that a method can throw.

## Basic Structure of Exception Handling

```
java
CopyEdit
try {
    // Code that may throw an exception
} catch (ExceptionType e) {
    // Code to handle the exception
} finally {
    // Code that will always execute (optional)
}
```

---

## Example 1: Handling an Exception

```
java
CopyEdit
public class ExceptionExample {
    public static void main(String[] args) {
        try {
            int data = 10 / 0; // This will throw ArithmeticException
        } catch (ArithmetricException e) {
            System.out.println("Cannot divide by zero: " + e.getMessage());
        } finally {
            System.out.println("Execution complete.");
        }
    }
}
```

### Output:

```
csharp
CopyEdit
Cannot divide by zero: / by zero
Execution complete.
```

### Example 2: Multiple Catch Blocks

```
java
CopyEdit
public class MultipleCatchExample {
    public static void main(String[] args) {
        try {
            int[] arr = new int[5];
            arr[10] = 30 / 0;
        } catch (ArithmetricException e) {
            System.out.println("Arithmetic Exception occurred: " + e.getMessage());
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array Index Out of Bounds: " + e.getMessage());
        } catch (Exception e) {
            System.out.println("Some other exception occurred: " + e.getMessage());
        }
    }
}
```

```
    }  
}
```

---

## Throwing Exceptions

You can explicitly throw exceptions using the `throw` keyword.

### Example:

```
java  
CopyEdit  
public class ThrowExample {  
    public static void validateAge(int age) {  
        if (age < 18) {  
            throw new IllegalArgumentException("Age must be 18 or above.");  
        }  
    }  
    public static void main(String[] args) {  
        validateAge(15);  
    }  
}
```

### Output:

```
arduino  
CopyEdit  
Exception in thread "main" java.lang.IllegalArgumentException: Age must be 18 or above.
```

---

## Using `throws` Keyword

The `throws` keyword is used to declare exceptions that a method might throw.

### **Example:**

```
java
CopyEdit
import java.io.*;

public class ThrowsExample {
    public static void readFile() throws IOException {
        FileReader file = new FileReader("nonexistent.txt");
        file.close();
    }

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

### **Finally Block**

The `finally` block always executes, whether an exception is handled or not.

### **Example:**

```
java
CopyEdit
public class FinallyExample {
    public static void main(String[] args) {
        try {
            int data = 10 / 0;
        } catch (ArithmetricException e) {
            System.out.println("Exception caught.");
        } finally {
```

```
        System.out.println("Finally block executed.");
    }
}
}
```

### Output:

```
php
CopyEdit
Exception caught.
Finally block executed.
```

## Custom Exceptions

You can create your own exception classes by extending the `Exception` class.

### Example:

```
java
CopyEdit
class MyException extends Exception {
    public MyException(String message) {
        super(message);
    }
}

public class CustomExceptionExample {
    public static void main(String[] args) {
        try {
            throw new MyException("Custom exception occurred.");
        } catch (MyException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

## **Output:**

```
vbnnet
CopyEdit
Custom exception occurred.
```

---

## **Best Practices for Exception Handling**

1. **Catch Specific Exceptions:** Avoid using a generic `Exception` unless necessary.
2. **Avoid Empty Catch Blocks:** Always handle or log the exception.
3. **Use `finally` for Cleanup:** For resources like files or database connections.
4. **Custom Exceptions:** Use custom exceptions for meaningful error reporting.
5. **Don't Swallow Exceptions:** Ensure exceptions are not silently ignored.

## **Summary**

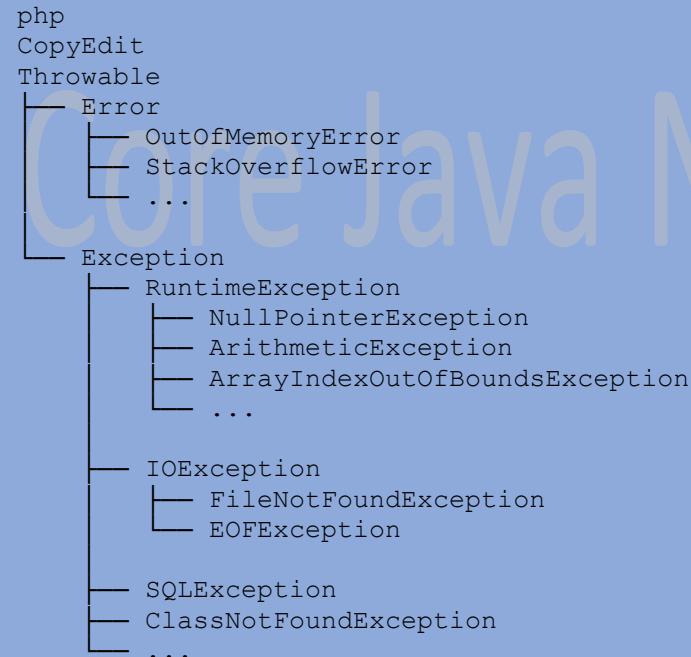
Exception handling in Java ensures that runtime errors do not crash the application. By using `try`, `catch`, `finally`, `throw`, and `throws`, you can write robust and error-resilient programs.

# Exception Hierarchy in Java

In Java, all exceptions and errors are subclasses of the `Throwable` class. The `Throwable` class is the root of the exception hierarchy, and it has two main branches: `Error` and `Exception`.

---

## Structure of the Exception Hierarchy



## Key Branches of the Hierarchy

## **1. Throwable**

- The base class for all exceptions and errors.
- It has two direct subclasses:
  - **Error**
  - **Exception**

## **2. Error**

- Represents serious problems that an application cannot typically recover from.
- Examples:
  - **OutOfMemoryError**: Indicates insufficient memory.
  - **StackOverflowError**: Indicates a stack overflow due to deep recursion.
  - **VirtualMachineError**: Indicates problems with the Java Virtual Machine.

Errors are not meant to be caught or handled in typical Java applications.

## **3. Exception**

- Represents conditions that an application might want to catch and handle.
- Subdivided into:
  - **Checked Exceptions**: Exceptions that are checked at compile-time.
  - **Unchecked Exceptions**: Runtime exceptions that are checked at runtime.

---

## **Checked Exceptions**

- **Definition**: Exceptions that must be declared in a method's `throws` clause or handled using `try-catch`.
- **Examples**:

- **IOException**: Input/output operations failure.
- **SQLException**: Issues with database operations.
- **ClassNotFoundException**: Class cannot be found.

#### Example:

```
java
CopyEdit
import java.io.*;

public class CheckedExceptionExample {
    public static void main(String[] args) {
        try {
            FileReader file = new FileReader("nonexistent.txt");
        } catch (FileNotFoundException e) {
            System.out.println("File not found: " + e.getMessage());
        }
    }
}
```

## Unchecked Exceptions

- **Definition**: Exceptions that are not checked at compile-time. They occur during runtime.
- **Subclass**: **RuntimeException**
- **Examples**:
  - **NullPointerException**: Null reference access.
  - **ArithmaticException**: Invalid arithmetic operation (e.g., division by zero).
  - **ArrayIndexOutOfBoundsException**: Accessing an array with an invalid index.

#### Example:

```
java
CopyEdit
public class UncheckedExceptionExample {
```

```

public static void main(String[] args) {
    int[] arr = {1, 2, 3};
    System.out.println(arr[5]); // ArrayIndexOutOfBoundsException
}

```

---

## Differences Between Checked and Unchecked Exceptions

Feature	Checked Exceptions	Unchecked Exceptions
<b>Compile-time Check</b>	Yes	No
<b>Class</b>	Subclasses of <code>Exception</code> (excluding <code>RuntimeException</code> )	Subclasses of <code>RuntimeException</code>
<b>Handling</b>	Must be explicitly handled or declared using <code>throws</code> .	Can be optionally handled.
<b>Examples</b>	<code>IOException</code> , <code>SQLException</code>	<code>NullPointerException</code> , <code>ArithmaticException</code>

## Errors vs Exceptions

Feature	Errors	Exceptions
<b>Definition</b>	Represent critical problems beyond application control.	Represent recoverable conditions.
<b>Handling</b>	Not meant to be caught or handled.	Should be handled using <code>try-catch</code> .
<b>Examples</b>	<code>OutOfMemoryError</code> , <code>StackOverflowError</code>	<code>IOException</code> , <code>NullPointerException</code>

## Practical Example: Handling Different Exceptions

```

java
CopyEdit
public class ExceptionHierarchyExample {
    public static void main(String[] args) {
        try {

```

```
        int[] arr = new int[5];
        arr[10] = 30 / 0; // Will throw ArithmeticException first
    } catch (ArithmaticException e) {
        System.out.println("Arithmatic Exception: " + e.getMessage());
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Array Index Out of Bounds: " + e.getMessage());
    } catch (Exception e) {
        System.out.println("General Exception: " + e.getMessage());
    }
}
}
```

## Output:

```
csharp
CopyEdit
Arithmatic Exception: / by zero
```

## Summary

1. **Throwable** is the root of the exception hierarchy.
2. **Error** represents unrecoverable issues like `OutOfMemoryError`.
3. **Exception** is for recoverable conditions and includes:
  - o **Checked Exceptions**: `IOException`, `SQLException`.
  - o **Unchecked Exceptions**: `NullPointerException`, `ArithmaticException`.
4. Use `try-catch` to handle exceptions gracefully.
5. Properly handling exceptions ensures a robust and reliable application.

# Try-Catch Blocks in Java

A **try-catch block** in Java is used to handle exceptions that may occur during the execution of a program. This ensures that the application does not crash when an exception is encountered and allows for graceful recovery.

---

## Structure of Try-Catch Block

**Syntax:**

```
java
CopyEdit
try {
    // Code that might throw an exception
} catch (ExceptionType1 e1) {
    // Code to handle ExceptionType1
} catch (ExceptionType2 e2) {
    // Code to handle ExceptionType2
} finally {
    // Optional block that will always execute
}
```

---

## How It Works

1. **try Block:**
  - o Contains code that may throw an exception.
  - o If no exception is thrown, the `catch` block is skipped, and the program continues normally.
2. **catch Block:**
  - o Catches and handles a specific type of exception.
  - o Multiple `catch` blocks can be used to handle different exceptions.

### 3. **`finally`** Block (Optional):

- o Contains cleanup code (e.g., closing files, releasing resources).
  - o Executes whether or not an exception occurs.
- 

## Basic Example

```
java
CopyEdit
public class TryCatchExample {
    public static void main(String[] args) {
        try {
            int data = 10 / 0; // Will throw ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Exception caught: " + e.getMessage());
        }
        System.out.println("Program continues...");
    }
}
```

## Output:

```
csharp
CopyEdit
Exception caught: / by zero
Program continues...
```

## Multiple Catch Blocks

You can use multiple `catch` blocks to handle different exceptions separately.

```
java
CopyEdit
```

```
public class MultipleCatchExample {  
    public static void main(String[] args) {  
        try {  
            int[] arr = new int[5];  
            arr[10] = 30 / 0; // ArithmeticException occurs first  
        } catch (ArithmetiException e) {  
            System.out.println("Arithmetic Exception: " + e.getMessage());  
        } catch (ArrayIndexOutOfBoundsException e) {  
            System.out.println("Array Index Out of Bounds: " + e.getMessage());  
        } catch (Exception e) {  
            System.out.println("General Exception: " + e.getMessage());  
        }  
    }  
}
```

### Output:

```
csharp  
CopyEdit  
Arithmetic Exception: / by zero
```

## Catching Multiple Exceptions in a Single Catch Block

Since Java 7, you can catch multiple exceptions in a single `catch` block using the `|` operator.

```
java  
CopyEdit  
public class MultiCatchExample {  
    public static void main(String[] args) {  
        try {  
            String str = null;  
            System.out.println(str.length()); // NullPointerException  
        } catch (NullPointerException | ArithmetiException e) {  
            System.out.println("Exception caught: " + e.getMessage());  
        }  
    }  
}
```

```
}
```

### Output:

```
csharp
CopyEdit
Exception caught: Cannot invoke "String.length()" because "str" is null
```

---

## Using Finally Block

The `finally` block is used to execute code regardless of whether an exception is thrown or caught. This is typically used for resource cleanup.

```
java
CopyEdit
public class FinallyExample {
    public static void main(String[] args) {
        try {
            int data = 10 / 0;
        } catch (ArithmetricException e) {
            System.out.println("Exception caught.");
        } finally {
            System.out.println("Finally block executed.");
        }
    }
}
```

### Output:

```
php
CopyEdit
Exception caught.
Finally block executed.
```

---

## Nested Try-Catch Blocks

You can have a `try` block inside another `try` block. This is useful for handling exceptions at multiple levels.

```
java
CopyEdit
public class NestedTryExample {
    public static void main(String[] args) {
        try {
            try {
                int data = 10 / 0; // Inner try block
            } catch (ArithmaticException e) {
                System.out.println("Inner catch: " + e.getMessage());
            }
            int[] arr = new int[5];
            arr[10] = 50; // Outer try block
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Outer catch: " + e.getMessage());
        }
    }
}
```

**Output:**

```
csharp
CopyEdit
Inner catch: / by zero
Outer catch: Index 10 out of bounds for length 5
```

---

### Example: Try-Catch-Finally with Resources

The `finally` block is commonly used for releasing resources like closing files or database connections.

```
java
CopyEdit
import java.io.*;

public class ResourceExample {
```

```
public static void main(String[] args) {  
    BufferedReader br = null;  
    try {  
        br = new BufferedReader(new FileReader("file.txt"));  
        System.out.println(br.readLine());  
    } catch (IOException e) {  
        System.out.println("File not found or unable to read.");  
    } finally {  
        try {  
            if (br != null) {  
                br.close();  
            }  
        } catch (IOException e) {  
            System.out.println("Error closing the resource.");  
        }  
    }  
}  
}
```

## Best Practices

1. **Catch Specific Exceptions:**
  - o Avoid catching generic `Exception` unless necessary.
2. **Always Clean Up Resources:**
  - o Use `finally` or `try-with-resources` to close resources.
3. **Log Exceptions:**
  - o Use a logging framework for better debugging.
4. **Avoid Empty Catch Blocks:**
  - o Always handle or log the exception.
  - o Example of a bad practice:

```
java  
CopyEdit  
try {  
    int data = 10 / 0;
```

```
    } catch (Exception e) {
        // Do nothing
    }
```

---

## Conclusion

- The **try-catch** block is the backbone of exception handling in Java.
- It provides a way to detect and recover from runtime errors.
- Proper use of `try`, `catch`, and `finally` ensures robust, reliable, and maintainable code.

Core Java Notes By AJAY RAZZ

# **finally and throw Keywords in Java**

---

## **1. finally Keyword**

The **finally** block in Java is used to define a section of code that will always execute, regardless of whether an exception occurs or not. It is primarily used for **cleanup operations** such as closing files, releasing database connections, or freeing up other resources.

---

### **Key Features of finally**

- 1. Always Executes:**
    - o The **finally** block executes whether an exception is thrown or not.
  - 2. Optional Block:**
    - o You can use **try-catch** without **finally**, but adding it improves resource management.
  - 3. Cannot Be Skipped:**
    - o Even if there's a **return** statement in the **try** or **catch** block, the **finally** block still executes.
- 

## **Syntax**

```
java
CopyEdit
try {
    // Code that may throw an exception
} catch (ExceptionType e) {
    // Code to handle the exception
} finally {
```

```
// Code that always executes  
}
```

---

### Example: Using **finally**

```
java  
CopyEdit  
public class FinallyExample {  
    public static void main(String[] args) {  
        try {  
            int data = 10 / 0;  
        } catch (ArithmetricException e) {  
            System.out.println("Exception caught: " + e.getMessage());  
        } finally {  
            System.out.println("Finally block executed.");  
        }  
    }  
}
```

### Output:

```
vbnnet  
CopyEdit  
Exception caught: / by zero  
Finally block executed.
```

---

### Example: With Resource Cleanup

```
java  
CopyEdit  
import java.io.*;  
  
public class FinallyResourceExample {  
    public static void main(String[] args) {  
        BufferedReader br = null;
```

```
try {
    br = new BufferedReader(new FileReader("file.txt"));
    System.out.println(br.readLine());
} catch (IOException e) {
    System.out.println("Exception: " + e.getMessage());
} finally {
    try {
        if (br != null) {
            br.close(); // Resource cleanup
            System.out.println("BufferedReader closed.");
        }
    } catch (IOException e) {
        System.out.println("Error closing BufferedReader.");
    }
}
}
```

### When `finally` Does Not Execute

The only scenarios where the `finally` block may not execute are:

1. `System.exit()` is called before the `finally` block.
2. **JVM crashes.**
3. **Infinite loop or deadlock** in the `try` or `catch` block.

---

## 2. `throw` Keyword

The `throw` keyword in Java is used to explicitly **throw an exception**. This is often used in custom logic to signal that an exceptional situation has occurred.

---

## Key Features of `throw`

1. **Throws an Exception:**
    - o Can throw both checked and unchecked exceptions.
  2. **Single Exception:**
    - o Can throw only one exception at a time.
  3. **Execution Stops:**
    - o Once an exception is thrown, the subsequent code in the method is not executed.
- 

## Syntax

```
java
CopyEdit
throw new ExceptionType("Exception message");
```

---

### Example: Throwing a Built-in Exception

```
java
CopyEdit
public class ThrowExample {
    public static void validateAge(int age) {
        if (age < 18) {
            throw new IllegalArgumentException("Age must be 18 or above.");
        }
        System.out.println("Valid age.");
    }

    public static void main(String[] args) {
        try {
            validateAge(16);
        } catch (IllegalArgumentException e) {
            System.out.println("Exception caught: " + e.getMessage());
        }
    }
}
```

```
    }
}
}
```

### Output:

```
php
CopyEdit
Exception caught: Age must be 18 or above.
```

---

### Custom Exceptions with `throw`

You can create and throw your own exception types by extending the `Exception` class.

#### Example:

```
java
CopyEdit
class MyCustomException extends Exception {
    public MyCustomException(String message) {
        super(message);
    }
}

public class CustomThrowExample {
    public static void main(String[] args) {
        try {
            throw new MyCustomException("This is a custom exception.");
        } catch (MyCustomException e) {
            System.out.println("Caught: " + e.getMessage());
        }
    }
}
```

### Output:

```
vbn  
CopyEdit  
Caught: This is a custom exception.
```

---

## Differences Between **throw** and **throws**

Feature	<b>throw</b>	<b>throws</b>
<b>Purpose</b>	Used to throw an exception explicitly. Declares exceptions a method might throw.	
<b>Location</b>	Inside a method or block.	In the method declaration.
<b>Usage</b>	<code>throw new Exception();</code>	<code>public void method() throws Exception</code>

---

## Combining **finally** and **throw**

### Example:

```
java  
CopyEdit  
public class ThrowFinallyExample {  
    public static void main(String[] args) {  
        try {  
            throw new ArithmeticException("Arithmetic error occurred.");  
        } catch (ArithmeticeException e) {  
            System.out.println("Caught: " + e.getMessage());  
        } finally {  
            System.out.println("Finally block executed.");  
        }  
    }  
}
```

### Output:

```
vbn
```

CopyEdit  
Caught: Arithmetic error occurred.  
Finally block executed.

---

## Summary

Keyword	Description
---------	-------------

**finally** Defines a block of code that will always execute, used for cleanup operations.

**throw** Used to explicitly throw an exception from a method or block of code.

Proper use of `finally` ensures resources are freed, while `throw` helps in signaling and managing exceptions explicitly.

Core Java Notes By AJAY RAZZ

In Java, **custom exceptions** allow you to create your own exception classes to handle specific error scenarios in your application. By defining a custom exception, you can provide more meaningful and specific error messages or handling mechanisms compared to using standard exception types like `IOException` or `SQLException`.

Here's how you can create and use a custom exception in Java:

## Steps to Create a Custom Exception

1. **Extend an existing exception class:** Your custom exception must extend either `Exception` (for checked exceptions) or `RuntimeException` (for unchecked exceptions). By extending these classes, your custom exception will inherit their behavior.
2. **Define constructors:** Usually, you will define at least two constructors—one that accepts a message and another that accepts both a message and a cause (the underlying exception).
3. **Optionally add custom functionality:** You can add additional functionality to your custom exception, such as custom fields or methods, depending on your needs.

## Example of a Custom Exception

Here's an example of a custom exception for handling invalid account operations:

```
java
Copy
// Custom checked exception class
public class InvalidAccountOperationException extends Exception {

    // Constructor that accepts a message
    public InvalidAccountOperationException(String message) {
        super(message); // Pass the message to the superclass (Exception)
    }

    // Constructor that accepts a message and a cause (another exception)
    public InvalidAccountOperationException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

```
}
```

## Using the Custom Exception

You can now throw this custom exception in your code to handle specific error situations.

```
java
Copy
public class BankAccount {

    private double balance;

    public BankAccount(double balance) {
        this.balance = balance;
    }

    // Method to withdraw money, throws custom exception if balance is insufficient
    public void withdraw(double amount) throws InvalidAccountOperationException {
        if (amount > balance) {
            throw new InvalidAccountOperationException("Insufficient balance for withdrawal");
        }
        balance -= amount;
    }
}
```

## Handling the Custom Exception

You can handle the custom exception using a `try-catch` block like this:

```
java
Copy
public class Main {
    public static void main(String[] args) {
        BankAccount account = new BankAccount(100.00);

        try {
            account.withdraw(150.00); // This will trigger the custom exception
        }
    }
}
```

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

## Types of Custom Exceptions

- **Checked Exceptions:** These must be either caught or declared in the method signature (using `throws`). In the example above, `InvalidAccountOperationException` extends `Exception`, making it a checked exception.
- **Unchecked Exceptions:** These extend `RuntimeException` and do not require explicit handling (i.e., no need for a `throws` declaration or try-catch block). They are often used for programming errors, like `NullPointerException` or `ArrayIndexOutOfBoundsException`.

## Summary

- Custom exceptions allow you to handle specific application scenarios more precisely.
- You create a custom exception by extending `Exception` or `RuntimeException`.
- You can add custom constructors and fields to the exception.
- Use `throw` to raise the exception and try-catch blocks to handle it.

This approach helps keep your error handling organized, improves clarity, and enhances code readability when dealing with unique application-specific errors.

# String handling in Java

is a fundamental topic since strings are one of the most commonly used data types in programming. Java provides a rich set of tools and classes for working with strings. Let's start from the basics and work through the essential concepts.

## What is a String in Java?

A **String** in Java is an object that represents a sequence of characters. It is used to store and manipulate text in Java programs. Unlike primitive types (like `int` or `char`), Strings are objects in Java and are instances of the `String` class.

## How Strings are Stored in Java

Strings in Java are stored in a special area of memory called the **String Pool**. This is an optimization technique that ensures the same string value is not duplicated in memory.

For example, if you create the string "hello" twice:

```
java
Copy
String str1 = "hello";
String str2 = "hello";
```

Both `str1` and `str2` point to the same string object in the String Pool.

## String Creation in Java

1. **Using String literals (recommended):** This is the most common way to create a string. When you use double quotes (" "), Java will automatically check the string pool first to see if the string already exists.

```
java
Copy
String s1 = "Hello";
String s2 = "Hello";
```

- Using the `new` keyword: You can also create a new string using the `new` keyword. This approach will create a new object, even if the string already exists in the String Pool.

```
java
Copy
String s3 = new String("Hello");
```

However, this is not commonly used unless you specifically need a new object in memory.

## Important String Methods

Java provides many methods for handling and manipulating strings. Here are some of the most commonly used ones:

- Length of a String: The `length()` method returns the number of characters in the string.

```
java
Copy
String s = "Hello";
int length = s.length(); // 5
```

- Accessing Characters: You can access individual characters in a string using the `charAt()` method. It returns the character at the specified index.

```
java
Copy
char c = s.charAt(1); // 'e'
```

- Concatenating Strings: You can join two or more strings together using the `+` operator or the `concat()` method.

```
java
Copy
String greeting = "Hello";
String name = "John";
String message = greeting + " " + name; // Using + operator
// or
```

```
String message2 = greeting.concat(" ").concat(name); // Using concat method
```

4. **Converting to Uppercase or Lowercase:** The `toUpperCase()` and `toLowerCase()` methods return a new string that is converted to uppercase or lowercase.

```
java
Copy
String lower = "hello".toLowerCase(); // "hello"
String upper = "hello".toUpperCase(); // "HELLO"
```

5. **Trimming Whitespace:** The `trim()` method removes leading and trailing whitespace from the string.

```
java
Copy
String text = "    Hello    ";
String trimmedText = text.trim(); // "Hello"
```

6. **Substring:** The `substring()` method returns a part of the string. You can specify the starting index, and optionally, an ending index.

```
java
Copy
String text = "Hello World";
String sub = text.substring(0, 5); // "Hello"
```

7. **Replacing Characters or Substrings:** The `replace()` method allows you to replace characters or substrings.

```
java
Copy
String sentence = "I like Java";
String newSentence = sentence.replace("Java", "Python"); // "I like Python"
```

8. **Checking String Equality:** Use the `equals()` method to compare strings for equality. The `==` operator checks for reference equality, but `equals()` checks for value equality.

```
java
```

```
Copy
String s1 = "hello";
String s2 = "hello";
boolean isEqual = s1.equals(s2); // true
```

9. **String Contains:** You can check if a string contains a specific substring using the `contains()` method.

```
java
Copy
String text = "Hello, World!";
boolean contains = text.contains("World"); // true
```

10. **Index of a Substring:** The `indexOf()` method returns the index of the first occurrence of a substring, or `-1` if not found.

```
java
Copy
String text = "Hello, World!";
int index = text.indexOf("World"); // 7
```

## String Immutability

In Java, strings are **immutable**, meaning once a string object is created, it cannot be changed. Any operation on a string (like concatenation or replacement) creates a new string object, rather than modifying the existing one.

Example:

```
java
Copy
String s1 = "Hello";
s1 = s1 + " World"; // Creates a new string "Hello World"
```

This behavior helps in performance optimization, especially when working with large numbers of strings in multi-threaded environments.

## StringBuilder and StringBuffer

Since strings are immutable, using them in situations where you need to modify a string repeatedly (e.g., in loops) can be inefficient. In these cases, you can use `StringBuilder` or `StringBuffer`, which are mutable and more efficient for such operations.

- `StringBuilder` is preferred for single-threaded applications because it's faster.
- `StringBuffer` is similar to `StringBuilder` but is thread-safe, meaning it has synchronization built in.

Example with `StringBuilder`:

```
java
Copy
StringBuilder sb = new StringBuilder("Hello");
sb.append(" World");
String result = sb.toString(); // "Hello World"
```

## String Formatting

Java provides the `String.format()` method to format strings, similar to how you might format strings in other languages using placeholders.

Example:

```
java
Copy
int age = 25;
String message = String.format("I am %d years old.", age);
```

This is very useful for creating dynamic output, like constructing sentences with variables.

## String Comparison

- `equals()` – Compares the content of two strings.
- `compareTo()` – Compares two strings lexicographically (like dictionary order). It returns an integer:
  - 0 if the strings are equal.
  - A negative number if the first string is lexicographically less than the second.

- A positive number if the first string is lexicographically greater than the second.

```
java
Copy
String s1 = "apple";
String s2 = "banana";
int result = s1.compareTo(s2); // Negative because "apple" comes before "banana"
```

## Conclusion

- **Strings** in Java are objects that represent sequences of characters.
- **Immutability** is a key characteristic of strings in Java.
- You can manipulate strings using built-in methods such as `substring()`, `toUpperCase()`, `replace()`, and others.
- For frequent string modifications, use `StringBuilder` or `StringBuffer` for better performance.
- **String Pooling** helps save memory by reusing identical string literals.

Understanding string handling in Java is critical because strings are widely used throughout almost every Java application.

The **String class in Java** is one of the most commonly used classes and is part of the `java.lang` package, which is automatically imported in every Java program. The `String` class represents a sequence of characters and is used to store and manipulate text in Java.

## Key Features of the `String` Class

1. **Immutable:** Once a `String` object is created, it cannot be changed. Any operation on a string (like concatenation, replacement, etc.) creates a new string object rather than modifying the original one.
2. **String Pool:** Strings in Java are stored in a special memory region known as the **String Pool**. This is an optimization to save memory when multiple identical string literals are used in the program. When you create a string literal, it's placed in this pool, and if the same string is used again, the existing reference is returned.

## Constructor of the `String` Class

While you typically create strings using literals, you can also create `String` objects using constructors:

1. **Using a string literal** (preferred):

```
java
Copy
String s1 = "Hello";
```

2. **Using the `new` keyword** (less efficient, creates a new object):

```
java
Copy
String s2 = new String("Hello");
```

3. **Using a character array:**

```
java
Copy
```

```
char[] chars = {'H', 'e', 'l', 'l', 'o'};
String s3 = new String(chars); // "Hello"
```

#### 4. Using byte array:

```
java
Copy
byte[] bytes = {72, 101, 108, 108, 111}; // ASCII values of "Hello"
String s4 = new String(bytes); // "Hello"
```

### Key Methods of the `String` Class

Here are the most commonly used methods in the `String` class:

#### 1. Length of a String

The `length()` method returns the number of characters in the string (its length).

```
java
Copy
String s = "Hello";
int length = s.length(); // 5
```

#### 2. Character at a Specific Index

The `charAt()` method returns the character at a specified index in the string.

```
java
Copy
String s = "Hello";
char c = s.charAt(1); // 'e'
```

#### 3. Substring

The `substring()` method extracts a portion of the string from a given start index, and optionally an end index.

```
java
Copy
String s = "Hello, World!";
String sub1 = s.substring(0, 5); // "Hello"
String sub2 = s.substring(7); // "World!"
```

## 4. String Comparison

- `equals()` – Compares the content of two strings.
- `equalsIgnoreCase()` – Compares two strings, ignoring case.
- `compareTo()` – Compares two strings lexicographically (dictionary order).

```
java
Copy
String s1 = "Hello";
String s2 = "hello";
boolean isEqual = s1.equals(s2); // false
boolean isEqualIgnoreCase = s1.equalsIgnoreCase(s2); // true
int comparison = s1.compareTo(s2); // Negative because "Hello" comes before "hello"
```

## 5. String Concatenation

You can concatenate strings using the `+` operator or the `concat()` method.

```
java
Copy
String s1 = "Hello";
String s2 = " World";
String result = s1 + s2; // "Hello World"
String result2 = s1.concat(s2); // "Hello World"
```

## 6. Case Conversion

- `toUpperCase()` – Converts the string to uppercase.
- `toLowerCase()` – Converts the string to lowercase.

```
java
Copy
String s = "Hello";
String upper = s.toUpperCase(); // "HELLO"
String lower = s.toLowerCase(); // "hello"
```

## 7. Trimming Whitespace

The `trim()` method removes leading and trailing whitespace from a string.

```
java
Copy
String s = "    Hello    ";
String trimmed = s.trim(); // "Hello"
```

## 8. Replacing Characters/Substrings

The `replace()` method allows you to replace characters or substrings in the string.

```
java
Copy
String s = "I like Java";
String replaced = s.replace("Java", "Python"); // "I like Python"
```

## 9. Checking if String Contains Another String

The `contains()` method checks whether the string contains a specified sequence of characters.

```
java
Copy
String s = "Hello, World!";
boolean contains = s.contains("World"); // true
```

## 10. Finding the Index of a Character or Substring

- `indexOf()` – Returns the index of the first occurrence of a character or substring.
- `lastIndexOf()` – Returns the index of the last occurrence.

```
java
Copy
String s = "Hello, World!";
int index1 = s.indexOf("World");    // 7
int index2 = s.lastIndexOf("o");    // 8
```

## 11. Splitting a String into Substrings

The `split()` method splits the string into an array of substrings based on a given delimiter.

```
java
Copy
String s = "apple,banana,cherry";
String[] fruits = s.split(","); // ["apple", "banana", "cherry"]
```

## 12. Checking for Empty or Null

- `isEmpty()` – Returns true if the string is empty (i.e., has a length of 0).
- `isBlank()` – Returns true if the string is empty or contains only whitespace (Java 11 and later).

```
java
Copy
String s = "";
boolean isEmpty = s.isEmpty(); // true
boolean isBlank = s.isBlank(); // true
```

## 13. String Join

The `join()` method allows you to join an array or a collection of strings using a specified delimiter.

```
java
Copy
String[] words = {"Hello", "World"};
String joined = String.join(" ", words); // "Hello World"
```

## 14. String Matching with Regular Expressions

The `matches()` method checks whether the string matches a given regular expression.

```
java
Copy
String s = "abc123";
boolean matches = s.matches("[a-z]+\d+"); // true (matches lowercase letters followed by digits)
```

## 15. String Format

The `String.format()` method is used for string formatting, allowing you to insert values into a formatted string.

```
java
Copy
int age = 25;
String formatted = String.format("I am %d years old.", age); // "I am 25 years old."
```

## 16. String.valueOf()

This method returns the string representation of any data type (primitive values, objects, etc.).

```
java
Copy
int number = 42;
String str = String.valueOf(number); // "42"
```

## Example Code Demonstrating String Methods

```
java
```

```
Copy
public class StringExample {
    public static void main(String[] args) {
        String s1 = "Hello";
        String s2 = " World";

        // Concatenate strings
        String result = s1 + s2; // "Hello World"
        System.out.println(result);

        // Convert to uppercase
        System.out.println(s1.toUpperCase()); // "HELLO"

        // Check if string contains a substring
        System.out.println(s1.contains("ell")); // true

        // Replace part of a string
        String replaced = s1.replace("Hello", "Hi");
        System.out.println(replaced); // "Hi"

        // Find the index of a character
        System.out.println(s1.indexOf("e")); // 1

        // Substring
        System.out.println(s1.substring(0, 3)); // "Hel"

        // Check if string is empty
        System.out.println(s1.isEmpty()); // false
    }
}
```

## Conclusion

The `String` class in Java provides a vast number of methods for creating, manipulating, and inspecting strings. Understanding and mastering these methods is essential for handling text-based data efficiently. Although strings are immutable in Java, this immutability offers benefits like security, thread-safety, and memory efficiency.

# StringBuilder and StringBuffer Classes in Java

In Java, `StringBuilder` and `StringBuffer` are two classes that provide mutable sequences of characters, allowing you to efficiently modify strings in your code. These classes are part of the `java.lang` package, and they are used when you need to modify the content of a string frequently, such as in loops or string concatenations. Unlike the `String` class, which is immutable, both `StringBuilder` and `StringBuffer` provide a way to modify the string directly without creating new objects every time a modification is made.

Let's break down both classes in detail.

---

## 1. StringBuilder Class

### What is StringBuilder?

- `StringBuilder` is a mutable sequence of characters.
- It is **not synchronized**, meaning it is not thread-safe. This makes it faster for single-threaded applications or when thread safety is not a concern.
- It is **recommended for use in most situations** where you need to modify strings frequently, especially when the application is not multi-threaded.

### Constructor

- `StringBuilder():` Creates an empty string builder with an initial capacity of 16 characters.
- `StringBuilder(String str):` Creates a string builder with the specified string `str` as its initial content.
- `StringBuilder(int capacity):` Creates an empty string builder with a specified initial capacity.

### Example:

```
java  
Copy
```

```
StringBuilder sb1 = new StringBuilder(); // Default capacity 16
StringBuilder sb2 = new StringBuilder("Hello"); // Initial content "Hello"
StringBuilder sb3 = new StringBuilder(50); // Capacity 50
```

## Common Methods of `StringBuilder`

- `append()`: Adds the specified string to the end of the current string builder.

```
java
Copy
StringBuilder sb = new StringBuilder("Hello");
sb.append(" World"); // sb = "Hello World"
```

- `insert()`: Inserts the specified string at the specified index.

```
java
Copy
StringBuilder sb = new StringBuilder("Hello");
sb.insert(5, " World"); // sb = "Hello World"
```

- `delete()`: Removes a substring from the string builder, defined by the start and end indices.

```
java
Copy
StringBuilder sb = new StringBuilder("Hello World");
sb.delete(5, 11); // sb = "Hello"
```

- `deleteCharAt()`: Removes a character at the specified index.

```
java
Copy
StringBuilder sb = new StringBuilder("Hello");
sb.deleteCharAt(1); // sb = "Hllo"
```

- `reverse()`: Reverses the contents of the string builder.

```
java
Copy
StringBuilder sb = new StringBuilder("Hello");
sb.reverse(); // sb = "olleH"
```

- **toString()**: Converts the `StringBuilder` to a `String` object.

```
java
Copy
StringBuilder sb = new StringBuilder("Hello");
String result = sb.toString(); // result = "Hello"
```

- **capacity()**: Returns the current capacity of the `StringBuilder`. If the capacity exceeds the size of the content, the extra capacity is available for future modifications.

```
java
Copy
StringBuilder sb = new StringBuilder("Hello");
int capacity = sb.capacity(); // capacity = 21 (default capacity is 16, but increases)
```

- **setLength()**: Sets the length of the `StringBuilder`. If the length is less than the current length, it truncates the content; if it's greater, it appends null characters.

```
java
Copy
StringBuilder sb = new StringBuilder("Hello");
sb.setLength(3); // sb = "Hel"
```

---

## 2. StringBuffer Class

### What is `StringBuffer`?

- `StringBuffer` is also a mutable sequence of characters.

- It is **synchronized**, meaning it is thread-safe. This makes it suitable for use in multi-threaded applications, where multiple threads might be modifying the same string.
- However, because of synchronization overhead, it tends to be slower than `StringBuilder` in single-threaded applications.

## Constructor

- `StringBuffer()`: Creates an empty string buffer with an initial capacity of 16 characters.
- `StringBuffer(String str)`: Creates a string buffer with the specified string `str` as its initial content.
- `StringBuffer(int capacity)`: Creates an empty string buffer with a specified initial capacity.

Example:

```
java
Copy
StringBuffer sb1 = new StringBuffer(); // Default capacity 16
StringBuffer sb2 = new StringBuffer("Hello"); // Initial content "Hello"
StringBuffer sb3 = new StringBuffer(50); // Capacity 50
```

## Common Methods of `StringBuffer`

- `append()`: Adds the specified string to the end of the current string buffer.

```
java
Copy
StringBuffer sb = new StringBuffer("Hello");
sb.append(" World"); // sb = "Hello World"
```

- `insert()`: Inserts the specified string at the specified index.

```
java
Copy
StringBuffer sb = new StringBuffer("Hello");
sb.insert(5, " World"); // sb = "Hello World"
```

- **delete():** Removes a substring from the string buffer, defined by the start and end indices.

```
java
Copy
StringBuffer sb = new StringBuffer("Hello World");
sb.delete(5, 11); // sb = "Hello"
```

- **deleteCharAt():** Removes a character at the specified index.

```
java
Copy
StringBuffer sb = new StringBuffer("Hello");
sb.deleteCharAt(1); // sb = "Hlo"
```

- **reverse():** Reverses the contents of the string buffer.

```
java
Copy
StringBuffer sb = new StringBuffer("Hello");
sb.reverse(); // sb = "olleH"
```

- **toString():** Converts the StringBuffer to a String object.

```
java
Copy
StringBuffer sb = new StringBuffer("Hello");
String result = sb.toString(); // result = "Hello"
```

- **capacity():** Returns the current capacity of the StringBuffer.

```
java
Copy
StringBuffer sb = new StringBuffer("Hello");
int capacity = sb.capacity(); // capacity = 21
```

- **setLength():** Sets the length of the `StringBuffer`.

```
java
Copy
StringBuffer sb = new StringBuffer("Hello");
sb.setLength(3); // sb = "Hel"
```

---

## StringBuilder vs StringBuffer: Key Differences

Feature	StringBuilder	StringBuffer
<b>Thread-Safety</b>	Not thread-safe (faster)	Thread-safe (slower due to synchronization)
<b>Performance</b>	Faster (in single-threaded applications)	Slower (due to synchronized methods)
<b>Use Case</b>	Best for single-threaded applications	Best for multi-threaded applications
<b>Synchronization</b>	No synchronization overhead	Synchronization overhead (due to synchronized methods)

### When to Use Which Class?

- **Use `StringBuilder`:** When you are working in a single-threaded environment or when you don't need synchronization. It provides better performance due to the lack of synchronization overhead.
- **Use `StringBuffer`:** When you need a mutable string and your application is multi-threaded, and thread safety is important. The synchronized nature of `StringBuffer` makes it suitable for concurrent access, but with a performance cost.

---

## Example Code Demonstrating StringBuilder and StringBuffer

```
java
Copy
public class StringBuilderStringBufferExample {
    public static void main(String[] args) {
        // Using StringBuilder
```

```
StringBuilder sb = new StringBuilder("Hello");
sb.append(" World");
sb.insert(6, "Java ");
sb.reverse();
System.out.println("StringBuilder Result: " + sb.toString());

// Using StringBuffer
StringBuffer sbf = new StringBuffer("Hello");
sbf.append(" World");
sbf.insert(6, "Java ");
sbf.reverse();
System.out.println("StringBuffer Result: " + sbf.toString());
}

}
```

## Output:

rust  
Copy  
StringBuilder Result: dlrow avaJ olleH  
StringBuffer Result: dlrow avaJ olleH

---

## Conclusion

- **StringBuilder** and **StringBuffer** are both mutable classes that allow you to modify the content of a string.
- **StringBuilder** is faster in single-threaded environments because it doesn't have the synchronization overhead that **StringBuffer** has.
- **StringBuffer** is thread-safe, making it a better choice when thread safety is important, but it is slower due to synchronization.
- Both classes provide many useful methods like `append()`, `insert()`, `delete()`, `reverse()`, and `toString()` for manipulating strings efficiently.

# The Collections Framework in Java

The Collections Framework in Java is a set of classes and interfaces that implement various data structures and algorithms to store and manipulate groups of objects. The framework provides a unified architecture for working with collections of objects, such as lists, sets, queues, and maps. It also provides algorithms to manipulate these collections, such as sorting and searching.

The goal of the Collections Framework is to simplify and standardize the way we work with collections, making code more flexible, reusable, and easier to understand.

## Key Components of the Collections Framework

1. **Interfaces:** These define the abstract behavior of collections (e.g., how a collection is accessed or modified).
2. **Classes:** These provide concrete implementations of the collection interfaces.
3. **Algorithms:** These are methods that operate on collections (e.g., sorting, searching, etc.).
4. **Utility Classes:** These provide helper methods (e.g., for manipulating or searching collections).

### 1. Core Interfaces of the Collections Framework

There are several important interfaces in the Collections Framework. These interfaces define the basic operations on collections.

#### Collection Interface

- The root interface in the Collections Framework hierarchy.
- It defines common operations such as adding elements (`add()`), removing elements (`remove()`), and checking for elements (`contains()`).

```
java
Copy
public interface Collection<E> {
    boolean add(E e);
    boolean remove(Object o);
    boolean contains(Object o);
```

```
    int size();
    void clear();
    boolean isEmpty();
}
```

## List Interface

- Extends `Collection` and represents an ordered collection (sequence) that allows duplicate elements.
- Lists maintain the order in which elements are inserted and allow positional access (i.e., accessing elements by index).
- Key classes implementing `List` include `ArrayList`, `LinkedList`, and `Vector`.

```
java
Copy
public interface List<E> extends Collection<E> {
    E get(int index);
    void add(int index, E element);
    E remove(int index);
    int indexOf(Object o);
}
```

## Set Interface

- Extends `Collection` and represents an unordered collection that does **not allow duplicate elements**.
- Key classes implementing `Set` include `HashSet`, `LinkedHashSet`, and `TreeSet`.

```
java
Copy
public interface Set<E> extends Collection<E> {
    boolean add(E e);
    boolean contains(Object o);
    boolean remove(Object o);
}
```

## Queue Interface

- Extends `Collection` and represents a collection designed for holding elements prior to processing (e.g., first-in, first-out - FIFO).
- Key classes implementing `Queue` include `LinkedList` (which also implements `Deque`), `PriorityQueue`, and `ArrayDeque`.

```
java
Copy
public interface Queue<E> extends Collection<E> {
    boolean add(E e);
    E remove();
    E peek();
}
```

## Map Interface

- Not a direct child of `Collection`, but part of the Collections Framework.
- Represents a collection of key-value pairs, where each key maps to a single value.
- Key classes implementing `Map` include `HashMap`, `TreeMap`, `LinkedHashMap`, and `Hashtable`.

```
java
Copy
public interface Map<K, V> {
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    int size();
}
```

---

## 2. Common Implementations of the Core Interfaces

### List Implementations

- **ArrayList:**
  - A resizable array implementation of the `List` interface.
  - Allows random access of elements (by index).

- Provides fast insertion and deletion at the end of the list but slower insertion/removal at other positions.

```
java
Copy
List<String> list = new ArrayList<>();
list.add("A");
list.add("B");
list.add("C");
```

- **LinkedList:**

- A doubly-linked list implementation of the `List` interface.
- Provides fast insertion and deletion at both ends (head and tail) of the list but slower random access.

```
java
Copy
List<String> list = new LinkedList<>();
list.add("A");
list.add("B");
list.add("C");
```

## Set Implementations

- **HashSet:**

- A set that uses a hash table for storing elements.
- Does not guarantee the order of elements.
- Ensures that no duplicate elements are present.

```
java
Copy
Set<String> set = new HashSet<>();
set.add("A");
set.add("B");
set.add("A"); // Duplicate, won't be added
```

- **TreeSet:**

- A set that implements the `NavigableSet` interface and uses a red-black tree.
- Elements are ordered in their natural order (or by a comparator provided at the time of construction).

```
java
Copy
Set<String> set = new TreeSet<>();
set.add("A");
set.add("C");
set.add("B"); // Elements will be sorted: ["A", "B", "C"]
```

## Queue Implementations

- **PriorityQueue:**

- A queue that orders elements according to their natural ordering or by a comparator.
- It's not thread-safe.

```
java
Copy
Queue<Integer> queue = new PriorityQueue<>();
queue.add(10);
queue.add(5);
queue.add(20); // Elements are ordered based on priority: [5, 10, 20]
```

- **LinkedList:**

- As `LinkedList` implements both `List` and `Queue` interfaces, it can be used as a queue.

```
java
Copy
Queue<String> queue = new LinkedList<>();
queue.add("A");
queue.add("B");
queue.poll(); // Removes "A"
```

## Map Implementations

- **HashMap:**
  - A map that uses a hash table for storing key-value pairs.
  - Keys are unordered, and the map allows null values and null keys.

```
java
Copy
Map<Integer, String> map = new HashMap<>();
map.put(1, "One");
map.put(2, "Two");
map.put(3, "Three");
```

- **TreeMap:**
  - A map that uses a red-black tree and stores keys in sorted order (either natural order or based on a comparator).

```
java
Copy
Map<Integer, String> map = new TreeMap<>();
map.put(3, "Three");
map.put(1, "One");
map.put(2, "Two");
```

### 3. Algorithms in the Collections Framework

The Collections Framework also provides a set of algorithms that can be used with various collection types.

- **Sorting:**
  - The `Collections.sort()` method can be used to sort lists.
  - `TreeSet` and `TreeMap` automatically keep elements sorted.

```
java
Copy
List<Integer> list = new ArrayList<>();
list.add(5);
list.add(2);
```

```
list.add(8);
Collections.sort(list); // [2, 5, 8]
```

- **Searching:**

- The `Collections.binarySearch()` method can be used to perform a binary search on a sorted list.

```
java
Copy
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);
int index = Collections.binarySearch(list, 3); // index = 2
```

- **Shuffling:**

- The `Collections.shuffle()` method can be used to shuffle elements randomly in a list.

```
java
Copy
List<Integer> list = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));
Collections.shuffle(list); // Randomly shuffled
```

- **Reversing:**

- The `Collections.reverse()` method can be used to reverse the order of elements in a list.

```
java
Copy
List<Integer> list = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));
Collections.reverse(list); // [5, 4, 3, 2, 1]
```

---

## 4. Utility Classes

The Collections Framework also includes utility classes like:

- **Collections:** Contains static methods for manipulating collections (e.g., sorting, shuffling, reversing).
- **Arrays:** Provides methods for manipulating arrays and converting arrays to collections.

```
java
Copy
List<Integer> list = Arrays.asList(1, 2, 3, 4);
System.out.println(Collections.max(list)); // 4
```

---

## Conclusion

The **Collections Framework** in Java provides a well-organized set of interfaces and classes that allow you to efficiently work with collections of objects. Whether you're using `List`, `Set`, `Queue`, or `Map`, the Collections Framework provides all the data structures and algorithms you need for manipulating and managing groups of objects.

By using the framework, you can leverage the pre-built classes and methods for common operations like adding, removing, searching, sorting, and iterating through elements, thus simplifying and optimizing your code.

Core Java Notes By AJAY RAZZ

In Java, the **Collection Framework** is a unified architecture for storing and manipulating a group of objects. It includes a hierarchy of interfaces and classes that define different types of collections, such as lists, sets, and maps. These interfaces provide common operations like adding, removing, and accessing elements. Let's take a detailed look at the three major interfaces in the Collection Framework: **List**, **Set**, and **Map**.

---

## 1. The Collection Interface

Before diving into **List**, **Set**, and **Map**, it's important to understand that `collection` is the root interface in the Java Collections Framework hierarchy. All collection interfaces extend `collection` (except for `Map`).

**Key methods of the Collection interface:**

- `boolean add(E e)`: Adds an element to the collection.
  - `boolean remove(Object o)`: Removes a single occurrence of the specified element.
  - `int size()`: Returns the number of elements in the collection.
  - `boolean isEmpty()`: Checks if the collection is empty.
  - `boolean contains(Object o)`: Checks if the collection contains the specified element.
  - `void clear()`: Removes all elements from the collection.
- 

## 2. List Interface

A **List** is an ordered collection that allows **duplicate elements**. Lists allow access to elements via an index (position in the list) and maintain the order in which elements were inserted.

**Key Features of List:**

- **Ordered:** Elements are stored in a sequence.
- **Allows Duplicates:** Lists can have repeated elements.
- **Indexed:** You can access elements by their index.

#### Important Methods of List:

- `E get(int index)`: Returns the element at the specified index.
- `void add(int index, E element)`: Inserts the specified element at the specified index.
- `E remove(int index)`: Removes the element at the specified index.
- `int indexOf(Object o)`: Returns the index of the first occurrence of the specified element.
- `int lastIndexOf(Object o)`: Returns the index of the last occurrence of the specified element.

#### Common Implementations of List:

- **ArrayList:** A dynamic array that allows random access of elements by index. It is the most commonly used implementation of List due to its fast access times.

```
java
Copy
List<String> list = new ArrayList<>();
list.add("A");
list.add("B");
list.add("C");
```

- **LinkedList:** A doubly-linked list that provides efficient insertion and deletion at both ends (head and tail), but slower access by index.

```
java
Copy
List<String> list = new LinkedList<>();
list.add("A");
list.add("B");
list.add("C");
```

- **Vector:** An older, synchronized version of ArrayList, which is now less commonly used.

```
java
Copy
List<String> list = new Vector<>();
list.add("A");
list.add("B");
list.add("C");
```

---

### 3. Set Interface

A **set** is a collection that does not allow duplicate elements and does not guarantee any specific order of the elements. It is useful when you need to store unique elements.

#### Key Features of **set**:

- **Unordered:** The elements in a set are not stored in any specific order.
- **No Duplicates:** Sets do not allow duplicate elements.

#### Important Methods of **set**:

- **boolean add(E e):** Adds the specified element to the set.
- **boolean contains(Object o):** Checks if the set contains the specified element.
- **boolean remove(Object o):** Removes the specified element from the set.
- **void clear():** Removes all elements from the set.

#### Common Implementations of **set**:

- **HashSet:** The most common implementation of **Set**, which stores elements in a hash table. It does not guarantee any specific order of elements.

```
java
Copy
Set<String> set = new HashSet<>();
```

```
set.add("A");
set.add("B");
set.add("A"); // Duplicate, will not be added
```

- **LinkedHashSet:** A `Set` implementation that maintains the order of insertion. It uses a hash table but also maintains a linked list to keep track of the order.

```
java
Copy
Set<String> set = new LinkedHashSet<>();
set.add("A");
set.add("B");
set.add("A"); // Duplicate, will not be added
```

- **TreeSet:** A `Set` implementation that stores elements in a red-black tree, which guarantees that elements are ordered. You can use a natural ordering or provide a custom comparator.

```
java
Copy
Set<String> set = new TreeSet<>();
set.add("C");
set.add("A");
set.add("B"); // Elements will be sorted: [A, B, C]
```

---

## 4. Map Interface

A `Map` is an object that maps **keys** to **values**. It does not extend `Collection`, but it is a part of the Collections Framework. Each key in a map is unique, and each key maps to exactly one value.

### Key Features of `Map`:

- **Key-Value Pairs:** Maps store key-value pairs. A key is associated with a single value.
- **Unique Keys:** A map cannot contain duplicate keys.
- **Unordered or Sorted:** Maps can be ordered or unordered, depending on the implementation.

## **Important Methods of Map:**

- `v put(K key, v value)`: Adds a key-value pair to the map.
- `v get(Object key)`: Returns the value associated with the specified key.
- `v remove(Object key)`: Removes the key-value pair associated with the specified key.
- `boolean containsKey(Object key)`: Checks if the map contains a mapping for the specified key.
- `Set<K> keySet()`: Returns a set view of the keys contained in the map.
- `Collection<V> values()`: Returns a collection view of the values contained in the map.
- `Set<Map.Entry<K, V>> entrySet()`: Returns a set view of the mappings in the map.

## **Common Implementations of Map:**

- **HashMap**: A hash table-based implementation of `Map`. It does not guarantee any specific order of elements. It allows `null` values and `null` keys.

```
java
Copy
Map<Integer, String> map = new HashMap<>();
map.put(1, "One");
map.put(2, "Two");
map.put(3, "Three");
```

- **TreeMap**: A map that uses a red-black tree to store the keys in sorted order. It allows you to define the order either by the natural ordering of the keys or by a custom comparator.

```
java
Copy
Map<Integer, String> map = new TreeMap<>();
map.put(3, "Three");
map.put(1, "One");
map.put(2, "Two"); // Elements will be sorted: {1=One, 2=Two, 3=Three}
```

- **LinkedHashMap**: A map that maintains the order of insertion while using a hash table to store key-value pairs.

```

java
Copy
Map<Integer, String> map = new LinkedHashMap<>();
map.put(1, "One");
map.put(2, "Two");
map.put(3, "Three"); // Maintains insertion order

```

- **Hashtable:** A synchronized version of `HashMap`. It is rarely used today in favor of `HashMap` with external synchronization.

```

java
Copy
Map<Integer, String> map = new Hashtable<>();
map.put(1, "One");
map.put(2, "Two");

```

## Comparison of List, Set, and Map

Feature	List	Set	Map
<b>Order</b>	Elements are ordered by index.	Unordered (except <code>LinkedHashSet</code> and <code>TreeSet</code> )	Unordered (except <code>LinkedHashMap</code> , <code>TreeMap</code> )
<b>Duplicates</b>	Allows duplicate elements.	Does not allow duplicates.	Does not allow duplicate keys.
<b>Access Type</b>	Indexed (access by position).	No index access (access by element).	Key-value pair access.
<b>Performance</b>	Allows fast random access (in <code>ArrayList</code> ).	Generally faster for lookups and membership tests.	Typically slower than <code>List</code> for lookups.

## Conclusion

- **List:** Use when you need an ordered collection that allows duplicates and you need random access via an index.
- **Set:** Use when you need a collection that stores unique elements and the order doesn't matter (or when you want elements sorted with `TreeSet`).

- **Map:** Use when you need to associate keys with values (key-value pairs) and ensure that each key is unique.

Each of these interfaces serves different use cases and provides different functionalities. By selecting the right implementation based on your requirements (e.g., whether you need ordered collections, fast lookups, or uniqueness), you can ensure that your code is efficient, clean, and easy to maintain.

Core Java Notes By AJAY RAZZ

In Java, there are several commonly used classes in the Collections Framework, each designed to handle different types of collections and provide varying functionalities depending on the requirements. Some of the most widely used classes are **ArrayList**, **LinkedList**, **HashSet**, and **HashMap**. These classes implement the **List**, **Set**, and **Map** interfaces, respectively. Let's break down each of these classes in detail to understand their internal workings, characteristics, advantages, and use cases.

---

## 1. ArrayList (Implements `List` Interface)

### Key Features:

- **Dynamic Array:** `ArrayList` is backed by a dynamically resizing array. It can grow or shrink as elements are added or removed.
- **Indexed Access:** Provides fast random access to elements by index.
- **Allows Duplicates:** It allows duplicate elements and maintains insertion order.
- **Null Elements:** Allows `null` elements.
- **Resizable:** As elements are added, the underlying array may grow automatically to accommodate more elements.

### Performance:

- **Get** (Access by index): **O(1)** (constant time).
- **Add** (to the end): **O(1)** on average, but can be **O(n)** in the worst case (when the array needs to be resized).
- **Remove**: **O(n)** due to shifting of elements after removal (for non-terminal positions).
- **Insert** (at an index other than the end): **O(n)** because elements need to be shifted.

### Example:

```
java
Copy
ArrayList<String> list = new ArrayList<>();
list.add("Java");
list.add("Python");
list.add("C++");
```

```
System.out.println(list.get(1)); // Output: Python
```

### Use Cases:

- When you need to frequently access elements by index.
  - When you don't perform many insertions or deletions (other than at the end).
  - When maintaining the insertion order is important.
- 

## 2. LinkedList (Implements List and Queue Interfaces)

### Key Features:

- **Doubly Linked List:** `LinkedList` uses a doubly linked list internally, where each element (node) contains references to both the previous and next elements.
- **Efficient Insertions/Deletions:** Insertion and removal of elements can be done efficiently at both ends (head and tail), especially for large lists.
- **Indexed Access:** Provides index-based access, but not as efficient as `ArrayList`.
- **Allows Duplicates:** Like `ArrayList`, it allows duplicate elements and maintains the order of insertion.
- **Null Elements:** Allows `null` elements.

### Performance:

- **Get (Access by index):**  $O(n)$  (linear time), since it needs to traverse the list.
- **Add (to the end or beginning):**  $O(1)$  (constant time), fast at both ends.
- **Remove:**  $O(1)$  for removing from the beginning or end, but  $O(n)$  for removing from arbitrary positions.
- **Insert:**  $O(1)$  for inserting at the beginning or end, but  $O(n)$  for arbitrary positions.

### Example:

```
java
```

```
Copy
LinkedList<String> list = new LinkedList<>();
list.add("Java");
list.add("Python");
list.add("C++");

System.out.println(list.get(1)); // Output: Python
```

### Use Cases:

- When you need frequent insertions and deletions at the beginning or end of the list.
  - When you don't need fast random access to elements (e.g., access by index).
- 

## 3. HashSet (Implements `Set` Interface)

### Key Features:

- **Hash Table:** HashSet is backed by a **hash table** (actually HashMap is used internally). It does not guarantee any specific order of elements.
- **No Duplicates:** It does not allow duplicate elements. If an attempt is made to add a duplicate element, it simply returns `false`.
- **Null Elements:** Allows at most one `null` element.
- **Unordered:** The elements are unordered; the order in which elements are retrieved is not predictable.

### Performance:

- **Add:**  $O(1)$  on average (constant time). However, it may take  $O(n)$  in rare cases (e.g., when rehashing is needed).
- **Contains:**  $O(1)$  on average (constant time).
- **Remove:**  $O(1)$  on average (constant time).

### Example:

```
java
Copy
HashSet<String> set = new HashSet<>();
set.add("Java");
set.add("Python");
set.add("C++");

System.out.println(set.contains("Python")); // Output: true
set.remove("Python");
System.out.println(set.contains("Python")); // Output: false
```

### Use Cases:

- When you need to store unique elements and do not care about the order.
- When you frequently check for membership (i.e., contains operation).
- When insertions and removals are expected to be frequent and fast.

## 4. HashMap (Implements Map Interface)

### Key Features:

- **Hash Table:** HashMap is backed by a **hash table**. It stores key-value pairs and allows fast retrieval of values based on their keys.
- **No Duplicate Keys:** HashMap does not allow duplicate keys, but values can be duplicated.
- **Unordered:** The keys are unordered, meaning that the order in which key-value pairs are stored or retrieved is not guaranteed.
- **Null Keys/Values:** Allows one `null` key and multiple `null` values.

### Performance:

- **Put** (insertion): **O(1)** on average (constant time), but may be **O(n)** in rare cases due to rehashing.
- **Get** (retrieval by key): **O(1)** on average (constant time).
- **Contains** (key or value): **O(1)** on average (constant time).
- **Remove**: **O(1)** on average (constant time).

### Example:

```
java
Copy
HashMap<Integer, String> map = new HashMap<>();
map.put(1, "Java");
map.put(2, "Python");
map.put(3, "C++");

System.out.println(map.get(1)); // Output: Java
map.remove(2);
System.out.println(map.containsKey(2)); // Output: false
```

### Use Cases:

- When you need to store key-value pairs and quickly look up values based on their keys.
- When the order of key-value pairs does not matter.
- When you need to frequently check for the existence of a key and retrieve corresponding values.

## 5. Other Useful Classes in the Collections Framework

### TreeSet (Implements `Set` Interface)

- **Sorted Set:** A set that stores elements in sorted order. It uses a red-black tree internally.
- **No Duplicates:** Like `HashSet`, it does not allow duplicates.
- **Order:** Elements are sorted according to their natural ordering or by a comparator.
- **Performance:** Operations like add, remove, and contains take  $O(\log n)$  time due to the underlying tree structure.

### Example:

```
java
Copy
```

```
TreeSet<String> set = new TreeSet<>();
set.add("Java");
set.add("Python");
set.add("C++");
System.out.println(set); // Output: [C++, Java, Python] (sorted order)
```

### TreeMap (Implements Map Interface)

- **Sorted Map:** A map that stores key-value pairs in a sorted order based on the keys.
- **No Duplicate Keys:** Like `HashMap`, it does not allow duplicate keys.
- **Order:** The keys are sorted according to their natural ordering or by a comparator.
- **Performance:** Operations like put, get, remove, and contains take **O(log n)** time.

#### Example:

```
java
Copy
TreeMap<Integer, String> map = new TreeMap<>();
map.put(1, "Java");
map.put(3, "Python");
map.put(2, "C++");
System.out.println(map); // Output: {1=Java, 2=C++, 3=Python} (sorted by keys)
```

### LinkedHashMap (Implements Map Interface)

- **Ordered Map:** A map that maintains the insertion order of key-value pairs.
- **Performance:** It provides the same **O(1)** performance for get and put operations as `HashMap`, but with the added benefit of predictable iteration order.

#### Example:

```
java
Copy
LinkedHashMap<Integer, String> map = new LinkedHashMap<>();
map.put(1, "Java");
```

```
map.put(2, "Python");
map.put(3, "C++");
System.out.println(map); // Output: {1=Java, 2=Python, 3=C++} (insertion order)
```

---

## Conclusion

Each collection class (`ArrayList`, `LinkedList`, `HashSet`, `HashMap`, etc.) is optimized for specific use cases, providing different trade-offs in terms of performance and functionality. Here's a quick summary:

- **ArrayList**: Use when you need fast random access by index and don't perform many insertions or deletions.
- **LinkedList**: Use when you need efficient insertions and deletions at the beginning or end of the list.
- **HashSet**: Use when you need to store unique elements and don't care about their order.
- **HashMap**: Use when you need to store key-value pairs and perform fast lookups based on keys.
- **TreeSet**: Use when you need a sorted set of elements.
- **TreeMap**: Use when you need a sorted map of key-value pairs.

Choosing the right implementation of the collection classes can improve the efficiency and readability of your Java programs.

In Java, **Iterators** are objects that allow you to traverse through the elements of a collection (such as `List`, `Set`, or `Queue`). They are part of the Java Collections Framework and are widely used to access and manipulate data in a collection in a systematic way.

# What is an Iterator?

An **Iterator** is an object that provides methods to iterate over the elements in a collection one by one, while abstracting away the underlying data structure. It allows you to access and remove elements from the collection during iteration. Iterators are a safer and more flexible alternative to using traditional `for` loops when iterating over collections.

---

## Iterator Interface

The `Iterator` interface is part of the `java.util` package and is implemented by most of the collections in the Java Collections Framework. It provides methods for iterating over a collection.

### Key Methods of the `Iterator` Interface:

1. `boolean hasNext()`:
  - o Returns `true` if there are more elements to iterate over in the collection.
  - o Returns `false` if all elements have been traversed.
  - o **Example:** `while (iterator.hasNext())`
2. `E next()`:
  - o Returns the next element in the collection.
  - o Throws a `NoSuchElementException` if there are no more elements to return.
  - o After calling `next()`, the iterator moves forward to the next element.
  - o **Example:** `E element = iterator.next()`
3. `void remove()`:
  - o Removes the last element returned by the iterator from the collection.
  - o This method can be called only once per call to `next()`.

- Throws an `IllegalStateException` if `next()` has not been called, or `remove()` has already been called after the last call to `next()`.
  - **Example:** `iterator.remove()`
- 

## Iterator Example

Here's an example of how you can use an `Iterator` to traverse through a `List` in Java:

```
java
Copy
import java.util.ArrayList;
import java.util.Iterator;

public class IteratorExample {
    public static void main(String[] args) {
        // Create a list and add some elements
        ArrayList<String> list = new ArrayList<>();
        list.add("Java");
        list.add("Python");
        list.add("C++");

        // Obtain the iterator for the list
        Iterator<String> iterator = list.iterator();

        // Iterate over the list
        while (iterator.hasNext()) {
            String element = iterator.next(); // Get the next element
            System.out.println(element);
        }

        // Remove an element using iterator
        iterator = list.iterator(); // Re-create the iterator since it's exhausted
        while (iterator.hasNext()) {
            String element = iterator.next();
            if ("Python".equals(element)) {
```

```
        iterator.remove(); // Remove "Python"
    }
}

// Print list after removal
System.out.println("List after removal: " + list);
}
}
```

## Output:

```
mathematica
Copy
Java
Python
C++
List after removal: [Java, C++]
```

## Important Notes:

- The iterator is **fail-fast** in most cases. This means if the collection is modified while an iterator is traversing (except through the iterator itself), it will throw a `ConcurrentModificationException`.
- The `remove()` method is optional and not supported by all collections (e.g., immutable collections like `List.of()`).

---

## Types of Iterators in Java

Java provides several variations of iterators depending on the type of collection being used. Here are some key iterators in the Java Collections Framework:

### 1. **ListIterator** (For `List` collections like `ArrayList`, `LinkedList`)

- A **ListIterator** extends **Iterator** and allows bidirectional iteration, meaning you can traverse the list in both **forward** and **reverse** directions.
- It also provides additional methods to modify the list during iteration (like adding elements or setting values).

### Key Methods in ListIterator:

- **boolean hasPrevious()**: Checks if there's an element before the current position.
- **E previous()**: Returns the previous element in the list.
- **void add(E e)**: Adds an element to the list at the current position.
- **void set(E e)**: Replaces the last element returned by `next()` or `previous()` with the specified element.

### Example of ListIterator:

```
java
Copy
import java.util.*;

public class ListIteratorExample {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>(Arrays.asList("Java", "Python", "C++"));

        ListIterator<String> listIterator = list.listIterator();

        // Iterate forward
        while (listIterator.hasNext()) {
            System.out.println(listIterator.next());
        }

        // Iterate backward
        while (listIterator.hasPrevious()) {
            System.out.println(listIterator.previous());
        }
    }
}
```

## 2. Iterator for Sets (For **Set** collections like **HashSet**, **LinkedHashSet**, **TreeSet**)

- **Iterator** for `Set` works similarly to that for `List` but without the concept of indexes. Since `Set` does not maintain any order, the order in which elements are returned by the iterator may not be the same as the insertion order.

#### Example for `Set`:

```
java
Copy
import java.util.*;

public class SetIteratorExample {
    public static void main(String[] args) {
        Set<String> set = new HashSet<>(Arrays.asList("Java", "Python", "C++"));

        Iterator<String> iterator = set.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }
}
```

### 3. Concurrent Iterators (For concurrent collections like `CopyOnWriteArrayList`)

- Java also provides **Concurrent Iterators**, which are part of concurrent collections in the `java.util.concurrent` package. These iterators are designed for safe iteration in multi-threaded environments.
- They provide an **iterator** that allows reading the elements even if the underlying collection is being modified by other threads, although some restrictions may apply (like not allowing certain modifications).

---

### Iterator vs. Enhanced for-loop (for-each loop)

While the traditional `Iterator` provides more flexibility (e.g., `remove()` method), the enhanced `for` loop (also known as the **for-each loop**) is a simpler, more concise way to iterate through collections in most cases.

### **Example: Using the Enhanced for loop (for-each loop):**

```
java
Copy
List<String> list = new ArrayList<>(Arrays.asList("Java", "Python", "C++"));
for (String language : list) {
    System.out.println(language);
}
```

### **Comparison:**

- **Iterator:** More flexible, allows removing elements during iteration, and works with all collections.
- **Enhanced for-loop:** Simpler syntax but doesn't allow removing elements from the collection.

### **When to Use Iterators?**

1. **When you need to remove elements while iterating:** The enhanced `for` loop does not support removal of elements during iteration, but `Iterator` provides a `remove()` method to do so.
2. **When you need to traverse a collection in a controlled and systematic manner:** Iterators give more control, especially when modifying the collection or iterating over different types of collections (like `Set`, `List`, etc.).
3. **When you need to avoid `ConcurrentModificationException`:** When modifying the collection inside the loop, use an iterator to avoid exceptions.

### **Conclusion**

- **Iterators** provide a standard way to traverse collections in Java, allowing sequential access to elements without exposing the underlying structure.
- The basic `Iterator` interface allows one-directional iteration with `hasNext()`, `next()`, and `remove()` methods.
- More specialized iterators like `ListIterator` offer bidirectional iteration and additional methods for modifying the collection.

- Iterators provide a more flexible and controlled way of iterating over collections, especially when removal or other modifications are necessary during iteration.

Understanding iterators is essential for working with Java collections effectively, as they provide a safe, convenient, and powerful mechanism for traversing collections.

# Core Java Notes By AJAY RAZZ

**Concurrency** in Java refers to the ability of the program to execute multiple tasks or processes simultaneously, improving the efficiency of execution, especially on multi-core processors. Java provides built-in support for managing concurrency through threads, synchronization mechanisms, and concurrent utilities.

To understand concurrency in Java, let's break it down into key concepts:

---

# 1. What is Concurrency?

Concurrency is the ability of a program to run multiple tasks or threads independently but in overlapping periods, instead of sequentially. Concurrency is not necessarily the same as parallelism, which involves executing tasks simultaneously on multiple processors or cores. Concurrency simply allows multiple tasks to progress without necessarily being executed at the same time.

In a **single-core** CPU, concurrent tasks are managed by quickly switching between them, giving the illusion of parallelism. In a **multi-core** CPU, multiple tasks can run at the same time, which is true parallelism.

---

## 2. Threads in Java

At the core of concurrency in Java are **threads**. A thread is the smallest unit of execution within a process. In Java, threads allow a program to perform multiple operations simultaneously. Each thread runs its own code independently but shares the resources of the parent process.

- **Main thread:** Every Java program starts execution with a single main thread.
- **Child threads:** Additional threads can be created by the program to perform tasks in parallel with the main thread.

### Thread Lifecycle:

A thread in Java follows a specific lifecycle:

1. **New:** A thread is created but has not yet started.
2. **Runnable:** The thread is ready for execution but might not be running yet.
3. **Blocked/Waiting:** The thread is waiting for a resource or condition to continue.
4. **Terminated:** The thread has completed its execution or has been terminated.

## Creating a Thread in Java:

There are two primary ways to create a thread in Java:

1. **By Extending the Thread class.**
2. **By Implementing the Runnable interface.**

Example 1: Using Thread Class:

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

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

Example 2: Using Runnable Interface:

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

```
public class RunnableExample {  
    public static void main(String[] args) {  
        MyRunnable myRunnable = new MyRunnable();  
        Thread thread = new Thread(myRunnable);  
        thread.start(); // Starts the thread  
    }  
}
```

---

### 3. Thread Synchronization

In a multithreaded environment, multiple threads may try to access shared resources simultaneously, leading to data inconsistency. To avoid such issues, **synchronization** is used. Synchronization ensures that only one thread can access a shared resource at a time.

#### Synchronized Methods

You can use the `synchronized` keyword to ensure that only one thread can execute a particular method at a time. When a method is marked as synchronized, the thread holds a lock on the object until it finishes executing.

```
java  
Copy  
class Counter {  
    private int count = 0;  
  
    // Synchronized method to ensure thread-safe access to the count variable  
    public synchronized void increment() {  
        count++;  
    }  
  
    public int getCount() {  
        return count;  
    }  
}
```

#### Synchronized Blocks

Instead of synchronizing the entire method, you can use synchronized blocks to control access to smaller critical sections of code.

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

---

## 4. Deadlock

Deadlock occurs when two or more threads are blocked forever, waiting for each other to release resources. It usually happens when two threads hold locks on resources that the other needs to proceed.

### Example of Deadlock:

```
java
Copy
class A {
    synchronized void methodA(B b) {
        b.last();
    }

    synchronized void last() {}
}

class B {
    synchronized void methodB(A a) {
        a.last();
    }

    synchronized void last() {}
}
```

In this example, if A and B each hold a lock on the other, both threads will be blocked forever.

### Preventing Deadlock:

- Avoid nested locks.
  - Lock resources in a consistent order.
  - Use timeout-based locking (e.g., `ReentrantLock.tryLock()`).
- 

## 5. Executor Framework

The **Executor Framework** (introduced in Java 5) simplifies managing threads and task execution. It abstracts the process of creating and managing threads and provides thread pools for efficient resource management.

### Key Components:

1. **Executor**: A simple interface for executing tasks.
2. **ExecutorService**: A more advanced interface that can manage tasks, track their progress, and shut down the thread pool.
3. **ThreadPoolExecutor**: A concrete implementation of `ExecutorService`.
4. **ScheduledExecutorService**: A service that can schedule tasks with fixed-rate or fixed-delay execution.

### Example of ExecutorService:

```
java
Copy
import java.util.concurrent.*;

public class ExecutorExample {
    public static void main(String[] args) {
        ExecutorService executorService = Executors.newFixedThreadPool(2);

        Runnable task = () -> System.out.println("Task is running");
```

```
        executorService.submit(task); // Submit a task to the executor
        executorService.submit(task);

        executorService.shutdown(); // Shutdown the executor service
    }
}
```

- **ThreadPoolExecutor** manages a pool of threads, which can be reused to execute multiple tasks, improving efficiency.
- The **ScheduledExecutorService** allows tasks to be scheduled to run at fixed intervals, much like cron jobs.

---

## 6. Java Concurrency Utilities (java.util.concurrent)

Java provides a set of concurrency utilities in the `java.util.concurrent` package. These utilities make working with multithreading easier and more efficient.

### Key Classes and Interfaces:

#### 1. **ReentrantLock**:

- Unlike the `synchronized keyword`, `ReentrantLock` allows more flexible control over locking.
- It provides methods like `lock()`, `unlock()`, and `tryLock()` for acquiring and releasing locks.

```
java
Copy
ReentrantLock lock = new ReentrantLock();
lock.lock(); // Acquire lock
try {
    // Critical section
} finally {
    lock.unlock(); // Release lock
}
```

#### 2. **CountDownLatch**:

- Used to block threads until a certain number of operations are complete. It allows one or more threads to wait until a set of operations completes.

```
java
Copy
CountDownLatch latch = new CountDownLatch(3); // Wait for 3 threads
// Perform tasks
latch.await(); // Wait until the latch count reaches 0
```

### 3. **Semaphore:**

- Controls access to a shared resource by a fixed number of threads. It maintains a set of permits and allows threads to acquire and release them.

```
java
Copy
Semaphore semaphore = new Semaphore(1); // Only one thread allowed
semaphore.acquire(); // Acquire a permit
// Critical section
semaphore.release(); // Release the permit
```

### 4. **CyclicBarrier:**

- Synchronizes multiple threads to wait for each other at a common barrier point.

```
java
Copy
CyclicBarrier barrier = new CyclicBarrier(3); // Wait for 3 threads
barrier.await(); // All threads reach this point before continuing
```

### 5. **ExecutorService** (as discussed earlier):

- Manages the execution of tasks asynchronously.

---

## 7. Best Practices for Concurrency

1. **Minimize Synchronization:** Overuse of synchronization can degrade performance. Synchronize only the critical sections of code that require access to shared resources.
  2. **Avoid Shared Mutable State:** Where possible, use immutable objects or ensure proper synchronization when sharing mutable state between threads.
  3. **Use Thread Pools:** Creating a new thread for each task is inefficient. Use thread pools to reuse threads and reduce the overhead of thread creation.
  4. **Use High-Level Concurrency Utilities:** Prefer using `ExecutorService`, `CountDownLatch`, `Semaphore`, and other classes from the `java.util.concurrent` package for easier and more efficient concurrency management.
- 

## Conclusion

Concurrency in Java is a powerful feature that helps improve performance and responsiveness, especially when dealing with time-consuming tasks or parallel operations. Java provides various tools and frameworks to handle multithreading, from the basic `Thread` class to advanced utilities in the `java.util.concurrent` package.

- **Threads** allow for concurrent execution of tasks.
- **Synchronization** ensures that shared resources are accessed in a thread-safe manner.
- The **Executor Framework** and **Concurrency Utilities** simplify thread management, resource sharing, and synchronization.

By understanding and properly applying these concurrency concepts, Java developers can build highly efficient, scalable, and thread-safe applications.

In Java, **threads** are fundamental units of execution. A **thread** is a lightweight process that has its own path of execution and allows concurrent execution of tasks. Threads enable multitasking and parallel execution of programs, making Java applications more responsive and efficient, especially for time-consuming operations.

# Basics of Threads in Java

## 1. What is a Thread?

- A **thread** is a single sequence of execution within a program. In a Java program, multiple threads can run concurrently, sharing resources like memory and CPU time, which enables parallel execution.
- Every Java program has at least one thread, called the **main thread**, which starts executing the `main()` method. Additional threads can be created to perform other tasks concurrently.

## 2. Thread vs Process:

- A **process** is an independent program with its own memory space, while a **thread** is a part of a process. Multiple threads within a process share the same memory space, which allows them to communicate with each other easily.
- Threads within the same process can share data, unlike processes that do not share memory space.

---

## Creating Threads in Java

In Java, there are two main ways to create threads:

1. **By Extending the `Thread` class**
2. **By Implementing the `Runnable` interface**

Let's look at both methods.

---

### 1. Creating Threads by Extending the `Thread` Class

Java provides a `Thread` class to represent a thread. You can create a custom thread by extending the `Thread` class and overriding its `run()` method. The `run()` method contains the code that will be executed by the thread when it starts.

#### Steps:

1. **Create a subclass of `Thread`.**
2. **Override the `run()` method** to define the task to be executed by the thread.
3. **Create an object of the subclass and call its `start()` method** to begin the thread execution.

#### Example:

```
java
Copy
class MyThread extends Thread {
    public void run() {
        // Task that the thread will execute
        System.out.println("Thread is running");
    }
    public static void main(String[] args) {
        MyThread thread = new MyThread(); // Create thread object
        thread.start(); // Start the thread (calls run method)
    }
}
```

#### Explanation:

- When you call the `start()` method, the thread is placed in the **Runnable state**. The `run()` method is then executed in a separate thread of control.
- The `run()` method contains the code you want the thread to execute, and the thread terminates once the `run()` method finishes execution.

## 2. Creating Threads by Implementing the `Runnable` Interface

Another way to create a thread is by implementing the `Runnable` interface. The `Runnable` interface has a single method, `run()`, which you override to define the task for the thread.

**Steps:**

1. **Implement the `Runnable` interface** and override its `run()` method.
2. **Create a `Thread` object**, passing the `Runnable` object to its constructor.
3. **Start the thread** by calling the `start()` method on the `Thread` object.

**Example:**

```
java
Copy
class MyRunnable implements Runnable {
    public void run() {
        // Task that the thread will execute
        System.out.println("Thread is running");
    }

    public static void main(String[] args) {
        MyRunnable task = new MyRunnable(); // Create a Runnable object
        Thread thread = new Thread(task); // Create a Thread object with Runnable
        thread.start(); // Start the thread (calls run method)
    }
}
```

**Explanation:**

- The `Runnable` interface separates the task (logic) from the thread management (starting and stopping). This is beneficial because you can create a `Runnable` object and pass it to multiple `Thread` objects if needed.
- The `run()` method in this case performs the task when the thread is started.

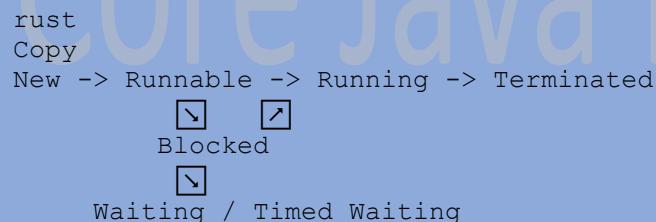
---

## Thread Lifecycle

The lifecycle of a thread in Java goes through several states, each representing the stage the thread is in. The key states are:

1. **New**: A thread is created but has not yet started execution.
2. **Runnable**: After calling the `start()` method, the thread becomes **runnable**, meaning it is ready to run but may not be currently running due to CPU scheduling.
3. **Blocked**: The thread is blocked while waiting for a resource (e.g., I/O operations or synchronization).
4. **Waiting**: The thread is waiting for another thread to perform a particular action (e.g., waiting for a signal or timeout).
5. **Timed Waiting**: The thread is waiting for a specified amount of time (e.g., `sleep()` or `join()`).
6. **Terminated**: The thread has finished execution (either by completing the `run()` method or being terminated by an exception).

### Thread States Diagram:



---

## Starting and Stopping Threads

1. **Starting a Thread**:
  - To start a thread, you call its `start()` method, which triggers the thread to execute the `run()` method.
  - Example: `thread.start();`
2. **Stopping a Thread**:
  - A thread stops executing when the `run()` method finishes or if it's interrupted.

- To stop a thread explicitly, you cannot directly call `stop()` (it's deprecated). Instead, you use an interrupt mechanism or a flag to signal the thread to stop.

### Example of stopping a thread using a flag:

```
java
Copy
class MyThread extends Thread {
    private boolean running = true;

    public void run() {
        while (running) {
            System.out.println("Thread is running");
            try {
                Thread.sleep(1000); // Simulating some work
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt(); // Handle interruption
            }
        }
    }

    public void stopThread() {
        running = false;
    }

    public static void main(String[] args) throws InterruptedException {
        MyThread thread = new MyThread();
        thread.start();
        Thread.sleep(5000); // Let the thread run for 5 seconds
        thread.stopThread(); // Stop the thread
    }
}
```

---

## Thread Methods

Java provides several methods in the `Thread` class to control thread behavior:

1. `start()`: Starts the thread's execution (calls `run()`).
  2. `run()`: Contains the code that the thread will execute.
  3. `sleep(long milliseconds)`: Makes the thread pause for a specified number of milliseconds.
  4. `join()`: Makes the calling thread wait for the thread to finish.
  5. `interrupt()`: Interrupts a thread, which can be used to stop it (or signal it to stop).
  6. `getName()` and `setName(String name)`: Get or set the name of the thread.
  7. `getPriority()` and `setPriority(int priority)`: Get or set the thread's priority.
- 

## Thread Scheduling and Priorities

Java threads can be scheduled based on priorities, though the actual behavior depends on the operating system and JVM. A thread's priority is an integer value (between `Thread.MIN_PRIORITY` and `Thread.MAX_PRIORITY`, with a default of `Thread.NORM_PRIORITY`).

### Example of setting thread priority:

```
java
Copy
Thread thread = new Thread(new MyRunnable());
thread.setPriority(Thread.MAX_PRIORITY); // Set thread priority to maximum
thread.start();
```

Thread priorities help the JVM determine which thread should be executed first, but there is no guarantee that a high-priority thread will always run before low-priority threads.

---

## Common Thread Methods

- `Thread.sleep(long millis)`: Causes the current thread to pause for a specified amount of time.
- `Thread.yield()`: Suggests to the thread scheduler that the current thread is willing to yield its current use of the CPU.

- `Thread.currentThread()`: Returns a reference to the currently executing thread.
- 

## Conclusion

Threads are a core concept in Java that enables multitasking and parallelism. By using threads, Java applications can perform multiple operations at the same time, making them more efficient and responsive. Java provides two main ways to create threads:

- **Extending the `Thread` class.**
- **Implementing the `Runnable` interface.**

In addition, Java offers methods for controlling thread execution, synchronization, and handling multiple threads effectively. Proper thread management is crucial for building high-performance and thread-safe Java applications.

Core Java Notes By AJAY RAZZ

# Synchronization in Java

Synchronization in Java is a mechanism that ensures that only one thread can access a resource (such as a method or block of code) at a time. This is crucial in a multithreaded environment to prevent data inconsistency and ensure thread safety when multiple threads access shared resources.

## Why is Synchronization Needed?

In a multithreaded program, multiple threads may try to access and modify shared data simultaneously. If proper synchronization is not applied, it can lead to issues like:

- **Data corruption:** One thread could overwrite data that another thread is in the process of updating.
- **Race conditions:** The behavior of the program can depend on the order in which threads are executed, leading to unpredictable results.

### Types of Synchronization in Java

1. **Method-level Synchronization:** You can synchronize an entire method to ensure that only one thread can execute the method at a time.

```
java
Copy
public synchronized void myMethod() {
    // Code that should be accessed by only one thread at a time
}
```

When a method is marked with the `synchronized` keyword, a thread must acquire the lock for the object (or class, if it's a static method) before executing it. This prevents other threads from accessing the method until the first thread finishes.

2. **Block-level Synchronization:** If only a specific block of code needs to be synchronized, you can use the `synchronized` keyword inside the method to specify a smaller scope for synchronization.

```
java
Copy
public void myMethod() {
    // Some code
    synchronized (this) {
        // Critical code block
    }
}
```

```
synchronized(this) {  
    // Critical section code  
}  
  
// Some code  
}
```

In this case, the thread must acquire the lock for the object `this` before executing the synchronized block. You can also synchronize on a different object if needed.

3. **Static Method Synchronization:** When synchronizing static methods, the lock is applied on the `Class` object itself rather than an instance of the class.

```
java  
Copy  
public static synchronized void staticMethod() {  
    // Static method code that only one thread can execute at a time  
}
```

Here, the class object (`Class`) is locked, ensuring only one thread can access any synchronized static method of the class at a time.

## Synchronized Blocks and Locks

You can also explicitly control the synchronization using `ReentrantLock` or `Lock` interface, which provides more flexibility (like timed locks, fairness, etc.).

```
java  
Copy  
Lock lock = new ReentrantLock();  
lock.lock();  
try {  
    // Critical section code  
} finally {  
    lock.unlock();  
}
```

## Deadlock

Be careful with synchronization to avoid **deadlocks**. A deadlock occurs when two or more threads are blocked forever because each is waiting for the other to release a lock. To avoid deadlocks:

- Always acquire locks in a consistent order.
- Use timeout mechanisms when acquiring locks.

## Key Points:

- **Thread Safety:** Synchronization ensures that shared data is not accessed by multiple threads concurrently, which can lead to inconsistent results.
- **Performance:** Excessive synchronization can slow down the program, as threads will spend time waiting to acquire locks.
- **Intrinsic Locks:** Every object in Java has an implicit lock that is used when a method or block is synchronized.

In summary, synchronization is crucial for ensuring thread safety in concurrent programming, but it needs to be used carefully to avoid performance problems and issues like deadlocks.

**Thread safety** in Java refers to the ability of a piece of code or a data structure to function correctly when accessed by multiple threads concurrently, without causing data corruption or unpredictable behavior. In other words, thread-safe code ensures that shared resources are properly managed so that threads do not interfere with each other, leading to consistent and correct results.

## Key Concepts of Thread Safety

1. **Shared Resources:** When multiple threads access shared resources (e.g., variables, objects, files, etc.), there's a risk of data inconsistency if the threads are modifying the resources simultaneously without proper synchronization. Thread safety ensures that these resources are accessed and modified in a way that prevents conflicts.
2. **Race Condition:** A race condition occurs when two or more threads attempt to modify shared data at the same time. If the proper synchronization is not implemented, this can lead to incorrect results, as the final value of the data depends on the order in which the threads execute, which is unpredictable.

### Methods to Achieve Thread Safety

1. **Synchronization:** One of the most common techniques for ensuring thread safety is synchronization. This involves using locks to restrict access to critical sections of the code, ensuring that only one thread can execute that section at a time.
  - o **Method Synchronization:** You can synchronize an entire method to make sure only one thread can access it at a time.

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

- o **Block Synchronization:** You can also synchronize specific blocks of code inside a method to reduce the scope of synchronization.

```
java
Copy
public void increment() {
```

```
        synchronized(this) {
            count++;
        }
    }
```

2. **Volatile Keyword:** The `volatile` keyword ensures that a variable is always read from and written to the main memory, instead of being cached in a thread's local memory. This is useful for simple flag variables or state variables where you want to ensure visibility across threads.

```
java
Copy
private volatile boolean flag;
```

It doesn't provide atomicity (it doesn't solve issues like race conditions), but ensures that the most up-to-date value of the variable is used by all threads.

3. **Immutable Objects:** Immutable objects are thread-safe because their state cannot be changed once they are created. If an object's state cannot be altered, then it's inherently safe from race conditions.

For example, `String` and `Integer` in Java are immutable. Once created, their values cannot be changed. You can create your own immutable class by ensuring that all fields are `final` and not modifiable.

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

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

    public String getName() { return name; }
    public int getAge() { return age; }
}
```

4. **Using Concurrency Utilities:** Java provides a set of classes in the `java.util.concurrent` package to make it easier to manage thread safety. Some of the key classes include:

- **ReentrantLock:** A more flexible lock than `synchronized`. It provides additional features like timed locks and fairness policies.

```
java
Copy
Lock lock = new ReentrantLock();
lock.lock();
try {
    // Critical section code
} finally {
    lock.unlock();
}
```

- **Atomic Classes:** Classes like `AtomicInteger`, `AtomicLong`, `AtomicReference`, etc., provide atomic operations that ensure thread safety without needing explicit synchronization.

```
java
Copy
AtomicInteger counter = new AtomicInteger(0);
counter.incrementAndGet(); // Thread-safe increment
```

- **CopyOnWrite Collections:** For situations where writes are rare, but reads are frequent, classes like `CopyOnWriteArrayList` provide a thread-safe way to handle collections.

```
java
Copy
List<Integer> list = new CopyOnWriteArrayList<>();
list.add(1);
list.add(2);
```

5. **Thread-Local Storage:** A **thread-local variable** is a variable that is unique to each thread, meaning that each thread has its own independent copy of the variable. The `ThreadLocal` class in Java allows you to create thread-local variables.

```
java
Copy
```

```
private ThreadLocal<Integer> threadLocalCounter = ThreadLocal.withInitial(() -> 0);

public void increment() {
    threadLocalCounter.set(threadLocalCounter.get() + 1);
}
```

6. **Using Synchronized Collections:** Some of Java's collection classes are thread-safe, such as `Vector` and `Hashtable`, but they tend to be slower because every method is synchronized. However, Java provides other options like `Collections.synchronizedList()` to wrap existing collections into thread-safe versions.

```
java
Copy
List<String> synchronizedList = Collections.synchronizedList(new ArrayList<>());
```

## Common Pitfalls and Considerations

1. **Deadlock:** Deadlock occurs when two or more threads are blocked forever because each thread is holding a lock and waiting for the other to release it. Deadlocks can be avoided by ensuring that threads acquire locks in a consistent order.
2. **Performance Issues:** Excessive synchronization can cause performance bottlenecks, especially if threads are frequently blocked while waiting for access to synchronized code. In such cases, you should consider alternatives like `Lock` objects, which can provide more flexible and efficient synchronization.
3. **Atomicity vs. Visibility:** Achieving thread safety often requires addressing two separate issues:
  - o **Atomicity:** Ensuring that a thread's operations on shared data are done in one uninterrupted sequence.
  - o **Visibility:** Ensuring that updates to shared data by one thread are visible to all other threads.

Using tools like `volatile`, `synchronized`, and `Atomic` classes can help manage both.

## Conclusion

Thread safety is essential for ensuring that a program behaves correctly in a multithreaded environment. Java provides several mechanisms, such as synchronization, atomic variables, and thread-local storage, to achieve thread safety. By carefully managing shared resources and using concurrency utilities, you can avoid common problems like race conditions and deadlocks, ensuring that your program is robust, reliable, and efficient in handling concurrent tasks.

**File Handling** in Java refers to the process of reading from and writing to files in the file system. Java provides a rich set of APIs in the `java.io` package that makes it easy to perform file operations such as creating, reading, writing, and deleting files.

## Types of File Operations in Java

1. **Creating Files:** Creating new files or checking if a file already exists.
2. **Reading Files:** Accessing and reading data from files.
3. **Writing to Files:** Saving data to files.
4. **Appending to Files:** Adding data to the end of an existing file without overwriting.
5. **Deleting Files:** Removing files from the filesystem.
6. **Renaming or Moving Files:** Renaming or moving files within directories.

## Basic File Handling in Java

### 1. Creating and Checking Files

You can create a new file or check if a file already exists using the `File` class.

```
java
Copy
import java.io.File;
import java.io.IOException;

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

        // Check if file exists
        if (file.exists()) {
            System.out.println("File already exists.");
        } else {
            try {
                // Create a new file
                if (file.createNewFile()) {
                    System.out.println("File created: " + file.getName());
                }
            } catch (IOException e) {
                System.out.println("An error occurred while creating the file.");
            }
        }
    }
}
```

```
        } else {
            System.out.println("File already exists.");
        }
    } catch (IOException e) {
        System.out.println("An error occurred.");
        e.printStackTrace();
    }
}
}
```

## 2. Reading Files

You can read files using classes like `FileReader`, `BufferedReader`, and `Scanner`. Here's how to read a file line by line using `BufferedReader`.

```
java
Copy
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class ReadFile {
    public static void main(String[] args) {
        String filename = "example.txt";
        try (BufferedReader reader = new BufferedReader(new FileReader(filename))) {
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println(line); // Print each line
            }
        } catch (IOException e) {
            System.out.println("An error occurred while reading the file.");
            e.printStackTrace();
        }
    }
}
```

## 3. Writing to Files

You can write to a file using `FileWriter` or `BufferedWriter`. Here's an example of writing text to a file using `FileWriter`.

```
java
Copy
import java.io.FileWriter;
import java.io.IOException;

public class WriteFile {
    public static void main(String[] args) {
        try (FileWriter writer = new FileWriter("output.txt")) {
            writer.write("Hello, world!");
            writer.write("\nThis is a test file.");
        } catch (IOException e) {
            System.out.println("An error occurred while writing to the file.");
            e.printStackTrace();
        }
    }
}
```

#### 4. Appending to Files

To append to a file (i.e., adding new data without overwriting existing content), you can pass a `true` value to the `FileWriter` constructor.

```
java
Copy
import java.io.FileWriter;
import java.io.IOException;

public class AppendToFile {
    public static void main(String[] args) {
        try (FileWriter writer = new FileWriter("output.txt", true)) {
            writer.write("\nThis is appended text.");
        } catch (IOException e) {
            System.out.println("An error occurred while appending to the file.");
            e.printStackTrace();
        }
    }
}
```

## 5. Deleting Files

You can delete a file using the `delete()` method of the `File` class.

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

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

        if (file.delete()) {
            System.out.println("File deleted successfully.");
        } else {
            System.out.println("Failed to delete the file.");
        }
    }
}
```

## 6. Renaming or Moving Files

Renaming or moving a file can be done using the `renameTo()` method of the `File` class.

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

public class RenameFile {
    public static void main(String[] args) {
        File oldFile = new File("oldfile.txt");
        File newFile = new File("newfile.txt");

        if (oldFile.renameTo(newFile)) {
            System.out.println("File renamed successfully.");
        } else {
            System.out.println("Failed to rename the file.");
        }
    }
}
```

```
        }
    }
}
```

## Classes for File Handling

1. **File Class:**
  - o Used for representing files and directories.
  - o Allows you to perform basic operations like creating, deleting, renaming files, and checking file attributes (like size, permissions, etc.).
  - o Example methods: `exists()`, `createNewFile()`, `delete()`, `renameTo()`, `length()`, etc.
2. **FileReader and FileWriter:**
  - o Used for reading from and writing to text files, character-based streams.
  - o Example: `FileReader` is used to read files, and `FileWriter` is used to write characters to a file.
3. **BufferedReader and BufferedWriter:**
  - o These provide buffering to read or write data more efficiently by reducing the number of I/O operations.
  - o `BufferedReader` is commonly used for reading files line by line.
  - o `BufferedWriter` is used to write data to a file efficiently.
4. **PrintWriter:**
  - o A more convenient class for writing formatted text to files, including automatic line flushing.
5. **Scanner:**
  - o Used for parsing input from files or user input.
  - o Can read input from various sources, including files, strings, and streams.

## Example of Using Scanner for File Reading

```
java
Copy
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

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

```
try {
    File file = new File("example.txt");
    Scanner scanner = new Scanner(file);

    while (scanner.hasNextLine()) {
        System.out.println(scanner.nextLine()); // Read each line
    }

    scanner.close();
} catch (FileNotFoundException e) {
    System.out.println("File not found.");
    e.printStackTrace();
}
}
```

## File Handling with `java.nio` (New I/O)

In addition to the `java.io` package, Java introduced a more modern and flexible way of handling files in the `java.nio` (New I/O) package, which includes classes like `Path`, `Files`, and `FileSystems`. The `java.nio` package is part of the Java 7 and later versions.

For example, using `Files` to read and write files:

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

public class NioFileExample {
    public static void main(String[] args) {
        Path path = Paths.get("example.txt");

        // Write to a file
        try {
            Files.write(path, "Hello NIO!".getBytes(), StandardOpenOption.CREATE, StandardOpenOption.APPEND);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
// Read from a file
try {
    Files.lines(path).forEach(System.out::println);
} catch (IOException e) {
    e.printStackTrace();
}
}
```

## Conclusion

File handling in Java is a fundamental concept for working with external data stored in files. Java provides multiple classes and methods in both the `java.io` and `java.nio` packages to perform file operations such as reading, writing, appending, and deleting files. The choice of which class or approach to use depends on the requirements of your program and the performance considerations for handling large amounts of data.

---

Reading and writing files in Java are common tasks when working with input and output (I/O) operations. Java provides multiple classes and techniques to handle file reading and writing, both in the older `java.io` package and the newer `java.nio` package.

# 1. Reading Files in Java

Java offers several classes for reading files, such as `FileReader`, `BufferedReader`, and `Scanner`. The most common method for reading a file line by line is using `BufferedReader`.

### Reading Files Using `BufferedReader`

`BufferedReader` is one of the most efficient classes for reading text files because it buffers the input, making file reading faster by reducing I/O operations.

Here's how you can read a file line by line using `BufferedReader`:

```
java
Copy
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class FileReadExample {
    public static void main(String[] args) {
        String filename = "example.txt";
        try (BufferedReader reader = new BufferedReader(new FileReader(filename))) {
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println(line); // Print each line of the file
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- **Explanation:**

- `FileReader` is used to read character files.
- `BufferedReader` is wrapped around `FileReader` to improve performance by buffering the input.
- The `readLine()` method reads one line at a time.
- The `try-with-resources` statement automatically closes the reader when done, preventing resource leaks.

## Reading Files Using `Scanner`

`Scanner` is another easy way to read files. It can read input from various sources like files, strings, and user input. Here's an example:

```
java
Copy
import java.io.File;
import java.io.FileNotFoundException;
```

```
import java.util.Scanner;

public class ScannerFileRead {
    public static void main(String[] args) {
        try {
            File file = new File("example.txt");
            Scanner scanner = new Scanner(file);

            while (scanner.hasNextLine()) {
                System.out.println(scanner.nextLine()); // Print each line of the file
            }

            scanner.close(); // Close the scanner
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

- **Explanation:**

- Scanner reads the file example.txt line by line.
- The hasNextLine() method checks if there is another line to read, and nextLine() reads the next line.
- Always remember to close the Scanner to free resources.

---

## 2. Writing Files in Java

Java provides classes like `FileWriter`, `BufferedWriter`, and `PrintWriter` for writing to files. The following examples show different ways to write data to a file.

### Writing to a File Using `FileWriter`

`FileWriter` is the simplest way to write text to a file. It writes character-based output.

```
java
Copy
import java.io.FileWriter;
import java.io.IOException;

public class FileWriteExample {
    public static void main(String[] args) {
        try (FileWriter writer = new FileWriter("output.txt")) {
            writer.write("Hello, world!"); // Write data to the file
            writer.write("\nThis is a test file.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- **Explanation:**

- `FileWriter` writes characters to the file `output.txt`.
- The `write()` method writes data to the file.
- The `try-with-resources` statement ensures that the writer is closed automatically.

### Writing to a File Using `BufferedWriter`

`BufferedWriter` is an enhanced version of `FileWriter` that improves performance by buffering the output.

```
java
Copy
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class BufferedWriterExample {
    public static void main(String[] args) {
        try (BufferedReader writer = new BufferedReader(new FileReader("output.txt"))) {
            writer.write("Hello, world!");
            writer.newLine(); // Adds a new line
            writer.write("Buffered writing is faster.");
        }
    }
}
```

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

- **Explanation:**

- `BufferedWriter` is wrapped around `FileWriter` to provide buffered output.
- The `newLine()` method is used to insert a newline character (platform-independent).

## Writing to a File Using `PrintWriter`

`PrintWriter` provides a convenient way to write formatted text to a file, and it also handles automatic flushing.

```
java
Copy
import java.io.FileNotFoundException;
import java.io.PrintWriter;

public class PrintWriterExample {
    public static void main(String[] args) {
        try (PrintWriter writer = new PrintWriter("output.txt")) {
            writer.println("Hello, world!"); // Prints a line with a newline
            writer.println("This is a test file.");
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

- **Explanation:**

- `PrintWriter` makes it easier to write formatted data and automatically handles line breaks with the `println()` method.
- It also handles flushing the data, making it more efficient for large outputs.

---

## 3. Appending to Files

Sometimes, you may need to add new content to an existing file without overwriting the existing data. To do this, you can pass `true` as the second argument when creating a `FileWriter` to enable appending.

```
java
Copy
import java.io.FileWriter;
import java.io.IOException;

public class AppendToFile {
    public static void main(String[] args) {
        try (FileWriter writer = new FileWriter("output.txt", true)) {
            writer.write("\nThis is appended text."); // Appends text to the file
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- **Explanation:**
  - The second argument `true` to `FileWriter` enables appending instead of overwriting the file.

---

## 4. Reading and Writing Files Using `java.nio` (New I/O)

Java also provides the `java.nio` package (introduced in Java 7) for more efficient file handling. This package offers classes like `Path`, `Files`, and `Paths` for reading and writing files.

### Reading Files Using `java.nio`

```
java
Copy
import java.nio.file.*;
import java.io.IOException;

public class NioFileRead {
    public static void main(String[] args) {
        Path path = Paths.get("example.txt");

        try {
            Files.lines(path).forEach(System.out::println); // Read and print each line
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- **Explanation:**

- `Files.lines()` reads all lines of the file as a `Stream<String>`, which allows you to process each line using lambda expressions or other stream operations.

## Writing Files Using `java.nio`

```
java
Copy
import java.nio.file.*;
import java.io.IOException;
import java.util.List;

public class NioFileWrite {
    public static void main(String[] args) {
        Path path = Paths.get("output.txt");
        List<String> lines = List.of("Hello, world!", "This is a test.");

        try {
            Files.write(path, lines); // Write the list of strings to the file
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
        }  
    }  
}
```

- **Explanation:**

- `Files.write()` writes a list of strings to a file, where each string is written on a new line.
  - This is a simple and efficient way to write data to a file using `java.nio`.
- 

## Conclusion

Java offers multiple ways to handle file reading and writing, each suitable for different use cases:

- **For simple file operations**, you can use `FileReader`, `FileWriter`, `BufferedReader`, and `BufferedWriter`.
- **For more flexibility and ease of use**, `PrintWriter` and `Scanner` provide convenient methods.
- **For modern and efficient file handling**, the `java.nio` package (with classes like `Path`, `Files`, and `Paths`) is preferred for better performance and more powerful I/O capabilities.

Each approach has its strengths, and the choice depends on the complexity of the task and the size of the file being processed.

---

In Java, file and directory operations are handled using classes from the `java.io` and `java.nio` packages. These operations allow you to create, delete, move, copy, and list files and directories in the filesystem.

## 1. File Operations in Java

### Using `java.io.File` Class

The `File` class in Java provides methods for performing basic file operations like creating, deleting, renaming, and checking file properties.

## Creating a File

To create a new file, you can use the `createNewFile()` method. If the file already exists, it will return `false`.

```
java
Copy
import java.io.File;
import java.io.IOException;

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

        try {
            if (file.createNewFile()) {
                System.out.println("File created: " + file.getName());
            } else {
                System.out.println("File already exists.");
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- **Explanation:**

- `createNewFile()` creates a new file if it doesn't exist. It returns `true` if successful, and `false` if the file already exists.

## Checking if a File Exists

You can check if a file exists using the `exists()` method.

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

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

        if (file.exists()) {
            System.out.println("File exists.");
        } else {
            System.out.println("File does not exist.");
        }
    }
}
```

- **Explanation:**

- o `exists()` checks if the file or directory exists.

## Deleting a File

To delete a file, you can use the `delete()` method. It returns `true` if the file was deleted successfully, and `false` otherwise.

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

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

        if (file.delete()) {
            System.out.println("File deleted successfully.");
        } else {
            System.out.println("Failed to delete the file.");
        }
    }
}
```

- **Explanation:**

- o `delete()` deletes the file from the filesystem. It returns `true` if the file is successfully deleted.

## Renaming a File

To rename a file, you can use the `renameTo()` method, which moves or renames a file.

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

public class RenameFile {
    public static void main(String[] args) {
        File oldFile = new File("oldfile.txt");
        File newFile = new File("newfile.txt");

        if (oldFile.renameTo(newFile)) {
            System.out.println("File renamed successfully.");
        } else {
            System.out.println("Failed to rename the file.");
        }
    }
}
```

- **Explanation:**
  - `renameTo()` renames the file or moves it to a different location. It returns `true` if the operation is successful.

---

## 2. Directory Operations in Java

### Using `java.io.File` Class for Directories

In addition to file operations, the `File` class also provides methods for directory operations.

## Creating a Directory

To create a directory, you can use the `mkdir()` or `mkdirs()` method. The `mkdir()` creates a single directory, while `mkdirs()` creates any missing parent directories as well.

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

public class CreateDirectory {
    public static void main(String[] args) {
        File dir = new File("myDirectory");

        if (dir.mkdir()) {
            System.out.println("Directory created.");
        } else {
            System.out.println("Failed to create directory.");
        }
    }
}
```

- **Explanation:**

- `mkdir()` creates a single directory if it doesn't already exist. If the directory cannot be created, it returns `false`.

## Listing Files in a Directory

You can list the files in a directory using the `listFiles()` method, which returns an array of `File` objects representing the files and directories inside.

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

public class ListFilesInDirectory {
    public static void main(String[] args) {
        File dir = new File("myDirectory");
    }
}
```

```
        if (dir.exists() && dir.isDirectory()) {
            File[] files = dir.listFiles();
            for (File file : files) {
                System.out.println(file.getName());
            }
        } else {
            System.out.println("Directory does not exist.");
        }
    }
}
```

- **Explanation:**

- o `listFiles()` returns an array of `File` objects representing all the files and subdirectories in the specified directory.

#### Deleting a Directory

To delete an empty directory, you can use the `delete()` method of the `File` class. The directory must be empty for this method to succeed.

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

public class DeleteDirectory {
    public static void main(String[] args) {
        File dir = new File("myDirectory");

        if (dir.delete()) {
            System.out.println("Directory deleted.");
        } else {
            System.out.println("Failed to delete directory. Make sure it is empty.");
        }
    }
}
```

- **Explanation:**

- o `delete()` will only delete the directory if it is empty. If there are files or subdirectories, you need to delete them first.

---

### 3. File and Directory Operations Using `java.nio` (New I/O)

Starting with Java 7, the `java.nio` (New I/O) package introduced better ways to work with files and directories, including the `Path` class and `Files` utility class.

#### Creating a Directory Using `java.nio`

You can create a directory using the `Files.createDirectory()` method.

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

public class Nio.CreateDirectory {
    public static void main(String[] args) {
        Path dirPath = Paths.get("myDirectory");

        try {
            Files.createDirectory(dirPath);
            System.out.println("Directory created.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- **Explanation:**
  - `Files.createDirectory()` creates a new directory. If the directory already exists, it throws an exception.

#### Listing Files in a Directory Using `java.nio`

You can list the files in a directory using the `Files.list()` method, which returns a stream of `Path` objects representing the files in the directory.

```
java
Copy
import java.nio.file.*;
import java.io.IOException;

public class NioListFiles {
    public static void main(String[] args) {
        Path dirPath = Paths.get("myDirectory");

        try {
            Files.list(dirPath).forEach(System.out::println);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- **Explanation:**

- `Files.list()` returns a stream of `Path` objects for the files in the directory.

## Deleting a Directory Using `java.nio`

To delete a directory using `java.nio`, you can use the `Files.delete()` method. Note that the directory must be empty before it can be deleted.

```
java
Copy
import java.nio.file.*;
import java.io.IOException;

public class NioDeleteDirectory {
    public static void main(String[] args) {
        Path dirPath = Paths.get("myDirectory");

        try {
            Files.delete(dirPath);
            System.out.println("Directory deleted.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
        }
    }
}

• Explanation:
    ○ Files.delete() deletes the directory. It throws an exception if the directory is not empty.
```

---

## 4. Copying and Moving Files Using `java.nio`

### Copying a File

You can copy a file using `Files.copy()`.

```
java
Copy
import java.nio.file.*;
import java.io.IOException;

public class NioCopyFile {
    public static void main(String[] args) {
        Path sourcePath = Paths.get("source.txt");
        Path destinationPath = Paths.get("destination.txt");

        try {
            Files.copy(sourcePath, destinationPath, StandardCopyOption.REPLACE_EXISTING);
            System.out.println("File copied.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- **Explanation:**

- `Files.copy()` copies a file from one path to another. The `StandardCopyOption.REPLACE_EXISTING` option is used to overwrite the destination file if it exists.

## Moving a File

You can move a file using `Files.move()`.

```
java
Copy
import java.nio.file.*;
import java.io.IOException;

public class NioMoveFile {
    public static void main(String[] args) {
        Path sourcePath = Paths.get("source.txt");
        Path destinationPath = Paths.get("destination.txt");

        try {
            Files.move(sourcePath, destinationPath, StandardCopyOption.REPLACE_EXISTING);
            System.out.println("File moved.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- **Explanation:**

- `Files.move()` moves a file from one path to another. You can also use `StandardCopyOption.REPLACE_EXISTING` to overwrite the destination file.

---

## Conclusion

Java provides powerful APIs for performing file and directory operations, whether you use the traditional `java.io.File` class or the more modern `java.nio` package. Depending on your use case, you can choose the most appropriate approach for tasks such as creating, deleting, copying, moving, or listing files and directories.

- For **basic file operations** (creating, deleting, renaming, etc.), you can use the `java.io.File` class.
  - For **advanced and efficient file I/O** (such as listing files, copying, moving, or handling symbolic links), the `java.nio` package is recommended.
- 

# Serialization in Java

is the process of converting an object into a byte stream so that it can be easily saved to a file, transmitted over a network, or stored in a database. This byte stream can then be deserialized (or restored) to recreate the original object. Serialization is essential for tasks like saving the state of an object or sending objects over a network.

In Java, **serialization** is implemented using the `java.io.Serializable` interface, and the `ObjectOutputStream` and `ObjectInputStream` classes are used for writing and reading the serialized object, respectively.

## 1. What is Serialization?

Serialization in Java refers to converting an object into a byte stream that can be sent over the network or saved into a file. This byte stream represents the object's state (i.e., the values of its fields) and its class metadata. When the byte stream is read, the object can be deserialized back to its original form.

## 2. What is Deserialization?

Deserialization is the reverse process of serialization. It involves converting the byte stream back into a copy of the original object. This is done using the `ObjectInputStream` class in Java.

### 3. How to Make a Class Serializable

To make a Java class serializable, you must implement the `java.io.Serializable` interface. The `Serializable` interface is a marker interface (it does not contain any methods) that tells the Java Virtual Machine (JVM) that the class instances can be serialized and deserialized.

#### Example of a Serializable Class

```
java
Copy
import java.io.Serializable;

public class Person implements Serializable {
    private String name;
    private int age;

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

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

    public int getAge() {
        return age;
    }

    // Overriding toString for easy printing
    @Override
    public String toString() {
        return "Person{name='" + name + "', age=" + age + '}';
    }
}
```

- **Explanation:**
  - Person implements the Serializable interface, meaning instances of Person can be serialized and deserialized.

## 4. Serializing an Object

To serialize an object in Java, you use the `ObjectOutputStream` class. It allows writing the object to an output stream, typically a file or network stream.

### Example: Serialize a Person Object to a File

```
java
Copy
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;

public class SerializeExample {
    public static void main(String[] args) {
        Person person = new Person("John Doe", 30);

        try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("person.ser"))) {
            oos.writeObject(person); // Serialize the person object
            System.out.println("Object has been serialized");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- **Explanation:**
  - `ObjectOutputStream` is used to serialize the `person` object and write it to a file named `person.ser`.
  - The `writeObject()` method is called to perform the actual serialization.
  - `FileOutputStream` creates a file output stream that writes bytes to a file.

## 5. Deserializing an Object

To deserialize an object, you use the `ObjectInputStream` class. It allows reading an object from an input stream, such as a file or network stream.

### **Example: Deserialize a Person Object from a File**

```
java
Copy
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

public class DeserializeExample {
    public static void main(String[] args) {
        try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream("person.ser"))) {
            Person person = (Person) ois.readObject(); // Deserialize the person object
            System.out.println("Object has been deserialized");
            System.out.println(person); // Print the deserialized object
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

- **Explanation:**

- `ObjectInputStream` is used to read the serialized object from the `person.ser` file.
- `readObject()` is called to deserialize the object. The object is then cast to the appropriate class (`Person` in this case).
- The deserialized object is printed to verify that the state was restored.

## **6. Transient Keyword**

In some cases, you might not want certain fields of an object to be serialized. In such cases, you can mark those fields with the `transient` keyword. Fields marked as `transient` will not be included in the serialized representation of the object.

### **Example: Using the `transient` Keyword**

```
java
Copy
import java.io.Serializable;

public class Person implements Serializable {
    private String name;
    private int age;
    private transient String password; // This field will not be serialized

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

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

    public int getAge() {
        return age;
    }

    public String getPassword() {
        return password;
    }

    @Override
    public String toString() {
        return "Person{name='" + name + "', age=" + age + ", password='" + password + "'}";
    }
}
```

- **Explanation:**

- The password field is marked as `transient`, meaning it will not be serialized and will be `null` when the object is deserialized.

## 7. Version Control in Serialization

Java provides a mechanism for version control in serialization through the `serialVersionUID` field. This is a unique identifier for the version of the class.

When a class is modified (e.g., adding/removing fields), the default serialization mechanism may fail if the `serialVersionUID` changes. To handle this, you should explicitly define a `serialVersionUID`.

### Example of `serialVersionUID`

```
java
Copy
import java.io.Serializable;

public class Person implements Serializable {
    private static final long serialVersionUID = 1L; // Explicitly defining serialVersionUID
    private String name;
    private int age;

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

    // Getters and other methods...
}
```

- **Explanation:**

- `serialVersionUID` ensures that the class version is consistent during deserialization. If the class definition changes but the `serialVersionUID` remains the same, deserialization will still work correctly.
- If you change the structure of the class (e.g., adding fields), you should also update `serialVersionUID`.

## 8. Custom Serialization

Sometimes, you might want more control over the serialization process. Java allows you to define custom methods to control how the serialization and deserialization process works.

- **writeObject()**: This method is invoked during serialization.
- **readObject()**: This method is invoked during deserialization.

### Example: Custom Serialization

```
java
Copy
import java.io.*;

public class CustomSerialization implements Serializable {
    private String name;
    private transient int age;

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

    // Custom serialization
    private void writeObject(ObjectOutputStream oos) throws IOException {
        oos.defaultWriteObject(); // Default serialization
        oos.writeInt(age * 2); // Custom serialization logic
    }

    // Custom deserialization
    private void readObject(ObjectInputStream ois) throws IOException, ClassNotFoundException {
        ois.defaultReadObject(); // Default deserialization
        this.age = ois.readInt() / 2; // Custom deserialization logic
    }

    @Override
    public String toString() {
        return "CustomSerialization{name='" + name + "', age=" + age + "}";
    }
}
```

}

- **Explanation:**

- The `writeObject()` and `readObject()` methods are used to customize how the object is serialized and deserialized. For example, the `age` field is stored differently.
- 

## 9. Common Exceptions During Serialization

- `java.io.NotSerializableException`: Thrown when an object that is not serializable is attempted to be serialized.
- `java.io.InvalidClassException`: Thrown if there is a version mismatch between the serialized object and the class being deserialized (e.g., if the `serialVersionUID` is different).

### Conclusion

Serialization is a powerful feature in Java that allows you to persist the state of an object or transmit it over a network. By implementing the `Serializable` interface, you can easily convert objects to a byte stream and later restore them. You can also use `transient` to exclude fields from serialization and control the process with `writeObject()` and `readObject()` methods. Understanding serialization is key for tasks like saving object states, object persistence, and communication between different systems.

---

# Java I/O (Input/Output)

refers to the mechanisms through which Java programs interact with external systems, such as files, networks, or devices. The Java I/O API provides a set of classes and interfaces that allow for reading from and writing to various data sources, such as files, memory, and network connections. It provides different abstractions and tools for handling input and output operations, enabling developers to perform tasks such as file manipulation, data processing, and communication between programs.

Java I/O is a key part of Java's `java.io` package, but since Java 7, the `java.nio` (New I/O) package was introduced to improve scalability and performance for certain types of I/O operations.

## 1. Java I/O Overview

Java I/O is divided into two main categories:

- **Byte-based I/O:** Deals with raw binary data (e.g., images, audio files).
- **Character-based I/O:** Deals with text data (e.g., files with textual content).

These categories provide different classes and interfaces for reading and writing data. The `java.io` package contains most of the classes for I/O, while `java.nio` introduced more advanced features like non-blocking I/O and buffer-based IO.

## 2. Byte-based I/O (Streams)

Byte-based I/O uses byte streams to read and write binary data. It is used for all types of data, including text files, images, audio files, and other raw byte data.

### Byte Streams Classes

- **InputStream** and **OutputStream** are the base classes for byte-based I/O.
  - **InputStream:** Reads data byte by byte.
  - **OutputStream:** Writes data byte by byte.

Example of Byte-based I/O

```
java
Copy
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class ByteStreamExample {
    public static void main(String[] args) {
        String inputFile = "input.txt";
        String outputFile = "output.txt";

        try (FileInputStream fis = new FileInputStream(inputFile);
             FileOutputStream fos = new FileOutputStream(outputFile)) {

            int byteData;
            while ((byteData = fis.read()) != -1) {
                fos.write(byteData); // Copy byte by byte from input to output
            }

            System.out.println("File copied successfully!");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- **Explanation:**

- `FileInputStream` reads the file byte by byte.
- `FileOutputStream` writes each byte to the output file.

### 3. Character-based I/O (Streams)

Character-based I/O is used for handling text data. It works with streams that handle characters rather than raw bytes, automatically handling character encoding (e.g., UTF-8, UTF-16).

#### Character Streams Classes

- **Reader** and **Writer** are the base classes for character-based I/O.
  - **Reader**: Reads data character by character.
  - **Writer**: Writes data character by character.

#### Example of Character-based I/O

```
java
Copy
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class CharStreamExample {
    public static void main(String[] args) {
        String inputFile = "input.txt";
        String outputFile = "output.txt";

        try (FileReader fr = new FileReader(inputFile);
             FileWriter fw = new FileWriter(outputFile)) {
            int charData;
            while ((charData = fr.read()) != -1) {
                fw.write(charData); // Copy characters from input to output
            }

            System.out.println("File copied successfully!");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- **Explanation:**
  - `FileReader` reads characters from the file.
  - `FileWriter` writes characters to the output file.

## 4. Buffered I/O

Buffered I/O is a technique used to optimize reading and writing by buffering data in memory, reducing the number of I/O operations. Buffered streams are commonly used in both byte and character streams.

## Buffered Streams Classes

- **BufferedInputStream** and **BufferedOutputStream** (for byte streams).
- **BufferedReader** and **BufferedWriter** (for character streams).

Example of Buffered I/O (Character Streams)

```
java
Copy
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.BufferedWriter;
import java.io.IOException;

public class BufferedStreamExample {
    public static void main(String[] args) {
        String inputFile = "input.txt";
        String outputFile = "output.txt";

        try (BufferedReader br = new BufferedReader(new FileReader(inputFile));
             BufferedWriter bw = new BufferedWriter(new FileWriter(outputFile))) {

            String line;
            while ((line = br.readLine()) != null) {
                bw.write(line); // Write each line of text
                bw.newLine(); // Write a newline after each line
            }

            System.out.println("Buffered File copied successfully!");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- **Explanation:**
  - `BufferedReader` reads text data line by line.
  - `BufferedWriter` writes text data, with buffering, improving efficiency.

## 5. Data Streams

Data streams allow reading and writing primitive data types (like `int`, `float`, `char`) in a machine-independent way.

### Data Input/Output Streams

- **DataInputStream:** Used to read primitive data types.
- **DataOutputStream:** Used to write primitive data types.

Example of Data Streams

```
java
Copy
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class DataStreamExample {
    public static void main(String[] args) {
        String fileName = "data.txt";

        try (DataOutputStream dos = new DataOutputStream(new FileOutputStream(fileName))) {
            dos.writeInt(100);      // Write integer
            dos.writeDouble(99.99); // Write double
            dos.writeUTF("Hello, World!"); // Write string

            System.out.println("Data written successfully!");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```

        try (DataInputStream dis = new DataInputStream(new FileInputStream(fileName))) {
            int intData = dis.readInt();
            double doubleData = dis.readDouble();
            String strData = dis.readUTF();

            System.out.println("Read data: " + intData + ", " + doubleData + ", " + strData);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

- **Explanation:**

- DataOutputStream **writes** primitive data types.
- DataInputStream **reads** the primitive data types.

## 6. Object Streams

Object streams are used to serialize and deserialize objects in Java. Objects are written to and read from streams as part of the serialization process.

### Object Streams Classes

- **ObjectOutputStream:** Used to serialize objects to an output stream.
- **ObjectInputStream:** Used to deserialize objects from an input stream.

#### Example of Object Serialization

```

java
Copy
import java.io.*;

class Person implements Serializable {
    String name;
    int age;
}

```

```
public Person(String name, int age) {  
    this.name = name;  
    this.age = age;  
}  
}  
  
public class ObjectStreamExample {  
    public static void main(String[] args) {  
        String fileName = "person.ser";  
  
        Person person = new Person("John", 30);  
  
        // Serialize the object to a file  
        try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(fileName))) {  
            oos.writeObject(person);  
            System.out.println("Object serialized successfully!");  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
  
        // Deserialize the object from the file  
        try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream(fileName))) {  
            Person serializedPerson = (Person) ois.readObject();  
            System.out.println("Serialized Person: " + serializedPerson.name + ", " + serializedPerson.age);  
        } catch (IOException | ClassNotFoundException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

- **Explanation:**

- `ObjectOutputStream` serializes the `Person` object into the file.
- `ObjectInputStream` deserializes the object back into memory.

## 7. File I/O in Java

File I/O operations allow Java programs to read from and write to files. In addition to the basic streams, Java provides classes like `FileReader`, `FileWriter`, `BufferedReader`, `BufferedWriter`, and `RandomAccessFile` for more specialized file handling.

### Example: Working with Files

```
java
Copy
import java.io.*;

public class FileOperationsExample {
    public static void main(String[] args) {
        String fileName = "test.txt";
        String data = "Hello, this is a test file.";

        // Writing to a file
        try (FileWriter fw = new FileWriter(fileName);
             BufferedWriter bw = new BufferedWriter(fw)) {
            bw.write(data);
            System.out.println("Data written to file successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Reading from a file
        try (FileReader fr = new FileReader(fileName);
             BufferedReader br = new BufferedReader(fr)) {
            String line;
            while ((line = br.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- **Explanation:**
  - `FileWriter` and `BufferedWriter` are used for writing to a file.

- o `FileReader` and `BufferedReader` are used for reading from a file.
- 

## Conclusion

Java I/O is a powerful set of APIs that enables interaction with various data sources, such as files, network connections, and devices. Java provides both **byte-based** and **character-based** I/O for different types of data and ensures efficient handling of large data with buffered streams. Additionally, Java supports serialization with object streams, allowing objects to be saved and restored. Understanding Java I/O is crucial for handling tasks like file manipulation, data storage, and network communication.

---

# Core Java Notes By AJAY RAZZ

In Java, **Streams** are a powerful mechanism for reading and writing data. Streams allow you to perform I/O (Input/Output) operations in a sequence of bytes or characters. Java provides two main types of streams to handle different types of data:

1. **Byte Streams:** Used for handling binary data (e.g., images, audio, video, files that are not necessarily text).
2. **Character Streams:** Used for handling text data (e.g., text files, XML files, or any content that involves characters).

Both of these streams work through the **InputStream** and **OutputStream** (for byte streams) or **Reader** and **Writer** (for character streams) classes in Java.

## 1. Byte Streams in Java

Byte streams are used to perform I/O operations on **raw binary data**. These are ideal for handling data that is not text-based, such as image files, video files, or other media files.

Byte streams read and write data in **bytes**, making them more flexible for non-textual data. They are subclassed from `InputStream` (for reading data) and `OutputStream` (for writing data).

### Key Classes for Byte Streams:

- **InputStream**: Abstract class for reading byte data.
  - **FileInputStream**: Reads bytes from a file.
  - **BufferedInputStream**: Buffers data from an input stream for more efficient reading.
  - **DataInputStream**: Reads primitive data types in a machine-independent manner.
- **OutputStream**: Abstract class for writing byte data.
  - **FileOutputStream**: Writes bytes to a file.
  - **BufferedOutputStream**: Buffers data for more efficient writing.
  - **DataOutputStream**: Writes primitive data types.

### Example: Byte Stream (Reading and Writing a File)

```
java
Copy
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class ByteStreamExample {
    public static void main(String[] args) {
        String inputFile = "input.txt";
        String outputFile = "output.txt";

        try (FileInputStream fis = new FileInputStream(inputFile);
             FileOutputStream fos = new FileOutputStream(outputFile)) {

            int byteData;
            while ((byteData = fis.read()) != -1) {
                fos.write(byteData); // Copy byte by byte from input to output
            }
            System.out.println("File copied successfully!");
        } catch (IOException e) {
```

```
        e.printStackTrace();
    }
}
}
```

- **Explanation:**

- `FileInputStream` reads bytes from the `input.txt` file.
- `FileOutputStream` writes each byte to the `output.txt` file.

## 2. Character Streams in Java

Character streams are designed for **handling text data**, i.e., sequences of characters. These streams handle the encoding and decoding of characters automatically, making them suitable for reading and writing text files.

Character streams are more efficient than byte streams when working with text because they read and write data in **characters**, rather than raw bytes. They are subclassed from `Reader` (for reading data) and `Writer` (for writing data).

### Key Classes for Character Streams:

- **Reader:** Abstract class for reading character data.
  - **FileReader:** Reads characters from a file.
  - **BufferedReader:** Buffers input data, improving efficiency, especially when reading line by line.
  - **InputStreamReader:** Converts byte streams into character streams (useful when dealing with non-UTF-8 byte data).
- **Writer:** Abstract class for writing character data.
  - **FileWriter:** Writes characters to a file.
  - **BufferedWriter:** Buffers character data, improving efficiency.
  - **PrintWriter:** Can be used for writing formatted text and has convenient methods like `println()`.

### Example: Character Stream (Reading and Writing a File)

```
java
Copy
import java.io.FileReader;
```

```
import java.io.FileWriter;
import java.io.IOException;

public class CharStreamExample {
    public static void main(String[] args) {
        String inputFile = "input.txt";
        String outputFile = "output.txt";

        try (FileReader fr = new FileReader(inputFile);
             FileWriter fw = new FileWriter(outputFile)) {

            int charData;
            while ((charData = fr.read()) != -1) {
                fw.write(charData); // Copy character by character from input to output
            }

            System.out.println("File copied successfully!");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- **Explanation:**

- `FileReader` reads characters from the `input.txt` file.
- `FileWriter` writes each character to the `output.txt` file.

### 3. Buffered Streams (Byte and Character)

Buffered streams are used to improve the performance of reading and writing operations by buffering the data in memory. Without buffering, each read or write operation would involve interacting with the underlying storage system, which can be slow. Buffered streams help minimize these operations by reading or writing larger chunks of data at once.

#### Buffered Classes:

- **BufferedInputStream** (for byte streams)

- **BufferedOutputStream** (for byte streams)
- **BufferedReader** (for character streams)
- **BufferedWriter** (for character streams)

### Example: Buffered Character Stream (Reading and Writing a File)

```
java
Copy
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class BufferedCharStreamExample {
    public static void main(String[] args) {
        String inputFile = "input.txt";
        String outputFile = "output.txt";

        try (BufferedReader br = new BufferedReader(new FileReader(inputFile));
             BufferedWriter bw = new BufferedWriter(new FileWriter(outputFile))) {

            String line;
            while ((line = br.readLine()) != null) {
                bw.write(line); // Write each line of text from input to output
                bw.newLine(); // Write a new line after each line
            }

            System.out.println("Buffered file copied successfully!");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- **Explanation:**
  - o `BufferedReader` reads text line by line.

- `BufferedWriter` writes the text, with buffering to optimize performance.

## 4. Differences between Byte Streams and Character Streams

Feature	Byte Streams	Character Streams
<b>Data type</b>	Deal with binary data (bytes)	Deal with character data (text)
<b>Base Classes</b>	<code>InputStream</code> , <code>OutputStream</code>	<code>Reader</code> , <code>Writer</code>
<b>Use Case</b>	Used for all kinds of data (e.g., images, audio, video, files containing non-text data)	Used for reading and writing text (e.g., text files, XML files, etc.)
<b>Encoding</b>	No automatic encoding or decoding. The data is raw and not dependent on encoding.	Automatically handles character encoding/decoding (e.g., UTF-8).
<b>Efficiency</b>	Typically slower for text files due to lack of encoding handling	More efficient for text files due to automatic handling of character encoding

## 5. When to Use Byte Streams and Character Streams

- **Byte Streams** should be used when you are working with raw binary data, such as:
  - Images
  - Audio and video files
  - Executable files
  - Any non-textual data (e.g., PDFs, ZIP files)
- **Character Streams** should be used when you are working with text data, such as:
  - Text files (e.g., `.txt`, `.csv`, `.html`)
  - JSON, XML files
  - Any data that consists of readable characters and where encoding matters

## 6. InputStream vs. Reader / OutputStream vs. Writer

- **InputStream** and **OutputStream** are for byte-based I/O, which can handle raw binary data.

- **Reader** and **Writer** are for character-based I/O, which is ideal for text files and data requiring character encoding (e.g., UTF-8).

## 7. Example with Data Streams (for Primitive Data Types)

If you need to handle primitive data types (e.g., int, float, double, etc.) in a machine-independent way, you can use **DataInputStream** and **DataOutputStream**.

```
java
Copy
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class DataStreamExample {
    public static void main(String[] args) {
        String fileName = "data.txt";

        try (DataOutputStream dos = new DataOutputStream(new FileOutputStream(fileName))) {
            dos.writeInt(100);      // Write integer
            dos.writeDouble(99.99); // Write double
            dos.writeUTF("Hello, World!"); // Write string
            System.out.println("Data written successfully!");
        } catch (IOException e) {
            e.printStackTrace();
        }

        try (DataInputStream dis = new DataInputStream(new FileInputStream(fileName))) {
            int intData = dis.readInt();
            double doubleData = dis.readDouble();
            String strData = dis.readUTF();

            System.out.println("Read data: " + intData + ", " + doubleData + ", " + strData);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

}

- **Explanation:**

- `DataOutputStream` is used to write primitive data types to the file.
- `DataInputStream` is used to read the primitive data types from the file.

## Conclusion

Java provides a rich set of stream classes for handling I/O operations. **Byte streams** are ideal for handling raw binary data, while **character streams** are optimized for working with text data. Both types of streams come with buffering and data classes that improve efficiency and allow for reading/writing primitive data types. Choosing the right stream type is essential depending on the kind of data you're working with.

---

**Readers and Writers** In Java, **Readers and Writers** are part of the **character stream API**, which is specifically designed for handling **text data** (i.e., sequences of characters). Unlike byte streams, which handle raw binary data, character streams handle data in a format that is easier to work with when dealing with textual content.

The classes in the `java.io` package provide an abstraction layer for reading from and writing to files, network connections, or memory in terms of **characters** instead of raw bytes. These streams automatically handle character encoding, making them more convenient and efficient when working with text-based data.

## 1. Overview of Readers and Writers

- **Readers:** Used to read data as characters from input sources like files, buffers, or network streams.
  - All classes that extend the `Reader` class read data in characters.
  - It decodes bytes (usually in a specific character encoding) into characters.

- **Writers:** Used to write character data to output destinations like files, buffers, or network streams.
  - All classes that extend the `Writer` class write data in characters.
  - It encodes characters into bytes (again, in a specific character encoding).

## 2. Key Classes for Readers and Writers

### Reader Class Hierarchy (for reading data)

The `Reader` class is the abstract superclass of all character input streams. Some important subclasses are:

- **FileReader:** A convenience class for reading characters from a file.
- **BufferedReader:** Reads text from a character-based input stream, buffering characters for efficient reading, especially when reading lines.
- **CharArrayReader:** Reads characters from a character array.
- **InputStreamReader:** A bridge class that converts byte streams into character streams by specifying the character encoding.

### Writer Class Hierarchy (for writing data)

The `Writer` class is the abstract superclass of all character output streams. Some important subclasses are:

- **FileWriter:** A convenience class for writing characters to a file.
- **BufferedWriter:** Writes text to a character-based output stream, buffering characters for efficient writing, especially when writing lines.
- **CharArrayWriter:** Writes characters to a character array.
- **PrintWriter:** A more convenient class for writing formatted text (it also provides methods like `println()`).

## 3. Common Reader and Writer Methods

- **Reader Methods:**
  - `int read():` Reads a single character.
  - `int read(char[] cbuf):` Reads characters into a buffer (array).
  - `String readLine():` Reads a line of text (only available in `BufferedReader`).
  - `void close():` Closes the stream.

- **Writer Methods:**
  - void write(int c): Writes a single character.
  - void write(char[] cbuf): Writes an array of characters.
  - void write(String str): Writes a string.
  - void newLine(): Writes a line separator (only available in BufferedWriter).
  - void close(): Closes the stream.

## 4. Examples of Using Readers and Writers

### Reading Data with Reader Classes

Here's an example of using `FileReader` and `BufferedReader` to read a file line by line:

```
java
Copy
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;

public class ReaderExample {
    public static void main(String[] args) {
        String fileName = "input.txt";

        try (BufferedReader br = new BufferedReader(new FileReader(fileName))) {
            String line;
            while ((line = br.readLine()) != null) {
                System.out.println(line); // Print each line read from the file
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- **Explanation:**

- o `FileReader` is used to read characters from the `input.txt` file.
- o `BufferedReader` is used to read the file efficiently, one line at a time.
- o `readLine()` method reads each line from the file until the end of the file (`null`).

## Writing Data with `Writer` Classes

Here's an example of using `FileWriter` and `BufferedWriter` to write data to a file:

```
java
Copy
import java.io.FileWriter;
import java.io.BufferedWriter;
import java.io.IOException;

public class WriterExample {
    public static void main(String[] args) {
        String outputFile = "output.txt";
        String data = "Hello, this is a sample text.";

        try (BufferedWriter bw = new BufferedWriter(new FileWriter(outputFile))) {
            bw.write(data); // Write the string to the output file
            bw.newLine(); // Write a new line (optional)
            System.out.println("Data written successfully!");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- **Explanation:**

- o `FileWriter` is used to write characters to the `output.txt` file.
- o `BufferedWriter` buffers the characters for more efficient writing, especially when dealing with large data.
- o `write()` writes the text to the file, and `newLine()` adds a new line to the output.

## 5. `BufferedReader` and `BufferedWriter` for Efficiency

The `BufferedReader` and `BufferedWriter` classes provide a performance improvement when reading and writing large amounts of data. By buffering the input and output, they reduce the number of actual read/write operations to the underlying file system or other resources, making I/O operations more efficient.

- **BufferedReader:** Efficiently reads text, particularly when reading line by line.
- **BufferedWriter:** Efficiently writes text, particularly when writing large blocks of data.

Here's an example of how to use `BufferedReader` to read text from a file line by line:

```
java
Copy
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;

public class BufferedReaderExample {
    public static void main(String[] args) {
        String fileName = "input.txt";

        try (BufferedReader br = new BufferedReader(new FileReader(fileName))) {
            String line;
            while ((line = br.readLine()) != null) {
                System.out.println(line); // Print each line from the file
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- **Explanation:**
  - `BufferedReader` reads the file efficiently line by line, rather than reading one character at a time.

## 6. PrintWriter for Formatted Output

The `PrintWriter` class is an extension of the `Writer` class, designed to provide convenient methods for formatted output. It supports writing formatted text to files or other output destinations.

Here's an example of using `PrintWriter` to write formatted text to a file:

```
java
Copy
import java.io.PrintWriter;
import java.io.IOException;

public class PrintWriterExample {
    public static void main(String[] args) {
        String outputFile = "output.txt";

        try (PrintWriter pw = new PrintWriter(outputFile)) {
            pw.println("Hello, world!");
            pw.printf("The value of pi is approximately: %.2f\n", Math.PI); // Formatted output
            pw.println("End of file.");
            System.out.println("Formatted data written successfully!");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- **Explanation:**
  - `PrintWriter` provides methods like `println()` for writing text with newlines, and `printf()` for formatted output.

## 7. Handling Encoding with InputStreamReader and OutputStreamWriter

If you need to deal with specific character encodings (e.g., UTF-8, UTF-16), you can use `InputStreamReader` and `OutputStreamWriter` to convert byte streams to character streams and vice versa, while specifying the encoding.

### Example: Using `InputStreamReader` to Specify Encoding

```
java
Copy
import java.io.FileInputStream;
import java.io.InputStreamReader;
import java.io.BufferedReader;
import java.io.IOException;

public class EncodingExample {
    public static void main(String[] args) {
        String fileName = "input.txt";

        try (BufferedReader br = new BufferedReader(new InputStreamReader(new FileInputStream(fileName), "UTF-8")));
        {
            String line;
            while ((line = br.readLine()) != null) {
                System.out.println(line); // Read and print each line with UTF-8 encoding
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- **Explanation:**
  - `InputStreamReader` reads bytes from the file and converts them into characters using the specified encoding (UTF-8 in this case).

## 8. Conclusion

- **Readers** and **Writers** are essential for working with character-based data in Java.
- **Readers** are used for reading characters, while **Writers** are used for writing characters.
- **BufferedReader** and **BufferedWriter** are more efficient than using simple `Reader` and `Writer` classes because they buffer data.
- **PrintWriter** provides convenient methods for formatted text output.
- The `InputStreamReader` and `OutputStreamWriter` classes are useful when dealing with specific encodings.

These classes simplify I/O operations with text data by abstracting away the complexities of character encoding and buffering. They are ideal for handling tasks such as reading and writing text files, processing logs, and handling other forms of textual content.

---

In Java, **annotations** are a form of metadata that provide additional information about the code. Annotations do not change the behavior of the code directly but can be used by tools, compilers, or frameworks to provide useful information during compilation, runtime, or development. They allow developers to embed extra information in the source code that can be processed by various tools and libraries.

## 1. What Are Annotations?

Annotations are special markers that can be added to Java code to convey information about the code, such as its purpose, behavior, or intended use. They are defined using the @ symbol and are typically placed above classes, methods, fields, or parameters.

Example of an annotation:

```
java
Copy
@Override
public String toString() {
    return "This is a string representation";
```

```
}
```

Here, `@Override` is an annotation indicating that the method `toString()` is intended to override a method in the superclass.

## 2. Basic Syntax of Annotations

An annotation is created by preceding an identifier with the `@` symbol. For example, `@AnnotationName`. Annotations can have optional elements that can store values, and these elements are called **members** or **attributes**.

```
java
Copy
// Defining an annotation with elements (optional)
public @interface MyAnnotation {
    String value() default "Default value"; // An attribute with a default value
    int count() default 0;                  // Another attribute with a default value
}
```

## 3. Types of Annotations

There are several categories of annotations in Java:

1. **Built-in Annotations:** Java provides several built-in annotations, such as `@Override`, `@Deprecated`, and `@SuppressWarnings`.
2. **Custom Annotations:** Developers can define their own annotations using the `@interface` keyword.
3. **Meta-Annotations:** Annotations that are used to define other annotations (e.g., `@Retention`, `@Target`, `@Documented`).
4. **Marker Annotations:** Annotations with no members, used for marking a class or method with a specific attribute or behavior.
5. **Single-Value Annotations:** Annotations with a single member, where the member can be set with a value directly.

## 4. Common Built-in Annotations

- **`@Override`:** Indicates that a method is overriding a method in a superclass.

```
java
Copy
```

```
@Override  
public String toString() {  
    return "Custom String";  
}
```

- **@Deprecated:** Marks a method, class, or field as deprecated. This indicates that the element is no longer recommended for use.

```
java  
Copy  
@Deprecated  
public void oldMethod() {  
    // This method is outdated  
}
```

- **@SuppressWarnings:** Instructs the compiler to suppress specific warnings.

```
java  
Copy  
@SuppressWarnings("unchecked")  
public void myMethod() {  
    // Suppress unchecked warnings here  
}
```

## 5. Custom Annotations

Java allows you to define your own annotations using the `@interface` keyword. Custom annotations can contain **elements (attributes)**, which can have default values.

```
java  
Copy  
// Defining a custom annotation  
public @interface MyAnnotation {  
    String description() default "No description";  
    int version() default 1;  
}
```

```
// Using the custom annotation
@MyAnnotation(description = "This is a custom annotation", version = 2)
public class MyClass {
    public void myMethod() {
        // Method code here
    }
}
```

- **Attributes:** You can define attributes (also called elements or members) in custom annotations. These attributes can have default values or can be required.
- **Default Value:** If an attribute has a default value, you can omit it when using the annotation.

## 6. Meta-Annotations

Meta-annotations are annotations that provide metadata about other annotations. Java provides the following meta-annotations:

- **@Retention:** Specifies whether an annotation is available at runtime, compile-time, or source-code level.

```
java
Copy
@Retention(RetentionPolicy.RUNTIME)
public @interface MyRuntimeAnnotation {
    // This annotation is available at runtime
}
```

- **@Target:** Specifies where the annotation can be applied (e.g., methods, classes, fields).

```
java
Copy
@Target(ElementType.METHOD) // Can only be applied to methods
public @interface MyMethodAnnotation {}
```

- **@Documented:** Indicates that an annotation should be included in Javadoc documentation.

```
java
```

```
Copy
@Documented
public @interface MyDocumentedAnnotation {}
```

- **@Inherited:** Indicates that an annotation is inherited by subclasses.

```
java
Copy
@Inherited
public @interface MyInheritedAnnotation {}
```

## 7. Using Annotations at Runtime (Reflection)

Annotations can be processed at runtime using Java reflection. The `java.lang.reflect` package allows you to inspect annotations applied to classes, methods, fields, etc.

```
java
Copy
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface MyAnnotation {
    String value();
}

public class AnnotationExample {

    @MyAnnotation(value = "Hello")
    public void myMethod() {
        // Method code
    }

    public static void main(String[] args) throws Exception {
        Method method = AnnotationExample.class.getMethod("myMethod");
    }
}
```

```
// Check if the method is annotated with MyAnnotation
if (method.isAnnotationPresent(MyAnnotation.class)) {
    MyAnnotation annotation = method.getAnnotation(MyAnnotation.class);
    System.out.println("Annotation value: " + annotation.value());
}
}
```

- **Explanation:**

- The annotation `@MyAnnotation` is defined with a `value` attribute.
- Reflection is used to inspect whether the `myMethod` is annotated with `@MyAnnotation`, and its value is printed.

## 8. Practical Uses of Annotations

Annotations are commonly used in the following scenarios:

1. **Frameworks:** Many Java frameworks like Spring, Hibernate, and JUnit use annotations to simplify configuration and enhance functionality. For example:
  - `@Autowired` (Spring) for dependency injection.
  - `@Entity` (Hibernate) to mark a class as a database entity.
2. **Code Quality:** Annotations like `@Deprecated` help indicate methods or classes that should no longer be used, while `@SuppressWarnings` can help avoid compiler warnings.
3. **Runtime Processing:** Annotations allow frameworks to modify behavior at runtime, such as in the case of custom annotations used in web frameworks or validation libraries.

## 9. Annotations and Reflection Example

Here's a practical example of using annotations with reflection to process metadata dynamically at runtime.

```
java
Copy
import java.lang.annotation.*;
import java.lang.reflect.*;
```

```

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface Info {
    String author() default "Unknown";
    String date();
}

public class AnnotationExample {

    @Info(author = "John", date = "01/01/2025")
    public void exampleMethod() {
        // Method code
    }

    public static void main(String[] args) throws Exception {
        Method method = AnnotationExample.class.getMethod("exampleMethod");

        if (method.isAnnotationPresent(Info.class)) {
            Info info = method.getAnnotation(Info.class);
            System.out.println("Author: " + info.author());
            System.out.println("Date: " + info.date());
        }
    }
}

```

- **Explanation:**
  - The `@Info` annotation is applied to `exampleMethod()`, with attributes `author` and `date`.
  - Using reflection, the `Info` annotation is accessed at runtime to print its values.

## 10. Conclusion

Annotations in Java provide a powerful way to add metadata to the code. They can be used for various purposes such as:

- Indicating that a method overrides a method in a superclass (`@Override`).
- Marking deprecated methods or classes (`@Deprecated`).
- Suppressing compiler warnings (`@SuppressWarnings`).

- Defining custom behaviors for frameworks or tools.

Annotations enhance code readability, maintainability, and provide a mechanism for frameworks to process metadata automatically at compile-time or runtime. Through reflection, developers can process annotations dynamically to implement flexible and configurable behavior.

---

## **built-in annotations**

In Java, **built-in annotations** are predefined annotations that serve various purposes during compilation and runtime. Three of the most commonly used built-in annotations are:

1. `@Override`
2. `@Deprecated`
3. `@SuppressWarnings`

These annotations help make your code more robust, readable, and maintainable. Let's dive into each of them in detail.

---

### **1. `@Override` Annotation**

#### **Purpose:**

The `@Override` annotation is used to indicate that a method is intended to override a method in a superclass or implement an abstract method from an interface. It provides compile-time checking to ensure that the method correctly overrides a method from the parent class or implements the required method from the interface.

## Key Points:

- It helps avoid errors by ensuring that you correctly override a method.
- If the method signature doesn't match the method in the superclass or interface (e.g., wrong method name, wrong parameters), the compiler will generate an error.

## Example:

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

class Dog extends Animal {
    @Override // Correctly overriding the method
    public void makeSound() {
        System.out.println("Dog barks");
    }

    // Uncommenting the following method would cause a compile-time error.
    // @Override
    // public void makeSund() { // Error: method doesn't match the superclass method
    //     System.out.println("Dog barks");
    // }
}

public class Main {
    public static void main(String[] args) {
        Animal animal = new Dog();
        animal.makeSound(); // Output: Dog barks
    }
}
```

- Explanation:

- The `@Override` annotation ensures that `makeSound()` in `Dog` correctly overrides the `makeSound()` method from `Animal`.
  - If we made a typo in the method signature (e.g., `makeSund()`), the compiler would raise an error because it wouldn't match the method in the superclass.
- 

## 2. `@Deprecated` Annotation

### Purpose:

The `@Deprecated` annotation is used to mark classes, methods, or fields as "deprecated," meaning they should no longer be used and are considered outdated. This annotation is typically used when a feature has been replaced by a better alternative or will be removed in future versions.

### Key Points:

- It does not prevent you from using the marked element, but it **signals to developers** that the element is no longer recommended for use.
- The compiler will show a warning if you try to use a deprecated element.

### Example:

```
java
Copy
class OldClass {
    @Deprecated
    public void oldMethod() {
        System.out.println("This is an old method.");
    }

    public void newMethod() {
        System.out.println("This is a new method.");
    }
}
```

```
public class Main {  
    public static void main(String[] args) {  
        OldClass obj = new OldClass();  
        obj.oldMethod(); // Warning: 'oldMethod()' is deprecated  
        obj.newMethod(); // No warning  
    }  
}
```

- **Explanation:**
  - The `@Deprecated` annotation marks `oldMethod()` as outdated.
  - If you use the deprecated method, the compiler will show a warning, notifying you that it's no longer recommended to use that method.

### Suppressing Warnings for Deprecated Methods:

To avoid getting warnings about deprecated methods, you can use the `@SuppressWarnings` annotation (explained below) to suppress such warnings in specific cases.

## 3. `@SuppressWarnings` Annotation

### Purpose:

The `@SuppressWarnings` annotation is used to instruct the compiler to **suppress specific warnings**. It is particularly useful when you are aware of the warning but choose to ignore it (e.g., when using deprecated methods, raw types, unchecked casts, etc.).

### Key Points:

- It allows you to **suppress specific warnings** from the compiler.
- Commonly used values for the `@SuppressWarnings` annotation include "unchecked", "deprecation", and "all".
- The annotation can be applied to classes, methods, fields, and local variables.

### **Example:**

```
java
Copy
public class Main {

    @SuppressWarnings("deprecation") // Suppress warnings for deprecated methods
    public static void main(String[] args) {
        OldClass obj = new OldClass();
        obj.oldMethod(); // No warning due to suppression
    }
}

class OldClass {
    @Deprecated
    public void oldMethod() {
        System.out.println("This is an old method.");
    }
}
```

- **Explanation:**

- In the above example, the `@SuppressWarnings("deprecation")` annotation suppresses the deprecation warning when calling `oldMethod()` even though the method is marked with `@Deprecated`.
- Without this annotation, the compiler would issue a warning when calling a deprecated method.

### **Other Common `SuppressWarnings` Options:**

- **unchecked:** Suppresses warnings related to unchecked generics (e.g., casting raw types).
- **rawtypes:** Suppresses warnings related to the use of raw types in generics.
- **all:** Suppresses all compiler warnings.

---

### **4. When to Use These Annotations?**

- **Use `@Override`:**
    - Always use `@Override` when overriding a method from a superclass or implementing an interface method. It helps catch errors during development, ensuring that you are correctly overriding the method.
  - **Use `@Deprecated`:**
    - Mark methods, classes, or fields as `@Deprecated` when they are no longer recommended for use and will be replaced in future versions of your program. This allows other developers to migrate away from old code.
  - **Use `@SuppressWarnings`:**
    - Use `@SuppressWarnings` when you are intentionally ignoring a specific warning (e.g., deprecated methods, unchecked type casts) and are confident that it is safe to do so in the given context. However, it's best to use this annotation sparingly as it could hide potential issues.
- 

## 5. Conclusion

- `@Override`: Helps prevent errors by ensuring that you are correctly overriding methods in the superclass or interface.
- `@Deprecated`: Marks elements that are outdated and no longer recommended for use.
- `@SuppressWarnings`: Suppresses specific compiler warnings, making the code cleaner but should be used carefully.

These built-in annotations play a significant role in improving code quality, enhancing readability, and preventing common errors during the development process.

---

# Custom Annotations in Java

In Java, you can define your own **custom annotations** to add metadata to your code. Custom annotations are user-defined annotations that you can create to serve your specific needs, such as marking certain methods or classes for special processing, validation, or processing by tools or frameworks.

Creating and using custom annotations allows you to add more meaning to your code, which can be processed later by tools, libraries, or frameworks.

## 1. Defining a Custom Annotation

To define a custom annotation in Java, you use the `@interface` keyword. The `@interface` keyword defines a new annotation type.

### Syntax of a Custom Annotation:

```
java
Copy
public @interface MyAnnotation {
    // elements or attributes (optional)
    String description() default "No description provided";
    int version() default 1;
}
```

- `@interface`: Defines a custom annotation.
- `description()` and `version()`: These are attributes or members of the annotation. These attributes can be used to store values when the annotation is applied.

## 2. Using a Custom Annotation

Once you've defined a custom annotation, you can apply it to classes, methods, fields, constructors, parameters, etc.

### **Example of Applying a Custom Annotation:**

```
java
Copy
// Define the custom annotation
public @interface MyAnnotation {
    String description() default "No description provided";
    int version() default 1;
}

// Apply the custom annotation to a method
public class MyClass {

    @MyAnnotation(description = "This is a custom annotation", version = 2)
    public void myMethod() {
        System.out.println("This is a method with a custom annotation.");
    }
}

```

- **Explanation:**
  - The custom annotation `@MyAnnotation` is applied to the `myMethod()` method.
  - `description` and `version` are passed as attributes while applying the annotation.

### **3. Accessing Custom Annotations at Runtime**

Annotations are often used for code analysis or to provide additional behavior through frameworks. You can access annotations using **reflection** at runtime.

### **Example of Using Reflection to Access Custom Annotations:**

```
java
Copy
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME) // Make the annotation available at runtime
```

```
@Target(ElementType.METHOD) // This annotation can be applied to methods only
public @interface MyAnnotation {
    String description() default "No description provided";
    int version() default 1;
}

public class MyClass {

    @MyAnnotation(description = "Custom annotation example", version = 2)
    public void myMethod() {
        System.out.println("This method is annotated.");
    }

    public static void main(String[] args) throws Exception {
        // Get the method of the class
        Method method = MyClass.class.getMethod("myMethod");

        // Check if the method is annotated with @MyAnnotation
        if (method.isAnnotationPresent(MyAnnotation.class)) {
            // Retrieve the annotation
            MyAnnotation annotation = method.getAnnotation(MyAnnotation.class);

            // Access the values of the annotation
            System.out.println("Description: " + annotation.description());
            System.out.println("Version: " + annotation.version());
        }
    }
}
```

- **Explanation:**
  - The `@MyAnnotation` annotation is applied to `myMethod()`.
  - The `Retention` policy is set to `RUNTIME` so that the annotation can be accessed at runtime.
  - Using reflection, we retrieve the annotation applied to `myMethod()` and access its `description` and `version` values.

#### 4. Commonly Used Meta-Annotations for Custom Annotations

In Java, annotations can have special characteristics defined by **meta-annotations**. These are annotations used to annotate other annotations. Some commonly used meta-annotations include:

### 1. @Retention:

- Defines whether the annotation is available at **runtime**, **compile-time**, or **source-code** level.
- `RetentionPolicy.RUNTIME`: The annotation is available at runtime (for reflection).
- `RetentionPolicy.CLASS`: The annotation is available during compilation but not at runtime.
- `RetentionPolicy.SOURCE`: The annotation is discarded by the compiler and is not included in the bytecode.

```
java
Copy
@Retention(RetentionPolicy.RUNTIME)
public @interface MyAnnotation {
    String value();
}
```

### 2. @Target:

- Specifies where the annotation can be applied (e.g., methods, classes, fields, etc.).
- It can take values from `ElementType` like `METHOD`, `FIELD`, `CLASS`, `PARAMETER`, etc.

```
java
Copy
@Target(ElementType.METHOD) // The annotation can only be applied to methods
public @interface MyMethodAnnotation {
    String info();
}
```

### 3. @Documented:

- Indicates that the annotation should be included in the Javadoc.
- If you want your custom annotation to show up in Javadoc, apply this meta-annotation.

```
java
Copy
@Documented
public @interface MyDocumentationAnnotation {
    String description();
}
```

#### 4. @Inherited:

- Indicates that annotations can be inherited by subclasses.
- If applied to an annotation, subclasses of a class that is annotated with this annotation will also inherit it.

```
java
Copy
@Inherited
public @interface MyInheritedAnnotation {
    String value();
}
```

#### 5. Example of Custom Annotation with Meta-Annotations

```
java
Copy
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME) // Retained at runtime
@Target(ElementType.METHOD)      // Can be applied to methods
@Documented                     // Included in Javadoc
public @interface MyCustomAnnotation {
    String author() default "Unknown";
    String date();
}

public class Example {

    @MyCustomAnnotation(author = "John Doe", date = "2025-01-01")
    public void myMethod() {
```

```

        System.out.println("This is a method with a custom annotation.");
    }

public static void main(String[] args) throws Exception {
    Method method = Example.class.getMethod("myMethod");

    // Checking if the method is annotated with MyCustomAnnotation
    if (method.isAnnotationPresent(MyCustomAnnotation.class)) {
        MyCustomAnnotation annotation = method.getAnnotation(MyCustomAnnotation.class);
        System.out.println("Author: " + annotation.author());
        System.out.println("Date: " + annotation.date());
    }
}
}

```

- **Explanation:**

- `@Retention(RetentionPolicy.RUNTIME)` ensures that the annotation is available at runtime.
- `@Target(ElementType.METHOD)` ensures that the annotation is only applicable to methods.
- `@Documented` ensures that the annotation appears in Javadoc.
- When the method is accessed via reflection, we retrieve the annotation values and print them.

## 6. Real-World Use Cases for Custom Annotations

- **Frameworks:** Many frameworks (e.g., Spring, Hibernate) use custom annotations for configuration, dependency injection, and mapping. For example, annotations like `@Autowired` in Spring are used to inject dependencies automatically.
- **Validation:** Custom annotations can be used in validation frameworks to apply rules to fields, methods, or parameters. For example, in a user registration system, you could create an annotation `@ValidEmail` to ensure the email field is formatted correctly.
- **Code Analysis:** Tools like checkers, linters, or custom code processors can use annotations to identify code that requires special treatment, such as logging, performance tracking, or security checks.

## 7. Conclusion

- **Custom annotations** provide a flexible way to add metadata to your code.
- They are defined using the `@interface` keyword and can have optional members (attributes).

- You can control the retention policy and targets of the annotation with **meta-annotations** like `@Retention`, `@Target`, and `@Documented`.
- Custom annotations are often processed via **reflection** or by frameworks that interpret them for specific purposes (e.g., dependency injection, validation, or code analysis).

Custom annotations are powerful tools that allow developers to introduce additional layers of meaning and behavior to the code, which can be automatically processed and acted upon by tools, frameworks, or custom-built processing systems.

---

Core Java Notes By AJAY RAZZ

# Lambda Expressions in Java

In Java, **lambda expressions** were introduced in **Java 8** as a way to provide a clear and concise syntax for writing **functional interfaces** (interfaces with just one abstract method). Lambda expressions help make code more readable, expressive, and less verbose by allowing you to treat functionality as a method argument, or to create a **function object**.

Lambda expressions are particularly useful in working with **functional programming** concepts in Java, such as filtering, mapping, and reducing data in collections.

## 1. Syntax of Lambda Expressions

The syntax of a lambda expression in Java is:

```
java
Copy
(parameters) -> expression
```

Where:

- **parameters**: The parameters that the lambda expression will take (like method parameters). These can be a single parameter, multiple parameters, or even none.
- **->**: The lambda operator, which separates parameters from the expression or block of code.
- **expression**: The body of the lambda, which can be a single expression or a block of statements.

### Basic Syntax Examples:

#### 1. No Parameters:

```
java
Copy
() -> System.out.println("Hello, World!");
```

## 2. One Parameter:

```
java
Copy
(x) -> x * x;
```

## 3. Multiple Parameters:

```
java
Copy
(x, y) -> x + y;
```

## 4. With Block of Code:

```
java
Copy
(x, y) -> {
    int sum = x + y;
    return sum;
}
```

## 2. Lambda Expression Example

Let's consider a simple example of a lambda expression:

```
java
Copy
// Define a functional interface
@FunctionalInterface
interface MyOperation {
    int operate(int a, int b); // Abstract method
}

public class LambdaExample {
    public static void main(String[] args) {
        // Lambda expression to add two numbers
```

```

        MyOperation add = (a, b) -> a + b;

        System.out.println("Result: " + add.operate(5, 3)); // Output: Result: 8
    }
}

```

- **Explanation:**

- The `MyOperation` interface is a functional interface with one abstract method `operate`.
- The lambda expression `(a, b) -> a + b` is used to define the behavior of the `operate` method, which adds two integers.
- The lambda expression is assigned to the `add` variable, and it's invoked using `add.operate(5, 3)`.

### 3. Functional Interfaces

A **functional interface** is an interface that has exactly one abstract method, and can have multiple default or static methods. Lambda expressions are used to provide implementations for the abstract method of these interfaces.

#### Common Functional Interfaces in Java:

Java provides several commonly used functional interfaces in the `java.util.function` package:

- `Function<T, R>`: Represents a function that takes an argument of type `T` and returns a result of type `R`.
- `Predicate<T>`: Represents a boolean-valued function of one argument of type `T`.
- `Consumer<T>`: Represents an operation that accepts a single input argument and returns no result.
- `Supplier<T>`: Represents a supplier of results of type `T`.
- `UnaryOperator<T>`: Represents a function that accepts a single argument of type `T` and returns a result of the same type `T`.
- `BinaryOperator<T>`: Represents a function that accepts two arguments of the same type `T` and returns a result of the same type `T`.

#### Example using `Predicate`:

```

java
Copy
import java.util.function.Predicate;

```

```

public class LambdaExample {
    public static void main(String[] args) {
        // Predicate to check if a number is even
        Predicate<Integer> isEven = num -> num % 2 == 0;

        System.out.println(isEven.test(4)); // Output: true
        System.out.println(isEven.test(7)); // Output: false
    }
}

```

- **Explanation:**

- The lambda expression `num -> num % 2 == 0` implements the `test` method of the `Predicate` interface to check if a number is even.

## 4. Benefits of Lambda Expressions

- **Conciseness:** Lambda expressions reduce boilerplate code, making your code more concise and readable.
- **Functional Style:** Encourages the use of functional programming techniques such as passing behavior as parameters (higher-order functions), making it easier to work with collections and stream-based APIs.
- **Cleaner Code:** Especially when using with `streams` or when working with API methods like `map()`, `filter()`, and `forEach()`, lambda expressions simplify operations that would require an anonymous class or verbose code otherwise.
- **Parallelism:** Lambda expressions, combined with streams, make it easier to express operations that can be parallelized.

## 5. Lambda Expressions with Collections (Streams)

The Java **Stream API** (introduced in Java 8) makes heavy use of lambda expressions, allowing for functional programming-style operations on collections of data.

### Example: Using `forEach()` with a Lambda Expression

```

java
Copy
import java.util.List;
import java.util.Arrays;

```

```
public class LambdaExample {  
    public static void main(String[] args) {  
        List<String> list = Arrays.asList("Apple", "Banana", "Cherry");  
  
        // Using lambda expression to print each element of the list  
        list.forEach(item -> System.out.println(item));  
    }  
}
```

- **Explanation:**

- The `forEach()` method is a terminal operation on a stream (in this case, a `List`) that takes a `Consumer` as an argument.
- The lambda expression `item -> System.out.println(item)` is passed as a `Consumer` to print each item of the list.

### Example: Using `map()` and `filter()` with Lambda Expressions

```
java  
Copy  
import java.util.List;  
import java.util.Arrays;  
import java.util.stream.Collectors;  
  
public class LambdaExample {  
    public static void main(String[] args) {  
        List<String> list = Arrays.asList("Apple", "Banana", "Cherry", "Date");  
  
        // Using lambda expressions with stream() to transform and filter data  
        List<String> result = list.stream()  
            .filter(item -> item.length() > 5) // Filter strings with length > 5  
            .map(String::toUpperCase) // Convert each string to uppercase  
            .collect(Collectors.toList());  
  
        System.out.println(result); // Output: [BANANA, CHERRY]  
    }  
}
```

- **Explanation:**

- The `stream()` method creates a stream from the list.
- The `filter()` method filters elements based on a condition defined in the lambda expression.
- The `map()` method transforms the elements (in this case, converting them to uppercase).
- The result is collected into a new list using `collect()`.

## 6. Lambda Expressions vs Anonymous Classes

Lambda expressions provide a more concise and readable alternative to **anonymous inner classes**.

### Example: Using an Anonymous Class

```
java
Copy
public class LambdaExample {
    public static void main(String[] args) {
        // Using an anonymous class to implement the Runnable interface
        Runnable r = new Runnable() {
            @Override
            public void run() {
                System.out.println("Running in a thread");
            }
        };
        new Thread(r).start();
    }
}
```

### Example: Using a Lambda Expression

```
java
Copy
public class LambdaExample {
    public static void main(String[] args) {
        // Using a lambda expression to implement the Runnable interface
        Runnable r = () -> System.out.println("Running in a thread");
        new Thread(r).start();
    }
}
```

}

- **Explanation:**

- The lambda expression simplifies the anonymous class syntax by eliminating boilerplate code like `new Runnable() { ... }`.

## 7. Conclusion

- **Lambda expressions** are a powerful feature in Java that enable functional programming techniques, making the code concise, readable, and expressive.
- They are primarily used in **functional interfaces**, and work seamlessly with Java's **Stream API** to perform operations on collections.
- Lambda expressions provide a cleaner, more maintainable alternative to using anonymous classes for implementing functional interfaces.

By leveraging lambda expressions, Java developers can write code that is more declarative, making it easier to express complex operations and transformations on data in a clear and concise manner.

Core Java Notes By AJAY RAZZ

---

# Functional Interfaces in Java

A **functional interface** in Java is an interface that contains **exactly one abstract method**. Functional interfaces are the foundation of **lambda expressions** and **method references** in Java, introduced in **Java 8**. They enable the use of functional programming techniques, allowing behavior to be passed as parameters or returned from methods.

Functional interfaces can have multiple **default** or **static methods**, but they must have **only one abstract method**.

## Key Characteristics of Functional Interfaces:

1. **One Abstract Method:** A functional interface must have exactly one abstract method. It can have multiple default and static methods.
2. **@FunctionalInterface Annotation:** While not required, Java provides the `@FunctionalInterface` annotation to indicate that the interface is intended to be a functional interface. This helps catch errors at compile time (e.g., if the interface has more than one abstract method).
3. **Can Be Used with Lambda Expressions:** Functional interfaces are commonly used with lambda expressions or method references, allowing concise and functional-style code.

## 1. Basic Example of a Functional Interface

Let's start by creating a simple functional interface:

```
java
Copy
@FunctionalInterface
interface MyOperation {
    int operate(int a, int b); // Abstract method
}
```

- `MyOperation` is a functional interface because it has exactly one abstract method, `operate`.
- The `@FunctionalInterface` annotation ensures that this interface is intended to be a functional interface. If you accidentally add another abstract method, the compiler will generate an error.

## Using the Functional Interface with Lambda Expression:

```
java
Copy
public class FunctionalInterfaceExample {
    public static void main(String[] args) {
        // Lambda expression implementing the 'operate' method of MyOperation
        MyOperation add = (a, b) -> a + b; // Addition operation

        System.out.println("Result: " + add.operate(5, 3)); // Output: Result: 8
    }
}
```

- The lambda expression `(a, b) -> a + b` provides an implementation for the `operate` method of the `MyOperation` interface.
- The lambda expression is assigned to the variable `add`, and you can invoke `add.operate(5, 3)` to perform the operation.

## 2. Common Built-in Functional Interfaces

Java 8 introduced a number of built-in functional interfaces in the `java.util.function` package. These interfaces are used for common tasks such as working with **predicates**, **functions**, **consumers**, and **suppliers**.

### a) `Predicate<T>`: Represents a condition that evaluates to a boolean value.

```
java
Copy
@FunctionalInterface
interface Predicate<T> {
    boolean test(T t); // Abstract method that tests the condition
}
```

Example of using `Predicate`:

```
java
Copy
import java.util.function.Predicate;
```

```

public class PredicateExample {
    public static void main(String[] args) {
        Predicate<Integer> isEven = num -> num % 2 == 0;

        System.out.println(isEven.test(4)); // Output: true
        System.out.println(isEven.test(5)); // Output: false
    }
}

```

- `Predicate<T>` has a single abstract method `test()` which returns a boolean value based on the input.

**b) `Function<T, R>`: Represents a function that accepts an argument of type `T` and produces a result of type `R`.**

```

java
Copy
@FunctionalInterface
interface Function<T, R> {
    R apply(T t); // Abstract method that applies a function
}

```

Example of using `Function`:

```

java
Copy
import java.util.function.Function;

public class FunctionExample {
    public static void main(String[] args) {
        Function<Integer, String> convertToString = num -> "Number: " + num;

        System.out.println(convertToString.apply(10)); // Output: Number: 10
    }
}

```

- `Function<T, R>` is used for transforming one type into another, such as converting an `Integer` to a `String`.

c) **Consumer<T>**: Represents an operation that accepts a single input argument of type  $\tau$  and returns no result.

```
java
Copy
@FunctionalInterface
interface Consumer<T> {
    void accept(T t); // Abstract method that accepts an argument and performs an action
}
```

Example of using Consumer:

```
java
Copy
import java.util.function.Consumer;

public class ConsumerExample {
    public static void main(String[] args) {
        Consumer<String> printMessage = message -> System.out.println(message);
        printMessage.accept("Hello, World!"); // Output: Hello, World!
    }
}
```

- `Consumer<T>` is used for performing an action without returning any result (like printing or modifying an object).

d) **Supplier<T>**: Represents a supplier that provides a result of type  $\tau$  without taking any input.

```
java
Copy
@FunctionalInterface
interface Supplier<T> {
    T get(); // Abstract method that supplies a result
}
```

Example of using Supplier:

```
java
Copy
import java.util.function.Supplier;

public class SupplierExample {
    public static void main(String[] args) {
        Supplier<String> getGreeting = () -> "Hello from Supplier!";
        System.out.println(getGreeting.get()); // Output: Hello from Supplier!
    }
}
```

- `Supplier<T>` is used when you want to generate or supply a value of type `T` without any input.

### 3. Default and Static Methods in Functional Interfaces

A functional interface can have **default** and **static** methods in addition to its single abstract method. These methods can be implemented directly in the interface.

- **Default methods** allow you to provide a default implementation of a method in the interface.
- **Static methods** can be called directly on the interface and provide utility methods.

#### Example:

```
java
Copy
@FunctionalInterface
interface MyFunctionalInterface {
    int operate(int a, int b); // Abstract method

    // Default method
    default void printMessage() {
        System.out.println("This is a default method in the functional interface.");
    }

    // Static method
}
```

```
static void staticMethod() {  
    System.out.println("This is a static method in the functional interface.");  
}  
}
```

- `printMessage()` is a default method that can be used without overriding in the implementing class.
- `staticMethod()` is a static method that can be called without an instance of the interface.

Example of using the default and static methods:

```
java  
Copy  
public class FunctionalInterfaceExample {  
    public static void main(String[] args) {  
        MyFunctionalInterface add = (a, b) -> a + b;  
  
        // Calling the abstract method  
        System.out.println("Result: " + add.operate(5, 3)); // Output: Result: 8  
  
        // Calling the default method  
        add.printMessage(); // Output: This is a default method in the functional interface.  
  
        // Calling the static method  
        MyFunctionalInterface.staticMethod(); // Output: This is a static method in the functional interface.  
    }  
}
```

## 4. Combining Functional Interfaces

You can **combine functional interfaces** using methods such as `andThen()`, `compose()`, etc., which are provided by functional interfaces like `Function<T, R>`.

Example with Function chaining:

```
java  
Copy
```

```

import java.util.function.Function;

public class FunctionChainingExample {
    public static void main(String[] args) {
        Function<Integer, Integer> multiplyBy2 = x -> x * 2;
        Function<Integer, Integer> add5 = x -> x + 5;

        // Chaining functions: first multiply by 2, then add 5
        Function<Integer, Integer> result = multiplyBy2.andThen(add5);

        System.out.println(result.apply(3)); // Output: 11 (3 * 2 = 6, then 6 + 5 = 11)
    }
}

```

- `andThen()` allows chaining functions where the output of one function becomes the input to the next function.

## 5. Conclusion

- **Functional interfaces** in Java are interfaces with exactly one abstract method and can have multiple default or static methods.
- Java 8's **lambda expressions** work hand-in-hand with functional interfaces, making it easy to implement them concisely and expressively.
- Common built-in functional interfaces like `Predicate`, `Function`, `Consumer`, and `Supplier` are used for a variety of purposes such as filtering, transforming, or consuming data.
- Functional interfaces enable the functional programming paradigm in Java, making it easier to handle operations on data (especially with the **Stream API**).

Understanding and using functional interfaces effectively allows you to write cleaner, more maintainable, and expressive code, especially when working with collections, parallel operations, or handling behavior as data.

---

# Using Lambda Expressions in Java

Lambda expressions, introduced in **Java 8**, are a concise way to represent an instance of a functional interface (an interface with exactly one abstract method). They allow you to write more compact, readable, and expressive code, especially when working with **functional programming** paradigms. Lambda expressions enable behavior to be passed around as arguments or return values in a more succinct form than using anonymous inner classes.

## 1. Syntax of Lambda Expressions

A lambda expression in Java follows this general syntax:

```
java
Copy
(parameters) -> expression
```

Where:

- **parameters**: The input parameters (comma-separated) for the lambda expression. This can be zero or more parameters.
- **->**: The arrow operator that separates the parameters from the body of the lambda expression.
- **expression**: The body of the lambda expression. It can either be a single expression or a block of code.

## 2. Lambda Expressions with Functional Interfaces

A lambda expression works with **functional interfaces** (interfaces with exactly one abstract method). Java provides several built-in functional interfaces in the `java.util.function` package, which are commonly used with lambda expressions.

## 3. Basic Examples of Lambda Expressions

Here are some examples of how lambda expressions can be used in Java:

### **Example 1: Lambda Expression with No Parameters**

```
java
Copy
public class LambdaExample {
    public static void main(String[] args) {
        // Lambda expression with no parameters
        Runnable helloWorld = () -> System.out.println("Hello, World!");
        helloWorld.run(); // Output: Hello, World!
    }
}
```

- In this example, the `Runnable` interface is a functional interface with a single abstract method `run()`. The lambda expression `() -> System.out.println("Hello, World!")` implements this method without the need for a separate class.

### **Example 2: Lambda Expression with One Parameter**

```
java
Copy
public class LambdaExample {
    public static void main(String[] args) {
        // Lambda expression with one parameter
        // This lambda expression calculates the square of a number
        Function<Integer, Integer> square = x -> x * x;

        System.out.println("Square of 5: " + square.apply(5)); // Output: 25
    }
}
```

- `Function<Integer, Integer>` is a functional interface that represents a function that takes one argument and returns a result. The lambda expression `(x) -> x * x` calculates the square of the input.

### **Example 3: Lambda Expression with Multiple Parameters**

```
java
Copy
```

```

public class LambdaExample {
    public static void main(String[] args) {
        // Lambda expression with multiple parameters
        // This lambda expression adds two numbers
        BiFunction<Integer, Integer, Integer> add = (a, b) -> a + b;

        System.out.println("Sum: " + add.apply(3, 7)); // Output: 10
    }
}

```

- `BiFunction<Integer, Integer, Integer>` is a functional interface that accepts two parameters and returns a result. The lambda expression `(a, b) -> a + b` adds the two input numbers.

#### **Example 4: Lambda Expression with Block of Code**

```

java
Copy
public class LambdaExample {
    public static void main(String[] args) {
        // Lambda expression with a block of code
        // This lambda expression checks if a number is positive
        Predicate<Integer> isPositive = num -> {
            if (num > 0) {
                return true;
            } else {
                return false;
            }
        };

        System.out.println("Is 5 positive? " + isPositive.test(5)); // Output: true
        System.out.println("Is -5 positive? " + isPositive.test(-5)); // Output: false
    }
}

```

- `Predicate<Integer>` is a functional interface that tests a condition and returns a boolean. The lambda expression checks if the number is positive using a block of code.

## 4. Lambda Expressions with Collections (Stream API)

Lambda expressions are commonly used with the **Stream API**, which was introduced in Java 8 to process collections of data in a functional style. Using lambda expressions with streams allows you to perform operations like filtering, mapping, and reducing in a concise and readable way.

### Example 5: Using Lambda Expressions with `forEach()`

```
java
Copy
import java.util.Arrays;
import java.util.List;

public class LambdaExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");
        // Using lambda expression with forEach to print each name
        names.forEach(name -> System.out.println(name));
    }
}
```

- The `forEach()` method in the `List` interface takes a `Consumer` (a functional interface), and the lambda expression `name -> System.out.println(name)` is used to print each element.

### Example 6: Using Lambda Expressions with `filter()` and `map()`

```
java
Copy
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

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

```

        // Using lambda expressions with filter() and map() to transform data
        List<String> result = names.stream()
            .filter(name -> name.length() > 3)    // Only names with length > 3
            .map(name -> name.toUpperCase())        // Convert names to uppercase
            .collect(Collectors.toList());           // Collect the result into a list

        System.out.println(result);   // Output: [ALICE, CHARLIE, DAVID]
    }
}

```

- The `filter()` method uses a lambda expression to filter names with more than 3 characters.
- The `map()` method uses a lambda expression to convert the names to uppercase.
- Finally, `collect()` is used to collect the results into a new list.

### **Example 7: Using Lambda Expressions with `reduce()`**

```

java
Copy
import java.util.Arrays;
import java.util.List;
import java.util.Optional;

public class LambdaExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

        // Using lambda expression with reduce to calculate the sum of the list
        Optional<Integer> sum = numbers.stream()
            .reduce((a, b) -> a + b); // Lambda expression to add two numbers

        sum.ifPresent(System.out::println); // Output: 15
    }
}

```

- The `reduce()` method uses a lambda expression to accumulate the elements of the list (in this case, by summing them).
- The result is wrapped in an `Optional` to handle cases where the stream is empty.

## 5. Lambda Expressions in Event Handling

Lambda expressions can also be used in **event handling** (e.g., GUI applications, button clicks) to implement event listeners more concisely. For instance, in Java's Swing framework, lambda expressions can be used to define actions for buttons.

### Example 8: Lambda Expression for Button Click (Swing)

```
java
Copy
import javax.swing.*;
import java.awt.event.ActionListener;

public class LambdaExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Lambda Example");
        JButton button = new JButton("Click Me");

        // Using lambda expression for button click event handling
        button.addActionListener(e -> System.out.println("Button clicked!"));

        frame.add(button);
        frame.setSize(200, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

- The `addActionListener()` method expects an `ActionListener` interface, and the lambda expression `e -> System.out.println("Button clicked!")` provides the implementation for the `actionPerformed()` method.

## 6. Benefits of Using Lambda Expressions

- Conciseness:** Lambda expressions reduce boilerplate code, making it more readable and concise.
- Improved Readability:** By using lambda expressions, you can express behavior (like filtering, transforming, and handling events) in a clear and declarative manner.

- **Functional Programming:** Lambda expressions support functional programming principles, such as passing behavior as parameters or returning functions from methods.
- **Seamless with Collections and Streams:** Lambda expressions work seamlessly with Java's **Stream API**, allowing you to perform complex operations on data in a more functional style.

## 7. Conclusion

- **Lambda expressions** are a powerful feature introduced in Java 8 that enable you to write more concise and expressive code, especially when dealing with **functional interfaces**.
- They can be used with **collections**, **streams**, **event handling**, and **higher-order functions**, allowing you to express complex operations in a cleaner and more functional style.
- Lambda expressions work well with Java's **Stream API** to process collections and data in a functional programming paradigm.

By mastering lambda expressions, you can make your Java code more modern, readable, and maintainable, especially when working with functional-style operations on data.

---

# Java 8 Features

Java 8, released in **March 2014**, introduced several new features and improvements that significantly enhanced the language and its APIs, making Java more expressive, concise, and powerful. These features brought Java closer to the functional programming paradigm, making it easier to work with collections, perform parallel operations, and create more expressive and efficient code.

Here are some of the key features introduced in **Java 8**:

---

## 1. Lambda Expressions

Lambda expressions allow you to write instances of functional interfaces in a more concise and readable way. They enable you to treat functionality as a method argument or to define a function object without the need for verbose anonymous classes.

**Syntax:**

```
java
Copy
(parameters) -> expression
```

**Example:**

```
java
Copy
// A lambda expression to calculate the sum of two numbers
BinaryOperator<Integer> sum = (a, b) -> a + b;
System.out.println(sum.apply(5, 3)); // Output: 8
```

Lambda expressions are commonly used with **functional interfaces** and **Stream API** for concise code.

---

## 2. Functional Interfaces

A **functional interface** is an interface with exactly one abstract method, and it may contain multiple default or static methods. Java 8 introduced functional interfaces and enables them to be implemented via lambda expressions.

**Example:**

```
java
Copy
@FunctionalInterface
interface MyFunctionalInterface {
    void myMethod();
}
```

Some common functional interfaces in Java 8 include:

- Runnable
- Comparator<T>
- Callable<V>
- Predicate<T>
- Function<T, R>
- Consumer<T>

---

## 3. Stream API

The **Stream API** provides a powerful and efficient way to process sequences of elements (like collections) in a functional style. It supports operations like **filtering**, **mapping**, **reducing**, **collecting**, and **sorting** in a more declarative and expressive manner.

- **Streams** can be **sequential** or **parallel**.

- You can chain multiple operations like **map()**, **filter()**, and **reduce()**.

**Example:**

```
java
Copy
import java.util.Arrays;
import java.util.List;

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

        // Filter names that have more than 3 characters and convert them to uppercase
        names.stream()
            .filter(name -> name.length() > 3)
            .map(String::toUpperCase)
            .forEach(System.out::println); // Output: ALICE, CHARLIE, DAVID
    }
}
```

- **Intermediate operations** (e.g., `filter()`, `map()`) return a new stream and are **lazy**.
- **Terminal operations** (e.g., `forEach()`, `collect()`, `reduce()`) trigger the actual processing of the stream.

---

## 4. Default and Static Methods in Interfaces

Java 8 allows **default methods** in interfaces. These are methods with a default implementation, so classes that implement the interface don't need to provide their own implementation.

- **Static methods** in interfaces are also allowed, which means you can have utility methods directly inside the interface.

**Example:**

```
java
Copy
interface MyInterface {
    // Default method
    default void defaultMethod() {
        System.out.println("This is a default method.");
    }

    // Static method
    static void staticMethod() {
        System.out.println("This is a static method.");
    }
}

public class InterfaceExample implements MyInterface {
    public static void main(String[] args) {
        MyInterface.staticMethod();    // Output: This is a static method.

        MyInterface obj = new InterfaceExample();
        obj.defaultMethod();    // Output: This is a default method.
    }
}
```

- Default methods allow adding new functionality to interfaces without breaking existing implementations.
- Static methods in interfaces are useful for utility functions.

---

## 5. Method References

Method references provide a shorthand for calling methods using lambda expressions. They allow you to refer to a method directly by its name.

### Types of Method References:

- **Static methods:** `ClassName::methodName`
- **Instance methods:** `instance::methodName`

- **Constructor references:** `ClassName::new`

**Example:**

```
java
Copy
import java.util.Arrays;
import java.util.List;

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

        // Using method reference to print each name
        names.forEach(System.out::println); // Output: Alice, Bob, Charlie
    }
}
```

- `System.out::println` is a method reference to the `println` method.

## 6. Optional Class

The `Optional` class is a container that may or may not contain a non-null value. It is used to prevent `NullPointerExceptions` and to make code more expressive when dealing with potentially null values.

**Example:**

```
java
Copy
import java.util.Optional;

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

```

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

// Using ifPresent to check if value is present
name.ifPresent(System.out::println); // Output: Alice

// Using orElse to provide a default value if empty
System.out.println(name.orElse("Default Name")); // Output: Alice
}

}

```

- `Optional` helps to make code more readable and safer by explicitly handling null values.
- 

## 7. Date and Time API (`java.time`)

The `java.time` package introduced a new, immutable, and thread-safe API for working with dates, times, and durations. It is a replacement for the old `Date` and `Calendar` classes.

### Key Classes:

- `LocalDate`: Represents a date (year, month, day).
- `LocalTime`: Represents a time (hour, minute, second).
- `LocalDateTime`: Combines date and time.
- `ZonedDateTime`: Represents date and time with a time zone.

### Example:

```

java
Copy
import java.time.LocalDate;
import java.time.LocalTime;
import java.time.LocalDateTime;

public class DateTimeExample {

```

```

public static void main(String[] args) {
    LocalDate date = LocalDate.now();
    LocalTime time = LocalTime.now();
    LocalDateTime dateTime = LocalDateTime.now();

    System.out.println("Current Date: " + date); // Output: Current Date: YYYY-MM-DD
    System.out.println("Current Time: " + time); // Output: Current Time: HH:MM:SS
    System.out.println("Current DateTime: " + dateTime); // Output: Current DateTime: YYYY-MM-DDTHH:MM:SS
}
}

```

The **Date and Time API** is more modern, precise, and easier to use compared to the previous `Date` and `Calendar` classes.

---

## 8. Nashorn JavaScript Engine

Java 8 introduced the **Nashorn** JavaScript engine, which allows Java applications to execute JavaScript code natively.

- Nashorn offers better performance and improved integration with Java applications compared to the older **Rhino** engine.
- It provides the ability to execute JavaScript code directly from Java.

### Example:

```

java
Copy
import javax.script.*;

public class NashornExample {
    public static void main(String[] args) throws ScriptException {
        ScriptEngine engine = new ScriptEngineManager().getEngineByName("nashorn");
        String script = "var x = 10; var y = 20; x + y;";
        Object result = engine.eval(script);
        System.out.println(result); // Output: 30
    }
}

```

Nashorn enables Java to integrate more easily with JavaScript code.

---

## 9. Parallel Streams

Java 8 introduced the ability to perform operations on streams in **parallel**. With parallel streams, Java can automatically divide the workload across multiple threads to take advantage of multi-core processors, improving performance in certain scenarios.

### Example:

```
java
Copy
import java.util.Arrays;
import java.util.List;

public class ParallelStreamsExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

        // Parallel stream to perform sum of squares
        int sumOfSquares = numbers.parallelStream()
            .mapToInt(x -> x * x)
            .sum();

        System.out.println("Sum of squares: " + sumOfSquares); // Output: 385
    }
}
```

- `parallelStream()` can be used to easily process collections in parallel.

---

## 10. CompletableFuture API

The **CompletableFuture** class introduced in Java 8 makes it easier to handle asynchronous programming. It allows you to write non-blocking code and process results asynchronously, improving performance in I/O-bound applications.

### Example:

```
java
Copy
import java.util.concurrent.CompletableFuture;

public class CompletableFutureExample {
    public static void main(String[] args) {
        CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> {
            return 10 + 20;
        });

        future.thenAccept(result -> System.out.println("Result: " + result)); // Output: Result: 30
    }
}
```

- **CompletableFuture** is used for composing asynchronous tasks and handling their results when they become available.

---

## Conclusion

Java 8 introduced a major shift in the language, bringing **functional programming features** to the forefront. The key features include:

- **Lambda expressions** and **functional interfaces** for more concise code.
- **Stream API** for handling collections and performing parallel processing.
- New **Date and Time API** for modern date and time handling.
- **Optional** to handle null values safely.
- **CompletableFuture** for easier asynchronous programming.

These features, combined with the other enhancements like **default methods in interfaces**, **method references**, and **Nashorn JavaScript engine**, have significantly improved Java, making it more powerful and expressive, especially for modern development practices.

---

# Default and Static Methods in Interfaces in Java

In Java 8, interfaces were enhanced with the ability to have **default** and **static** methods. This was a significant change because, prior to Java 8, interfaces could only contain **abstract methods** (methods without implementations). The addition of **default** and **static** methods made interfaces more powerful, allowing them to provide method implementations while still maintaining their role as abstract contracts.

Let's explore **default** and **static** methods in interfaces in more detail:

---

## 1. Default Methods in Interfaces

**Default methods** are methods with a default implementation in an interface. They allow you to add new methods to interfaces without breaking existing implementations of the interface. This means that classes implementing the interface don't need to provide an implementation for default methods unless they want to override them.

**Syntax:**

```
java
Copy
interface MyInterface {
    // Default method
    default void defaultMethod() {
        System.out.println("This is a default method.");
    }
}
```

### Key Points About Default Methods:

- **Provides a default implementation:** A method in the interface can provide its own implementation.
- **Can be overridden:** Classes implementing the interface can override the default method if they need to provide a specific implementation.
- **Helps with backward compatibility:** You can add new methods to an interface without affecting the existing implementing classes.

### Example: Using Default Methods

```
java
Copy
interface Vehicle {
    // Default method
    default void start() {
        System.out.println("Starting the vehicle.");
    }

    // Abstract method (to be implemented by classes)
    void drive();
}

class Car implements Vehicle {
    // Overriding the abstract method
    public void drive() {
        System.out.println("Driving the car.");
    }
}
```

```
public class DefaultMethodExample {  
    public static void main(String[] args) {  
        Vehicle vehicle = new Car();  
        vehicle.start(); // Output: Starting the vehicle.  
        vehicle.drive(); // Output: Driving the car.  
    }  
}
```

In the above example, the `Vehicle` interface defines a default method `start()`. The `Car` class implements the `Vehicle` interface and overrides the `drive()` method but doesn't need to provide an implementation for `start()`, as it's already provided by the interface.

### Overriding a Default Method:

```
java  
Copy  
class ElectricCar implements Vehicle {  
    // Overriding the default method  
    @Override  
    public void start() {  
        System.out.println("Starting the electric car silently.");  
    }  
  
    public void drive() {  
        System.out.println("Driving the electric car.");  
    }  
}  
  
public class DefaultMethodOverrideExample {  
    public static void main(String[] args) {  
        Vehicle vehicle = new ElectricCar();  
        vehicle.start(); // Output: Starting the electric car silently.  
        vehicle.drive(); // Output: Driving the electric car.  
    }  
}
```

In this case, the `ElectricCar` class overrides the default method `start()` with its own implementation.

---

## 2. Static Methods in Interfaces

**Static methods** in interfaces are methods that belong to the interface itself, not to the instances of the implementing classes. Static methods can be invoked using the interface name directly.

Static methods in interfaces are similar to static methods in classes:

- **Can be called without creating an instance** of the interface.
- **Cannot be overridden** by implementing classes (because static methods belong to the interface, not the instance).

**Syntax:**

```
java
Copy
interface MyInterface {
    // Static method
    static void staticMethod() {
        System.out.println("This is a static method in the interface.");
    }
}
```

### Key Points About Static Methods:

- **Belongs to the interface, not the instance:** Static methods are not inherited by implementing classes. They are called using the interface name.
- **Cannot be overridden:** Static methods cannot be overridden in the implementing classes. They are tied to the interface.
- **Can be called using the interface:** You access static methods through the interface itself.

### Example: Using Static Methods

```
java
Copy
```

```

interface Calculator {
    // Static method
    static int add(int a, int b) {
        return a + b;
    }
}

public class StaticMethodExample {
    public static void main(String[] args) {
        // Calling the static method using the interface name
        int result = Calculator.add(10, 20);
        System.out.println("Result of addition: " + result); // Output: Result of addition: 30
    }
}

```

In the example above, the `add()` method is a static method of the `Calculator` interface. It is called using the interface name (`calculator.add()`), not through an instance.

### 3. Key Differences Between Default and Static Methods

Feature	Default Methods	Static Methods
<b>Definition</b>	Methods with a default implementation that can be overridden.	Methods that belong to the interface, not to instances of implementing classes.
<b>Usage</b>	Can be called on an instance of a class that implements the interface.	Must be called on the interface itself.
<b>Inheritance</b>	Inherited by implementing classes, but can be overridden.	Not inherited by implementing classes.
<b>Override</b>	Can be overridden by implementing classes if needed.	Cannot be overridden by implementing classes.
<b>Instance vs. Interface</b>	Works on instances of implementing classes.	Belongs to the interface itself and is not tied to an instance.
<b>Access</b>	Accessed via the instance of the class.	Accessed via the interface name.

---

## 4. Use Cases for Default and Static Methods

- **Default Methods:**
  - Use them when you want to provide a **default implementation** for methods, ensuring that existing classes implementing the interface do not break when a new method is added to the interface.
  - Use them when you want to share common behavior across multiple classes but still allow for custom behavior when needed (through overriding).
- **Static Methods:**
  - Use static methods in interfaces for **utility functions** that are relevant to the interface but do not require access to instance-specific data.
  - They are useful for providing default functionality or helper methods that can be called without creating an instance of the implementing class.

## 5. Example with Both Default and Static Methods

```
java
Copy
interface Payment {
    // Default method
    default void makePayment() {
        System.out.println("Payment made through default method.");
    }

    // Static method
    static void paymentInfo() {
        System.out.println("Payment system information.");
    }
}

class CreditCardPayment implements Payment {
    @Override
```

```

public void makePayment() {
    System.out.println("Payment made using Credit Card.");
}
}

public class PaymentExample {
    public static void main(String[] args) {
        // Call static method using the interface name
        Payment.paymentInfo(); // Output: Payment system information.

        // Call default method through an instance
        Payment payment = new CreditCardPayment();
        payment.makePayment(); // Output: Payment made using Credit Card.
    }
}

```

- **Static Method:** The `paymentInfo()` method is called directly on the interface.
- **Default Method:** The `makePayment()` method is overridden in the `CreditCardPayment` class, and its behavior is called through the instance.

## Conclusion

- **Default methods** allow you to add method implementations to interfaces without breaking existing code.
  - **Static methods** belong to the interface itself and can be called using the interface name, providing utility methods or common functionality.
  - Both **default** and **static methods** provide more flexibility in interfaces, making them more powerful and useful in a wide range of scenarios.
-

# Streams API in Java (Java 8)

The **Streams API** was introduced in **Java 8** as part of the **java.util.stream** package. It enables a functional approach to working with collections of data by allowing operations to be performed in a declarative manner. The Stream API helps simplify the code that deals with bulk data processing, such as **filtering**, **mapping**, **reducing**, and **collecting**.

The Streams API provides a **high-level abstraction** for processing sequences of elements (such as collections or arrays) in a functional style. With streams, you can easily perform operations like searching, sorting, filtering, and transforming data in a more concise and readable manner.

---

## Key Concepts of Streams API

1. **Stream:**
  - A **Stream** represents a sequence of elements that can be processed in parallel or sequentially. It is not a data structure (like a `List` or `Set`), but rather an abstraction that allows you to perform operations on data.
2. **Stream Operations:** There are two types of stream operations:
  - **Intermediate Operations:** These operations transform a stream into another stream. They are **lazy**, meaning they are not executed until a terminal operation is invoked. Examples include `filter()`, `map()`, and `sorted()`.
  - **Terminal Operations:** These operations produce a result or a side-effect. Once a terminal operation is invoked, the stream is considered consumed, and no further operations can be performed. Examples include `collect()`, `forEach()`, and `reduce()`.
3. **Internal Iteration:**
  - Streams provide **internal iteration**, which means that the framework handles the iteration of data behind the scenes. You don't need to write explicit `for` or `while` loops. Instead, you can apply operations on the stream, and the stream will handle iteration automatically.
4. **Laziness:**
  - Streams are **lazy** by nature, meaning that intermediate operations (like `map()` or `filter()`) are not executed until a terminal operation is invoked. This allows for optimization, especially when working with large datasets.

## Types of Streams

- **Sequential Streams:** By default, streams are processed sequentially (one element after another).
  - **Parallel Streams:** You can process a stream in parallel to leverage multi-core processors, which can improve performance for large datasets. Parallel streams divide the data into smaller chunks and process them concurrently.
- 

## Stream API: Common Operations

### 1. Creating Streams

Streams can be created from different data sources like collections, arrays, or I/O channels.

- **From a Collection** (e.g., List, Set):

```
java
Copy
import java.util.Arrays;
import java.util.List;

public class StreamExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");
        names.stream().forEach(System.out::println); // Output: Alice, Bob, Charlie, David
    }
}
```

- **From an Array:**

```
java
Copy
import java.util.Arrays;

public class StreamExample {
```

```
public static void main(String[] args) {
    int[] numbers = {1, 2, 3, 4, 5};
    Arrays.stream(numbers).forEach(System.out::println); // Output: 1, 2, 3, 4, 5
}
}
```

- **From a Range:**

```
java
Copy
import java.util.stream.IntStream;

public class StreamExample {
    public static void main(String[] args) {
        IntStream.range(1, 6).forEach(System.out::println); // Output: 1, 2, 3, 4, 5
    }
}
```

## 2. Intermediate Operations

Intermediate operations transform a stream into another stream. They are **lazy**, meaning they don't do any processing until a terminal operation is invoked.

- **filter()**: Filters elements based on a predicate condition.

```
java
Copy
List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");
names.stream()
    .filter(name -> name.startsWith("A")) // Filters names that start with "A"
    .forEach(System.out::println); // Output: Alice
```

- **map()**: Transforms each element of the stream.

```
java
```

```
Copy
List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");
names.stream()
    .map(String::toUpperCase) // Converts each name to uppercase
    .forEach(System.out::println); // Output: ALICE, BOB, CHARLIE, DAVID
```

- **distinct()**: Removes duplicate elements.

```
java
Copy
List<Integer> numbers = Arrays.asList(1, 2, 2, 3, 3, 4);
numbers.stream()
    .distinct() // Removes duplicates
    .forEach(System.out::println); // Output: 1, 2, 3, 4
```

- **sorted()**: Sorts elements in natural order or using a comparator.

```
java
Copy
List<Integer> numbers = Arrays.asList(4, 3, 2, 1);
numbers.stream()
    .sorted() // Sorts in natural order
    .forEach(System.out::println); // Output: 1, 2, 3, 4
```

- **flatMap()**: Flattens nested collections into a single stream.

```
java
Copy
List<List<String>> namesList = Arrays.asList(
    Arrays.asList("Alice", "Bob"),
    Arrays.asList("Charlie", "David")
);
namesList.stream()
    .flatMap(List::stream) // Flattens the nested lists
    .forEach(System.out::println); // Output: Alice, Bob, Charlie, David
```

---

### 3. Terminal Operations

Terminal operations produce a result or side-effect. They trigger the processing of the stream.

- **forEach()**: Iterates over each element in the stream.

```
java
Copy
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
names.stream()
    .forEach(System.out::println); // Output: Alice, Bob, Charlie
```

- **collect()**: Converts the stream into a collection (e.g., List, Set, Map).

```
java
Copy
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
List<String> upperCaseNames = names.stream()
    .map(String::toUpperCase)
    .collect(Collectors.toList()); // Converts to List
System.out.println(upperCaseNames); // Output: [ALICE, BOB, CHARLIE]
```

- **reduce()**: Performs a reduction on the elements of the stream, using an associative accumulation function and returns an `Optional`.

```
java
Copy
List<Integer> numbers = Arrays.asList(1, 2, 3, 4);
int sum = numbers.stream()
    .reduce(0, (a, b) -> a + b); // Sum of all elements
System.out.println(sum); // Output: 10
```

- **count()**: Returns the number of elements in the stream.

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

```
long count = names.stream().count(); // Output: 3
System.out.println(count);
```

- **anyMatch(), allMatch(), noneMatch()**: Match operations that check if any, all, or none of the elements meet a given predicate.

```
java
Copy
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
boolean anyStartsWithA = names.stream().anyMatch(name -> name.startsWith("A"));
System.out.println(anyStartsWithA); // Output: true
```

---

## 4. Parallel Streams

A **parallel stream** allows you to process elements in parallel, taking advantage of multi-core processors. You can convert a stream to parallel by calling the `parallelStream()` method or using the `parallel()` method on an existing stream.

### Example:

```
java
Copy
import java.util.Arrays;
import java.util.List;

public class ParallelStreamExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

        // Parallel stream to calculate sum
        int sum = numbers.parallelStream()
            .reduce(0, Integer::sum);
        System.out.println(sum); // Output: 55
    }
}
```

- Parallel streams are typically useful for **CPU-intensive operations** on large datasets.

- They split the data into smaller chunks, process them concurrently, and then combine the results.
- 

## 5. Benefits of Using Streams API

- **Conciseness:** It allows for more concise code by reducing the boilerplate code for iterating, filtering, and transforming collections.
  - **Readability:** The declarative style of writing operations makes the code easier to read and understand.
  - **Parallelism:** Streams can be processed in parallel without requiring explicit management of threads.
  - **Functional Style:** Supports functional programming concepts like **map**, **filter**, and **reduce**, which can lead to cleaner and more maintainable code.
- 

## Conclusion

The **Streams API** in Java provides a powerful way to process sequences of elements in a declarative and functional style. With its rich set of intermediate and terminal operations, you can easily perform complex data manipulations. Additionally, the ability to process streams in parallel enhances performance for large datasets, making the Streams API an essential feature in modern Java programming.

---

# Optional Class in Java (Java 8)

The `Optional` class is a container object which may or may not contain a non-null value. It was introduced in **Java 8** to address the common problem of `NullPointerExceptions` and provide a more expressive way of dealing with the possibility of `null` values in Java applications.

Instead of returning `null` to represent the absence of a value, the `Optional` class encapsulates the presence or absence of a value in a more explicit and type-safe manner. This can help reduce `NullPointerExceptions` by forcing developers to handle the case where a value might be absent.

---

## Key Points about the Optional Class

1. **Container for a Value:**
    - o `Optional` is essentially a container that holds a value or is empty (if no value is present). It forces you to explicitly deal with the possibility of a value being absent, rather than silently returning `null`.
  2. **Helps Prevent `NullPointerException`:**
    - o By using `Optional`, you can avoid manually checking for `null` values, as `Optional` provides methods to handle the absence of values gracefully.
  3. **Not a Replacement for All Null Handling:**
    - o `Optional` is not meant to replace `null` everywhere in Java. It's intended to be used primarily in method return types to indicate the possibility of no value being returned.
- 

## Creating an Optional Object

You can create an `Optional` object in different ways, depending on whether or not you have a value to present.

- **Optional.empty():** Creates an empty Optional, representing the absence of a value.

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

- **Optional.of(T value):** Creates an Optional with a non-null value. If the value is null, it will throw a NullPointerException.

```
java
Copy
Optional<String> nonNullValue = Optional.of("Hello, World!");
```

- **Optional.ofNullable(T value):** Creates an Optional that can hold a nullable value. If the value is null, it will create an empty Optional instead.

```
java
Copy
Optional<String> nullableValue = Optional.ofNullable("Hello, Java!"); // Non-null value
Optional<String> nullValue = Optional.ofNullable(null); // Empty Optional
```

---

## Methods of the Optional Class

Here are some important methods of the `Optional` class that are commonly used:

### 1. `isPresent():`

- Returns `true` if the value is present (non-null), otherwise `false`.

```
java
Copy
Optional<String> value = Optional.of("Java");
if (value.isPresent()) {
    System.out.println(value.get()); // Output: Java
}
```

## **2. ifPresent(Consumer<? super T> action):**

- Executes the provided action if the value is present (i.e., not `null`), otherwise does nothing.

```
java
Copy
Optional<String> value = Optional.of("Java");
value.ifPresent(val -> System.out.println(val)); // Output: Java
```

## **3. get():**

- Retrieves the value if present. If the value is not present, it throws a `NoSuchElementException`.

```
java
Copy
Optional<String> value = Optional.of("Java");
System.out.println(value.get()); // Output: Java
```

- Be cautious when using `get()` because it can throw an exception if the value is absent.

## **4. orElse(T other):**

- Returns the value if present; otherwise, returns the provided fallback value.

```
java
Copy
Optional<String> value = Optional.ofNullable(null);
System.out.println(value.orElse("Default Value")); // Output: Default Value
```

## **5. orElseGet(Supplier<? extends T> other):**

- Returns the value if present; otherwise, invokes the provided `Supplier` to generate a fallback value.

```
java
```

```
Copy
Optional<String> value = Optional.ofNullable(null);
System.out.println(value.orElseGet(() -> "Generated Default Value")); // Output: Generated Default Value
```

#### 6. orElseThrow():

- Returns the value if present; otherwise, it throws a `NoSuchElementException`.

```
java
Copy
Optional<String> value = Optional.ofNullable(null);
System.out.println(value.orElseThrow()); // Throws NoSuchElementException
```

- This is useful when you want to explicitly throw an exception when the value is missing.

#### 7. filter(Predicate<? super T> predicate):

- If a value is present, it filters the value using the provided predicate. If the value does not satisfy the predicate, it returns an empty `Optional`.

```
java
Copy
Optional<Integer> number = Optional.of(10);
Optional<Integer> filtered = number.filter(n -> n > 5);
filtered.ifPresent(System.out::println); // Output: 10
```

- In the example above, `filter()` allows you to apply a condition to the value, returning an empty `Optional` if the condition isn't met.

#### 8. map(Function<? super T, ? extends U> mapper):

- If the value is present, applies the given function to the value and returns an `Optional` of the transformed value. If the value is absent, it returns an empty `Optional`.

```
java
```

```
Copy
Optional<String> value = Optional.of("Java");
Optional<String> upperCaseValue = value.map(String::toUpperCase);
upperCaseValue.ifPresent(System.out::println); // Output: JAVA
```

- The `map()` method is often used to transform the value inside the `Optional` (if it is present).

#### 9. `flatMap(Function<? super T, Optional<U>> mapper)`:

- Similar to `map()`, but the function provided must return an `Optional` instead of a value. This helps in avoiding nested `Optional` objects.

```
java
Copy
Optional<String> value = Optional.of("Java");
Optional<String> result = value.flatMap(v -> Optional.of(v.toUpperCase()));
result.ifPresent(System.out::println); // Output: JAVA
```

#### 10. `stream()`:

- Converts the `Optional` into a **Stream**. If the value is present, it returns a stream containing the value; otherwise, it returns an empty stream.

```
java
Copy
Optional<String> value = Optional.of("Java");
value.stream().forEach(System.out::println); // Output: Java
```

---

## Use Cases for the `Optional` Class

### 1. Method Return Values:

- Instead of returning `null` to indicate the absence of a value, you can return an `Optional`. This makes it clear that the value might be absent, and forces the caller to handle that case.

```
java
Copy
public Optional<String> getUserNameById(int id) {
    User user = userDatabase.findUserById(id);
    return Optional.ofNullable(user).map(User::getName);
}
```

## 2. Avoiding Null Checks:

- Instead of manually checking for `null`, you can use `Optional`'s methods like `ifPresent()`, `orElse()`, etc., to handle the value or its absence.

## 3. Improving Code Readability:

- `Optional` can make your code more readable by clearly indicating that a value may be absent, and by providing more fluent, expressive methods for handling that possibility.

### Example: Using `Optional` to Prevent `NullPointerException`

```
java
Copy
import java.util.Optional;

public class OptionalExample {
    public static void main(String[] args) {
        String name = null;

        // Using Optional to safely handle null
        Optional<String> optionalName = Optional.ofNullable(name);

        // Using ifPresent to avoid NPE
        optionalName.ifPresent(n -> System.out.println("Name is: " + n));

        // Providing a default value using orElse
        String result = optionalName.orElse("Default Name");
        System.out.println("Result: " + result); // Output: Default Name
    }
}
```

```
}
```

In this example:

- `Optional.ofNullable(name)` creates an `Optional` that is empty if `name` is `null`.
  - `ifPresent()` is used to only print the value if it's present.
  - `orElse()` is used to provide a default value if the `Optional` is empty.
- 

## Conclusion

The `Optional` class in Java provides a robust and type-safe way to handle cases where a value may or may not be present. It encourages developers to think explicitly about the possibility of missing values, reducing the chances of encountering `NullPointerException`. The various methods provided by `Optional` allow for clear and concise handling of missing values, making the code more readable and maintainable. However, it should be used judiciously, primarily in method return types, and not to replace `null` checks throughout your codebase.

---

# Unit Testing with JUnit in Java

Unit testing is an essential part of software development that ensures individual components of your code (like methods and classes) are functioning correctly. In Java, **JUnit** is one of the most widely used frameworks for writing and running unit tests. JUnit helps developers to test code automatically by providing a set of assertions and annotations to verify the behavior of a program.

## Key Concepts of Unit Testing and JUnit

### 1. Unit Testing:

- Unit testing is the practice of testing individual units of code (usually methods or classes) to verify their correctness.
- A unit test typically checks that a function or method behaves as expected for different input values.

### 2. JUnit:

- JUnit is a **Java framework** used for writing and running tests. It is one of the most popular unit testing frameworks in the Java ecosystem.
- It is based on the **xUnit architecture** and provides annotations and assertions to manage the lifecycle of tests, setup, execution, and validation.

---

## Setting Up JUnit

JUnit 5 (the latest version) is used widely, but earlier versions like JUnit 4 are still in use. Here, we'll focus on **JUnit 5**.

To get started with JUnit 5, you'll need to include the required dependencies in your **pom.xml** (for Maven) or **build.gradle** (for Gradle) file.

### For Maven:

```
xml
Copy
<dependencies>
    <!-- JUnit 5 dependencies -->
```

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.8.2</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.8.2</version>
    <scope>test</scope>
</dependency>
</dependencies>
```

## For Gradle:

```
gradle
Copy
dependencies {
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.8.2'
    testImplementation 'org.junit.jupiter:junit-jupiter-engine:5.8.2'
}
```

Once you have the dependencies set up, you're ready to write and run your unit tests.

---

## Writing Unit Tests with JUnit

### 1. Test Class Structure

A typical JUnit test class is structured as follows:

- It contains methods annotated with `@Test` that perform the actual test.
- It may use other annotations like `@BeforeEach`, `@AfterEach`, `@BeforeAll`, and `@AfterAll` to set up or clean up the test environment.

## 2. Basic Annotations in JUnit 5

- **@Test**: Marks a method as a test case.
- **@BeforeEach**: Runs before each test method.
- **@AfterEach**: Runs after each test method.
- **@BeforeAll**: Runs once before any tests are executed.
- **@AfterAll**: Runs once after all tests are executed.

### Example: Basic Unit Test in JUnit

```
java
Copy
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class CalculatorTest {
    // A simple method to test
    public int add(int a, int b) {
        return a + b;
    }

    @Test
    void testAdd() {
        CalculatorTest calculator = new CalculatorTest();

        // Assert that 2 + 3 equals 5
        assertEquals(5, calculator.add(2, 3));
    }
}
```

In this example:

- **@Test** marks the `testAdd()` method as a test case.
- **assertEquals(expected, actual)** is an assertion that checks if the actual value matches the expected value. If it doesn't, the test will fail.

### 3. Assertions in JUnit

JUnit provides various assertion methods to verify conditions in tests. Common assertions include:

- `assertEquals(expected, actual)`: Checks if the expected value matches the actual value.
- `assertNotEquals(expected, actual)`: Checks if the expected value does not match the actual value.
- `assertTrue(condition)`: Asserts that the condition is true.
- `assertFalse(condition)`: Asserts that the condition is false.
- `assertNull(object)`: Asserts that the object is null.
- `assertNotNull(object)`: Asserts that the object is not null.
- `assertArrayEquals(expected, actual)`: Compares two arrays for equality.

Example of using assertions:

```
java
Copy
@Test
void testAssertions() {
    assertEquals(4, 2 + 2);
    assertTrue(5 > 3);
    assertFalse(3 > 5);
    assertNotNull("Hello");
    assertNull(null);
}
```

### 4. Setup and Teardown with `@BeforeEach` and `@AfterEach`

JUnit allows you to set up and clean up resources before and after each test case.

- `@BeforeEach`: This method runs before each test method.
- `@AfterEach`: This method runs after each test method.

```
java
Copy
```

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.AfterEach;

public class CalculatorTest {

    private Calculator calculator;

    @BeforeEach
    void setUp() {
        // Initialize resources before each test
        calculator = new Calculator();
    }

    @AfterEach
    void tearDown() {
        // Clean up resources after each test
        calculator = null;
    }

    @Test
    void testAdd() {
        assertEquals(5, calculator.add(2, 3));
    }
}
```

In this example, `setUp()` runs before each test and initializes a new instance of the `Calculator` class, while `tearDown()` cleans up after each test.

## 5. Running Tests

You can run JUnit tests in various ways:

- From the command line using build tools like Maven or Gradle.
- From your IDE (e.g., IntelliJ IDEA, Eclipse), which typically supports running tests with a right-click on the test class or method.

## Advanced JUnit Features

### 1. Parameterized Tests

JUnit 5 also supports **parameterized tests**, where the same test method can run with different sets of data.

```
java
Copy
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;

public class CalculatorTest {

    @ParameterizedTest
    @ValueSource(ints = {1, 2, 3, 4, 5})
    void testAdd(int number) {
        assertTrue(number > 0);
    }
}
```

In this example:

- **@ParameterizedTest**: Marks a method to be run multiple times with different arguments.
- **@ValueSource**: Provides input values for the test method.

### 2. Test Lifecycle with **@BeforeAll** and **@AfterAll**

For setup or cleanup that should happen once for the entire test class (before or after all tests), use **@BeforeAll** and **@AfterAll**. These methods must be static.

```
java
Copy
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.AfterAll;
```

```
public class CalculatorTest {  
  
    @BeforeAll  
    static void setupClass() {  
        // Run once before all tests  
        System.out.println("Setting up the test class...");  
    }  
  
    @AfterAll  
    static void tearDownClass() {  
        // Run once after all tests  
        System.out.println("Tearing down the test class...");  
    }  
  
    @Test  
    void testAdd() {  
        assertEquals(5, 2 + 3);  
    }  
}
```

### 3. Assertions with Exception Handling

You can assert that a specific exception is thrown in a test method using `assertThrows()`.

```
java  
Copy  
@Test  
void testException() {  
    Exception exception = assertThrows(ArithmeticException.class, () -> {  
        int result = 1 / 0; // This will throw ArithmeticException  
    });  
    assertEquals("/ by zero", exception.getMessage());  
}
```

---

## Benefits of Unit Testing with JUnit

1. **Early Bug Detection:** Unit tests help identify issues early in the development cycle, making it easier and cheaper to fix them.
  2. **Automated Testing:** Once written, unit tests can be run automatically whenever code changes, ensuring that the software continues to work as expected.
  3. **Improved Code Design:** Writing unit tests encourages developers to write modular, loosely coupled, and more maintainable code.
  4. **Documentation:** Tests provide a form of documentation for how a piece of code is supposed to behave.
  5. **Confidence in Refactoring:** With a solid set of unit tests, you can confidently make changes to the codebase and be assured that existing functionality will not break.
- 

## Conclusion

JUnit is a powerful and essential framework for performing unit testing in Java. It provides a simple and efficient way to test individual units of your code, detect issues early, and maintain high-quality code. The annotations like `@Test`, `@BeforeEach`, `@AfterEach`, and others, combined with assertions, allow you to easily validate and verify the behavior of your code. With advanced features like parameterized tests, exception handling, and lifecycle methods, JUnit 5 offers the flexibility needed for both simple and complex testing scenarios.

---

# Writing and Executing Test Cases in Unit Testing in Java

Unit testing involves writing individual tests to verify the correctness of small units of code such as methods or functions. In Java, **JUnit** is the most commonly used framework for writing and executing unit tests. Writing and executing test cases helps ensure that your code works as expected and that changes to the code do not introduce new bugs.

---

## 1. Writing Test Cases in JUnit

To write test cases in JUnit, follow these steps:

1. **Create a test class:** A test class contains test methods that test the functionality of your application's classes and methods.
2. **Write test methods:** Each method in the test class is a unit test for a particular piece of functionality in your application.
3. **Use assertions:** Assertions compare the expected output of the code under test with the actual result, and if they do not match, the test fails.
4. **Use annotations:** JUnit provides several annotations to define the structure of your tests.

### Basic Structure of a JUnit Test

1. **Test Class:** A class with `@Test` methods to test your functionality.
2. **Test Method:** A method annotated with `@Test` that performs a specific test.
3. **Assertions:** To compare expected and actual outcomes.

### Example: Writing Test Cases for a Calculator

Let's consider a simple `Calculator` class with methods to add and subtract numbers. We will write test cases to verify the correctness of these methods.

#### Calculator Class

```
java
Copy
public class Calculator {

    // Method to add two numbers
    public int add(int a, int b) {
        return a + b;
    }

    // Method to subtract two numbers
    public int subtract(int a, int b) {
        return a - b;
    }
}
```

## JUnit Test Class

Now, we'll write a test class to verify the `add` and `subtract` methods of the `Calculator` class.

```
java
Copy
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class CalculatorTest {

    // Create an instance of the Calculator class
    Calculator calculator = new Calculator();

    // Test the add method
    @Test
    void testAdd() {
        int result = calculator.add(2, 3);
        assertEquals(5, result); // Assert that 2 + 3 equals 5
    }

    // Test the subtract method
    @Test
```

```
void testSubtract() {
    int result = calculator.subtract(5, 3);
    assertEquals(2, result); // Assert that 5 - 3 equals 2
}

// Test adding negative numbers
@Test
void testAddNegative() {
    int result = calculator.add(-2, -3);
    assertEquals(-5, result); // Assert that -2 + (-3) equals -5
}
}
```

## Explanation of the Test Class

1. **@Test**: This annotation marks a method as a test method. JUnit runs this method as part of the test suite.
2. **assertEquals(expected, actual)**: This assertion checks if the expected value matches the actual result. If they do not match, the test fails.

## Test Methods Breakdown

- `testAdd()`: Verifies that the `add` method correctly adds two numbers.
- `testSubtract()`: Verifies that the `subtract` method correctly subtracts two numbers.
- `testAddNegative()`: Verifies that the `add` method works with negative numbers.

---

## 2. Executing Test Cases in JUnit

Once your test cases are written, you can execute them to verify that the code is working as expected.

### Running Tests in an IDE

Most modern IDEs, like **IntelliJ IDEA** or **Eclipse**, support running JUnit tests directly from the IDE.

- **In IntelliJ IDEA:**
  - Right-click on the test class or method and select **Run 'CalculatorTest'** or **Run 'testAdd'**.
- **In Eclipse:**
  - Right-click on the test class or method and select **Run As → JUnit Test**.

## Running Tests from the Command Line

If you're using Maven or Gradle, you can run your tests from the command line.

### For Maven:

```
bash
Copy
mvn test
```

This command will compile your project and run all tests that are part of the Maven project.

### For Gradle:

```
bash
Copy
gradle test
```

This command will run all the tests in your Gradle project.

## Example of Running the Test with Maven

If your `CalculatorTest` class is in the `/src/test/java` directory and you have a `pom.xml` file set up, running the following Maven command in the project's root directory will execute your test cases:

```
bash
Copy
mvn clean test
```

This command does two things:

- `clean`: Cleans the project (deletes previously compiled files).
  - `test`: Runs the unit tests using the JUnit framework.
- 

### 3. Understanding Test Results

After running the tests, you will receive results indicating whether each test passed or failed. Here's how to interpret the results:

- **Pass**: If the test passes, it means the actual output matched the expected output.
- **Fail**: If the test fails, it means there was a mismatch between the expected and actual outputs. The output will provide details about what went wrong.

#### JUnit Test Result Example

When you run the tests, the output might look something like this:

```
yaml
Copy
Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.005 sec
```

This means that all 3 tests ran successfully, with no failures or errors.

If a test fails, you might see something like:

```
makefile
Copy
org.junit.jupiter.api.AssertionFailedError: expected:<5> but was:<4>
```

This indicates that the `expected` value was 5, but the `actual` value was 4. You'll need to fix the bug in your code to ensure the correct output.

---

## 4. Advanced Test Case Features in JUnit

JUnit offers many advanced features for more complex test scenarios, including:

### 1. Setup and Teardown with `@BeforeEach` and `@AfterEach`

These annotations help you prepare and clean up resources for each test.

- `@BeforeEach`: Runs before each test method.
- `@AfterEach`: Runs after each test method.

```
java
Copy
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.AfterEach;

public class CalculatorTest {

    Calculator calculator;

    @BeforeEach
    void setUp() {
        calculator = new Calculator(); // Initialize before each test
    }

    @AfterEach
    void tearDown() {
        calculator = null; // Clean up after each test
    }

    @Test
    void testAdd() {
        assertEquals(5, calculator.add(2, 3));
    }
}
```

```
}
```

## 2. Test Execution Order with `@TestMethodOrder`

In some cases, you might want to specify the order in which tests should be executed. The `@TestMethodOrder` annotation allows you to control the execution order of the test methods.

```
java
Copy
import org.junit.jupiter.api.TestMethodOrder;
import org.junit.jupiter.api.MethodOrderer;
import org.junit.jupiter.api.Test;

@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
public class CalculatorTest {

    @Test
    @org.junit.jupiter.api.Order(1)
    void testAdd() {
        // Test 1
    }

    @Test
    @org.junit.jupiter.api.Order(2)
    void testSubtract() {
        // Test 2
    }
}
```

## 3. Grouping Tests with `@Tag`

You can group tests with tags and then run specific tests based on tags.

```
java
Copy
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;
```

```
public class CalculatorTest {  
  
    @Test  
    @Tag("Addition")  
    void testAdd() {  
        // Test code  
    }  
  
    @Test  
    @Tag("Subtraction")  
    void testSubtract() {  
        // Test code  
    }  
}
```

## 5. Running Tests with Parameters

JUnit supports **parameterized tests**, which allow you to run the same test with multiple sets of data.

```
java  
Copy  
import org.junit.jupiter.params.ParameterizedTest;  
import org.junit.jupiter.params.provider.ValueSource;  
  
public class CalculatorTest {  
  
    @ParameterizedTest  
    @ValueSource(ints = {1, 2, 3, 4, 5})  
    void testAdd(int number) {  
        assertTrue(number > 0);  
    }  
}
```

In this example, the test will run 5 times, once for each of the provided values.

---

## Conclusion

Writing and executing unit tests in Java using JUnit is a crucial part of ensuring that your code is correct, reliable, and free from regressions. Here's a summary of key steps:

1. **Write test cases** using annotations like `@Test`, `@BeforeEach`, and `@AfterEach`.
2. **Use assertions** (e.g., `assertEquals`, `assertTrue`) to validate that your code produces the correct results.
3. **Execute the tests** using an IDE, command-line tools (e.g., Maven, Gradle), or CI/CD pipelines.
4. **Review the test results** to identify any failures and fix them.
5. **Use advanced features** like parameterized tests, setup/teardown methods, and test ordering to manage more complex scenarios.

Unit testing with JUnit helps ensure that your code is robust, maintainable, and behaves as expected, making it a critical practice in modern software development.

Core Java Notes By AJAY RAZZ