

Java Notes: Beginner to Advanced (with Examples)

Introduction to Java

What Java Is

Java is a general-purpose, object-oriented programming language. You write source code once, compile it, and run the result on any machine that has a Java runtime. These notes move through three parts: Beginner, Intermediate (object-oriented programming) and Advanced. Where it matters, they use modern Java (17 and 21).

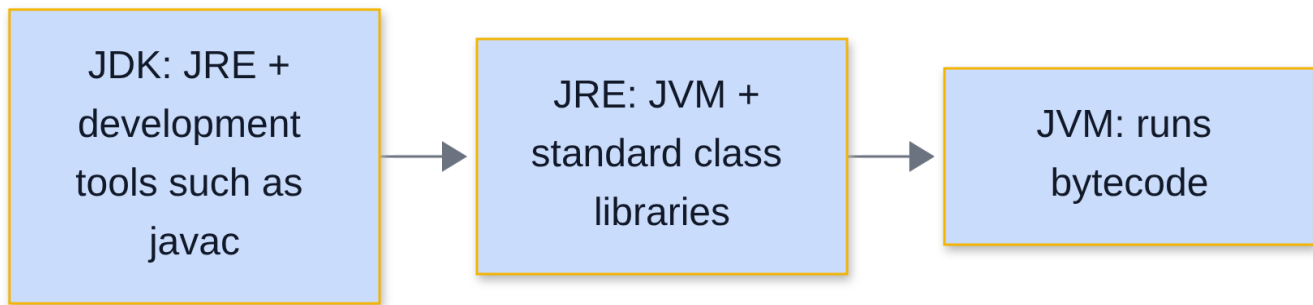
Key Features

Feature	What it means for you
Object-oriented	Programs are built from classes and objects, using encapsulation, inheritance, polymorphism and abstraction.
Platform independent	Source compiles to bytecode, which any compatible JVM can run. This is often described as "write once, run anywhere".
Strongly typed	Every variable has a declared type, so many mistakes are caught at compile time.
Automatic memory management	A garbage collector reclaims objects that are no longer used.
Robust	Exception handling and type checking help programs fail in controlled ways.
Multithreaded	The language and libraries support running tasks concurrently.
Rich standard library	Collections, I/O, networking, concurrency utilities and more ship with the platform.

JDK, JRE and JVM

These three terms are easy to confuse, but they nest inside one another.

- **JVM (Java Virtual Machine):** executes compiled bytecode. It loads classes, manages memory, runs garbage collection and turns bytecode into instructions the host machine can execute. Each operating system has its own JVM implementation, which is what makes bytecode portable.
- **JRE (Java Runtime Environment):** what you need to *run* Java programs. It contains the JVM plus the standard class libraries.
- **JDK (Java Development Kit):** what you need to *develop* Java programs. It contains the JRE plus development tools such as the compiler (`javac`) and the launcher (`java`).



In short: **JDK** $\supset$ **JRE** $\supset$ **JVM**. As a learner you should install the JDK, because it gives you everything. Recent distributions are generally supplied as a JDK, so a separate JRE download is less common than it once was.

How the Pieces Work Together

1. You write a `.java` source file.
2. The compiler from the JDK turns it into `.class` bytecode.
3. The JVM, supplied through the JRE, loads and runs that bytecode.

The next section walks through this compilation flow in detail, then installs the JDK and runs a first Hello World program.

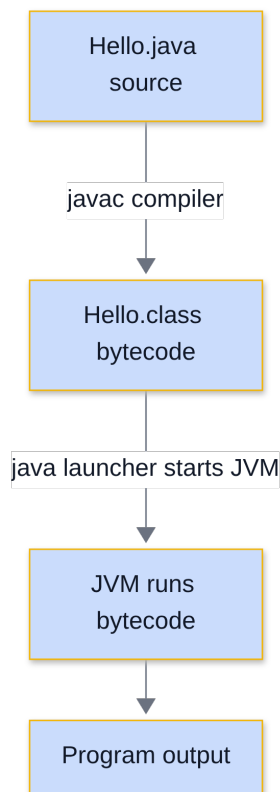
Key Points and Common Mistakes

- Java is both a language and a platform: the language is the syntax you write, and the platform is the JVM plus libraries.
- The JVM is not the compiler. `javac` compiles; the JVM runs.
- Do not say "Java is interpreted only". Source is compiled to bytecode first, and the JVM then executes that bytecode.
- Install a JDK, not just a runtime, if you intend to write code.
- Java and JavaScript are unrelated languages despite the similar names.

Compilation Flow, Setup and Hello World

From Source Code to Running Program

Java programs are written as plain-text source files with the `.java` extension. They do not run directly on the operating system. Instead, the compiler first translates them into an intermediate form called bytecode, and the JVM then executes that bytecode. This two-step design is what lets the same compiled program run on any platform that has a JVM.



The stages are:

1. **Write** the source code in a `.java` file.
2. **Compile** it with `javac`, which produces a `.class` file containing bytecode.
3. **Run** it with the `java` command, which starts the JVM. The JVM loads the class and executes the bytecode.

Recall the tool roles from the previous section: the JDK contains the compiler and development tools, the JRE provides what is needed to run programs, and the JVM is the engine that executes bytecode.

Setup Steps

The notes assume a modern JDK (Java 17 or 21 where relevant). A typical setup is:

1. Download and install a JDK (Java 17 or 21 is recommended for these notes).
2. Make sure the JDK's `bin` directory is available on your system path, so the `javac` and `java` commands can be found. Some installers do this for you.
3. Open a terminal and verify the installation:

BASH

```
java -version
javac -version
```

Both commands should print a version number. If `javac` is not found, you most likely installed only a runtime or the path is not set. You may use a plain text editor and the terminal, or an IDE; both work for every example in these notes.

Your First Program: Hello World

Save the following in a file named `Hello.java`:

JAVA

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

Compile and run it:

BASH

```
javac Hello.java
java Hello
```

Expected output:

```
Hello, World!
```

Understanding Each Part

Part	Meaning
<code>public class Hello</code>	Declares a class named <code>Hello</code> . Every Java program lives inside classes.
<code>public static void main(String[] args)</code>	The entry point the JVM calls to start the program.
<code>String[] args</code>	Holds any command-line arguments.
<code>System.out.println(...)</code>	Prints text to the console, followed by a new line.

Key Points and Common Mistakes

- A public class must be saved in a file with the same name, so `Hello` goes in `Hello.java`. A mismatch causes a compile error.
- Java is case-sensitive: `main` and `Main` are different, and `String` must be capitalized.
- Run with `java Hello`, not `java Hello.class`; you pass the class name, not the file name.

- Do not forget the semicolon at the end of statements.
- Compile errors appear at `javac` time; errors that occur while the program runs appear at the `java` step.

Variables, Data Types and Casting

Variables

A variable is a named memory location that holds a value. In Java every variable has a declared type, and that type cannot change after declaration. You declare a variable by giving its type and name, and you may assign a value at the same time.

JAVA

```
int age = 20;
double price = 99.5;
String name = "Asha";
System.out.println(name + " " + age + " " + price);
```

Expected output:

```
Asha 20 99.5
```

Variable names are case-sensitive, must not start with a digit, and cannot be a reserved keyword. By convention Java uses camelCase for variables (`studentCount`) and UPPER_SNAKE_CASE for constants declared with `final`.

Primitive Data Types

Java has eight primitive types. They store raw values directly and are not objects. Apart from `boolean`, their sizes are fixed and do not depend on the platform.

Type	Size	Typical use	Default (fields)
byte	8 bits	Small integers, raw data	0
short	16 bits	Small integers	0
int	32 bits	Default whole-number type	0
long	64 bits	Large whole numbers (suffix L)	0L
float	32 bits	Decimal numbers (suffix f)	0.0f
double	64 bits	Default decimal type	0.0
char	16 bits	A single UTF-16 character	'\u0000'
boolean	Not precisely defined (JVM-dependent)	true or false	false

Local variables have no default value and must be assigned before use.

Reference Data Types

Reference types (classes, interfaces, arrays, and `String`) store a reference to an object located on the heap, not the object itself. A reference that points to nothing holds `null`.

JAVA

```
String a = "Java";
String b = a;      // both refer to the same object
String c = null;   // refers to nothing
System.out.println(b);
```

Expected output:

Java

Type Casting

Casting converts a value from one type to another.

Widening (implicit) goes from a smaller to a larger type and is done automatically: `byte` → `short` → `int` → `long` → `float` → `double`.

Narrowing (explicit) goes from a larger to a smaller type, needs a cast operator, and may lose information.

JAVA

```
int i = 10;
double d = i;      // widening
double pi = 3.99;
int n = (int) pi;   // narrowing: fraction truncated
int big = 130;
byte b = (byte) big; // overflow wraps around
System.out.println(d + " " + n + " " + b);
```

Expected output:

10.0 3 -126

Key Points and Common Mistakes

- Casting a decimal to an integer truncates; it does not round.
- Narrowing can overflow silently, as the `byte` example shows.
- Write `100L` for long literals and `3.5f` for float literals; otherwise the compiler treats them as `int` and `double`.
- Use `==` to compare primitives, but use `equals()` for the contents of reference types such as `String`.
- Using a local variable before assigning it is a compile-time error.

Operators, Input and Comments

Operators

Operators perform actions on values (operands). Java groups them into several families.

Category	Operators	Example	Result
Arithmetic	+ - * / %	7 / 2, 7 % 2	3, 1
Relational	== != < > <= >=	5 > 3	true
Logical	&& !	true && false	false
Assignment	= += -= *= /=	x += 4	adds 4 to x
Unary	++ -- + - !	i++	increments i
Ternary	? :	a > b ? a : b	the larger value

Key behaviors to remember:

- Dividing two integers discards the fractional part, so `7 / 2` is 3. Use `7 / 2.0` to get 3.5.
- `i++` (postfix) yields the old value and then increments; `++i` (prefix) increments first.
- `&&` and `||` short-circuit: the right side is skipped when the left side already decides the result.
- `==` compares primitive values. For objects, including `String`, it compares references; use `.equals()` for content.

Comments

Comments are ignored by the compiler and explain intent to readers.

JAVA

```
// Single-line comment
/* Multi-line
   comment */
/** Javadoc comment: documents a class or method. */
```

Reading Input with Scanner

The `java.util.Scanner` class reads text from a source such as the keyboard (`System.in`). Common methods are `nextInt()`, `nextDouble()`, `next()` (one word), and `nextLine()` (a full line).

```
import java.util.Scanner;

public class InputDemo {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Enter your name: ");
        String name = sc.nextLine();    // reads the whole line

        System.out.print("Enter two integers: ");
        int a = sc.nextInt();
        int b = sc.nextInt();

        System.out.println("Hello, " + name);
        System.out.println("Sum = " + (a + b));
        System.out.println("Average = " + (a + b) / 2.0);
        System.out.println("a is larger? " + (a > b));

        sc.close();
    }
}
```

Expected output when the user types Asha, then 4 7:

```
Enter your name: Asha
Enter two integers: 4 7
Hello, Asha
Sum = 11
Average = 5.5
a is larger? false
```

Key Points and Common Mistakes

- **Leftover newline:** after `nextInt()` or `nextDouble()`, the line break stays in the buffer. A following `nextLine()` returns an empty string. Call an extra `sc.nextLine()` to consume it.
- **Wrong input type:** typing text where `nextInt()` expects a number throws `InputMismatchException`.
- **Integer division:** $(a + b) / 2$ is integer division; divide by `2.0` for a decimal result.
- **Assignment vs comparison:** `=` assigns, `==` compares.
- **Closing Scanner:** close it when finished, but avoid closing a `Scanner` on `System.in` if you still need input elsewhere in the program.

Decision Making: if/else and switch

if / else

An `if` statement runs a block only when its condition is `true`. Chain alternatives with `else if`, and use `else` as the fallback.

```
int score = 72;
if (score >= 90) {
    System.out.println("A");
} else if (score >= 70) {
    System.out.println("B");
} else {
    System.out.println("C");
}
```

JAVA

Expected output: B

Classic switch

A classic `switch` compares one value against `case` labels. Without `break`, execution *falls through* to the next case.

```
int day = 2;
switch (day) {
    case 1: System.out.println("Mon"); break;
    case 2: System.out.println("Tue"); break;
    default: System.out.println("Other");
}
```

JAVA

Expected output: Tue

Switch expression (Java 14+, standard in 17/21)

The arrow form produces a value, does not fall through, and needs no `break`.

```
int day = 6;
String type = switch (day) {
    case 6, 7 -> "Weekend";
    default -> "Weekday";
};
System.out.println(type);
```

JAVA

Expected output: Weekend

Use `yield` to return a value from a multi-statement block inside a switch expression.

Comparison

Feature	Classic switch	Switch expression
Returns a value	No	Yes
Fall-through	Yes, unless <code>break</code>	No
Multiple labels	Repeated <code>case</code> lines	<code>case 6, 7 -></code>
Exhaustive	Not required	Required (cover all values, e.g. add <code>default</code>)

Common mistakes

- Using `=` instead of `==` in a condition; `if (x = 5)` does not compile for `int`.
- Forgetting `break` in a classic switch, causing fall-through.
- Comparing strings with `==` instead of `.equals()`.
- Omitting braces, so only the first statement belongs to the `if`.
- Leaving out `default` in a switch expression, which is a compile error when not all values are covered.

Loops and Loop Control

Loops repeat a block of code. Java offers four forms, and choosing the right one makes code shorter and less error-prone.

Comparing the Four Loops

Loop	Condition checked	Best used when
for	Before each iteration	The number of iterations is known
while	Before each iteration	Repetition depends on a condition; may run zero times
do-while	After each iteration	The body must run at least once
for-each	Handled automatically	Reading every element of an array or collection

Example

JAVA

```
public class LoopDemo {
    public static void main(String[] args) {
        for (int i = 1; i <= 3; i++) System.out.print(i + " ");
        int n = 3;
        while (n > 0) { System.out.print(n + " "); n--; }
        int k = 5;
        do { System.out.print(k + " "); k++; } while (k < 5);
        for (int v : new int[]{7, 8}) System.out.print(v + " ");
    }
}
```

Expected output:

```
1 2 3 3 2 1 5 7 8
```

The do-while printed 5 even though $k < 5$ is false, because its condition is tested after the body.

break and continue

- `break` ends the nearest enclosing loop immediately. It also ends a `switch` case.
- `continue` skips the rest of the current iteration and moves to the next one. In a `for` loop, the update expression still runs.

JAVA

```
for (int i = 1; i <= 5; i++) {
    if (i == 2) continue; // skip 2
    if (i == 4) break;    // stop at 4
    System.out.print(i + " ");
}
```

Output: 1 3

Key Points and Common Mistakes

- Forgetting to update the loop variable in a `while` loop causes an infinite loop.
- Using `<=` instead of `<` with array length causes `ArrayIndexOutOfBoundsException`.
- For-each gives no index and should not be used to remove elements from a collection while iterating.
- `break` and `continue` affect only the innermost loop unless a label is used.

Arrays: 1D and 2D

An array stores a fixed number of values of the same type in contiguous, zero-based positions. Its length is set when it is created and cannot change afterward.

One-Dimensional Arrays

JAVA

```
int[] scores = new int[3];           // default values: 0, 0, 0
int[] marks = {70, 85, 92};         // literal initialization

scores[0] = 10;
for (int i = 0; i < marks.length; i++) {
    System.out.println("marks[" + i + "] = " + marks[i]);
}
for (int m : marks) {                // for-each: read-only traversal
    System.out.print(m + " ");
}
```

Expected output:

```
marks[0] = 70
marks[1] = 85
marks[2] = 92
70 85 92
```

Note that `length` is a field, not a method (unlike `String.length()`).

Two-Dimensional Arrays

A 2D array is an array of arrays, so rows may even have different lengths.

JAVA

```
int[][] grid = {
    {1, 2, 3},
    {4, 5, 6}
};
for (int r = 0; r < grid.length; r++) {
    for (int c = 0; c < grid[r].length; c++) {
        System.out.print(grid[r][c] + " ");
    }
    System.out.println();
}
```

Expected output:

```
1 2 3
4 5 6
```

Common Mistakes

Mistake	Result
Using <code>i <= arr.length</code> in a loop	<code>ArrayIndexOutOfBoundsException</code> (valid indexes are 0 to <code>length - 1</code>)
Accessing <code>arr[-1]</code>	<code>ArrayIndexOutOfBoundsException</code>
Using an array variable never created (<code>int[] a = null;</code>)	<code>NullPointerException</code> on access
Writing <code>arr.length()</code>	Compile error
Printing an array directly	Shows a type/hash text; use <code>Arrays.toString(arr)</code>

Key Points

- Indexes start at 0; the last index is `length - 1`.
- Elements get default values (0, false, or null) when not initialized.
- For 2D arrays, use `grid[r].length` for the column count of row `r`.

Strings and StringBuilder

Immutability

A `String` object cannot be changed after it is created. Methods such as `toUpperCase()` or `concat()` return a new `String` and leave the original untouched.

```
String s = "java";
s.toUpperCase();           // result discarded
System.out.println(s);    // java
s = s.toUpperCase();
System.out.println(s);    // JAVA
```

JAVA

Common mistake: calling a `String` method and ignoring the returned value.

The String Pool

String literals are stored in a shared area called the String pool, so identical literals reuse one object. Using `new String(...)` forces a separate object.

```
String a = "hi";
String b = "hi";
String c = new String("hi");
System.out.println(a == b);           // true
System.out.println(a == c);           // false
System.out.println(a.equals(c));      // true
```

JAVA

Use `equals()` to compare content; `==` compares references.

StringBuilder

Because Strings are immutable, repeated concatenation in a loop creates many temporary objects. `StringBuilder` is a mutable alternative.

```
StringBuilder sb = new StringBuilder("Java");
sb.append(" Notes").insert(0, ">> ").reverse();
System.out.println(sb); // setoN avaJ >>
```

JAVA

Common String Methods

Method	Purpose	Example result
<code>length()</code>	Number of characters	<code>"Java".length()</code> → 4
<code>charAt(i)</code>	Character at index	<code>"Java".charAt(1)</code> → a
<code>substring(b, e)</code>	Part of string	<code>"Java".substring(1,3)</code> → av
<code>indexOf(x)</code>	First position, or -1	<code>"Java".indexOf("v")</code> → 2
<code>equals(x)</code>	Content comparison	<code>"a".equals("a")</code> → true
<code>trim()</code>	Remove edge whitespace	<code>" hi ".trim()</code> → hi
<code>split(r)</code>	Split into array	<code>"a,b".split(",")</code> → [a, b]
<code>replace(x, y)</code>	Replace text	<code>"aab".replace("a","c")</code> → ccb

Key Points

- Strings are immutable; reassign the result.
- Compare with `equals()`, not `==`.
- Use `StringBuilder` for heavy concatenation.

Methods, Overloading and Recursion

A **method** is a named block of code that performs a task and can be reused. Methods reduce duplication and make programs easier to test.

Parameters and Return Values

JAVA

```
public class MethodDemo {  
    static int add(int a, int b) { // two parameters, returns int  
        return a + b;  
    }  
  
    static void greet(String name) { // void: returns nothing  
        System.out.println("Hello, " + name);  
    }  
  
    public static void main(String[] args) {  
        int sum = add(3, 4);  
        greet("Asha");  
        System.out.println("Sum = " + sum);  
    }  
}
```

Output:

```
Hello, Asha  
Sum = 7
```

Java passes arguments **by value**: a method receives a copy of a primitive, or a copy of an object reference.

Overloading

Overloading means several methods share a name but differ in parameter lists (number, types, or order). The return type alone cannot distinguish them.

JAVA

```
static int square(int x) { return x * x; }  
static double square(double x) { return x * x; }  
  
System.out.println(square(5));  
System.out.println(square(2.5));
```

Output:

```
25  
6.25
```

Varargs

A varargs parameter (`int... nums`) accepts zero or more values and is treated as an array. It must be the last parameter.

JAVA

```
static int total(int... nums) {  
    int s = 0;  
    for (int n : nums) s += n;  
    return s;  
}  
System.out.println(total(1, 2, 3));  
System.out.println(total());
```

Output:

```
6  
0
```

Recursion

A recursive method calls itself and needs a **base case** to stop.

JAVA

```
static int factorial(int n) {  
    if (n <= 1) return 1;    // base case  
    return n * factorial(n - 1);  
}  
System.out.println(factorial(5));
```

Output:

```
120
```

Key Points and Common Mistakes

- Missing a base case causes `StackOverflowError`.
- A non-void method must return a value on every path.
- Overloads that differ only by return type do not compile.
- Very deep recursion is often better rewritten as a loop.

Classes, Objects and Constructors

Classes and Objects

A **class** is a blueprint that describes the state (fields) and behavior (methods) of a kind of thing. An **object** is a concrete instance of that class, created with the `new` keyword. Classes are the foundation of object-oriented programming (OOP) in Java.

Constructors

A **constructor** runs when an object is created and initializes its state. It has the same name as the class and no return type. If you write none, Java supplies a default no-argument constructor; once you declare any constructor, that default is no longer generated.

The `this` Keyword

`this` refers to the current object. It is commonly used to separate a field from a parameter with the same name, and `this(...)` lets one constructor call another in the same class (it must be the first statement).

Example

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

    Student() {
        this("Unknown", 0);
    }

    Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    void show() {
        System.out.println(name + " (" + age + ")");
    }

    public static void main(String[] args) {
        new Student("Asha", 20).show();
        new Student().show();
    }
}
```

JAVA

Expected output:

Key Points

- A class defines the type; each `new` creates a separate object with its own field values.
- Constructors can be overloaded by varying their parameter lists.
- Fields left unset receive defaults (`0`, `false`, `null`).

Common Mistakes

- Writing `name = name;` in a constructor, which assigns the parameter to itself and leaves the field unchanged.
- Giving a constructor a return type (such as `void`), which turns it into an ordinary method.
- Calling `new Student()` after defining only a parameterized constructor, which causes a compile error.

Static Members and Initialization Blocks

What Static Means

A **static** member belongs to the class itself, not to any single object. All instances share one copy, and you can reach it through the class name without creating an object.

Static Variables and Methods

JAVA

```
public class Counter {
    static int count = 0;    // shared by all objects
    int id;                 // one per object

    Counter() { id = ++count; }

    static int getCount() { return count; }
}

public class Main {
    public static void main(String[] args) {
        new Counter();
        new Counter();
        System.out.println(Counter.getCount());
    }
}
```

Expected output:

2

A static method has no **this** reference, so it cannot directly use instance variables or instance methods.

Initialization Blocks

A **static block** runs once, when the class is first loaded and initialized. An **instance block** runs every time an object is created, before the constructor body.

JAVA

```
public class Demo {
    static { System.out.println("1 static block"); }
    { System.out.println("2 instance block"); }
    Demo() { System.out.println("3 constructor"); }

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

Expected output:

```
1 static block
2 instance block
3 constructor
2 instance block
3 constructor
```

Order of Initialization

1. Static variables and static blocks, in the order written, once per class.
2. Instance variables and instance blocks, in the order written, for each object.
3. The constructor body.

With inheritance, the parent class's static initialization runs first, then the child's static initialization. For each object, the parent's instance initialization and constructor run before the child's.

Key Points and Common Mistakes

- Calling an instance member from a static method causes a compile error.
- Access static members through the class name, not an object reference.
- Static state is shared, so mutable static fields can cause surprising bugs, especially with threads.
- Static blocks run only once, however many objects you create.

Encapsulation and Inheritance

Encapsulation

Encapsulation bundles data and the methods that operate on it into one class, and restricts direct access to that data. Fields are declared `private`, and controlled access is offered through public methods. This protects the object's state: every change passes through code that can validate it.

JAVA

```
class Account {
    private double balance;

    public double getBalance() { return balance; }

    public void deposit(double amount) {
        if (amount <= 0) {
            throw new IllegalArgumentException("Amount must be positive");
        }
        balance += amount;
    }
}

public class Main {
    public static void main(String[] args) {
        Account a = new Account();
        a.deposit(500);
        System.out.println(a.getBalance());
    }
}
```

Expected output:

```
500.0
```

Because `balance` is private, code outside the class cannot set it to an invalid value.

Inheritance

Inheritance lets a subclass reuse the fields and methods of a superclass using `extends`. It models an "is-a" relationship and avoids duplicated code.

```
class Animal {  
    void eat() { System.out.println("Eating"); }  
}  
  
class Dog extends Animal {  
    void bark() { System.out.println("Barking"); }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Dog d = new Dog();  
        d.eat();  
        d.bark();  
    }  
}
```

Expected output:

```
Eating  
Barking
```

Dog gains `eat()` without redefining it. Java supports single inheritance for classes: a class extends only one superclass.

Key Points and Common Mistakes

- Make fields private by default; expose only what callers need.
- Setters should validate input rather than assign blindly.
- Use inheritance only for true "is-a" relationships.
- A common mistake is extending a class just to reuse a method, which creates misleading hierarchies.
- Private members are not directly accessible in a subclass.

Polymorphism and Abstraction

Polymorphism: One Interface, Many Forms

Polymorphism lets the same call behave differently depending on the types involved. Java offers two kinds: **compile-time** polymorphism (method overloading) and **runtime** polymorphism (method overriding).

Overloading vs Overriding

Aspect	Overloading	Overriding
Where	Same class (or subclass)	Subclass redefines a superclass method
Signature	Parameter list must differ	Must match the parent method
Resolved	At compile time	At runtime, by the object's actual type
Annotation	None	@Override (recommended)

JAVA

```
class Animal {
    void speak() { System.out.println("..."); }
}
class Dog extends Animal {
    @Override
    void speak() { System.out.println("Woof"); }
}
class Printer {
    void print(int n)    { System.out.println("int " + n); }
    void print(String s) { System.out.println("String " + s); }
}
public class Demo {
    public static void main(String[] args) {
        Animal a = new Dog(); // parent reference, child object
        a.speak();
        Printer p = new Printer();
        p.print(5);
        p.print("hi");
    }
}
```

Expected output:

```
Woof
int 5
String hi
```

`a.speak()` prints `Woof` because the JVM chooses the method from the actual object (`Dog`), not the reference type.

Abstraction: The Fourth Pillar

Abstraction hides implementation details and exposes only what a caller needs. In Java it is expressed through abstract classes and interfaces (compared in the next section). Together, the four pillars work as follows: encapsulation protects data, inheritance reuses behavior, polymorphism varies behavior, and abstraction defines the contract.

JAVA

```
abstract class Shape {  
    abstract double area();  
}  
class Circle extends Shape {  
    double r = 1;  
    double area() { return Math.PI * r * r; }  
}
```

Key Points and Common Mistakes

- Changing only the return type does not overload a method.
- Always add `@Override` so the compiler catches signature typos.
- An abstract class cannot be instantiated directly.
- Fields are not overridden; only instance methods are.

Abstract Classes vs Interfaces

An **abstract class** and an **interface** both define a contract that other classes must fulfil, but they serve different design goals. Use an abstract class to share state and partial implementation among closely related classes. Use an interface to describe a capability that unrelated classes can offer.

Comparison Table

Feature	Abstract class	Interface
Keyword	<code>abstract class</code>	<code>interface</code>
Inheritance	A class extends only one	A class can implement many
Instance fields	Allowed, any access level	Only <code>public static final</code> constants
Constructors	Allowed (called via <code>super</code>)	Not allowed
Abstract methods	Zero or more	Implicitly abstract unless marked otherwise
Concrete methods	Allowed freely	Only <code>default</code> , <code>static</code> or <code>private</code> methods
<code>default</code> methods	Not applicable	Provide an overridable implementation
<code>static</code> methods	Allowed	Allowed; called on the interface name
<code>private</code> methods	Allowed	Allowed; share code between <code>default</code> / <code>static</code> methods
Best used for	Shared state and base behavior	Capabilities and contracts

Interface Method Types

JAVA

```
interface Greeter {
    String name();                // abstract
    default String greet() {      // default
        return decorate("Hello, " + name());
    }
    static Greeter of(String n) { // static
        return () -> n;
    }
    private String decorate(String s) { // private helper
        return "*** " + s + " ***";
    }
}

public class Demo {
    public static void main(String[] args) {
        System.out.println(Greeter.of("Asha").greet());
    }
}
```

Expected output:

```
*** Hello, Asha ***
```

Key Points and Common Mistakes

- Neither type can be instantiated directly with `new`.
- A default method can be overridden by an implementing class; a private method cannot be called from outside the interface.
- Static interface methods are not inherited; call them as `Greeter.of(...)`.
- Mistake: choosing an abstract class when no shared state exists, which needlessly blocks a class from extending anything else.

super, final, Access Modifiers and Packages

The super keyword

`super` refers to the parent class. Use it to call a parent constructor (`super(args)`, which must be the first statement in a constructor), to call an overridden method (`super.show()`), or to access a hidden parent field.

JAVA

```
class Animal {
    String sound() { return "generic sound"; }
}
class Dog extends Animal {
    String sound() { return super.sound() + " -> Woof"; }
}
public class Demo {
    public static void main(String[] args) {
        System.out.println(new Dog().sound());
    }
}
```

Expected output: generic sound -> Woof

The final keyword

Applied to	Effect
Variable	Value can be assigned only once
Method	Cannot be overridden
Class	Cannot be extended (e.g., String)

Note that a `final` reference cannot be reassigned, but the object it points to may still change internally.

Access modifiers

Modifier	Same class	Same package	Subclass (other package)	Anywhere
private	Yes	No	No	No
default (none)	Yes	Yes	No	No
protected	Yes	Yes	Yes	No
public	Yes	Yes	Yes	Yes

Packages

A package groups related classes and prevents name clashes. Declare it as the first statement of a file, and bring other classes in with `import`.

```
package com.example.app;  
  
import java.util.List;
```

By convention, package names are lowercase and the folder structure mirrors them.

Key points and common mistakes

- Prefer the most restrictive modifier that works; keep fields `private`.
- Placing `super(...)` after other statements causes a compile error.
- Assuming a `final` object is immutable is a frequent error.
- Forgetting that default access is package-private, not public.

Inner, Nested and Anonymous Classes

Nested Classes at a Glance

Java lets you declare a class inside another class. Use this when the inner type exists only to serve its outer class. The result is tighter encapsulation and less clutter in the package.

Kind	Declaration	Needs outer instance?	Typical use
Static nested	<code>static class</code> inside a class	No	Helper types such as a <code>Node</code> or <code>Builder</code>
Inner (non-static)	<code>class</code> inside a class	Yes	Objects tied to one outer object, such as an iterator
Anonymous	<code>new Type() { ... }</code>	Created in place	One-off implementation of an interface or class

Example

JAVA

```
public class Outer {
    private int value = 10;

    static class Nested {
        void show() { System.out.println("Static nested"); }
    }

    class Inner {
        void show() { System.out.println("Inner sees value " + value); }
    }

    public static void main(String[] args) {
        Outer.Nested n = new Outer.Nested();
        n.show();

        Outer outer = new Outer();
        Outer.Inner in = outer.new Inner();
        in.show();

        Runnable r = new Runnable() {
            @Override public void run() { System.out.println("Anonymous run"); }
        };
        r.run();
    }
}
```

Expected output:

```
Static nested
Inner sees value 10
Anonymous run
```

Choosing the Right One

- **Static nested:** the default choice when the class does not need the outer object's state.
- **Inner:** choose it when each instance must read or modify a specific outer object's fields, including private ones.
- **Anonymous:** suited to short, single-use implementations. For a functional interface, a lambda is usually clearer (covered in a later section).

Key Points and Common Mistakes

- Creating an inner instance without an outer instance (`new Inner()` from a static method) causes a compile error.
- Inner classes hold a reference to the outer object, which can keep it alive longer than intended.
- Anonymous classes cannot declare constructors and are hard to reuse; name the class if you need it twice.

Enums and Records

Why Enums and Records?

Java offers two compact constructs for modelling data safely. An **enum** represents a fixed set of named constants, such as days of the week or order states. A **record** represents immutable data with minimal boilerplate. Both are type-safe: the compiler rejects invalid values and mismatched types.

Enums

An enum declares every legal value up front. Enums can also hold fields, constructors, and methods.

```
JAVA

enum Level {
    LOW(1), MEDIUM(5), HIGH(10);

    private final int weight;
    Level(int weight) { this.weight = weight; }
    int getWeight() { return weight; }
}

public class EnumDemo {
    public static void main(String[] args) {
        Level l = Level.HIGH;
        System.out.println(l + " " + l.getWeight());
        for (Level x : Level.values()) System.out.println(x);
    }
}
```

Expected output:

```
HIGH 10
LOW
MEDIUM
HIGH
```

Enums work well in `switch` statements and `switch` expressions.

Records

A record declares its components once; the compiler generates the constructor, accessors, `equals`, `hashCode`, and `toString`.

```
JAVA

record Point(int x, int y) { }

Point p = new Point(2, 3);
System.out.println(p);           // Point[x=2, y=3]
System.out.println(p.x());       // 2
```

Record fields are final, so a record is shallowly immutable.

Comparison

Feature	Enum	Record
Purpose	Fixed set of constants	Immutable data carrier
Instances	Predefined only	Created with <code>new</code>
Generated members	<code>values()</code> , <code>valueOf()</code>	<code>Accessors</code> , <code>equals</code> , <code>hashCode</code> , <code>toString</code>

Key Points and Common Mistakes

- Use `==` to compare enum constants.
- Enum constructors are implicitly private.
- Records cannot extend other classes, though they may implement interfaces.
- A record holding a mutable object (such as a list) can still change through that object.

Exception Handling Fundamentals

Exception handling lets a program respond to runtime problems without crashing. An exception is an object that describes an abnormal event; when one is thrown, normal flow stops and Java looks for a matching handler.

try, catch and finally

Place risky code in `try`, handle specific failures in `catch`, and put cleanup code in `finally`, which runs whether or not an exception occurs.

JAVA

```
public class Demo {
    public static void main(String[] args) {
        try {
            int result = 10 / 0;
            System.out.println(result);
        } catch (ArithmeticException e) {
            System.out.println("Error: " + e.getMessage());
        } finally {
            System.out.println("Cleanup done");
        }
    }
}
```

Expected output:

```
Error: / by zero
Cleanup done
```

Checked vs unchecked exceptions

Aspect	Checked	Unchecked
Base type	Exception (not RuntimeException)	RuntimeException and Error
Compiler enforcement	Must be caught or declared	Not enforced
Typical cause	External conditions, such as a missing file	Programming bugs
Examples	IOException	NullPointerException, ArithmeticException

throw vs throws

- `throw` is a statement that raises one exception object: `throw new IllegalArgumentException("Age must be positive");`
- `throws` appears in a method signature and declares which checked exceptions the caller must handle: `void read() throws IOException.`

Key points and common mistakes

- Order `catch` blocks from most specific to most general; a superclass catch placed first makes later ones unreachable and causes a compile error.
- Avoid empty `catch` blocks, which hide failures.
- Catch the most specific type you can rather than `Exception`.
- Use `finally` for cleanup, not for normal logic.

Custom Exceptions and try-with-resources

Custom Exceptions

When built-in exceptions do not describe a domain problem well, define your own by extending `Exception` (checked) or `RuntimeException` (unchecked). Provide constructors that accept a message and, optionally, a cause.

JAVA

```
class InsufficientFundsException extends Exception {
    public InsufficientFundsException(String message) {
        super(message);
    }
}

class Account {
    private double balance = 100;
    void withdraw(double amount) throws InsufficientFundsException {
        if (amount > balance)
            throw new InsufficientFundsException("Balance too low: " + balance);
        balance -= amount;
    }
}
```

try-with-resources

Any object implementing `AutoCloseable` can be declared in the `try` parentheses. It is closed automatically, in reverse order of declaration, even if an exception is thrown.

JAVA

```
class Resource implements AutoCloseable {
    private final String name;
    Resource(String name) { this.name = name; System.out.println("Open " + name); }
    public void close() { System.out.println("Close " + name); }
}

public class Demo {
    public static void main(String[] args) {
        try (Resource a = new Resource("A");
             Resource b = new Resource("B")) {
            new Account().withdraw(500);
        } catch (InsufficientFundsException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

Expected output:

```
Open A
Open B
Close B
Close A
Error: Balance too low: 100.0
```

Note that resources are closed before the `catch` block runs.

Key Points and Common Mistakes

- Use checked exceptions for recoverable conditions callers must handle; use unchecked ones for programming errors.
- Name custom exceptions with an `Exception` suffix.
- Preserve the original cause by passing it to `super(message, cause)`.
- Do not swallow exceptions with an empty `catch` block.
- Prefer try-with-resources over manual `finally` cleanup; exceptions thrown by `close()` are attached as suppressed exceptions.

Wrappers, Autoboxing and Object Methods

Wrapper Classes

Every primitive type has a matching wrapper class in `java.lang`: `byte`→`Byte`, `short`→`Short`, `int`→`Integer`, `long`→`Long`, `float`→`Float`, `double`→`Double`, `char`→`Character`, `boolean`→`Boolean`. Wrappers let primitives be used where objects are required, such as in collections and generics (`List<Integer>`).

Autoboxing and Unboxing

Autoboxing is the automatic conversion of a primitive to its wrapper; unboxing is the reverse.

JAVA

```
Integer a = 10;      // autoboxing
int b = a;           // unboxing
Integer x = 100, y = 100;
Integer p = 1000, q = 1000;
System.out.println(x == y);
System.out.println(p == q);
System.out.println(p.equals(q));
```

Expected output (on a standard JVM):

```
true
false
true
```

Small `Integer` values (typically -128 to 127) are cached, so `==` may appear to work; larger values are separate objects. Use `equals` to compare wrapper values.

The Object Methods

Every class inherits from `Object`, including three methods you will often override.

Method	Purpose	Contract
<code>equals(Object)</code>	Logical equality	Reflexive, symmetric, transitive, consistent; false for <code>null</code>
<code>hashCode()</code>	Integer used by hash-based collections	Equal objects must return equal hash codes
<code>toString()</code>	Readable description	Default prints class name and hash, so override it

JAVA

```
record Point(int x, int y) {}
System.out.println(new Point(1, 2));
```

Output: `Point[x=1, y=2]`

Key Points and Common Mistakes

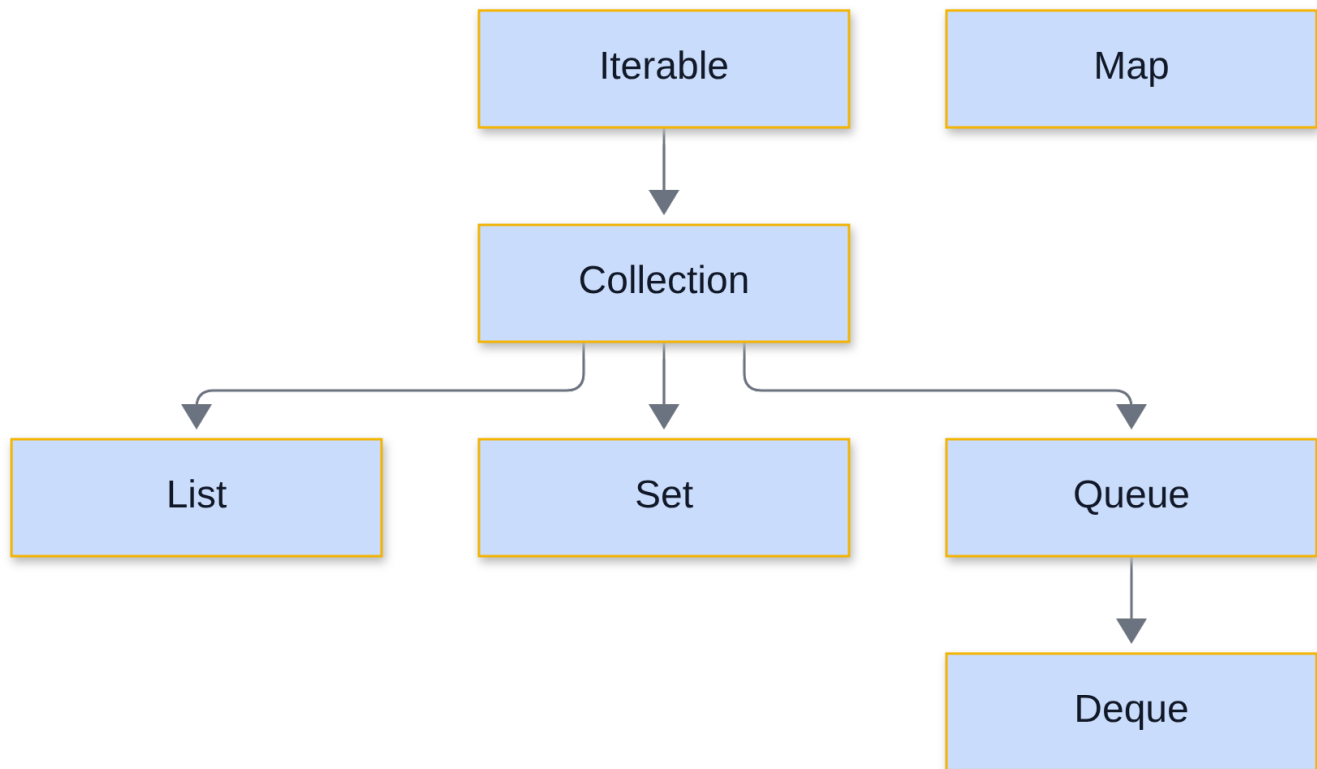
- Overriding `equals` without `hashCode` breaks `HashMap` and `HashSet` lookups.
- Unboxing a `null` wrapper throws `NullPointerException`.
- Avoid `==` for wrapper or `String` content comparison.
- Records generate `equals`, `hashCode` and `toString` automatically.

Collections Framework Overview and Hierarchy

What the Collections Framework Is

The Collections Framework is Java's standard set of interfaces and classes for storing and manipulating groups of objects. Programming against the interfaces (for example `List`) rather than a concrete class (`ArrayList`) lets you swap implementations later without rewriting your code.

Hierarchy



`Map` stands apart: it does not extend `Collection` because it stores key-value pairs.

Choosing the Right Interface

Interface	Order / Rules	Use it when
<code>List</code>	Ordered, indexed, allows duplicates	You need positions or repeated items
<code>Set</code>	No duplicates	You need uniqueness
<code>Map</code>	Unique keys mapped to values	You look up values by key
<code>Queue</code>	Items processed in turn, typically FIFO	You handle tasks in arrival order
<code>Deque</code>	Add/remove at both ends	You need a queue or a stack

Example

```
import java.util.*;

public class Demo {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>(List.of("A", "B", "A"));
        Set<String> set = new HashSet<>(list);
        Map<String, Integer> map = new HashMap<>();
        map.put("A", 1);
        Deque<Integer> dq = new ArrayDeque<>();
        dq.addFirst(1);
        dq.addLast(2);
        System.out.println(list);
        System.out.println(set.size());
        System.out.println(map);
        System.out.println(dq);
    }
}
```

Expected output:

```
[A, B, A]
2
{A=1}
[1, 2]
```

Key Points and Common Mistakes

- Declare variables with the interface type, such as `List<String> names = new ArrayList<>();`.
- Do not expect `Map` to be a `Collection`.
- A `Set` silently ignores a duplicate add; it does not throw.
- `Deque` can replace the legacy `Stack` class.
- Collections hold objects, so primitives are boxed (covered in section 20).

List Implementations: ArrayList vs LinkedList

Overview

`ArrayList` and `LinkedList` both implement the `List` interface, so they share the same API: `add`, `get`, `remove`, `size` and so on. They differ in how they store elements, and that difference decides which one performs better for a given task.

- **ArrayList** keeps elements in a resizable array.
- **LinkedList** keeps elements in nodes linked to their neighbours (it also implements `Deque`).

Comparison Table

Aspect	ArrayList	LinkedList
Internal structure	Resizable array	Doubly linked nodes
Access by index (<code>get(i)</code>)	Fast, constant time	Slow, must walk the nodes
Add/remove at the end	Fast (amortized)	Fast
Add/remove at the start or middle	Slow, elements shift	Fast once the position is reached; finding it is linear
Memory per element	Low	Higher (each node stores two links)
Typical use	Default choice, read-heavy	Queue/deque-style work at both ends

Runnable Example

JAVA

```
import java.util.*;

public class ListDemo {
    public static void main(String[] args) {
        List<Integer> arrayList = new ArrayList<>();
        List<Integer> linkedList = new LinkedList<>();

        for (int i = 0; i < 5; i++) {
            arrayList.add(i);
            linkedList.add(i);
        }

        arrayList.add(0, 99); // shifts every element right
        linkedList.add(0, 99); // relinks only the head

        System.out.println(arrayList.get(3));
        System.out.println(linkedList.get(3));
        System.out.println(arrayList);
        System.out.println(linkedList);
    }
}
```

Expected output:

```
2  
2  
[99, 0, 1, 2, 3]  
[99, 0, 1, 2, 3]
```

Both lists give identical results; only the cost of the operations differs, and that cost becomes visible with large lists.

Key Points

- Choose `ArrayList` by default; random access and iteration are its strengths.
- Choose `LinkedList` mainly when you frequently add or remove at the ends, for example as a queue.
- **Common mistake:** assuming `LinkedList` is faster for middle insertions. Reaching the position is itself linear, so the advantage is often lost.
- Calling `get(i)` in a loop over a `LinkedList` is slow; use a for-each loop or an iterator instead.

HashMap Internals

How HashMap Stores Entries

A `HashMap<K, V>` keeps its entries in an internal array of **buckets**. When you call `put(key, value)`, Java calls `key.hashCode()`, mixes the bits of the result, and maps it to a bucket index. The entry (key, value, and its hash) is stored in that bucket. `get(key)` repeats the same steps to find the bucket, then compares keys inside it.

Collisions

Different keys can land in the same bucket; this is a **collision**. Entries in one bucket are chained together. Within a bucket, `HashMap` first compares hash values and then calls `equals()` to find the exact key. In modern Java (8 and later), a bucket that grows long can be converted from a linked list into a balanced tree, which keeps worst-case lookups from degrading badly.

Why equals and hashCode Must Agree

The contract is: if `a.equals(b)` is true, then `a.hashCode() == b.hashCode()`. If you override `equals()` but not `hashCode()`, two equal keys may hash to different buckets, so `get()` cannot find an entry you stored.

JAVA

```
import java.util.*;

class Point {
    int x, y;
    Point(int x, int y) { this.x = x; this.y = y; }
    @Override public boolean equals(Object o) {
        return o instanceof Point p && x == p.x && y == p.y;
    }
    @Override public int hashCode() { return Objects.hash(x, y); }
}

public class Demo {
    public static void main(String[] args) {
        Map<Point, String> map = new HashMap<>();
        map.put(new Point(1, 2), "A");
        System.out.println(map.get(new Point(1, 2)));
    }
}
```

Expected output:

A

Removing the `hashCode()` method would typically make this print `null`.

Key Points and Common Mistakes

- Use immutable keys; changing a key's fields after insertion can make it unreachable.
- Always override `equals()` and `hashCode()` together.
- `HashMap` allows one `null` key and does not guarantee iteration order.
- Average `put` and `get` are fast, but depend on a well-distributed `hashCode()`.

Sets and TreeMap: Ordering Choices

Choosing by Ordering

A `Set` stores unique elements. The implementation you pick decides iteration order and lookup cost. `TreeMap` applies the same sorted-key idea to key-value pairs.

Type	Iteration order	Typical lookup	Null allowed
<code>HashSet</code>	Unspecified	$O(1)$ average	One null
<code>LinkedHashSet</code>	Insertion order	$O(1)$ average	One null
<code>TreeSet</code>	Sorted (natural or <code>Comparator</code>)	$O(\log n)$	No null with natural ordering
<code>TreeMap</code>	Sorted by key	$O(\log n)$	No null key with natural ordering

Example

```
import java.util.*;

public class OrderDemo {
    public static void main(String[] args) {
        String[] data = {"pear", "apple", "fig", "apple", "banana"};

        Set<String> hash = new HashSet<>(Arrays.asList(data));
        Set<String> linked = new LinkedHashSet<>(Arrays.asList(data));
        Set<String> tree = new TreeSet<>(Arrays.asList(data));

        System.out.println("LinkedHashSet: " + linked);
        System.out.println("TreeSet: " + tree);

        TreeMap<String, Integer> map = new TreeMap<>();
        for (String s : data) map.merge(s, 1, Integer::sum);
        System.out.println("TreeMap: " + map);
        System.out.println("First key: " + map.firstKey());
    }
}
```

JAVA

Output:

```
LinkedHashSet: [pear, apple, fig, banana]
TreeSet: [apple, banana, fig, pear]
TreeMap: {apple=2, banana=1, fig=1, pear=1}
First key: apple
```

`HashSet` is omitted from the output because its order is unspecified and may differ between runs or Java versions. The duplicate `apple` is stored once in every set.

Key Points

- Use `HashSet` when only membership speed matters.
- Use `LinkedHashSet` when you need predictable insertion order.
- Use `TreeSet` or `TreeMap` for sorted data and range-style methods such as `firstKey()`.

Common Mistakes

- Relying on `HashSet` order.
- Adding elements to a `TreeSet` that are not `Comparable` and supplying no `Comparator`, which fails at runtime.
- Overriding `equals` without `hashCode` for hash-based sets.

Iterator, Sorting and Collections Utilities

Iterator: Safe Traversal

An `Iterator` walks a collection using `hasNext()`, `next()` and `remove()`. It is the safe way to remove elements while traversing.

JAVA

```
List<Integer> nums = new ArrayList<>(List.of(1, 2, 3, 4, 5, 6));
Iterator<Integer> it = nums.iterator();
while (it.hasNext()) {
    if (it.next() % 2 == 0) it.remove();
}
System.out.println(nums);
```

Output: [1, 3, 5]

Common mistake: calling `nums.remove(...)` inside a for-each loop typically throws `ConcurrentModificationException`, because the list was changed behind the iterator's back. Use `it.remove()` or `nums.removeIf(n -> n % 2 == 0)`.

Comparable vs Comparator

Aspect	Comparable	Comparator
Package	<code>java.lang</code>	<code>java.util</code>
Method	<code>compareTo(T o)</code>	<code>compare(T a, T b)</code>
Defined	Inside the class	Separately, outside the class
Orderings	One natural order	Many custom orders

JAVA

```
record Person(String name, int age) {}

List<Person> people = new ArrayList<>(List.of(
    new Person("Ravi", 30), new Person("Asha", 25), new Person("Meena", 25)));

people.sort(Comparator.comparingInt(Person::age)
    .thenComparing(Person::name));
System.out.println(people);
```

Output: [Person[name=Asha, age=25], Person[name=Meena, age=25], Person[name=Ravi, age=30]]

The Collections Utility Class

```
List<Integer> list = new ArrayList<>(List.of(4, 1, 3, 2));
Collections.sort(list);           // [1, 2, 3, 4]
Collections.reverse(list);        // [4, 3, 2, 1]
System.out.println(Collections.max(list)); // 4
List<Integer> ro = Collections.unmodifiableList(list);
```

Other helpers include `shuffle`, `min`, `binarySearch` (the list must be sorted first) and `swap`.

Key Points and Mistakes

- Never modify a collection structurally during for-each; use the iterator or `removeIf`.
- Make `compareTo` consistent with `equals` where possible.
- Avoid subtracting integers in comparators (overflow risk); use `Integer.compare`.
- Unmodifiable views still reflect changes to the underlying list.

Generics: Classes, Methods and Wildcards

Generics let a class or method work with many types while the compiler checks type safety. They remove casts and turn many runtime `ClassCastException` errors into compile-time errors.

Generic Classes

```
class Box<T> {
    private T value;
    Box(T value) { this.value = value; }
    T get() { return value; }
}

Box<String> b = new Box<>("Java");
System.out.println(b.get());
```

JAVA

Output:

```
Java
```

`T` is a type parameter. Writing `Box<String>` fixes it to `String`, so `b.get()` needs no cast.

Generic Methods

```
static <T> void printAll(T[] items) {
    for (T t : items) System.out.print(t + " ");
}

// printAll(new Integer[]{1, 2, 3});
```

JAVA

Output: 1 2 3

The type parameter is declared before the return type.

Bounded Types

```
static <T extends Comparable<T>> T max(T a, T b) {
    return a.compareTo(b) >= 0 ? a : b;
}

// max(3, 7) returns 7
```

JAVA

`T extends Comparable<T>` restricts callers to types that can be compared, so `compareTo` is available.

Wildcards

Form	Meaning	Typical use
<code>List<?></code>	list of unknown type	read as <code>Object</code>
<code>List<? extends Number></code>	<code>Number</code> or a subtype	reading (producer)
<code>List<? super Integer></code>	<code>Integer</code> or a supertype	writing (consumer)

A common guide is "producer extends, consumer super".

Key Points and Common Mistakes

- Generics work with reference types only; use `Box<Integer>`, not `Box<int>`.
- `List<Integer>` is not a `List<Number>`; use a wildcard instead.
- Type information is erased at runtime, so `new T()` is not allowed.
- Avoid raw types such as `List`; they bypass compile-time checks.

Lambdas, Functional Interfaces and Method References

Part 3 begins here. Lambdas let you pass behavior as a value, replacing the verbose anonymous classes from the inner-class section.

From Anonymous Class to Lambda

A **functional interface** has exactly one abstract method, such as `Runnable` or `Comparator`. A lambda supplies that method's body directly.

JAVA

```
import java.util.*;

public class LambdaDemo {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>(List.of("Charlie", "Al", "Bob"));

        // Anonymous class (verbose)
        names.sort(new Comparator<String>() {
            @Override
            public int compare(String a, String b) {
                return a.length() - b.length();
            }
        });

        // Lambda (concise)
        names.sort((a, b) -> a.length() - b.length());

        // Method reference (most concise)
        names.sort(Comparator.comparingInt(String::length));

        names.forEach(System.out::println);
    }
}
```

Expected output:

```
Al
Bob
Charlie
```

Common Functional Interfaces

Interface	Method	Purpose
Runnable	<code>void run()</code>	Action, no input or result
Supplier<T>	<code>T get()</code>	Produces a value
Consumer<T>	<code>void accept(T)</code>	Uses a value
Function<T,R>	<code>R apply(T)</code>	Transforms a value
Predicate<T>	<code>boolean test(T)</code>	Tests a condition

You can mark your own with `@FunctionalInterface`, and the compiler will check the single-method rule.

Method Reference Forms

- Static method: `Integer::parseInt`
- Instance method of a particular object: `System.out::println`
- Instance method of an arbitrary object of a type: `String::length`
- Constructor: `ArrayList::new`

Key Points and Common Mistakes

- Lambdas may use local variables only if they are effectively final.
- A lambda is not valid for an interface with more than one abstract method.
- Prefer a method reference when it is clearer than the equivalent lambda; keep lambda bodies short.

Streams API and Optional

A **stream** is a pipeline that processes a sequence of elements: a source, zero or more intermediate operations (lazy), and one terminal operation that triggers execution. Streams do not modify their source.

Pipeline Example

JAVA

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

public class StreamDemo {
    public static void main(String[] args) {
        List<String> words = List.of("apple", "banana", "avocado", "cherry", "blueberry");

        List<String> result = words.stream()
            .filter(w -> w.length() > 5)           // keep long words
            .map(String::toUpperCase)               // transform
            .collect(Collectors.toList());           // terminal
        System.out.println(result);

        int total = Stream.of(1, 2, 3, 4)
            .reduce(0, Integer::sum);               // combine
        System.out.println(total);

        Map<Character, List<String>> byFirst = words.stream()
            .collect(Collectors.groupingBy(w -> w.charAt(0)));
        System.out.println(byFirst);
    }
}
```

Expected output (the map's key order may vary):

```
[BANANA, AVOCADO, CHERRY, BLUEBERRY]
10
{a=[apple, avocado], b=[banana, blueberry], c=[cherry]}
```

Optional

`Optional<T>` represents a value that may be absent, making "no result" explicit.

JAVA

```
Optional<String> first = words.stream()
    .filter(w -> w.startsWith("c"))
    .findFirst();
System.out.println(first.orElse("none"));
first.ifPresent(System.out::println);
```

Output:

```
cherry  
cherry
```

Key Points and Common Mistakes

- Intermediate operations (`filter`, `map`) do nothing until a terminal operation runs.
- A stream can be consumed only once; reusing it throws `IllegalStateException`.
- Avoid calling `Optional.get()` without checking; prefer `orElse`, `orElseGet`, or `ifPresent`.
- Do not use `Optional` for fields or method parameters.

Multithreading Basics and Thread Lifecycle

A **thread** is an independent path of execution inside a program. Java supports multithreading so that several tasks, such as handling requests or doing background work, can make progress at the same time.

Creating Threads

There are two basic ways: extend `Thread`, or implement `Runnable` and pass it to a `Thread`. Implementing `Runnable` is usually preferred because the class remains free to extend another class and the task is separate from the thread that runs it.

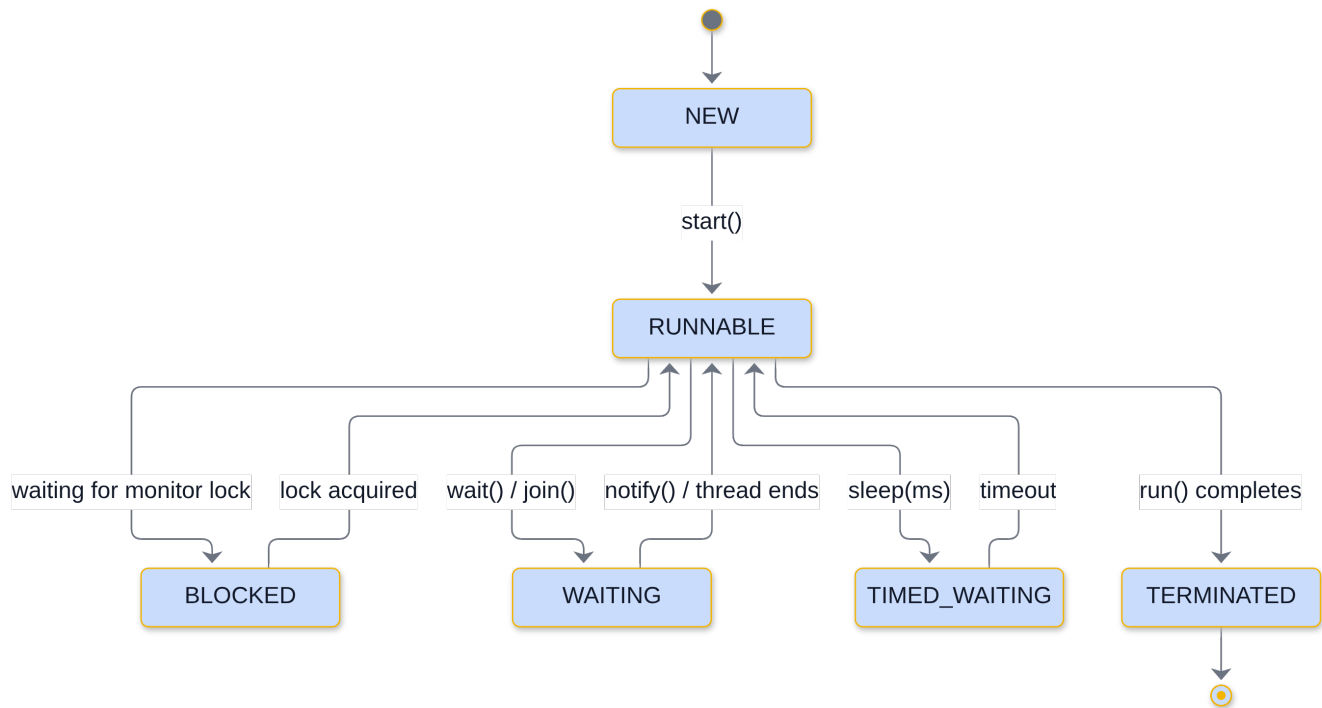
```
public class ThreadDemo {  
    public static void main(String[] args) throws InterruptedException {  
        Runnable task = () -> System.out.println("Running in " +  
Thread.currentThread().getName());  
        Thread t = new Thread(task, "worker-1");  
        t.start();    // schedules the thread  
        t.join();    // wait for it to finish  
        System.out.println("Done");  
    }  
}
```

JAVA

Expected output:

```
Running in worker-1  
Done
```

Thread Lifecycle



State	Meaning
NEW	Created, not yet started
RUNNABLE	Ready or running
BLOCKED	Waiting for a lock
WAITING / TIMED_WAITING	Waiting indefinitely or for a set time
TERMINATED	Finished execution

Key Points and Common Mistakes

- Call `start()`, not `run()`; calling `run()` directly executes on the current thread without creating a new one.
- A thread cannot be started twice; a second `start()` throws `IllegalThreadStateException`.
- Use `join()` to wait for completion, and handle `InterruptedException`.
- Output order between threads is not guaranteed.

Synchronization, volatile and Deadlock

The Problem: Shared Mutable State

When two threads update the same variable, their reads and writes can interleave. This is a **race condition**. For example, `count++` is really three steps (read, add, write), so two threads can both read the same value and one increment is lost.

JAVA

```
class Counter {  
    private int count = 0;  
    public synchronized void increment() { count++; }  
    public synchronized int get() { return count; }  
}
```

If two threads each call `increment()` 1000 times and are then joined, `get()` returns 2000. Without `synchronized`, the result may be lower and varies between runs.

synchronized

`synchronized` gives **mutual exclusion**: only one thread at a time may hold an object's monitor lock. It can be applied to a method or to a block (`synchronized (lock) { ... }`). Prefer small blocks on a private lock object to keep the locked region short.

volatile

`volatile` guarantees **visibility**: a write by one thread is seen by other threads reading that variable. It does **not** make compound actions such as `count++` atomic.

Feature	synchronized	volatile
Mutual exclusion	Yes	No
Visibility	Yes	Yes
Atomic compound actions	Yes	No
Typical use	Critical sections	Flags such as <code>running</code>

wait/notify

`wait()`, `notify()` and `notifyAll()` coordinate threads and must be called while holding the object's monitor. `wait()` releases the lock and suspends the thread until notified. Always call `wait()` inside a `while` loop that rechecks the condition, since a thread may wake without the condition being true.

Deadlock

Deadlock occurs when threads each hold a lock the other needs, so none can proceed. For example, thread A holds lock 1 and wants lock 2, while thread B holds lock 2 and wants lock 1.

Key Points and Common Mistakes

- Acquire multiple locks in a **consistent order** to avoid deadlock.
- Do not use `volatile` for counters; use `synchronized` or atomic classes.
- Keep synchronized regions short.
- Never call `wait()` with an `if` instead of a `while`.

Concurrency Utilities and Virtual Threads

Manually creating and managing threads does not scale well. Java's `java.util.concurrent` package and Java 21 virtual threads provide higher-level tools that separate *what* to run from *how* it is run.

The Modern Concurrency Toolkit

Tool	Purpose
<code>ExecutorService</code>	Runs submitted tasks on a managed pool of threads
<code>Callable</code> / <code>Future</code>	A task that returns a result, and a handle to retrieve it
<code>CompletableFuture</code>	Chains asynchronous steps without blocking
Locks	Explicit mutual exclusion with more control than <code>synchronized</code>
Concurrent collections	Thread-safe collections designed for concurrent access
Virtual threads (Java 21)	Lightweight threads for many blocking tasks

ExecutorService, Callable and Future

An `ExecutorService` accepts tasks and runs them on pooled threads. A `Runnable` returns nothing; a `Callable<V>` returns a value and may throw a checked exception. Submitting a `Callable` returns a `Future<V>`, whose `get()` waits for the result.

```
ExecutorService pool = Executors.newFixedThreadPool(2);
Future<Integer> f = pool.submit(() -> 6 * 7);
System.out.println(f.get());
pool.shutdown();
```

JAVA

Expected output: 42

CompletableFuture

`CompletableFuture` lets you compose asynchronous steps with methods such as `supplyAsync`, `thenApply` and `thenCombine`.

Locks and Concurrent Collections

`ReentrantLock` offers explicit `lock()` and `unlock()`; always release it in a `finally` block. Collections such as `ConcurrentHashMap` are built for concurrent use.

Virtual Threads (Java 21)

Virtual threads are cheap threads managed by the JVM, suited to many tasks that spend time waiting.

```
try (var ex = Executors.newVirtualThreadPerTaskExecutor()) {  
    ex.submit(() -> System.out.println("hello"));  
}
```

Key Points and Common Mistakes

- Always call `shutdown()` on pools you create.
- `Future.get()` blocks the calling thread.
- Unlocking outside `finally` risks leaving a lock held.

File I/O, NIO and Serialization

File I/O lets a program persist data. Java offers the classic `java.io` streams and the newer `java.nio.file` API. For most new code, prefer NIO's `Files` and `Path`.

Reading and Writing with NIO

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

public class FileDemo {
    public static void main(String[] args) throws Exception {
        Path p = Path.of("notes.txt");
        Files.writeString(p, "Hello\nJava");
        List<String> lines = Files.readAllLines(p);
        System.out.println(lines);
    }
}
```

JAVA

Expected output:

```
[Hello, Java]
```

Classic `java.io`

`BufferedReader` and `BufferedWriter` wrap `FileReader` and `FileWriter`. Always use `try-with-resources` so streams close automatically.

```
try (BufferedReader br = Files.newBufferedReader(Path.of("notes.txt"))) {
    String line;
    while ((line = br.readLine()) != null) System.out.println(line);
}
```

JAVA

Serialization

Serialization converts an object to bytes. The class implements `Serializable`; `ObjectOutputStream` writes it and `ObjectInputStream` reads it back. Fields marked `transient` are skipped.

```
record User(String name) implements java.io.Serializable {}
```

JAVA

Key Points and Common Mistakes

- Forgetting to close streams leaks resources; use `try-with-resources`.

- `readAllLines` and `readString` load the whole file, so avoid them for very large files; use `Files.lines` or a buffered reader.
- Declare a `serialVersionUID` to keep versions compatible.
- Mark sensitive fields `transient`, such as passwords.
- Never deserialize untrusted data; it is a known security risk.
- Handle `IOException`, a checked exception.

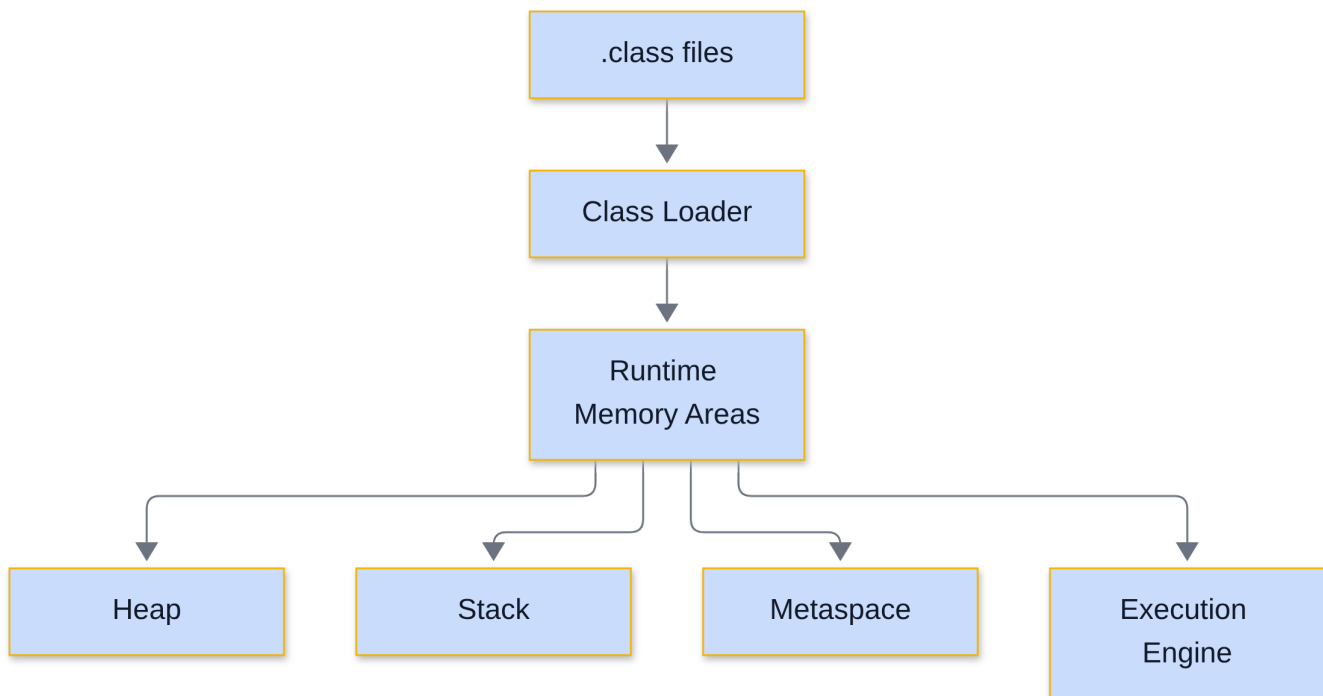
Aspect	java.io	NIO (java.nio.file)
Path type	File	Path
Convenience	Manual loops	Files helper methods
Recommended	Legacy code	New code

JVM Internals: Memory and Class Loading

Overview

The JVM runs compiled bytecode and manages memory for your program. Understanding its memory areas and how classes are loaded helps you diagnose errors such as `StackOverflowError` and `OutOfMemoryError`.

Architecture Diagram



Memory Areas

Area	Holds	Shared by threads?
Heap	Objects and arrays	Yes
Stack	Frames with local variables and method calls	No, one per thread
Metaspace	Class metadata (in native memory)	Yes

- **Heap:** every new object lives here. The garbage collector reclaims unreachable objects (covered in the next section).
- **Stack:** each method call pushes a frame; it is popped on return. Primitive locals and object references live in frames, while the objects themselves stay on the heap. Very deep recursion causes `StackOverflowError`.
- **Metaspace:** stores class structure information. Loading too many classes can cause `OutOfMemoryError: Metaspace`.

```
int x = 10;           // x in the stack frame
String s = new String("Hi"); // s (reference) on stack, object on heap
```

Class Loading

Classes are loaded on demand in three stages: **loading** (read the `.class` bytes), **linking** (verify, prepare, resolve) and **initialization** (run static initializers). Loaders typically delegate to their parent first:

1. Bootstrap loader – core Java classes.
2. Platform loader – platform modules.
3. Application loader – your classpath classes.

Key Points and Common Mistakes

- Stack memory is per thread; heap is shared, so shared heap objects need synchronization.
- Confusing a reference (stack) with the object it points to (heap) is a frequent error.
- Infinite recursion without a base case exhausts the stack.

Garbage Collection and Memory Leaks

What Garbage Collection Does

Garbage collection (GC) automatically reclaims heap memory held by objects that can no longer be reached from live references (GC roots such as stack variables and static fields). You never free objects yourself, but you still control whether they stay reachable.

Comparing GC Algorithms

The table below gives a general comparison of commonly used collectors in modern Java (17/21). Exact behavior depends on the JVM version and tuning.

Collector	Typical focus	Notes
Serial	Small heaps, single thread	Simple; pauses the application while collecting
Parallel	Throughput	Uses multiple threads; pauses can be longer
G1	Balanced pause time and throughput	Heap split into regions; common default on server-class machines
ZGC	Very low pause times	Mostly concurrent; suited to large heaps

Most collectors are generational: new objects start in a young area, and survivors are promoted to an older area, because most objects die young.

How Memory Leaks Arise

Java can still leak: an object that is reachable but no longer needed is never collected. Common causes:

- Static collections (for example a static Map) that only grow.
- Listeners or callbacks registered but never removed.
- Unclosed resources such as streams or connections.
- Caches without a size limit or eviction policy.

JAVA

```
class Registry {  
    static final java.util.List<byte[]> CACHE = new java.util.ArrayList<>();  
    static void add() { CACHE.add(new byte[1_000_000]); }  
}
```

Repeated calls to `add()` keep every array reachable, so heap use climbs until an `OutOfMemoryError` occurs.

Diagnosing Leaks in Practice

Suggested procedure:

1. Watch heap usage over time; steady growth after collections suggests a leak.

2. Enable GC logging to see how much memory each collection recovers.
3. Capture a heap dump and inspect it with a profiler to find the objects retained and the references holding them.
4. Fix the retaining reference, then re-test.

Key Points

- Reachability, not usefulness, decides collection.
- Remove listeners, bound caches, and close resources (try-with-resources).
- Avoid calling `System.gc()` as a fix; it is only a hint.

Reflection and Annotations

What Reflection Does

Reflection lets a running program inspect and use classes, methods, fields and constructors without knowing them at compile time. Annotations attach metadata to code. Together they let frameworks discover what to do by reading your code instead of requiring explicit wiring.

Custom Annotation Example

```
JAVA

import java.lang.annotation.*;
import java.lang.reflect.Method;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface RunMe { String label() default "task"; }

class Jobs {
    @RunMe(label = "first") public void a() { System.out.println("A ran"); }
    public void b() { System.out.println("B ran"); }
    @RunMe public void c() { System.out.println("C ran"); }
}

public class Runner {
    public static void main(String[] args) throws Exception {
        Jobs jobs = new Jobs();
        for (Method m : Jobs.class.getDeclaredMethods()) {
            if (m.isAnnotationPresent(RunMe.class)) {
                System.out.println("Running " + m.getAnnotation(RunMe.class).label());
                m.invoke(jobs);
            }
        }
    }
}
```

Expected output (the order of `getDeclaredMethods()` is not guaranteed, so lines may appear in a different order):

```
Running first
A ran
Running task
C ran
```

Method `b` is skipped because it lacks the annotation.

How Frameworks Use This

A framework scans classes, reads annotations, then creates objects or calls methods reflectively. This is the general idea behind annotation-driven configuration and test discovery; the details differ by framework.

Key Points

- `@Retention(RUNTIME)` is required for reflection to see an annotation; the default retention does not make it visible at runtime.
- `@Target` limits where an annotation may be applied.
- Core entry points are `Class`, `Method`, `Field` and `Constructor`.

Common Mistakes

- Forgetting runtime retention, so the annotation is never found.
- Using reflection where plain code works: it is slower, bypasses compile-time checks, and failures appear only at runtime.
- Handling checked exceptions such as `InvocationTargetException` carelessly.

Design Patterns and SOLID Principles

Design patterns are reusable solutions to recurring design problems. The SOLID principles explain why many of them work. Each pattern below is shown briefly; adapt the code to your own project.

Five Core Patterns

Pattern	Purpose	Typical use
Singleton	One shared instance	Configuration, registry
Factory	Hide object creation	Choosing an implementation
Builder	Step-by-step construction	Objects with many optional fields
Observer	Notify dependents of changes	Event listeners
Strategy	Swap algorithms at runtime	Pricing, sorting rules

Singleton (an enum is the simplest thread-safe form):

JAVA

```
enum AppConfig { INSTANCE; String env = "dev"; }  
// AppConfig.INSTANCE.env -> dev
```

Factory returns an interface type, so callers never name the concrete class:

JAVA

```
interface Shape { double area(); }  
record Circle(double r) implements Shape { public double area() { return Math.PI * r * r; } }  
record Square(double s) implements Shape { public double area() { return s * s; } }  
class ShapeFactory {  
    static Shape of(String type, double v) {  
        return switch (type) {  
            case "circle" -> new Circle(v);  
            case "square" -> new Square(v);  
            default -> throw new IllegalArgumentException(type);  
        };  
    }  
}
```

Builder avoids long constructors:

```

class User {
    final String name; final int age;
    private User(Builder b) { name = b.name; age = b.age; }
    static class Builder {
        String name; int age;
        Builder name(String n) { name = n; return this; }
        Builder age(int a) { age = a; return this; }
        User build() { return new User(this); }
    }
}
// new User.Builder().name("Asha").age(20).build();

```

Observer and **Strategy** are often written with lambdas:

```

List<Consumer<String>> listeners = new ArrayList<>();
listeners.add(m -> System.out.println("Got: " + m));
listeners.forEach(l -> l.accept("hello")); // Got: hello

Function<Double, Double> discount = p -> p * 0.9; // strategy
System.out.println(discount.apply(100.0)); // 90.0

```

SOLID Principles

- **S – Single Responsibility:** a class has one reason to change.
- **O – Open/Closed:** extend behavior without modifying existing code (Strategy, Factory).
- **L – Liskov Substitution:** subtypes must work wherever the parent type is expected.
- **I – Interface Segregation:** prefer small, focused interfaces.
- **D – Dependency Inversion:** depend on abstractions, not concrete classes.

Common Mistakes

- Overusing Singleton as hidden global state, which makes testing hard.
- Applying a pattern where simple code would do.

JDBC, Build Tools, JUnit and Spring Boot

JDBC Basics

JDBC (Java Database Connectivity) is the standard API for talking to relational databases. The usual flow is: obtain a `Connection`, create a `PreparedStatement`, execute it, read a `ResultSet`, and close everything.

JAVA

```
String sql = "SELECT name FROM users WHERE id = ?";
try (Connection con = DriverManager.getConnection(url, user, pass);
    PreparedStatement ps = con.prepareStatement(sql)) {
    ps.setInt(1, 42);
    try (ResultSet rs = ps.executeQuery()) {
        while (rs.next()) System.out.println(rs.getString("name"));
    }
}
```

The output depends on the data in your table. **Key points:** use try-with-resources so resources close automatically, and prefer `PreparedStatement` over string concatenation to prevent SQL injection. The database driver must be on the classpath.

Maven and Gradle

Build tools compile code, manage dependencies, run tests and package applications.

Aspect	Maven	Gradle
Config file	pom.xml (XML)	build.gradle (Groovy/Kotlin DSL)
Style	Convention-based, fixed lifecycle	Flexible, task-based
Typical command	mvn package	gradle build

Both download libraries declared as dependencies from a repository, so you avoid copying JAR files by hand.

JUnit Testing

JUnit is the common framework for unit tests.

JAVA

```
@Test
void addsNumbers() {
    assertEquals(5, 2 + 3);
}
```

A passing test is reported as successful by the build tool or IDE. Keep tests small and independent.

Spring Boot Overview

Spring Boot simplifies building Java applications, such as REST services, through auto-configuration, starter dependencies and an embedded server. A single class annotated with `@SpringBootApplication` can start an application. Typical projects combine all four topics here: Maven or Gradle builds the project, JDBC (often via Spring) accesses data, and JUnit verifies behavior.

Common mistake: skipping tests or hard-coding credentials in source code.

Modern Java Features (17/21)

Modern Java (17 LTS and 21 LTS) reduces boilerplate and makes intent clearer. This section summarizes five features with before/after examples. Output comments show what each snippet would print.

var (local variable type inference)

var lets the compiler infer a local variable's type from its initializer. The type is still static; it is not dynamic typing.

JAVA

```
// Before
Map<String, List<Integer>> scores = new HashMap<String, List<Integer>>();
// After
var scores = new HashMap<String, List<Integer>>();
```

Common mistakes: var needs an initializer (var x; fails), cannot be used for fields or method parameters, and var x = null; is invalid.

Text blocks

Triple quotes create multi-line strings without escape noise.

JAVA

```
// Before
String json = "{\n  \"name\": \"Ava\"\n}";
// After
String json = """
    {
      "name": "Ava"
    }
    """;
```

Incidental indentation is stripped based on the closing delimiter's position.

Records

A record is a concise immutable data carrier; it generates the constructor, accessors, equals, hashCode and toString.

JAVA

```
// Before: class with fields, constructor, getters, equals, hashCode, toString
// After
record Point(int x, int y) {}
System.out.println(new Point(1, 2)); // Point[x=1, y=2]
```

Sealed classes

sealed restricts which classes may extend a type.

JAVA

```
sealed interface Shape permits Circle, Square {}  
record Circle(double r) implements Shape {}  
record Square(double side) implements Shape {}
```

Permitted subclasses must be final, sealed or non-sealed.

Pattern matching

JAVA

```
// Before  
if (obj instanceof String) { String s = (String) obj; System.out.println(s.length()); }  
// After  
if (obj instanceof String s) { System.out.println(s.length()); }
```

In Java 21, switch also supports patterns, and with a sealed type the compiler can check exhaustiveness:

JAVA

```
double area = switch (shape) {  
    case Circle c -> Math.PI * c.r() * c.r();  
    case Square s -> s.side() * s.side();  
};
```

Key points

- Prefer these features to cut boilerplate, not to obscure code.
- Sealed types plus records plus pattern-matching switch model closed data cleanly.

Best Practices and Interview Quick Revision

Best Practices

These are suggested habits that keep Java code readable and safe:

- Prefer **immutability**: use `final` fields, records and unmodifiable collections where possible.
- Program to interfaces (`List<String> items = new ArrayList<>();`).
- Override `equals` and `hashCode` together.
- Use **try-with-resources** for anything that must be closed.
- Catch specific exceptions; never swallow them silently.
- Use `Optional` for return values, not for fields or parameters.
- Favor composition over inheritance; follow SOLID.

Performance Tips

- Use `StringBuilder` for string building inside loops.
- Set an initial capacity when the size of an `ArrayList` or `HashMap` is roughly known.
- Avoid needless autoboxing in hot loops; use primitives.
- Use thread pools (`ExecutorService`) rather than creating threads by hand; consider virtual threads (Java 21) for many blocking tasks.
- Measure before optimizing, and release references to avoid memory leaks (for example, unbounded static caches).

Quick-Revision Cheat Sheet

Question	Short answer
JDK vs JRE vs JVM	JDK is the development kit, JRE the runtime environment, JVM executes bytecode
<code>==</code> vs <code>equals</code>	<code>==</code> compares references (or primitive values); <code>equals</code> compares logical content
String vs StringBuilder	String is immutable; StringBuilder is mutable
Abstract class vs interface	Abstract class can hold state and constructors; interface defines a contract (default, static and private methods allowed)
Overloading vs overriding	Overloading: same name, different parameters, resolved at compile time; overriding: subclass redefines a method, resolved at runtime
Checked vs unchecked	Checked exceptions must be handled or declared; unchecked need not be
<code>throw</code> vs <code>throws</code>	<code>throw</code> raises an exception; <code>throws</code> declares one
ArrayList vs LinkedList	ArrayList: fast indexed access; LinkedList: efficient insertion at the ends
Comparable vs Comparator	Natural order inside the class vs external custom order
Thread vs Runnable	Runnable describes a task; Thread runs it
<code>synchronized</code> vs <code>volatile</code>	Synchronized gives mutual exclusion and visibility; volatile gives visibility only

Common Mistakes to Revisit

- Comparing strings with `==`.
- Modifying a collection while iterating over it.
- Forgetting `hashCode` when overriding `equals`.
- Ignoring deadlock risk when acquiring several locks.

Learning Roadmap

A roadmap turns these notes into a practice plan. The stages below follow the order of the three parts: Beginner, Intermediate (OOP) and Advanced. Time estimates are deliberately left out, because pace varies from learner to learner.

Stage 1: Beginner foundations

- **Study:** the JDK, JRE and JVM, the compilation flow, variables and types, operators, control flow, arrays, Strings and methods.
- **Build:** small console programs that read input with `Scanner`, such as a calculator, a number-guessing game or a text-menu tool.
- **Checkpoint:** you can write, compile and run a program from the command line and explain what each step does.

Stage 2: Object-oriented Java

- **Study:** classes, constructors, the four pillars, abstract classes and interfaces, enums, records, exceptions, wrappers and `Object` methods.
- **Study next:** the Collections Framework (including `HashMap` internals and `Comparable` vs `Comparator`) and generics.
- **Build:** a modelling project such as a library or inventory manager. It should use custom exceptions and collections to store and sort its data.
- **Checkpoint:** you can choose the right collection for a task and justify the choice.

Stage 3: Advanced Java

- **Study:** lambdas, Streams and `Optional`, then multithreading and the concurrency utilities, file I/O and NIO, and JVM internals including garbage collection.
- **Study next:** reflection, annotations, design patterns, SOLID, and the modern features in Java 17 and 21.
- **Build:** a concurrent task processor using `ExecutorService` and `CompletableFuture`, then a small application that applies a few design patterns.
- **Checkpoint:** you can explain thread lifecycle, synchronization and deadlock, and reason about memory leaks.

Stage 4: Ecosystem and practice

- **Study:** JDBC, Maven or Gradle, JUnit and a Spring Boot overview.
- **Build:** a tested application that stores data in a database and exposes it through Spring Boot.
- **Revisit:** the best-practices and interview quick-revision section, and write tests for everything you build.

At every stage, write code yourself instead of only reading it. Return to earlier sections whenever a concept feels unclear.