Ars Digitale
Engineering Notes Series

JAVA 21 ENGINEERING GUIDE

Java Language, Architecture and Best Practices

Alessandro Fabri

2026 Edition

info@ars-digitale.com

Java 21 Engineering Guide

Alessandro Fabri

Ars Digitale

All rights reserved.

Java 21 Engineering Guide

A structured study guide for modern Java development and certification

Alessandro Fabri

2026 Edition

Contact: info@ars-digitale.com

Web: https://www.ars-digitale.com

Copyright

Java 21 Engineering Guide
Author: **Alessandro Fabri**

All rights reserved.

This book is provided for personal study, training, and educational use.

Version: 1.0
Language: English

Contact: info@ars-digitale.com

Web: https://www.ars-digitale.com

2026 Edition

Preface

This guide is designed as a structured and practical roadmap for studying Java 21 with a focus on clarity, correctness, and long-term understanding.

The material is organized by modules, progressing from language fundamentals to APIs, concurrency, I/O, and the Java Platform Module System.

Each chapter is intended to work both as part of a sequential learning path and as a standalone technical reference.

About This Book

This book was designed as a technical learning companion for Java 21.

It aims to combine:

- structured progression
- concise explanations
- technical precision
- exam-oriented clarity
- long-term usefulness as a reference

The EPUB edition is optimized for digital reading and chapter-based navigation.

Contact: info@ars-digitale.com

Web: https://www.ars-digitale.com

🌍 **Language:** English | Italiano | Français

Course Index (Java 21)

This index provides the complete **English (EN)** curriculum for the **Java 21 Study Guide**.

Modules are designed to be read **sequentially**, but each topic can also be used as a standalone reference.

Module 00 — Prerequisites & Setup

- Prerequisite material
- Eclipse shortcuts

Module 01 — Java Language Basics

- Syntax building blocks
- Basic language building blocks
- Java naming rules
- Java data types and casting
- Java operators
- Instantiating types

Module 02 — Control Flow

- Control flow statements
- Loops

Module 03 — Core Standard APIs

- Strings in Java
- Arrays in Java
- Math utilities
- Dates and time
- Formatting and localization

Module 04 — Object-Oriented Fundamentals

- Methods, attributes, and variables
- Class loading, initialization, and object construction
- Inheritance
- Beyond classes
- Generics in Java
- Exceptions and error handling

Module 05 — Functional Programming

- Functional programming in Java
- Java streams

Module 06 — Collections Framework

- Introduction to the collections framework
- Shared collection operations and equality
- Sorting and comparing
- List APIs
- Set APIs
- Queue and deque APIs
- Map APIs
- Sequenced collections

Module 07 — Concurrency and Threads

- Thread fundamentals
- Concurrency APIs

Module 08 — Java I/O and NIO

- Files and paths fundamentals
- Files and paths APIs
- Java I/O streams
- I/O streams APIs
- Interacting with the user

Module 09 — Java Platform Module System (JPMS)

- JPMS fundamentals
- Compiling, packaging, and running modules
- Services in JPMS

◀ | ▲ Index | Prerequisite material for the course ▶

Module 00

Prerequisites & Setup

Prerequisite material for the course

This is all the prerequisite material you will need for the course

DOCUMENTATION

- **Java 21 APIs** - Java 21 API Specification
- **Eclipse shortcuts** - Eclipse IDE shortcuts

EDITOR

- **Eclipse IDE** - Download Eclipse here

PANDOC

- **Pandoc** - Download Pandoc here

◀ Course Index (Java 21) | ▲ Index | Eclipse main shortcuts ▶

Eclipse main shortcuts

WIN	APPLE	DESCRIPTION
<kbd>Ctrl</kbd> + 3	<kbd>⌘</kbd> + 3	Go to quick access search for available views, actions, wizards, menus and more
<kbd>Alt</kbd> + <kbd>⇧</kbd> + Q Q	<kbd>⌥</kbd> + <kbd>⌘</kbd> + Q Q	Show all available views and select one or more to open
F2	F2	Show Javadoc for the selected element
F3 or <kbd>Ctrl</kbd> + Left click	F3 or <kbd>⌘</kbd> + Left click	In a code editor, go to the declaration of the selected symbol
F4	F4	Show selected symbol in the “Type Hierarchy” view
<kbd>Ctrl</kbd> + <kbd>⇧</kbd> + T	<kbd>⌘</kbd> + <kbd>⇧</kbd> + T	Open dialog to search for a type (class, interface, enum)
<kbd>Ctrl</kbd> + Alt + H	^ + <kbd>⌥</kbd> + H	Open selected callable symbol in the “Call Hierarchy” view
<kbd>Ctrl</kbd> + <kbd>⇧</kbd> + G	<kbd>⇧</kbd> + <kbd>⌘</kbd> + G	Search workspace for all references to the symbol
<kbd>Ctrl</kbd> + <kbd>⇧</kbd> + R	<kbd>⌘</kbd> + <kbd>⇧</kbd> + R	Open dialog to search resources (e.g. text files) by filename
<kbd>Ctrl</kbd> + F	<kbd>⌘</kbd> + F	Find/replace in the current file
<kbd>Ctrl</kbd> + H	<kbd>⌘</kbd> + H	Find/replace in current file, project, or workspace
<kbd>Ctrl</kbd> + L	<kbd>⌘</kbd> + L	Go to a line number
<kbd>Ctrl</kbd> + .	<kbd>⌘</kbd> + .	Jump to next occurrence, warning or error
<kbd>Ctrl</kbd> + ,	<kbd>⇧</kbd> + <kbd>⌘</kbd> + .	Jump to previous occurrence, warning or error
<kbd>Ctrl</kbd> + D	<kbd>⌘</kbd> + D	Delete line at cursor position
<kbd>Alt</kbd> + ↑ or <kbd>Alt</kbd> + ↓	<kbd>⌥</kbd> + ↑ or <kbd>⌥</kbd> + ↓	Move current line one line above or one line below
<kbd>Ctrl</kbd> + Space	<kbd>⌘</kbd> + Space	Open content assist dialog (based on current context)
Type "main", "if", "for", "while", "do", "syso" + <kbd>Ctrl</kbd> + Space	(same as before) + <kbd>⌘</kbd> + Space	Autocomplete element
<kbd>Ctrl</kbd> + <kbd>⇧</kbd> + F		Format code
<kbd>Alt</kbd> + <kbd>⇧</kbd> + Z	<kbd>⌥</kbd> + <kbd>⌘</kbd> + Z	Toggle Try Catch and other predefined blocks of code
<kbd>Alt</kbd> + <kbd>⇧</kbd> + A	<kbd>⌥</kbd> + <kbd>⌘</kbd> + A	Toggle block / column selection in the current text editor

WIN	APPLE	DESCRIPTION
<kbd>Alt</kbd> + <kbd>⇧</kbd> + R	<kbd>⌥</kbd> + <kbd>⌘</kbd> + R	Rename (variable, field, method, class...)
<kbd>Alt</kbd> + <kbd>⇧</kbd> + S	<kbd>⌥</kbd> + <kbd>⌘</kbd> + S	Show advanced editing operations for current selection
<kbd>Alt</kbd> + <kbd>⇧</kbd> + T	<kbd>⌥</kbd> + <kbd>⌘</kbd> + T	Show available refactoring operations for current selection
<kbd>Ctrl</kbd> + 1	<kbd>⌘</kbd> + 1	Show all possible fixes for a problem (on a text element with a problem marker, or in the problem view)
<kbd>Ctrl</kbd> + <kbd>⇧</kbd> + C	<kbd>⌘</kbd> + /	Add/Remove line comment
<kbd>Ctrl</kbd> + <kbd>⇧</kbd> + /	^ + <kbd>⌘</kbd> + /	Add/Remove block line comment

---

◀ Prerequisite material for the course | ▲ Index | 1. Syntax Building Blocks ▶

Module 01

Java Language Basics

1. Syntax Building Blocks

Table of Contents

- 1.1 Value
- 1.2 Literal
- 1.3 Identifier
- 1.4 Variable
- 1.5 Type
- 1.6 Operator
- 1.7 Expression
- 1.8 Statement
- 1.9 Code Block
- 1.10 Function / Method
- 1.11 Class / Object
- 1.12 Module / Package
- 1.13 Program
- 1.14 System
- 1.15 Summary as a Growing Scale
- 1.16 Hierarchy Diagram ASCII
- 1.17 Hierarchy Diagram Mermaid

Every software system or computer program is composed of a set of **data** and a set of **operations** that are applied to this data in order to produce a result.

More formally:

> A computer program consists of a collection of data structures that represent the state of the system, together with algorithms that specify the operations to be performed on this state in order to produce outputs.

This document describes a **hierarchy of abstractions**: the *elementary building blocks* which, combined into increasingly complex structures, form software.

The sequence is presented in **increasing order of complexity**, with general definitions (computer science) and Java references.

1.1 Value

- **Definition:** An abstract entity representing information (number, character, boolean, string, etc.).
- **Theory:** A value belongs to a mathematical domain (set), such as $\mathbb{N}$ for natural numbers or Σ^* for strings.
- **Example (abstract):** the number forty-two, the truth value *true*, the character "a".

Java example (values):

```
// These are values:
42        // an int value
true      // a boolean value
'a'       // a char value
"Hello"   // a String value
```

1.2 Literal

- **Definition:** A **literal** is the concrete notation in source code that directly denotes a fixed value.
- **In Java:** `42`, `'a'`, `true`, `"Hello"`.
- **Theory:** A literal is *syntax*, while a value is its *semantics*.
- **Note:** Literals are the most common way to introduce values into programs.

Java example (literals):

```
int answer = 42;           // 42 is an int literal
char letter = 'a';         // 'a' is a char literal
boolean flag = true;       // true is a boolean literal
String msg = "Hello";      // "Hello" is a String literal
```

1.3 Identifier

- **Definition:** A symbolic name that associates a value (or a structure) with a readable label.
- **In Java:**
  - **User-defined identifiers:** chosen by the programmer to name variables, methods, classes, etc.
    Examples: `x`, `counter`, `MyClass`, `calculateSum`.
  - **Keywords (reserved words):** predefined names reserved by the Java language and cannot be redefined.
    Examples: `class`, `public`, `static`, `if`, `return`.

> **Note**
>
> Identifiers must follow Java naming rules: see Java Naming Rules.

- **Theory:** Binding function: connects a name to a value or resource.

Java example (identifiers):

```
int counter = 0;           // counter is an identifier (variable name)
String userName = "Bob";   // userName is an identifier
class MyService { }        // MyService is a class identifier
```

1.4 Variable

- **Definition:** A “memory cell” labeled by an identifier, which can hold and change value.
- **In Java:** `int counter = 0; counter = counter + 1;`.
- **Theory:** A mutable state that can vary over time during execution.

Java example (variable changing over time):

```
int counter = 0;        // variable initialized
counter = counter + 1;  // variable updated
counter++;              // another update (post-increment)
```

1.5 Type

- **Definition:** A type is a set of values and a set of operations permitted on those values.
- **In Java:**
  - **Primitive (simple) types:** directly represent basic values.
    Examples: `int`, `double`, `boolean`, `char`, `byte`, `short`, `long`, `float`.
  - **Reference types:** represent references (pointers) to objects in memory.
    Examples: `String`, arrays (e.g., `int[]`), classes, interfaces, and user-defined types.

> **Note**
>
> See [Java Data Types](#).

- **Theory:** A type system is the set of rules that associates sets of values and admissible operations.

Java example (types):

```
int age = 30;            // int type
double price = 9.99;     // double type
boolean active = true;   // boolean type
String name = "Alice";   // reference type (class String)
```

1.6 Operator

- **Definition:** A **symbol or keyword** that performs a computation or action on one or more operands.
- **Role:** Operators combine values, variables, and expressions to produce new values or to modify program state.
- **In Java:**

> **Note**
>
> See [Java Operators](#).

- **Theory:** Operators define allowable computations over types; together with values and variables, they form **expressions**.

Java example (operators in context):

```
int a = 5 + 3;           // + arithmetic
boolean ok = a > 3;      // > comparison
ok = ok && true;         // && logical
a += 2;                  // += assignment
int sign = (a >= 0) ? 1 : -1; // ?: ternary
```

1.7 Expression

- **Definition:** A combination of values, literals, variables, operators, and functions that produces a new value.
- **In Java:** `x + 3`, `Math.sqrt(25)`, `"Hello" + " world"`.
- **Theory:** A syntax tree that evaluates to a result.

Java example (expressions):

```
int x = 10;
int y = x + 3;                // x + 3 is an expression
double r = Math.sqrt(25);     // Math.sqrt(25) is an expression
String msg = "Hello" + " ";   // "Hello" + " " is an expression
msg = msg + "world";          // msg + "world" is another expression
```

1.8 Statement

- **Definition:** A unit of execution that modifies state or controls flow.
- **In Java:** `x = x + 1;`, `if (x > 0) { ... }`.
- **Theory:** A sequence of actions that does not return a value as a result of the statement itself, but changes the configuration of the abstract machine.

Java example (statements):

```java
int x = 0;                    // declaration statement
x = x + 1;                    // assignment statement

if (x > 0) {                  // if statement
    System.out.println("Positive");
}
```

1.9 Code Block

- **Definition:** A set of statements enclosed between delimiters forming an executable unit.
- **In Java:** `{ int y = 5; x = x + y; }`.
- **Theory:** A sequential composition of statements, with rules of *scope* (visibility).

Java example (code block and scope):

```java
int x = 10;

{
    int y = 5;          // y is only visible inside this block
    x = x + y;          // OK: x is visible here
}

// y is not visible here
// x is still visible here
```

1.10 Function / Method

- **Definition:** A sequence of encapsulated statements, identified by a name, which can receive inputs (parameters) and return an output (value).
- **In Java:**

```java
int square(int n) {
    return n * n;
}
```

- **Theory:** A mapping between input and output domains, with an operational body.

Java usage example:

```java
int result = square(5); // result = 25
```

1.11 Class / Object

- **Definition:**
- - **Class:** abstract description of a set of objects (state + behavior).
- - **Object:** a concrete instance of the class.
- **In Java:**

```java
class Point {
    int x, y;

    void move(int dx, int dy) {
        x += dx;
        y += dy;
    }
}

Point p = new Point();    // p is an object (instance of Point)
p.move(1, 2);             // method call on the object
```

- **Theory:** Abstraction of an *ADT* (Abstract Data Type).

1.12 Module / Package

- **Definition:** Logical grouping of classes, functions, and resources with a common purpose.
- **In Java:** `package java.util;` → collects utilities.
- **Theory:** Mechanism of organization and reuse, reducing complexity.

Java example (package):

```
package com.example.app;

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

1.13 Program

- **Definition:** A coherent set of modules, classes, and functions that, when executed on a machine, realizes a global behavior.
- **In Java:** The `main` method and everything it invokes.
- **Theory:** A specification of transformations from input to output on an abstract machine.

Java example (minimal program):

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, Java 21!");
    }
}
```

1.14 System

- **Definition:** A set of cooperating programs that interact with external resources (user, network, devices).
- **Example:** An enterprise Java platform with database, REST services, UI.
- **Theory:** Complex architecture of software and hardware components.

Example (conceptual):

- A Java backend (Spring Boot service)
- A database (PostgreSQL)
- A front-end web app
- External services (REST APIs, message queues)

Together they form a *system*.

1.15 📌 Summary as a Growing Scale

`Value → Literal → Identifier → Variable → Type → Operator → Expression → Statement → Code Block → Function/Method → Class/Object → Module/Package → Program → System`

This scale shows how small conceptual units are combined into larger and more complex structures.

1.16 📊 Hierarchy Diagram (ASCII)

Description: This ASCII diagram shows the hierarchical relation between building blocks, from the most complex (System) down to the simplest (Value and its concrete form, the Literal).

```
System
└── Program
    └── Module / Package
        └── Class / Object
            └── Function / Method
                └── Code Block
                    └── Statement
                        └── Expression
                            └── Operator
                                └── Type
                                    └── Variable
                                        └── Identifier
                                            └── Literal
                                                └── Value
```

1.17 📊 Hierarchy Diagram (Mermaid)

Description: The Mermaid diagram renders the same hierarchy in a top-down tree. It highlights that a Literal is the syntactic form of a Value.

```
graph TD
    A[System]
    A --> B[Program]
    B --> C[Module / Package]
    C --> D[Class / Object]
    D --> E[Function / Method]
    E --> F[Code Block]
    F --> G[Statement]
    G --> H[Expression]
    H --> H2[Operator]
    H2 --> I[Type]
    I --> J[Variable]
    J --> K[Identifier]
    K --> L[Literal]
    L --> M[Value]
```

◀ Eclipse main shortcuts | ▲ Index | 2. Basic Language Java Building Blocks ▶

2. Basic Language Java Building Blocks

Table of Contents

- 2.1 Class Definition
- 2.2 Comments
- 2.3 Access Modifiers
- 2.4 Packages
  - 2.4.1 Organization and Purpose
  - 2.4.2 Mapping with the File System and Declaration of a Package
  - 2.4.3 Belonging to the Same Package
  - 2.4.4 Importing from a Package
  - 2.4.5 Static Imports
    - 2.4.5.1 Precedence Rules
  - 2.4.6 Standard vs. User-Defined Packages
- 2.5 The main Method
  - 2.5.1 main Method Signature
- 2.6 Compiling and Running Your Code
  - 2.6.1 Compiling One File, Default Package (Single Directory)
  - 2.6.2 Multiple Files, Default Package (Single Directory)
  - 2.6.3 Code Inside Packages (standard src → out layout)
  - 2.6.4 Compiling to Another Directory (-d)
  - 2.6.5 Multiple Files Across Packages (Compile Whole Source Tree)
  - 2.6.6 Single-File Source Execution (Quick Runs Without javac)
  - 2.6.7 Passing Parameters to a Java Program

This chapter introduces the essential structural elements of a Java program: `classes`, `methods`, `comments`, `access modifiers`, `packages`, the `main` method, and the basic command-line tools (`javac` and `java`).

These are the minimum essential components required to write, compile, organize, and execute Java code using the JDK (Java Development Kit) — without the use of any IDE (Integrated Development Environment).

2.1 Class definition

A Java `class` is the fundamental building block of a Java program.

A `class` represents a **user-defined data type**, composed of a set of internal data (`fields`) and the operations that can act upon them (`methods`).

A `class` is a **blueprint**, while `objects` are **concrete instances** created at runtime.

A Java class is composed of two main elements, known as its **members**:

- **Fields** (or variables): they represent the data that define the state of this newly created type.
- **Methods** (or functions): they represent the operations that can be performed on this data.

Some members can be declared with the keyword **static**.

A static member belongs to the class itself, not to the objects created from it.

This means that:

- there is only one shared copy across all instances
- it can be used without creating an object of the class
- it is loaded into memory when the class is loaded by the JVM

Non-static members (called **instance members**) instead belong to individual objects, and each instance has its own copy.

Normally, each class is defined in its own "**.java**" file; for example, a class named Person will be defined in the corresponding file Person.java.

Any class that is independently defined in its own source file is called a **top-level class**.

Such a class can only be declared as `public` or `package-private` (i.e., with no access modifier).

A single file, however, may contain more than one class definition. **In this case, only one class can be declared public, and the file name must match that class**.

Nested class, which are classes declared inside another class, can declare any access modifier: `public`, `protected`, `private`, `default` (package-private).

- Example:

```
public class Person {

    // This is a comment: explains the code but is ignored by the compiler. See section below.

    // Field → defines data/state
    String personName;

    // Method → defines behavior (this one take a parameter, newName, in input but does not re
    void setPersonName(String newName) {
        personName = newName;
    }

    // Method → defines behavior  (this one does not take parameters in input but does return
    String getPersonName(){
        return personName;
    }
}
```

> **Note**
>
> In its simplest form, we could theoretically have a class with no methods and no fields. Although such a class would compile, it would hardly make much sense.

Token / Identifier	Category	Meaning	Optional?
public	Keyword / access modifier	determines which other classes can use or see that element	Mandatory (when absent is, by default, package-private)
class	Keyword	Declares a class type.	Mandatory
Person	Class name (identifier)	The name of the class.	Mandatory
personName	Field name (identifier)	Stores the name of the person.	Optional
String	Type / Keyword	Type of the field `personName` and of the parameter `newName`.	Mandatory
setPersonName, getPersonName	Method names (identifier)	name a behavior of the class.	Optional
newName	Parameter name (identifier)	input passed to the method `setPersonName`.	Mandatory (if the method needs a parameter)
return	Keyword	Exits a method and gives back a value.	Mandatory (in methods with a non-void return type)
void	Return Type / Keyword	Indicates the method does not return a value.	Mandatory (if the method does not return a value)

> **Note**
>
> Mandatory = required by Java syntax, Optional = not required by syntax; depends on design.

2.2 Comments

Comments are not executable code: they **explain** the code but are ignored by the compiler.

In Java there are 3 types of comments: - Single-line (`//`) - Multi-line (`/* ... */`) - Javadoc (`/** ... */`)

A **single-line comment** starts with 2 slashes: all the text after that, on the same line, is ignored by the compiler.

- Example:

```
// This is a single-line comment. It starts with 2 slashes and ends at the end of the line.
```

A **multiline comment** includes anything between the symbols /* and */.

- Example:

```
/*
* This is a multi-line comment.
* It can span multiple lines.
* All the text between its opening and closing symbols is ignored by the compiler.
*
*/
```

A **Javadoc comment** is similar to a **multiline comment**, except it starts with /**: all the text between its opening and closing symbols is processed by the Javadoc tool to generate API documentation.

```
/**
 * This is a Javadoc comment
 *
 * This class represents a Person.
 * It stores a name and provides methods
 * to set and retrieve it.
 *
 * <p>Javadoc comments can include HTML tags,
 * and special annotations like @param and @return.</p>
 */
```

> **Warning**
>
> In Java, every block comment must be properly closed with */.

- Example:

```
/* valid block comment */
```

is fine, but

```
/* */ */
```

will cause a compilation error because, while the first two symbols are part of the comment, the last symbol don't. The extra symbol `*/` is not valid syntax then and the compiler will complain.

2.3 Access modifiers

In Java, an **access modifier** is a keyword that specifies the visibility (or accessibility) of a **class**, **method**, or **field**. It determines which other classes can use or see that element.

> **Note**
>
> **Table of the access modifiers available in Java**

Token / Identifier	Category	Meaning	Optional?
public	Keyword / access modifier	Visible from any class in any package	Yes
no modifier (default)	Keyword / access modifier	Visible only within the same package	Yes
protected	Keyword / access modifier	Visible within the same package and by subclasses (even in other packages)	Yes
private	Keyword / access modifier	Visible only within the same class	Yes

> **Tip**
>
> **private > default > protected > public** Think "visibility grows outward".

2.4 Packages

Java packages are logical groupings of classes, interfaces, and sub-packages. They help organize large codebases, avoid name conflicts, and provide controlled access between different parts of an application.

2.4.1 Organization and Purpose

- Naming of packages follow the same rules of variable names. see: Java Naming Rules
- Packages are like **folders** for your Java source code.
- They let you group related classes together (e.g., all utility classes in `java.util`, all networking classes in `java.net`).
- By using packages, you can prevent **naming conflicts**: for example, you may have two classes named `Date`, but one is `java.util.Date` and another is `java.sql.Date`.

2.4.2 Mapping with the File System and declaration of a package

- Packages map directly to the **directory hierarchy** on your file system.
- You declare the package at the top of the source file (**before any imports**).
- If you do not declare a package, the class belongs to the default package.
  - This is not recommended for real projects, since it complicates organization and imports.
- Example:

```
package com.example.myapp.utils;

public class MyApp{

}
```

> **Important**
>
> This declaration means the class must be located in the directory: **com/example/myapp/utils/MyApp.java**

2.4.3 Belonging to the Same Package

Two classes belong to the same package if and only if:

- They are declared with the same package statement at the top of their source file.
- They are placed in the same directory of the source hierarchy.
- Example:

A class in package A.B.C; belongs to A.B.C only, not to A.B. Classes in A.B cannot directly access **package-private** members of classes in A.B.C, because they are different packages.

Classes in the same package:

- Can access each other's package-private members (i.e., members without an access modifier).
- Share the same namespace, so you don't need to import them to use them.

Example: Two files in the same package

```
// File: src/com/example/tools/Tool.java
package com.example.tools;

public class Tool {
    static void hello() { System.out.println("Hi!"); }
}
```

```java
// File: src/com/example/tools/Runner.java
package com.example.tools;

public class Runner {
    public static void main(String[] args) {
        Tool.hello();    // OK: same package, no import needed
    }
}
```

2.4.4 Importing from a Package

To use classes from another package, you need to import them:

- Example:

```java
import java.util.List;       // imports a specific class
import java.util.*;          // imports all classes in java.util

import java.nio.file.*.*     // ERROR! only one wildcard is allowed and it must be at the end!
```

> **Note**
>
> The wildcard character * imports all types in the package but not its subpackages.

You can always use the fully qualified name instead of importing all the classes in that package:

```java
java.util.List myList = new java.util.ArrayList<>();
```

> **Note**
>
> If you explicitely import a class name, it takes precedence over any wildcard import; if you want two use two class with the same name (ex. `Date` from java.util and from java.sql), it is better to use a fully qualified name import.

2.4.5 Static imports

In addition to importing classes from a package, Java allows another kind of import: the **static import**.

A *static import* lets you import **static members** of a class — such as static methods and static variables — so you can use them **without referencing the class name**.

You may import either **specific** static members or use a **wildcard** to import all static members of a class.

Example — Specific Static Import

```java
import static java.util.Arrays.asList;   // Imports Arrays.asList()

public class Example {

    List<String> arr = asList("first", "second");
    // We can call asList() directly, without using Arrays.asList()
}
```

Example — Static Import of a Constant

```java
import static java.lang.Math.PI;
import static java.lang.Math.sqrt;

public class Circle {
    double radius = 3;

    double area = PI * radius * radius;
    double diagonal = sqrt(2);
}
```

Example — Wildcard Static Import

```
import static java.lang.Math.*;

public class Calculator {
    double x = sqrt(49);    // 7.0
    double y = max(10, 20);
    double z = random();    // calls Math.random()
}
```

Wildcard static imports behave exactly like normal wildcard imports:
they bring **all static members** of the class into scope.

> **Warning**
>
> You can **always** call a static member with the class name: `Math.sqrt(16)` always works — even if imported statically.

2.4.5.1 Precedence Rules

If the current class already declares a method or variable with the same name as the statically imported one:

- The **local member takes precedence**.
- The imported static member is **shadowed**.

Example:

```
import static java.lang.Math.max;

public class Test {

    static int max(int a, int b) {    // local version
        return a > b ? a : b;
    }

    void run() {
        System.out.println(max(2, 5));
        // Calls the LOCAL max(), not Math.max()
    }
}
```

> **Warning**
>
> - A static import **must** follow the exact syntax: `import static`.
> - The compiler forbids importing **two static members with the same simple name** if it creates ambiguity — even if they come from different classes or packages.

Example — **Not allowed**:

```
import static java.util.Collections.emptyList;
import static java.util.List.of;

// ❌ ERROR: both methods have the same name `of()`
import static java.util.Set.of;
```

The compiler does not know which `of()` you intend to call → compilation fails.

Tip

- If two static imports introduce the same name, **any attempt to use that name causes a compile error**.
- Static imports do **not** import classes, only static members.
- You can still call the static member using the class name even if statically imported.

Example:

```
import static java.lang.Math.sqrt;

double a = sqrt(16);         // imported
double b = Math.sqrt(25);    // fully qualified — always allowed
```

2.4.6 Standard vs. User-Defined Packages

Standard packages: shipped with the JDK (e.g., java.lang, java.util, java.io).

User-defined packages: created by developers to organize application code.

2.5 The `main` Method

In Java, the `main` method serves as the **entry point** of a standalone application. Its correct declaration is critical for the JVM to recognize it:

2.5.1 `main` method signature

Let's review the `main` method signature inside two of the simplest possible classes:

- Example: without optional modifiers

```
public class MainFirstExample {

    public static void main(String[] args){

        System.out.print("Hello World!!");

    }

}
```

- Example: with both, optional, `final` modifiers

```
public class MainSecondExample {

    public final static void main(final String options[]){

        System.out.print("Hello World!!");

    }

}
```

Note

Table of the access modifiers for the main method

Token / Identifier	Category	Meaning	Optional?
public	Keyword / Access Modifier	Makes the method accessible from anywhere. Required so the JVM can call it from outside the class.	Mandatory
static	Keyword	Means the method belongs to the class itself and can be called without creating an object. Required because the JVM has no object instance when starting the program.	Mandatory
final (before return type)	Modifier	Prevents the method from being overridden. It can legally appear before the return type, but it has no effect on `main` and is not required.	Optional
main	Method name (predefined)	The exact name that the JVM looks for as the entry point of the program. Must be spelled exactly as `main` (lowercase).	Mandatory
void	Return Type / Keyword	Declares that the method does not return any value to the JVM.	Mandatory
String[] args	Parameter list	An array of `String` values that holds the command-line arguments passed to the program. May also be written as `String args[]` or `String... args`. The parameter name (`args`) is arbitrary.	Mandatory (the parameter type is required, but the name can vary)
final (in parameter)	Modifier	Marks the parameter as unchangeable inside the method body (you cannot reassign `args` to another array).	Optional

Important

Modifiers `public`, `static` (mandatory) and `final` (if present) can be swapped in order; `public` and `static` can't be omitted.

Java treats `String[] args` and `String... args` the same.
Both compile and run correctly as entry points.

2.6 Compiling and running your code

This chapter shows **correct, working** `javac` and `java` command lines for common situations in Java 21: single files, multiple files, packages, separate output directories, and classpath/module-path usage.

Follow the directory layouts exactly.

> check your tools

```
javac -version   # should print: javac 21.x
java  -version   # should print: java version "21.0.7" ... (the output could be different depe
```

> **Warning**
>
> When running a class inside a package, **java requires the fully qualified name**, NEVER the path:
>
> `java com.example.app.Main` ✓
>
> `java src/com/example/app/Main` ❌

2.6.1 Compiling one file, default package (single directory)

Files

```
.
└── Hello.java
```

Hello.java

```
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, Java 21!");
    }
}
```

Compile (in the same directory)

```
javac Hello.java
```

This command will create, in the same directory, a file with the same name of your ".java" file but with ".class" filename extension ; this is the bytecode file which will be interpreted and compiled by the jvm.

Once you have the .class file, in this case Hello.class, you can run the program with:

Run

```
java Hello
```

> **Important**
>
> You don't have to specify the ".class" extension when executing the program

2.6.2 Multiple files, default package (single directory)

Files

```
.
├── A.java
└── B.java
```

Compile everything

```
javac *.java
```

Or, if the classes belong to a specific package:

```
javac packagex/*.java
```

Or, specifying each of them

```
javac A.java B.java
```

and

```
javac packagex/A.java packagey/B.java
```

Run the entry point: The class which has a `main` method

```
java A    # if A has main(...)
# or
java B
```

> **Important**
>
> The path to your classes is, in Java, the **classpath**. You can specify the **classpath** with one of the following options:

- **-cp** `<classpath>`
- **-classpath** `<classpath>`
- **–class-path** `<classpath>`

2.6.3 Code inside packages (standard src → out layout)

Files

```
.
├── src/
│   └── com/
│       └── example/
│           └── app/
│               └── Main.java
└── out/
```

> **Note**
>
> The `src` and `out` folders are not part of our packages, being only the directory containing all our source files and the compiled .class files;

Main.java

```
package com.example.app;

public class Main {
    public static void main(String[] args) {
        System.out.println("Packages done right.");
    }
}
```

Compile to the same directory

```
# Creates .class file just beside the source file
javac src/com/example/app/Main.java
```

2.6.4 Compiling to another directory (`-d`)

`-d out` places compiled `.class` files into the `out/` directory, creating package subfolders that mirror your `package` names:

```
javac -d out -sourcepath src src/com/example/app/Main.java
```

Run (use the classpath to point at out/)

```
# Unix/macOS
java -cp out com.example.app.Main

# Windows
java -cp out com.example.app.Main
```

2.6.5 Multiple files across packages (compile whole source tree)

Files

```
.
├── src/
│   └── com/
│       └── example/
│           ├── util/
│           │   └── Utils.java
│           └── app/
│               └── Main.java
│
└── out/
```

Compile entire source tree to `out/`

```
# Option A: point javac at the top package(s)
javac -d out   src/com/example/util/Utils.java   src/com/example/app/Main.java

# Option B: use -sourcepath to let javac find dependencies
javac -d out -sourcepath src   src/com/example/app/Main.java
```

> **Important**
>
> **-sourcepath** `<sourcepath>` tells `javac` where to look for other `.java` files that a given source depends on.

2.6.6 Single-file source execution (quick runs without `javac`)

Java 21 (since Java 11) lets you run small programs directly from source:

```
# Default package only
java Hello.java
```

Multiple source files are allowed if they're in the **default package** and in the **same directory**:

```
java Main.java Helper.java
```

> If you use **packages**, prefer compiling to `out/` and running with `-cp`.

2.6.7 Passing Parameters to a Java program

You can send data to your Java program through the parameters of the `main` entry point.

As we learned before, the `main` method can receive an array of strings in the form: **String[] args**. See the section on main.

Main.java printing out two parameters received in input by the "main" method:

```java
package com.example.app;

public class Main {
    public static void main(String[] args) {
        System.out.println(args[0]);
        System.out.println(args[1]);
    }
}
```

To pass parameters, just type (for example):

```
java Main.java Hello World  #spaces are used to separate the two arguments
```

If you want to pass an argument containing spaces, just use quotes:

```
java Main.java Hello "World Mario" #space are used to separate the two arguments
```

> If you declare to use (in this case print) the first two element of the parameter's array (as in our previous example) but, in fact, you pass less arguments, the jvm will notify you of a problem through a `java.lang.ArrayIndexOutOfBoundsException`.

> If, on the other hand, you pass more arguments than the method expects, it will print out just the two (in this case) expected.

> `args.length` tells you how many arguments were provided.

◀ 1. Syntax Building Blocks | ▲ Index | 3. Java Naming Rules ▶

3. Java Naming Rules

Table of Contents

- 3.1 Rules for Identifiers
  - 3.1.1 Reserved Words
    - 3.1.1.1 Java Reserved Keywords
    - 3.1.1.2 Reserved Literals
  - 3.1.2 Case Sensitivity
  - 3.1.3 Beginning of Identifiers
  - 3.1.4 Numbers in Identifiers
  - 3.1.5 Single `underscore` Token
  - 3.1.6 Numeric Literals & Underscore Character

Java defines precise rules for **identifiers**, which are the names given to variables, methods, classes, interfaces, and packages.

As long as you follow the naming rules described below, you are free to choose meaningful names for your program elements.

3.1 Rules for Identifiers

3.1.1 Reserved Words

`Identifiers` **cannot** be the same as Java's **keywords** or **reserved literals**.

`Keywords` are predefined, special words in the Java language which you are not allowed to use (see Table below).

`Literals` such as `true`, `false`, and `null` are also reserved and cannot be used as identifiers.

- Example:

```
int class = 5;         // invalid: 'class' is a keyword
boolean true = false; // invalid: 'true' is a literal
int year = 2024;     // valid
```

3.1.1.1 Java Reserved Keywords

a -> c	**c -> f**	**f -> n**	**n -> s**	**s -> w**
abstract	continue	for	new	switch
assert	default	goto*	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const*	float	native	super	while

> **Note**
>
> `goto` and `const` are reserved but not used.

3.1.1.2 Reserved Literals

- `true`
- `false`
- `null`

3.1.2 Case Sensitivity

Identifiers in Java are **case sensitive**.
This means `myVar`, `MyVar`, and `MYVAR` are all different identifiers.

- Example:

```
int myVar = 1;
int MyVar = 2;
int MYVAR = 3;
int CLASS = 6; // legal but, please, don't do it!!
```

> **Tip**
>
> Java treats identifiers literally: Count, count, and COUNt are unrelated and may exist together.
>
> Because of case sensitivity, you could use versions of keywords which differ in case. While legal, such naming is discouraged because it reduces readability and it is considered a very bad practice.

3.1.3 Beginning of Identifiers

Identifiers in Java must begin with a letter, a currency symbol ($, €, £, ₹...) or a _ symbol.

Example:

```
int myVarA;
int $myVarB;
int _myVarC;
String €uro = "currency"; // legal (rarely seen in practice)
```

> **Note**
>
> Currency symbols are legal but not recommended in real-world code.

3.1.4 Numbers in Identifiers

Identifiers in Java can include numbers but they cannot start with them.

Example:

```
int my33VarA;
int $myVar44;
int 3myVarC; // invalid: identifier cannot start with a digit
int var2024 = 10; // valid
```

3.1.5 Single `underscore` token

- A single underscore (`_`) is not allowed as an identifier.
- Since Java 9, `_` is a reserved token for future language use.
- Example:

```
int _;  // invalid since Java 9
```

> **Warning**
>
> `_` is legal inside number literals (see next section), but not as a standalone identifier.

3.1.6 Numeric literals & Underscore character

You can have one or more `_` (underscore) character in number literals in order to make them easier to read.

You can have underscores anywhere except at the beginning, at the end or right around (before/after) a decimal point.

- Example:

```
int firstNum = 1_000_000;
int secondNum = 1 _____________ 2;

double firstDouble = _1000.00   // DOES NOT COMPILE
double secondDouble = 1000_.00  // DOES NOT COMPILE
double thirdDouble = 1000._00   // DOES NOT COMPILE
double fourthDouble = 1000.00_  // DOES NOT COMPILE

double pi = 3.14_159_265; // valid
long mask = 0b1111_0000;  // valid in binary literals
```

> **Tip**
>
> Underscores improve readability: 1_000_000 is easier than 1000000.

◀ 2. Basic Language Java Building Blocks | ▲ Index | 4. Java Data Types and Casting ▶

4. Java Data Types and Casting

Table of Contents

- 4.1 Primitive Types
- 4.2 Reference Types
- 4.3 Primitive Types Table
- 4.4 Notes
- 4.5 Recap
- 4.6 Arithmetic and Primitive Numeric Promotion
  - 4.6.1 Numeric Promotion Rules in Java
    - 4.6.1.1 Rule 1 – Mixed Data Types → Smaller type promoted to larger type
    - 4.6.1.2 Rule 2 – Integral + Floating-point → Integral promoted to floating-point
    - 4.6.1.3 Rule 3 – byte short and char are promoted to int during arithmetic
    - 4.6.1.4 Rule 4 – Result type matches the promoted operand type
  - 4.6.2 Summary of Numeric Promotion Behavior
    - 4.6.2.1 Key Takeaways
- 4.7 Casting in Java
  - 4.7.1 Primitive Casting
    - 4.7.1.1 Widening Implicit Casting
    - 4.7.1.2 Narrowing Explicit Casting
    - 4.7.1.3 Compile-Time Implicit Narrowing
  - 4.7.2 Data Loss Overflow and Underflow
  - 4.7.3 Casting Values versus Variables
  - 4.7.4 Reference Casting Objects
    - 4.7.4.1 Upcasting Widening Reference Cast
    - 4.7.4.2 Downcasting Narrowing Reference Cast
  - 4.7.5 Key Points Summary
  - 4.7.6 Examples
- 4.8 Summary

As we saw before in the Syntax Building Blocks, Java has two categories of data types:

- **Primitive types**
- **Reference types**

👉 For a complete overview of primitive types with their sizes, ranges, defaults, and examples, see the Primitive Types Table.

4.1 Primitive Types

`Primitives` represent **single raw values** stored directly in memory.
Each primitive type has a fixed size that determines how many bytes it occupies.

Conceptually, a primitive is just a **cell in memory** holding a value:

```
+--------+
|   42   |   ← value of type short (2 bytes in memory)
+--------+
```

4.2 Reference Types

A `reference` type does not hold the `object` itself, but a **reference (pointer)** to it.
The reference has a fixed size (JVM-dependent, often 4 or 8 bytes), which points to a memory location where the actual object is stored.

- Example: a `String` reference variable points to a string object in the heap, which internally is composed of an array of `char` primitives.

Diagram:

```
Reference (4 or 8 bytes)
+---------+
| address | ───────────────▶  Object in Heap
+---------+                  +-------------------+
                             |   "Hello"         |
                             | ['H','e','l','l','o']  ← array of char
                             +-------------------+
```

4.3 Primitive Types Table

Keyword	Type	Size	Min Value	Max Value	Default Value	Example
`byte`	8-bit int	1 byte	−128	127	0	`byte b = 100;`
`short`	16-bit int	2 bytes	−32,768	32,767	0	`short s = 2000;`
`int`	32-bit int	4 bytes	−2,147,483,648 (`-2^31`)	2,147,483,647 (`2^31-1`)	0	`int i = 123456;`
`long`	64-bit int	8 bytes	−2^63	2^63−1	0L	`long l = 123456789L;`
`float`	32-bit FP	4 bytes	see note	see note	0.0f	`float f = 3.14f;`
`double`	64-bit FP	8 bytes	see note	see note	0.0	`double d = 2.718;`
`char`	UTF-16	2 bytes	`'\u0000'` (0)	`'\uffff'` (65,535)	`'\u0000'`	`char c = 'A';`
`boolean`	true/false	JVM-dep. (often 1 byte)	`false`	`true`	`false`	`boolean b = true;`

4.4 Notes

`float` and `double` do not have fixed integer bounds like integral types.
Instead, they follow the IEEE 754 standard:

- **Smallest positive nonzero values**:
  - `Float.MIN_VALUE ≈ 1.4E-45`
  - `Double.MIN_VALUE ≈ 4.9E-324`
- **Largest finite values**:

- `Float.MAX_VALUE ≈ 3.4028235E+38`
- `Double.MAX_VALUE ≈ 1.7976931348623157E+308`

They also support special values: **`+Infinity`**, **`-Infinity`**, and **`NaN`** (Not a Number).

- **FP** = floating point.
- `boolean` size is JVM-dependent but behaves logically as `true` / `false`.
- Default values apply to **fields** (class variables). **Local variables** must be explicitly initialized before use.

4.5 Recap

- **Primitive** = actual value, stored directly in memory.
- **Reference** = pointer to an object; the object itself may contain primitives and other references.
- For details of primitives, see the Primitive Types Table.

4.6 Arithmetic and Primitive Numeric Promotion

When applying arithmetic or comparison operators to **primitive data types**, Java automatically converts (or *promotes*) values to compatible types according to well-defined **numeric promotion rules**.

These rules ensure consistent calculations and prevent data loss when mixing different numeric types.

4.6.1 🔹 Numeric Promotion Rules in Java

4.6.1.1 Rule 1 – Mixed Data Types → Smaller type promoted to larger type

If two operands belong to **different numeric data types**, Java automatically promotes the **smaller** type to the **larger** type before performing the operation.

Example	Explanation
`int x = 10; double y = 5.5;` `double result = x + y;`	The `int` value `x` is promoted to `double`, so the result is a `double` (`15.5`).

Valid type promotion order (smallest → largest):
`byte → short → int → long → float → double`

4.6.1.2 Rule 2 – Integral + Floating-point → Integral promoted to floating-point

If one operand is an **integral type** (`byte`, `short`, `char`, `int`, `long`) and the other is a **floating-point type** (`float`, `double`),
the **integral value is promoted** to the **floating-point** type before the operation.

Example	Explanation
`float f = 2.5F; int n = 3;` `float result = f * n;`	`n` (int) is promoted to `float`. The result is a `float` (`7.5`).
`double d = 10.0; long l = 3;` `double result = d / l;`	`l` (long) is promoted to `double`. The result is a `double` (`3.333...`).

4.6.1.3 Rule 3 – `byte`, `short`, and `char` are promoted to `int` during arithmetic

When performing arithmetic **with variables** (not literal constants) of type `byte`, `short`, or `char`, Java automatically promotes them to `int`, even if **both operands are smaller than `int`**.

Example	Explanation
`byte a = 10, b = 20;` `byte c = a + b;`	❌ Compile-time error: result of `a + b` is `int`, not `byte`. Must cast → `byte c = (byte)(a + b);`
`short s1 = 1000, s2 = 2000;` `short sum = (short)(s1 + s2);`	The operands are promoted to `int`, so explicit casting is required to assign to `short`.
`char c1 = 'A', c2 = 2;` `int result = c1 + c2;`	`'A'` (65) and `2` are promoted to `int`, result = `67`.

Note

This rule applies only when **using variables**. When **using constant literals**, the compiler can sometimes evaluate the expression at compile time and assign it safely.

```
byte a = 10 + 20;   // ✅ OK: constant expression fits in byte
byte b = 10;
byte c = 20;
byte d = b + c;     // ❌ Error: b + c is evaluated at runtime → int
```

4.6.1.4 Rule 4 – Result type matches the promoted operand type

After promotions are applied, and both operands are of the same type,
the **result** of the expression has that **same promoted type**.

Example	Explanation
`int i = 5; double d = 6.0;` `var result = i * d;`	`i` is promoted to `double`, result is `double`.
`float f = 3.5F; long l = 4L;` `var result = f + l;`	`l` is promoted to `float`, result is `float`.
`int x = 10, y = 4;` `var div = x / y;`	Both are `int`, result = `int` (`2`), fractional part truncated.

Warning

Integer division always produces an **integer result**. To obtain a decimal result, **at least one operand must be floating-point**:

```
double result = 10.0 / 4; // ✅ 2.5
int result = 10 / 4;      // ❌ 2 (fraction discarded)
```

4.6.2 ✅ Summary of Numeric Promotion Behavior

Situation	Promotion Result	Example
Mixing smaller and larger numeric types	Smaller type promoted to larger	`int + double → double`
Integral + Floating-point	Integral promoted to floating-point	`long + float → float`
`byte`, `short`, `char` arithmetic	Promoted to `int`	`byte + byte → int`
Result after promotion	Result matches promoted type	`float + long → float`

4.6.2.1 🧠 Key Takeaways

- Always consider **type promotion** when mixing data types in arithmetic.

- For smaller types (`byte` , `short` , `char`), promotion to `int` is automatic when operands of an arithmetic operation containing variables.
- Use **explicit casting** only when you are sure the result fits the target type.
- Remember: **integer division truncates**, **floating-point division keeps decimals**.
- Understanding promotion rules is crucial for avoiding **unexpected precision loss** or **compile-time errors**.

4.7 Casting in Java

`Casting` in Java is the process of explicitly converting a value from one type to another. It applies both to `primitive types` (numbers) and to `reference types` (objects in a class hierarchy).

4.7.1 Primitive Casting

Primitive casting changes the type of a numeric value.

There are two categories of casting:

Type	Description	Example	Explicit?	Risk
Widening	smaller type → larger type	int → double	No	no loss
Narrowing	larger type → smaller type	double → int	Yes	possible loss

4.7.1.1 Widening Implicit Casting

Automatic conversion from a "smaller" type to a compatible "larger" type.
Handled by the compiler, **does not require explicit syntax**.

```
int i = 100;
double d = i;  // implicit cast: int → double
System.out.println(d); // 100.0
```

✅ **Safe** – no overflow (though still be aware of floating-point precision).

4.7.1.2 Narrowing Explicit Casting

Manual conversion from a "larger" type to a "smaller" one.
Requires a **cast expression** because it may cause data loss.

```
double d = 9.78;
int i = (int) d;  // explicit cast: double → int
System.out.println(i); // 9 (fraction discarded)
```

> **Warning**
>
> ⚠ Use only when you are sure the value fits in the target type.

4.7.1.3 Compile-Time Implicit Narrowing

In some specific cases, the compiler allows a narrowing conversion **without an explicit cast**.

If a variable is declared `final` and initialized with a constant expression whose value fits into the target type, the compiler can safely perform the conversion at compile time.

```
final int k = 11;
byte b = k;  // allowed: value 11 fits into byte range

final int x = 200;
byte c = x;  // does NOT compile: 200 is outside byte range
```

This works because the compiler knows the exact value of a `final` variable and can verify that it is within the range of the smaller type.

This kind of narrowing is allowed between: - `byte` - `short` - `char` - `int`

However, it does **not** apply to: - `long` - `float` - `double`

For example:

```
final float f = 10.0f;
int n = f;    // does not compile
```

Even though the value seems compatible, floating-point types are not eligible for this form of implicit narrowing.

4.7.2 Data Loss, Overflow and Underflow

When a value exceeds a type's capacity, you may get:

- **Overflow**: result greater than the maximum representable value
- **Underflow**: result lower than the minimum representable value
- **Truncation**: data that does not fit is lost (e.g., decimals)
- Example – overflow/underflow with int

```
int max = Integer.MAX_VALUE;
int overflow = max + 1;      // "wrap-around" to negative

int min = Integer.MIN_VALUE;
int underflow = min - 1;     // "wrap-around" to positive
```

- Example: truncation

```
double d = 9.99;
int i = (int) d; // 9 (fraction discarded)
```

> **Note**
>
> Floating-point types (`float`, `double`) **do not wrap**: - overflow → `Infinity` / `-Infinity` - underflow (very small values) → 0.0 or denormalized values.

4.7.3 Casting Values versus Variables

Java treats:

- Integer **literals** as `int` by default
- Floating-point **literals** as `double` by default

The compiler **does not require a cast** when a literal fits within the target type range:

```
byte first = 10;         // OK: 10 fits in byte
short second = 9 * 10;   // OK: constant expression evaluated at compile time
```

But:

```
long a = 5729685479;     // ❌ error: int literal out of range
long b = 5729685479L;    // ✅ long literal (L suffix)

float c = 3.14;          // ❌ double → float: requires F or cast
float d = 3.14F;         // ✅ float literal

int e = 0x7FFF_FFFF;     // ✅ max int in hex
int f = 0x8000_0000;     // ❌ out of int range (needs L)
```

However, when numeric promotion rules apply:

> With variables of type `byte`, `short`, and `char` in an arithmetic expression, operands are promoted to `int` and the result is `int`.

```
byte first = 10;
short second = 9 + first;        // ❌ 9 (int literal) + first (byte → int) = int
// second = (short) (9 + first);  // ✅ cast entire expression
```

```
short b = 10;
short a = 5 + b;              // ❌ 5 (int) + b (short → int) = int
short a2 = (short) (5 + b);   // ✅ cast entire expression
```

> **Warning**
>
> Cast is a **unary operator**:
>
> `short a = (short) 5 + b;` The cast applies only to `5` → the expression result remains int → assignment still fails.

4.7.4 Reference Casting Objects

Casting also applies to **object references** in a class hierarchy.
It does not change the object in memory — only **the reference type** used to access it.

Key rules:

- The **real object type** determines which fields/methods actually exist.
- The **reference type** determines what you may access at that point in code.

4.7.4.1 Upcasting (Widening Reference Cast)

Conversion from **subclass** to **superclass**.

Implicit and **always safe**: every `Dog` is also an `Animal`.

```
class Animal { }
class Dog extends Animal { }

Dog dog = new Dog();
Animal a = dog;     // implicit upcast: Dog → Animal
```

4.7.4.2 Downcasting (Narrowing Reference Cast)

Conversion from **superclass** to **subclass**.

- **Explicit**
- Can fail at runtime with `ClassCastException` if not truly that type

```
Animal a = new Dog();
Dog d = (Dog) a;    // OK: a really points to a Dog

Animal x = new Animal();
Dog d2 = (Dog) x;   // ⚠ Runtime error: ClassCastException
```

For safety, use `instanceof`:

```
if (x instanceof Dog) {
    Dog safeDog = (Dog) x;   // safe cast
}
```

4.7.5 Key Points Summary

Casting Type	Applies To	Direction	Syntax	Safe?	Performed By
Widening Primitive	Primitives	small → large	Implicit	Yes	Compiler
Narrowing Primitive	Primitives	large → small	Explicit	No	Programmer
Upcasting	Objects	subclass → superclass	Implicit	Yes	Compiler
Downcasting	Objects	superclass → subclass	Explicit	Runtime check	Programmer

4.7.6 Examples

```
// Primitive casting
short s = 50;
int i = s;          // widening
byte b = (byte) i;  // narrowing (possible loss)

// Object casting
Object obj = "Hello";
String str = (String) obj; // OK: obj points to a String

Object n = Integer.valueOf(10);
// String fail = (String) n;  // ClassCastException at runtime
```

4.8 Summary:

- **Primitive casting** changes the numeric type.
- **Reference casting** changes the “view” of an object in the hierarchy.
- **Upcasting** → safe and implicit.
- **Downcasting** → explicit, to be used carefully (often after `instanceof`).

◀ 3. Java Naming Rules | ▲ Index | 5. Java Operators ▶

5. Java Operators

Table of Contents

- 5.1 Definition
- 5.2 Types of Operators
- 5.3 Categories of Operators
- 5.4 Operator Precedence and Order of Evaluation
- 5.5 Summary Table of Java Operators
  - 5.5.1 Additional Notes
- 5.6 Unary Operators
  - 5.6.1 Categories of Unary Operators
  - 5.6.2 Examples
- 5.7 Binary Operators
  - 5.7.1 Categories of Binary Operators
  - 5.7.2 Division and Modulus Operators
    - 5.7.2.1 Division Operator
    - 5.7.2.2 Modulus Operator
  - 5.7.3 The Return Value of an Assignment Operator
  - 5.7.4 Compound Assignment Operators
  - 5.7.5 Equality Operators == and !=
    - 5.7.5.1 Equality with Primitive Types
    - 5.7.5.2 Equality with Reference Types Objects
  - 5.7.6 The instanceof Operator
    - 5.7.6.1 Compile-Time Check vs Runtime Check
    - 5.7.6.2 Pattern Matching for instanceof
    - 5.7.6.3 Flow Scoping & Short-Circuit Logic
    - 5.7.6.4 Arrays and Reifiable Types
- 5.8 Ternary Operator
  - 5.8.1 Type Rules for the Ternary Operator
    - 5.8.1.1 Numeric Operands
    - 5.8.1.2 Reference Types
  - 5.8.2 Syntax
  - 5.8.3 Example
  - 5.8.4 Nested Ternary Example
  - 5.8.5 Notes

5.1 Definition

In Java, **operators** are special symbols that perform operations on variables and values.
They are the building blocks of expressions and allow developers to manipulate data, compare values, perform arithmetic, and control logic flow.

An **expression** is a combination of operators and operands that produces a result.

For example:

```
int result = (a + b) * c;
```

Here, `+` and `*` are operators, and `a`, `b`, and `c` are operands.

5.2 Types of Operators

Java defines three types of operators, grouped by the number of operands they use:

Type	Description	Examples
Unary	Operate on a single operand	`+x`, `-x`, `++x`, `--x`, `!flag`, `~num`
Binary	Operate on two operands	`a + b`, `a - b`, `x * y`, `x / y`, `x % y`
Ternary	Operate on three operands (only one in Java)	`condition ? valueIfTrue : valueIfFalse`

5.3 Categories of Operators

Operators can also be grouped, by their purpose, into categories:

Category	Description	Examples
Assignment	Assign values to variables	`=`, `+=`, `-=`, `*=`, `/=`, `%=`
Relational	Compare values	`==`, `!=`, `<`, `>`, `<=`, `>=`
Logical	Combine or invert boolean expressions	`|`, `&`, `^`
Conditional	Combine or invert boolean expressions	`||`, `&&`
Bitwise	Manipulate bits	`&`, `|`, `^`, `~`, `<<`, `>>`, `>>>`
Instanceof	Test object type	`obj instanceof ClassName`
Lambda	Used in lambda expressions	`(x, y) -> x + y`

5.4 Operator Precedence and Order of Evaluation

Operator precedence determines how operators are grouped in an expression — that is, which operations are performed first.
Associativity (or **order of evaluation**) determines whether the expression is evaluated from **left to right** or **right to left** when operators have the same precedence.

Example:

```
int result = 10 + 5 * 2;  // Multiplication happens before addition → result = 20
```

Parentheses `()` can be used to **override precedence**:

```
int result = (10 + 5) * 2;  // Parentheses evaluated first → result = 30
```

> **Note**
>
> - Operator **precedence** is about *grouping*, not evaluation order.
> - Use parentheses for precedence and clarity in complex expressions.

5.5 Summary Table of Java Operators

Precedence (High → Low)	Type	Operators	Example	Evaluation Order	Applicable To
1	**Postfix Unary**	`expr++`, `expr--`	`x++`	Left → Right	Numeric types
2	**Prefix Unary**	`++expr`, `--expr`	`--x`	Left → Right	Numeric
3	**Other Unary**	`(type)`, `+`, `-`, `~`, `!`	`-x`, `!flag`	Right → Left	Numeric, boolean
4	**Cast Unary**	`(Type) reference`	`(short) 22`	Right → Left	reference, primitive
5	**Multiplication/division/modulus**	`*`, `/`, `%`	`a * b`	Left → Right	Numeric types
6	**Additive**	`+`, `-`	`a + b`	Left → Right	Numeric, String (concatenation)
7	**Shift**	`<<`, `>>`, `>>>`	`a << 2`	Left → Right	Integral types
8	**Relational**	`<`, `>`, `<=`, `>=`, `instanceof`	`a < b`, `obj instanceof Person`	Left → Right	Numeric, reference
9	**Equality**	`==`, `!=`	`a == b`	Left → Right	All types (except boolean for `<`, `>`)
10	**Logical AND**	`&`	`a & b`	Left → Right	boolean
11	**Logical exclusive OR**	`^`	`a ^ b`	Left → Right	boolean
12	**Logical inclusive OR**	`|`	`a | b`	Left → Right	boolean
13	**Conditional AND**	`&&`	`a && b`	Left → Right	boolean
14	**Conditional OR**	`||`	`a || b`	Left → Right	boolean
15	**Ternary (Conditional)**	`? :`	`a > b ? x : y`	Right → Left	All types
16	**Assignment**	`=`, `+=`, `-=`, `*=`, `/=`, `%=`	`x += 5`	Right → Left	All assignable types
17	**Arrow operator**	`->`	`(x, y) -> x + y`	Right → Left	Lambda expressions, switch rules

5.5.1 Additional Notes

- **String concatenation (`+`)** has lower precedence than arithmetic `+` with numbers.
- Use parentheses `()` for precedence and readability — they don't change semantics but make intent explicit.

5.6 Unary Operators

Unary operators operate on **a single operand** to produce a new value.
They are used for operations like incrementing/decrementing, negating a value, inverting a boolean, or performing bitwise complement.

5.6.1 Categories of Unary Operators

Operator	Name	Description	Example	Result
`+`	Unary plus	Indicates a positive value (usually redundant).	`+x`	Same as `x`
`-`	Unary minus	Indicates a literal number is negative or negates an expression.	`-5`	`-5`
`++`	Increment	Increases a variable by 1. Can be prefix or postfix.	`++x`, `x++`	`x+1`
`--`	Decrement	Decreases a variable by 1. Can be prefix or postfix.	`--x`, `x--`	`x-1`
`!`	Logical complement	Inverts a boolean value.	`!true`	`false`
`~`	Bitwise complement	Inverts each bit of an integer. Quick rule: ~n = -(n + 1)	`~5`	`-6`
`(type)`	Cast	Converts value to another type.	`(int) 3.9`	`3`

5.6.2 Examples

```
int x = 5;
System.out.println(++x);  // 6  (prefix: increments x to 6, then returns 6)
System.out.println(x++);  // 6  (postfix: returns 6, then increments x to 7)
System.out.println(x);    // 7

boolean flag = false;
System.out.println(!flag);  // true

int a = 5;                  // binary: 0000 0101
System.out.println(~a);     // -6 → binary: 1111 1010 (two's complement)
```

> **Note**
>
> - Prefix (`++x` / `--x`): updates the value first, then returns the new value.
> - Postfix (`x++` / `x--`): returns the current value first, then updates it.
> - The `!` operator applies to boolean values; `~` applies to integral numeric types.

5.7 Binary Operators

Binary operators require **two operands**.
They perform arithmetic, relational, logical, bitwise, and assignment operations.

5.7.1 Categories of Binary Operators

<table>
<tr><th>Category</th><th>Operators</th><th>Example</th><th>Description</th></tr>
<tr><td>Arithmetic</td><td><code>+</code>, <code>-</code>, <code>*</code>, <code>/</code>, <code>%</code></td><td><code>a + b</code></td><td>Basic math operations.</td></tr>
<tr><td>Relational</td><td><code><</code>, <code>></code>, <code><=</code>, <code>>=</code>, <code>==</code>, <code>!=</code></td><td><code>a < b</code></td><td>Compare values.</td></tr>
<tr><td>Logical (boolean)</td><td><code>&</code>, <code>|</code>, <code>^</code></td><td><code>a & b</code></td><td>See note below</td></tr>
<tr><td>Conditional</td><td><code>&&</code>, <code>||</code></td><td><code>a && b</code></td><td>See note below</td></tr>
<tr><td>Bitwise (integral)</td><td><code>&</code>, <code>|</code>, <code>^</code>, <code><<</code>, <code>>></code>, <code>>>></code></td><td><code>a << 2</code></td><td>Operate on bits.</td></tr>
<tr><td>Assignment</td><td><code>=</code>, <code>+=</code>, <code>-=</code>, <code>*=</code>, <code>/=</code>, <code>%=</code></td><td><code>x += 3</code></td><td>Modify and assign.</td></tr>
<tr><td>String Concatenation</td><td><code>+</code></td><td><code>"Hello " + name</code></td><td>Joins strings together.</td></tr>
</table>

Important

- **Logical operators** (`&` , `|` , `^`) *always evaluate both sides*.
- **Conditional operators** (`&&` , `||`) are **short-circuiting**:
  - `a && b` → `b` evaluated only if `a` is true
  - `a || b` → `b` evaluated only if `a` is false

Important

Cheat Sheet Pattern Bitwise and Boolean

```
a ^ a = 0

a ^ 0 = a

a ^ -1 = ~a

a ^ ~a = -1

a & a = a

a | 0 = a
```

Arithmetic Example:

```java
int a = 10, b = 4;
System.out.println(a + b);  // 14
System.out.println(a - b);  // 6
System.out.println(a * b);  // 40
System.out.println(a / b);  // 2
System.out.println(a % b);  // 2
```

Relational Example:

```java
int a = 5, b = 8;
System.out.println(a < b);   // true
System.out.println(a >= b);  // false
System.out.println(a == b);  // false
System.out.println(a != b);  // true
```

Logical Example:

```java
boolean x = true, y = false;
System.out.println(x && y);  // false
System.out.println(x || y);  // true
System.out.println(!x);      // false
```

Bitwise Example:

```java
int a = 5;   // 0101
int b = 3;   // 0011
System.out.println(a & b);  // 1  (0001)
System.out.println(a | b);  // 7  (0111)
System.out.println(a ^ b);  // 6  (0110)
System.out.println(a << 1); // 10 (1010)
System.out.println(a >> 1); // 2  (0010)
```

5.7.2 Division and Modulus Operators

5.7.2.1 Division Operator

Dividing an `integer` by zero (for example, 10 / 0) causes the JVM to throw a `java.lang.ArithmeticException: / by zero`.

However, floating-point division behaves differently.

When a `float` or `double` value is divided by 0 or 0.0, no exception is thrown. Instead, the result is:

- **Float.POSITIVE_INFINITY** or **Float.NEGATIVE_INFINITY**
- **Double.POSITIVE_INFINITY** or **Double.NEGATIVE_INFINITY**

The sign depends on the operands involved in the operation.

To determine whether a floating-point value represents infinity, the `Float` and `Double` classes provide utility methods:

Static methods:

- **Float.isInfinite(float value)**
- **Double.isInfinite(double value)**

Instance methods:

- **Float.isInfinite()**
- **Double.isInfinite()**

These methods return true if the value corresponds to either positive or negative infinity.

5.7.2.2 Modulus Operator

The modulus operator is the remainder when two numbers are divided. If two numbers divide evenly, the reminder is 0: for example **10 % 5** is 0. On the other hand, **13 % 4** gives the reminder of 1.

We can use modulus with negative numbers according to the following rules:

- if the **divisor** is negative (Ex: 7 % -5), then the sign is ignored and the result is **2**;
- if the **dividend** is negative (Ex: -7 % 5), then the sign is preserved and the result is **-2**;

```java
System.out.println(8 % 5);      // GIVES 3
System.out.println(10 % 5);     // GIVES 0
System.out.println(10 % 3);     // GIVES 1
System.out.println(-10 % 3);    // GIVES -1
System.out.println(10 % -3);    // GIVES 1
System.out.println(-10 % -3);   // GIVES -1

System.out.println(8 % 9);      // GIVES 8
System.out.println(3 % 4);      // GIVES 3
System.out.println(2 % 4);      // GIVES 2
System.out.println(-8 % 9);     // GIVES -8
```

5.7.3 The Return Value of an Assignment Operator

In Java, the **assignment operator (`=`)** not only stores a value in a variable —
it also **returns the assigned value** as the result of the entire expression.

This means that the assignment operation itself can be **used as part of another expression**,
such as inside an `if` statement, a loop condition, or even another assignment.

```java
int x;
int y = (x = 10);   // the assignment (x = 10) returns 10
System.out.println(y);  // 10

// x = 10 assigns 10 to x.
// The expression (x = 10) evaluates to 10.
// That value is then assigned to y.
// So both x and y end up with the same value (10).
```

Because assignment returns a value, it can also appear inside an **if** statement. However, this often leads to logical errors if used unintentionally.

```java
boolean flag = false;

if (flag = true) {
    System.out.println("This will always execute!");
}

// Here the condition (flag = true) assigns true to flag, and then evaluates to true, so the if i

// Correct usage (comparison instead of assignment):

if (flag == true) {
    System.out.println("Condition checked, not assigned");
}
```

Warning

If you see `if (x = something)`, stop: it's **assignment**, not comparison.

5.7.4 Compound Assignment Operators

Compound assignment operators in Java combine an arithmetic or bitwise operation with assignment in a single step.
Instead of writing `x = x + 5`, you can use the shorthand `x += 5`.
They automatically perform **type casting** to the left-hand variable type when necessary.

Common compound operators include:
`+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`, and `>>>=`.

```java
int x = 10;

// Arithmetic compound assignments
x += 5;   // same as x = x + 5 → x = 15
x -= 3;   // same as x = x - 3 → x = 12
x *= 2;   // same as x = x * 2 → x = 24
x /= 4;   // same as x = x / 4 → x = 6
x %= 5;   // same as x = x % 5 → x = 1

// Bitwise compound assignments
int y = 6;    // 0110 (binary)
y &= 3;       // y = y & 3 → 0110 & 0011 = 0010 → y = 2
y |= 4;       // y = y | 4 → 0010 | 0100 = 0110 → y = 6
y ^= 5;       // y = y ^ 5 → 0110 ^ 0101 = 0011 → y = 3

// Shift compound assignments
int z = 8;    // 0000 1000
z <<= 2;      // z = z << 2 → 0010 0000 → z = 32
z >>= 1;      // z = z >> 1 → 0001 0000 → z = 16
z >>>= 2;     // z = z >>> 2 → 0000 0100 → z = 4

// Type casting example
byte b = 10;
// b = b + 1;   // ❌ compile-time error: int result cannot be assigned to byte
b += 1;         // ✅ works: implicit cast back to byte
```

> **Note**
>
> Compound assignments **perform an implicit cast** to the variable type on the left. That's why `b += 1` compiles even though `b = b + 1` does not.

5.7.5 Equality Operators (`==` and `!=`)

The **equality operators** in Java `==` (equal to) and `!=` (not equal to) are used to compare two operands for equality.
However, their behavior differs **depending on whether they are applied to primitive types or reference types (objects)**.

> **Note**
>
> - `==` compares **values** for primitives
> - `==` compares **references** for objects
> - `.equals()` compares **object content** (if implemented)

5.7.5.1 Equality with Primitive Types

When comparing **primitive values**, `==` and `!=` compare the **actual stored values**.

```
int a = 5, b = 5;
System.out.println(a == b);  // true  → both have the same value
System.out.println(a != b);  // false → values are equal
```

> **Important**
>
> - If the operands are of different numeric types, Java automatically promotes them to a common type before comparison.
> - However, comparing float and double can produce unexpected results due to precision errors (check example below)

```
int x = 10;
double y = 10.0;
System.out.println(x == y);  // true → x promoted to double (10.0)


double d = 0.1 + 0.2;
System.out.println(d == 0.3); // false → floating-point rounding issue
```

5.7.5.2 Equality with Reference Types (Objects)

For objects, == and != compare references, not object content. They return true only if both references point to the exact same object in memory.

```
String s1 = new String("Java");
String s2 = new String("Java");
System.out.println(s1 == s2);      // false → different objects in memory
System.out.println(s1 != s2);      // true  → not the same reference
```

Even if two objects have identical content, == compares their **addresses**, not values. To compare the **contents** of objects, use the **.equals()** method instead.

```
System.out.println(s1.equals(s2)); // true → same string content
```

Special Case: null and String Literals

- Any reference can be compared with null using == or !=.

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

- String literals are interned by the Java Virtual Machine (JVM): This means identical literal strings may point to the same reference in memory:

```
String a = "Java";
String b = "Java";
System.out.println(a == b);        // true → same interned literal
```

- Equality with Mixed Types: When using == between operands of different categories (e.g., primitive vs. object), the compiler tries to perform unboxing if one of them is a **wrapper class**.

```
Integer i = 100;
int j = 100;
System.out.println(i == j);   // true → unboxed before comparison
```

5.7.6 The `instanceof` Operator

`instanceof` is a **relational operator** that tests whether a reference value is an **instance of** a given **reference type** at runtime.
It returns a `boolean`.

```
Object o = "Java";
boolean b1 = (o instanceof String);    // true
boolean b2 = (o instanceof Number);    // false
```

null behavior: If expr is null, **expr instanceof Type** is always **false**.

```
Object n = null;
System.out.println(n instanceof Object);   // false
```

> **Warning**
>
> `instanceof` always returns `false` when the left operand is `null`.

5.7.6.1 Compile-Time Check vs Runtime Check

- At compile time, the compiler rejects inconvertible types (types that cannot possibly relate at runtime).
- At runtime, if the compile-time check passed, the JVM evaluates the actual object type.

```
// ❌ Compile-time error: inconvertible types (String is unrelated to Integer)

boolean bad = ("abc" instanceof Integer);

// ✅ Compiles, but runtime result depends on actual object:

Number num = Integer.valueOf(10);
System.out.println(num instanceof Integer); // true at runtime
System.out.println(num instanceof Double);  // false at runtime
```

5.7.6.2 Pattern Matching for instanceof

Java supports type patterns with instanceof, which both test and bind the variable when the test succeeds. Adding a variable after the type instructs the compiler to treat it as Pattern Matching

Syntax (pattern form):

```
Object obj = "Hello";

if (obj instanceof String str) {
    // Adding the variable str after the type instructs the compiler to treat it as Pattern Matc

    System.out.println(str.toUpperCase()); // identifier is in scope here, of type Type: (safe: 
}
```

Key properties:

- If the test succeeds, the pattern variable (e.g., s) is definitely assigned and in scope in the true branch.

- Pattern variables are implicitly final (cannot be reassigned).
- The name must not clash with an existing variable in the same scope.

5.7.6.3 Flow Scoping & Short-Circuit Logic

Pattern variables become available based on flow analysis:

```
Object obj = "data";

// Negated test, variable available in the else branch
if (!(obj instanceof String s)) {
    // s not in scope here
} else {
    System.out.println(s.length()); // s is in scope here
}

// With &&, pattern variable can be used on the right side if the left side established it
if (obj instanceof String s && s.length() > 3) {
    System.out.println(s.substring(0, 3)); // s in scope
}

// With ||, the pattern variable is NOT safe on the right side (short-circuit may fail to establ
if (obj instanceof String s || s.length() > 3) {  // ❌ s not in scope here
    // ...
}

// Parentheses can help group logic
if ((obj instanceof String s) && s.contains("a")) { // ✅ s in scope after grouped test
    System.out.println(s);
}
```

Pattern matching with `null` evaluates, like always for `instanceof`, to `false`

```
String str = null;

// Regular instanceof
if (str instanceof String) {
    System.out.print("NOT EXECUTED"); // instanceof evaluates to false
}

// Pattern matching
if (str instanceof String s) {
    System.out.print("NOT EXECUTED"); // instanceof still evaluates to false
}
```

Supported Types:

The type of the pattern variable must be a subtype, a supertype or of the same type of the reference variable.

```
Number num = Short.valueOf(10);

if (num instanceof String s) {}  // ❌ Compile-time error
if (num instanceof Short s) {}   // ✅ Ok
if (num instanceof Object s) {}  // ✅ Ok
if (num instanceof Number s) {}  // ✅ Ok
```

5.7.6.4 Arrays and Reifiable Types

instanceof works with arrays (which are reifiable) and with erased or wildcard generic forms. **Reifiable types** are those whose runtime representation fully preserves their type (e.g., raw types, arrays, non-generic classes, wildcard ?). Due to type erasure, List cannot be tested directly at runtime.

```
Object arr = new int[]{1,2,3};
System.out.println(arr instanceof int[]); // true

Object list = java.util.List.of(1,2,3);
// System.out.println(list instanceof List<Integer>); // ❌ Compile-time error: parameterized ty
System.out.println(list instanceof java.util.List<?>); // ✅ true
```

5.8 Ternary Operator

The **ternary operator** (`? :`) is the only operator in Java that takes **three operands**.
It acts as a concise form of an `if-else` statement.

5.8.1 Type Rules for the Ternary Operator

The type of a conditional (ternary) expression is determined by the types of its second and third operands.

5.8.1.1 Numeric Operands

- If one operand is `byte` and the other is `short`, the result type is `short`.
- If one operand is of type `T` (`byte`, `short`, or `char`) and the other operand is an `int` constant expression whose value fits within type `T`, then the result type is `T`.
- In all other numeric cases, **binary numeric promotion** is applied to the second and third operands.
  The type of the conditional expression becomes the promoted type.

> Binary numeric promotion includes **unboxing conversion** and **value set conversion**.

5.8.1.2 Reference Types

- If one operand is `null` and the other is a reference type, the result type is that reference type.
- If the operands are different reference types, one type must be assignment-compatible with the other. The resulting type is the more general type (the one to which the other can be assigned).
- If neither type can be assigned to the other, the expression causes a **compile-time error**.

In short, the ternary operator determines its type by applying:

- Special narrowing rules for small integral types
- Binary numeric promotion for numeric values
- Assignment compatibility rules for reference types

Tip

The ternary operator **must** produce a value of a *compatible type*. If the two branches produce unrelated types, compilation fails.

```
String s = true ? "ok" : 5; // ❌ compile error: incompatible types
```

5.8.2 Syntax

```
condition ? expressionIfTrue : expressionIfFalse;
```

5.8.3 Example

```
int age = 20;
String access = (age >= 18) ? "Allowed" : "Denied";
System.out.println(access);  // "Allowed"
```

5.8.4 Nested Ternary Example

```
int score = 85;
String grade = (score >= 90) ? "A" :
               (score >= 75) ? "B" :
               (score >= 60) ? "C" : "F";
System.out.println(grade);  // "B"
```

5.8.5 Notes

> **Warning**
>
> - Nested ternary expressions can reduce readability. Use parentheses for clarity.
> - The ternary operator returns a **value**, unlike `if-else`,

◀ 4. Java Data Types and Casting | ▲ Index | 6. Instantiating Types ▶

6. Instantiating Types

Table of Contents

- 6.1 Introduction
  - 6.1.1 Handling Primitive Types
    - 6.1.1.1 Declaring a Primitive
    - 6.1.1.2 Assigning a Primitive
  - 6.1.2 Handling Reference Types
    - 6.1.2.1 Creating and Assigning a Reference
    - 6.1.2.2 Constructors
    - 6.1.2.3 Instance Initializer Blocks
- 6.2 Default Variable Initialization
  - 6.2.1 Instance and Class Variables
  - 6.2.2 Final Instance Variables
  - 6.2.3 Local Variables
    - 6.2.3.1 Inferring Types with var
- 6.3 Wrapper Types
  - 6.3.1 Purpose of Wrapper Types
  - 6.3.2 Autoboxing and Unboxing
  - 6.3.3 Parsing and Conversion
  - 6.3.4 Helper Methods
  - 6.3.5 Null Values
- 6.4 Equality in Java
  - 6.4.1 Equality with Primitive Types
    - 6.4.1.1 Key Points
  - 6.4.2 Equality with Reference Types
    - 6.4.2.1 Identity Comparison
    - 6.4.2.2 equals Logical Comparison
    - 6.4.2.3 Key Points
  - 6.4.3 String Pool and Equality
    - 6.4.3.1 The intern Method
  - 6.4.4 Equality with Wrapper Types
    - 6.4.4.1 Wrapper Caching
    - 6.4.4.2 The `new` keyword bypasses the cache
    - 6.4.4.3 Comparing wrapper objects
    - 6.4.4.4 Different Wrapper Types Cannot Be Compared
  - 6.4.5 Equality and null
  - 6.4.6 Summary Table

6.1 Introduction

In Java, a **type** can be either a **primitive type** (such as `int`, `double`, `boolean`, etc.) or a **reference type** (classes, interfaces, arrays, enums, records, etc.). See: Java Data Types and Casting

The way instances are created depends on the category of the type:

- **Primitive types**
  Instances of primitive types are created simply by declaring a variable.
  The JVM automatically allocates memory to hold the value, and no explicit keyword is needed.

```
int age = 30;          // creates a primitive int with value 30
boolean flag = true;  // creates a primitive boolean with value true
double pi = 3.14159;  // creates a primitive double with value 3.14159
```

- **Reference types (objects)**
  Instances of class types are created using the new keyword (except for a few special cases such as string literals, records with canonical constructors, or factory methods). The new keyword allocates memory on the heap and invokes a constructor of the class.

```
String name = new String("Alice"); // creates a new String object explicitly
Person p = new Person();           // creates a new Person object using its constructor
```

It is also common to rely on literals or factory methods for object creation.

```
String text = "Hello World";

List<String> list = List.of("A", "B", "C");             // factory method immutabile
Map<String, Integer> map = Map.of("one", 1, "two", 2);  // factory method immutabile
Optional<String> opt = Optional.of("value");            // factory method

LocalDate date = LocalDate.of(2025, 3, 15);
Integer boxed = Integer.valueOf(10);
```

> **Important**
>
> String literals **do not require `new`** and are stored in the **String pool**. Using `new String("x")` always creates a new object on the heap.

6.1.1 Handling Primitive Types

6.1.1.1 Declaring a Primitive

Declaring a primitive type (as with reference types) means reserving space in memory for a variable of a given type, without necessarily giving it a value.

> **Warning**
>
> Unlike primitives, whose size depends on their specific type (e.g., `int` vs `long`), reference types always occupy the same fixed size in memory — what varies is the size of the object they point to.

- Syntax examples for declaration only:

```
int number;

boolean active;

char letter;

int x, y, z;          // Multiple declarations in one statement: Java allows declaring multipl
```

Important

The `modifiers` and the `type` declared at the beginning of a variable declaration apply to all variables declared in the same statement.

Exception: when declaring arrays using brackets after the variable name, the brackets are part of the individual variable declarator, not the base type.

- Examples

```
static int a, b, c;

// is equivalent to :

static int a;
static int b;
static int c;


int[] a, b;   // both are arrays of int
int c[], d;   // only c is an array, d is a regular int
```

6.1.1.2 Assigning a Primitive

Assigning a primitive type (as with reference types) means storing a value into a declared variable of that given type.
For primitives, the variable holds the value itself, while for reference types the variable holds the memory address (a reference) of the object being pointed to.

- Syntax examples:

```
int number;                      // Declaring an int type: a variable called "number"

number = 10;                 // Assigning the value 10 to this variable

char letter = 'A';           // Declaring and Assigning in a single statement: declaration and

int a1, a2;                    // Multiple declarations

int a = 1, b = 2, c = 3;     // Multiple declarations & assignements

char b1, b2, b3 = 'C';       // Mixed declarations (2 declarations + 1 assignment)

double d1, double d2;          // ERROR - NOT LEGAL

int v1; v2;                      // ERROR - NOT LEGAL
```

Important

When you write a number directly in the code (a numeric literal), Java assumes by default that it is of type **int**. If the value does not fit into an `int`, the code will not compile unless you explicitly mark it with the correct suffix.

- Syntax example for a numeric literal:

```
long exNumLit = 5729685479; // ❌ Does not compile.
                            // Even though the value would fit in a long,
                            // a plain numeric literal is assumed to be an int,
                            // and this number is too large for int.

Changing the declaration adding the correct suffix (L or l) will solve:

long exNumLit = 5729685479L;

or

long exNumLit = 5729685479l;
```

Declaring a `reference` type means reserving space in memory for a variable that will contain a reference (pointer) to an object of the specified type.

At this stage, no object is created yet — the variable only has the potential to point to one.

> **Warning**
>
> Unlike primitives, whose size depends on their specific type (e.g., `int` vs `long`), reference variables always occupy the same fixed size in memory (enough to store a reference). What varies is the size of the object they point to, which is allocated separately on the heap.

- Syntax examples for declaration only:

```
String name;
Person person;
List<Integer> numbers;

Person p1, p2, p3;    // Multiple declarations in one statement

String a = "abc", b = "def", c = "ghi";      // Multiple declarations & assignements

String b1, b2, b3 = "abc"                    // Mixed declarations (b1, b2) with one assignemer

String d1, String d2;                    // ERROR - NOT LEGAL

String v1; v2;                           // ERROR - NOT LEGAL
```

6.1.2 Handling Reference Types

6.1.2.1 Creating and Assigning a Reference

Assigning a `reference` type means storing into the variable the memory address of an object.

This is normally done after the creation of the object with the **new** keyword and a Constructor, or by using a literal or a factory method.

A reference can also be assigned to another object of the same or compatible type.

Reference types can also be assigned **null**, which means that they do not refer to an object.

- Syntax examples:

```
Person person = new Person(); // Example with 'new' and a constructor 'Person()':
                              // 'new Person()' creates a new Person object on the heap
                              // and returns its reference, which is stored in the variable 'per

String greeting = "Hello";   // Example with literal (for String).

List<Integer> numbers = List.of(1, 2, 3);    // Example with a factory method.
```

6.1.2.2 Constructors

In the example, `Person()` is a constructor — a special kind of method used to initialize new objects.

Whenever you call `new Person()`, the constructor runs and sets up the newly created instance.

Constructors have three main characteristics:

- The constructor name **must match the class name** exactly (case-sensitive).
- Constructors **do not declare a return type** (not even `void`).
- If you do not define any constructor in your class, the compiler will automatically provide a **default no-argument constructor** that does nothing.

> **Warning**
>
> If you see a method that has the same name as the class **but also declares a return type**, it is **not** a constructor. It is simply a regular method (though starting method names with a capital letter is against Java naming conventions).

The **purpose of a constructor** is to initialize the state of a newly created object — typically by assigning values to its fields, either with default values or using parameters passed to the constructor.

- Example 1: Default constructor (no parameters)

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

    // Default constructor
    public Person() {
        name = "Unknown";
        age = 0;
    }
}

Person p1 = new Person(); // name = "Unknown", age = 0
```

- Example 2: Constructor with parameters

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

    // Constructor with parameters
    public Person(String newName, int newAge) {
        name = newName;
        age = newAge;
    }
}

Person p2 = new Person("Alice", 30); // name = "Alice", age = 30
```

- Example 3: Multiple constructors (constructor overloading)

```java
public class Person {
    String name;
    int age;

    // Default constructor
    public Person() {
        this("Unknown", 0); // calls the other constructor
    }

    // Constructor with parameters
    public Person(String newName, int newAge) {
        name = newName;
        age = newAge;
    }
}

Person p1 = new Person();            // name = "Unknown", age = 0
Person p2 = new Person("Bob", 25);   // name = "Bob", age = 25
```

> **Important**
>
> - Constructors are not inherited: if a superclass defines constructors, they are not automatically available in the subclass — you must declare them explicitly.
> - If you declare any constructor in a class, the compiler does not generate the default no-argument constructor: if you still need a no-argument constructor, you must declare it manually.

6.1.2.3 Instance Initializer Blocks

In addition to constructors, Java provides a mechanism called **initializer blocks** to help initialize objects. These are blocks of code inside a class, enclosed in `{ }`, that run **every time an instance is created**, just before the constructor body is executed.

Characteristics

- Also called **instance initializer blocks**.
- Executed, along with fields initializers, in the order in which they appear in the class definition but always before Constructors.
- Useful when multiple constructors need to share common initialization code.

Example: Using an Instance Initializer Block

```java
public class Person {
    String name;
    int age;

    // Instance initializer block
    {
        System.out.println("Instance initializer block executed");
        age = 18; // default age for every Person
    }

    // Default constructor
    public Person() {
        name = "Unknown";
    }

    // Constructor with parameters
    public Person(String newName) {
        name = newName;
    }
}

Person p1 = new Person();          // prints "Instance initializer block executed"
Person p2 = new Person("Alice");   // prints "Instance initializer block executed"
```

> **Note**
>
> In this example, the initializer block runs before either constructor body. Both p1 and p2 will start with age = 18, regardless of which constructor is used.

Multiple Initializer Blocks: if a class contains multiple initializer blocks, they are executed in the order they appear in the source file:

- Example:

```
public class Example {
    {
        System.out.println("First block");
    }

    {
        System.out.println("Second block");
    }
}

Example ex = new Example();
// Output:
// First block
// Second block
```

> **Note**
>
> Instance initializer blocks are less common in practice, because similar logic can often be placed directly in constructors. It is important to know that: - They always run before the constructor body. - They are executed in the order of declaration in the class. - They can be combined with constructors to avoid code duplication.

> **Warning**
>
> **Order of initialization when creating an object** 1. Static fields 2. Static initializer blocks 3. Instance fields 4. Instance initializer blocks 5. Constructor body

---

6.2 Default Variable Initialization

6.2.1 Instance and Class variables

- An **instance variable (a field)** is a value defined within an instance of an object;
- A **class variable** (defined with the keyword **static**) is defined at class level and it is shared among all the objects (instances of the class)

Instance and class variables are given a default value, by the compiler, if not initialized.

- Table of default values for instance & class variables;

Type	Default Value
Object	null
Numeric	0
boolean	false
char	'000' (NUL)

6.2.2 Final Instance Variables

Unlike regular instance and class variables, `final` **variables are *not* default-initialized by the compiler**.

A `final` variable **must be explicitly assigned exactly once**, otherwise the code does not compile.

This applies to both:

- **final instance variables**
- **static final class variables**

> **Note**
>
> We can assign a `null` value to a `final` instance or class instance variables as long they are explicitely set.

Java enforces this rule because a `final` variable represents a value that must be *known and fixed* before use.

`Final Instance Variables`

A **final instance variable** must be assigned **exactly once**, and the assignment must occur in *one* of the following:

1. **At the point of declaration**
2. **In an instance initializer block**
3. **Inside every constructor**

If the class has *multiple constructors*, the variable must be assigned in **all** of them.

- Example:

```java
public class Person {
    final int id;   // must be assigned before constructor ends
    String name;

    // Constructor 1
    public Person(int id, String name) {
        this.id = id;         // ok
        this.name = name;
    }

    // Constructor 2
    public Person() {
        this.id = 0;          // also required here
        this.name = "Unknown";
    }
}
```

> **Warning**
>
> Trying to compile without assigning `id` inside **every** constructor produces a compile-time error: variable id might not have been initialized

`static final` Class Variables (Constants)

A **static final variable** belongs to the class rather than to any instance.
It must also be assigned exactly once, but assignment can occur in one of the following places:

1. **At the point of declaration**
2. **Inside a static initializer block**

- Example:

```java
public class AppConfig {

    static final int TIMEOUT = 5000;     // assigned at declaration

    static final String VERSION;         // assigned in static initializer

    static {
        VERSION = "1.0.0";               // ok
    }
}
```

Attempting to assign a `static final` in a constructor is illegal.

Key Rules for `final` Fields

Scenario	Allowed?	Notes
Assign at declaration	✓	Most common pattern
Assign in constructor	✓	All constructors must assign it
Assign in instance initializer	✓	Before constructor body runs
Assign in static initializer (`static final` only)	✓	For class-level constants
Assign multiple times	❌	Compilation error
Default initialization	❌	Must be explicitly assigned

Example of an **illegal** situation:

```java
public class Example {
    final int x;        // not initialized
}

Example e = new Example(); // ❌ compile-time error
```

Why `final` Variables Are Not Default-Initialized?

Because:

- Their value must be **known and immutable**, and
- Java must guarantee that the value is set **before use**,
- Default initialization would create a situation where `0`, `null`, or `false` might unintentionally become the permanent value.

Thus, Java forces developers to explicitly initialize `final` fields.

> **Tip**
>
> `final` means **assigned once**, not **immutable object**.
>
> A final reference can still point to a mutable object.

```java
final List<String> list = new ArrayList<>();
list.add("ok");       // allowed
list = new ArrayList<>(); // ❌ cannot reassign reference
```

6.2.3 Local variables

Local variables are variables defined within a constructor, method or inizializer block;

Local variables do not have default values and they must be initialized before they can be used. If you try to use a not initialized local variable the compiler will report an ERROR.

- Example

```
public int localMethod {

    int firstVar = 25;
    int secondVar;
    secondVar = 35;
    int firstSum = firstVar + secondVar;    // OK variables are both initialized before use

    int thirdVar;
    int secondSum = firstSum + thirdVar;    // ERROR: variable thirdVar has not been initializ
}
```

6.2.3.1 Inferring Types with var

Under certain conditions you can use the keyword **var** in place of the appropriate type when declaring **local** variables;

> **Warning**
>
> - **var** IS NOT a reserved word in java;
> - **var** can be used only for local variables: it CAN'T be used for **constructor parameters**, **instance variables** or **method parameters**;
> - The compiler infere the type by looking ONLY at the code **on the line of the declaration**; once the right type has been inferred you can't reassign to another type.

- example

```
public int localMethod {

    var inferredInt = 10;   // The compiler infer int by the context;
    inferredInt = 25;       // OK

    inferredInt = "abcd";   // ERROR: the compiler has already inferred the type of the variab

    var notInferred;
    notInferred = 30;       // ERROR: in order to infer the type, the compiler looks ONLY at t

    var first, second = 15; // ERROR: var cannot be used to define two variables on the same s

    var x = null;           // ERROR: var cannot be initialized with null but it can be reassi
}
```

> **Warning**
>
> Local variables **never** get default values. Instance & static fields **always** do.

6.3 Wrapper Types

In Java, **wrapper types** are object representations of the eight primitive types.
Each primitive has a corresponding wrapper class in the `java.lang` package:

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

Wrapper objects are immutable — once created, their value cannot change.

6.3.1 Purpose of Wrapper Types

- Allow primitives to be used in contexts that require objects (e.g., collections, generics).
- Provide utility methods for parsing, converting, and working with values.
- Support constants such as `Integer.MAX_VALUE` or `Double.MIN_VALUE`.

6.3.2 Autoboxing and Unboxing

Since Java 5, the compiler automatically converts between primitives and their wrappers:

- **Autoboxing**: primitive → wrapper
- **Unboxing**: wrapper → primitive

```
Integer i = 10;        // autoboxing: int → Integer
int n = i;             // unboxing: Integer → int

Integer int1 = Integer.valueOf(11);
long long1 = int1;  // Unboxing --> implicit cast OK

Long long2 = 11;   // ❌ Does not compile.
                   // 11 is an int literal → requires autoboxing + widening → illegal

Character char1 = null;
char char2 = char1;  // WARNING: NullPointerException

Integer  arr1 = {11.5, 13.6}  // WARNING: Does not compile!!
Double[] arr2 = {11, 22};     // WARNING: Does not compile!!
```

Tip

Java **never** performs autoboxing + widening/narrowing in one step.

Warning

- **AUTOBOXING** and **Implicit cast** are not allowed in the same statement: you can't do both at the same time. (see example above)
- This rule apply also in method calls.

6.3.3 Parsing and Conversion

Wrappers provide static methods to convert strings or other types into primitives:

```java
int x = Integer.parseInt("123");    // returns primitive int
Integer y = Integer.valueOf("456"); // returns Integer object
double d = Double.parseDouble("3.14");

// On the numeric wrapper class valueOf() throws a NumberFormatException on invalid input.
// Example:

Integer w = Integer.valueOf("two"); // NumberFormatException

// On Boolean, the method returns Boolean.TRUE for any value that matches "true" ignoring case
// Example:

Boolean.valueOf("true");     // true
Boolean.valueOf("TrUe");     // true
Boolean.valueOf("TRUE");     // true
Boolean.valueOf("false");    // false
Boolean.valueOf("FALSE");    // false
Boolean.valueOf("xyz");      // false
Boolean.valueOf(null);       // false

// The numeric integral classes Byte, Short, Integer and Long include an overloaded **valueOf(
// Example with base 16 (hexadecimal) which includes character 0 -> 9 and A -> F (ignore case)

Integer.valueOf("6", 16);   // 6
Integer.valueOf("a", 16);   // 10
Integer.valueOf("A", 16);   // 10
Integer.valueOf("F", 16);   // 15
Integer.valueOf("G", 16);   // NumberFormatException
```

> **Note**
>
> methods **parseXxx()** return a primitive while **valueOf()** returns a wrapper object.

6.3.4 Helper methods

All the numeric wrapper classes extend the Number class and, for that, they inherit some helper methods such as: byteValue(), shortValue(), intValue(), longValue(), floatValue(), doubleValue().

The Boolean and Character wrapper classes include: booleanValue() and charValue().

- Example:

```java
        // In trying to convert those helper methods can result in a loss of precision.

        Double baseDouble = Double.valueOf("300.56");

        double wrapDouble = baseDouble.doubleValue();
        System.out.println("baseDouble.doubleValue(): " + wrapDouble);  // 300.56

        byte wrapByte = baseDouble.byteValue();
        System.out.println("baseDouble.byteValue(): " + wrapByte);      // 44  -> There is no

        int wrapInt = baseDouble.intValue();
        System.out.println("baseDouble.intValue(): " + wrapInt);        // 300 -> The value is
```

6.3.5 Null Values

Unlike primitives, wrapper types can hold **null**. Attempting to unbox null causes a NullPointerException:

```java
Integer val = null;
int z = val; // ❌ NullPointerException at runtime
```

6.4 Equality in Java

Java provides two different mechanisms for checking equality:

- `==` (equality operator)
- `.equals()` (method defined in `Object` and overridden in many classes)

Understanding the difference is essential.

6.4.1 Equality with Primitive Types

For **primitive values** (`int` , `double` , `char` , `boolean` , etc.),
the operator `==` compares their actual **numeric or boolean value**.

Example:

```
int a = 5;
int b = 5;
System.out.println(a == b);      // true

char c1 = 'A';
char c2 = 65;                    // same Unicode code point
System.out.println(c1 == c2);    // true
```

6.4.1.1 Key points

- `==` performs **value comparison** for primitives.
- Primitive types have **no** **`.equals()`** method.
- Mixed primitive types follow numeric **promotion rules**
  (e.g., `int == long` → `int` promoted to `long`).

6.4.2 Equality with Reference Types

With objects (reference types), the meaning of `==` changes.

6.4.2.1 `==` (Identity Comparison)

`==` checks whether **two references point to the same object in memory**.

```
String s1 = new String("Hi");
String s2 = new String("Hi");

System.out.println(s1 == s2);      // false → different objects
```

Even if contents are identical, `==` is false unless both variables refer to
the exact same object.

6.4.2.2 `.equals()` (Logical Comparison)

Many classes override `.equals()` to compare **values**, not memory addresses.

```
System.out.println(s1.equals(s2)); // true → same content
```

6.4.2.3 Key points

- `.equals()` is defined in `Object`.
- If a class does *not* override `.equals()` , it behaves like `==` .
- Classes like `String` , `Integer` , `List` , etc. override `.equals()`
  to provide meaningful value comparison.

6.4.3 String Pool and Equality

String literals are stored in the **String pool**, so identical literals
refer to the **same object**.

```
String a = "Java";
String b = "Java";
System.out.println(a == b);       // true → same pooled literal
```

But using `new` creates a different object:

```java
String x = new String("Java");
String y = "Java";

System.out.println(x == y);        // false → x is not pooled
System.out.println(x.equals(y));   // true
```

Common Pitfalls

```java
String x = "Java string literal";
String y = " Java string literal".trim();

System.out.println(x == y);        // false → x and y are not the same at compile time

String a = "Java string literal";
String b = "Java ";
b += "string literal";

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

> **Warning**
>
> Any String created at **runtime** does *not* go into the pool automatically. Use `intern()` if you want pooling.

> **Tip**
>
> `"Hello" == "Hel" + "lo"` → true (compile-time constant)
>
> `"Hello" == getHello()` → false (runtime concatenation)

```java
String x = "Hello";
String y = "Hel" + "lo";    // compile-time → same literal
String z = "Hel";
z += "lo";                  // runtime → new String

System.out.println(x == y); // true
System.out.println(x == z); // false
```

6.4.3.1 The intern method

You can also tell Java to use a String from the String Pool (in case it already exist) through the `intern()` method:

```java
String x = "Java";
String y = new String("Java").intern();


System.out.println(x == y);        // true
```

6.4.4 Equality with Wrapper Types

Wrapper classes (`Integer` , `Double` , `Boolean` , etc.) behave like normal objects.

Therefore:

- `==` → compares **object references**
- `.equals()` → compares **numeric values**

Example:

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

Integer c = 1000;
Integer d = 1000;
System.out.println(c == d);         // false → different objects

System.out.println(c.equals(d));    // true → same numeric value
```

Since, as previously noted, all wrapper classes are **immutable**, their internal value **cannot be modified** once they are created.

Operations that appear to modify a wrapper actually create **a new object**.

Example:

```java
Integer i = 5;
i++;
```

This is conceptually equivalent to:

```java
i = Integer.valueOf(i.intValue() + 1);
```

Therefore a **new `Integer` object** is created and assigned back to `i`.

6.4.4.1 Wrapper Caching

To reduce memory usage and object creation, Java **reuses certain wrapper instances**.

The following values are cached:

- All `Boolean` values (`true` and `false`)
- All `Byte` values
- All `Character` values from `\u0000` to `\u007f` (0–127)
- All `Short` values from **−128 to 127**
- All `Integer` values from **−128 to 127**

Because of this caching mechanism:

```java
Integer a = 100;
Integer b = 100;

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

Both variables refer to **the same cached object**.

However, values outside the cache range produce **distinct objects**:

```java
Integer c = 1000;
Integer d = 1000;

System.out.println(c == d);   // false
System.out.println(c.equals(d)); // true
```

6.4.4.2 The `new` keyword bypasses the cache

When a wrapper object is created using `new`, **a new instance is always created**, even if a cached value exists.

Example:

```
Integer i = 10;          // cached object
Integer j = 10;          // same cached object
Integer k = new Integer(10); // new object (not cached)

System.out.println(i == j); // true
System.out.println(i == k); // false
```

However, wrapper constructors were **deprecated in Java 9** and marked for removal.
Modern code should use **autoboxing** or factory methods such as `Integer.valueOf()`.

6.4.4.3 Comparing wrapper objects

When two wrapper references are compared using `==`, the result depends on whether they **refer to the same object**, not on whether their values are equal.

Therefore, equality tests between wrappers should normally use:

```
equals()
```

instead of `==`.

6.4.4.4 Different Wrapper Types Cannot Be Compared

Wrapper objects of **different types** cannot be compared using `==`.

Example:

```
Byte b = 1;
Integer i = 1;

b == i;   // compilation error
```

The operands must be **compatible types**, otherwise the comparison is not valid.

> **Warning**
>
> Be very careful when comparing wrapper objects with `==`.
> Due to wrapper caching, comparisons may sometimes return `true` and sometimes `false` depending on the value.

6.4.5 Equality and `null`

- `== null` is always safe.
- Calling `.equals()` on a `null` reference throws a **NullPointerException**.

```
String s = null;
System.out.println(s == null);    // true
// s.equals("Hi");                // ❌ NullPointerException
```

6.4.6 Summary Table

Comparison	Primitives	Objects / Wrappers	Strings
`==`	compares **value**	compares **reference**	identity (affected by String pool)
`.equals()`	N/A	compares **content** if overridden	**content** comparison

---

Control Flow

7. Control Flow

Table of Contents

- 7.1 The if Statement
- 7.2 The switch Statement & Expression
  - 7.2.1 The switch target variable can be
  - 7.2.2 Acceptable Case Values
  - 7.2.3 Type Compatibility Between Selector and Case
  - 7.2.4 Pattern Matching in Switch
    - 7.2.4.1 Variable Names and Scope Across Branches
    - 7.2.4.2 Ordering Dominance and Exhaustiveness in Pattern Switches
- 7.3 Two Forms of switch Statement vs switch Expression
  - 7.3.1 The Switch Statement
    - 7.3.1.1 Fall-Through Behavior
  - 7.3.2 The Switch Expression
    - 7.3.2.1 yield in Switch Expression Blocks
    - 7.3.2.2 Exhaustiveness for Switch Expressions
- 7.4 Null Handling

Control flow in Java refers to the **order in which individual statements, instructions, or method calls are executed** during program runtime.

By default, statements run sequentially from top to bottom, but control flow statements allow the program to **make decisions**, **repeat actions**, or **branch execution paths** based on conditions.

Java provides three main categories of control flow constructs:

- **Decision-making statements** — `if`, `if-else`, `switch`
- **Looping statements** — `for`, `while`, `do-while`, and the enhanced `for`
- **Branching statements** — `break`, `continue`, and `return`

> **Tip**
>
> Understanding control flow is essential to seeing how data moves through your program and how each logic decision is evaluated step by step.

7.1 The `if` Statement

The `if` statement is a conditional control-flow structure that executes a block of code only if a specified boolean expression evaluates to `true`. It allows the program to make decisions at runtime.

Syntax:

```
if (condition) {
    // executed only when condition is true
}
```

An optional `else` clause handles the alternative path:

```
if (score >= 60) {
    System.out.println("Passed");
} else {
    System.out.println("Failed");
}
```

Multiple conditions can be chained using `else if`:

```
if (grade >= 90) {
    System.out.println("A");
} else if (grade >= 80) {
    System.out.println("B");
} else if (grade >= 70) {
    System.out.println("C");
} else {
    System.out.println("D or below");
}
```

> **Note**
>
> The `if` condition must evaluate to a **boolean**; numeric or object types cannot be used directly as conditions.
>
> Curly braces `{}` are optional for single statements but strongly recommended to prevent subtle logic errors.
>
> An `if-else` chain is evaluated from top to bottom, and only the first branch with a condition evaluating to `true` is executed.

7.2 The `switch` Statement & Expression

The `switch` construct is a control-flow structure that selects one branch among multiple alternatives based on the value of an expression (the **selector**).

Compared to long chains of `if-else-if`, a `switch`:

- Is often **easier to read** when testing many discrete values (constants, enums, strings).
- Can be **safer and more concise** when used as a **switch expression**

because:

- It **produces a value**.
- The compiler can enforce **exhaustiveness** and **type consistency**.

Java 21 supports:

- The classic `switch` **statement** (control flow only).
- The `switch` **expression** (produces a result).
- **Pattern matching** inside `switch`, including type patterns and guards.

Both forms of `switch` share the same rules concerning the selector (switch **target variable**) and acceptable case values.

7.2.1 The switch `target variable` can be

Control Variable type
`byte` / `Byte`
`short` / `Short`
`char` / `Character`
`int` / `Integer`
`String`
Enum types (selectors of an `enum`)
Any reference type (with pattern matching)
`var` (if it resolves to one of the allowed types)

> **Warning**
>
> **Not allowed** as selector types for switch:
>
> - `boolean`
> - `long`
> - `float`
> - `double`

7.2.2 Acceptable `case` Values

For a non-pattern switch, each `case` label must be a compile-time constant compatible with the selector type.

Allowed as case labels:

- **Literals** such as `0`, `'A'`, `"ON"`.
- **Enum constants**, e.g., `RED` or `Color.GREEN`.
- **Final constant variables** (compile-time constants).

A compile-time constant variable:

- Must be declared with `final` and initialized in the same statement.
- Its initializer must itself be a constant expression (typically using literals and other compile-time constants).

7.2.3 Type Compatibility Between Selector and Case

The selector type and each `case` label must be compatible:

- Numeric case constants must be within the range of the selector type.
- For an `enum` selector, case labels must be constants of that `enum`.
- For a `String` selector, case labels must be string constants.

7.2.4 Pattern Matching in Switch

Switch in Java 21 supports pattern matching, including:

- **Type patterns**: `case String s`
- **Guarded patterns**: `case String s when s.length() > 3`
- **Null pattern**: `case null`

Example:

```
String describe(Object o) {
    return switch (o) {
        case null -> "null";
        case Integer i -> "int " + i;
        case String s when s.isEmpty() -> "empty string";
        case String s -> "string (" + s.length() + ")";
        default -> "other";
    };
}
```

Key points:

- Each pattern introduces a pattern variable (such as `i` or `s`).
- Pattern variables are in scope only within their own arm (or paths where the pattern is known to match).
- Order matters because of **dominance**: more specific patterns must precede more general ones.

7.2.4.1 Variable Names and Scope Across Branches

With pattern matching, the pattern variable exists only in the scope of the arm in which it is defined. This means you can reuse the same variable name in different case branches.

- Example:

```
switch (o) {
    case String str -> System.out.println(str.length());
    case CharSequence str -> System.out.println(str.charAt(0));
    default -> { }
}
```

> **Note**
>
> This last example does not return a value, so it is a **statement switch**, not a switch expression.

7.2.4.2 Ordering, Dominance and Exhaustiveness in Pattern Switches

When dealing with pattern matching, the ordering of branches is crucial because of **dominance** and potential **unreachable code**.

A more general pattern must **not** appear before a more specific one, or the specific one becomes unreachable.

- Example (unreachable branch):

```
return switch (o) {
    case Object obj -> "object";
    case String s -> "string"; // ❌ DOES NOT COMPILE: unreachable, String is already matched
};
```

- Another example with a guard:

```
return switch (o) {
    case Integer a -> "First";
    case Integer a when a > 0 -> "Second"; // ❌ DOES NOT COMPILE: unreachable, the first case
    // ...
};
```

When using pattern matching, switches must be **exhaustive**; that is, they must handle all possible selector values.

This can be achieved by:

- Providing a `default` case that handles all values not matched by any other case.
- Providing a final case clause with a pattern type that matches the selector reference type.
- Example (not exhaustive):

```
Number number = Short.valueOf(10);

switch (number) {
    case Short s -> System.out.println("A"); // ❌ DOES NOT COMPILE: not exhaustive, selector
}
```

To fix this, you can:

- Change the reference type of `number` to `Short` (then exhaustiveness is satisfied by the single case).
- Add a `default` clause that covers all remaining values.
- Add a final case clause covering the type of the selector variable, for example:

```
Number number = Short.valueOf(10);

switch (number) {
    case Short s -> System.out.println("A");
    case Number n -> System.out.println("B");
}
```

> **Warning**
>
> The following example, which uses both a `default` clause and a final clause with the same type as the selector variable, does **not** compile: the compiler considers one of the two cases as always dominating the other.

```
Number number = Short.valueOf(10);

switch (number) {
    case Short s -> System.out.println("A");
    case Number n -> System.out.println("B"); // ❌ DOES NOT COMPILE: dominated by either the
    default -> System.out.println("C");
}
```

7.3 Two Forms of `switch`: `switch` Statement vs `switch` Expression

7.3.1 The Switch Statement

A **switch statement** is used as a control-flow construct.

It does not, by itself, evaluate to a value, although its branches may contain `return` statements that return from the enclosing method.

```
switch (mode) { // switch statement
    case "ON":
        start();
        break; // prevents fall-through
    case "OFF":
        stop();
        break;
    default:
        reset();
}
```

Key points:

- Each `case` clause includes one or more matching values separated by commas `,`. A separator follows, which can be either a colon `:` or, less commonly for statements, the arrow operator `->`. Finally, an expression or a block (enclosed in `{}`) defines the code to execute when a match occurs. If you use the arrow operator for one clause, you must use it for all clauses in that switch statement.

- Fall-through is possible for colon-style cases unless a branch uses `break`, `return`, or `throw`. When present, `break` terminates the switch after executing its case; without it, execution continues, in order, into the following branches.
- A `default` clause is optional and can appear anywhere in the switch statement. It runs if there is no match for previous cases.
- A switch statement does not yield a value as an expression; you cannot assign a switch statement directly to a variable.

7.3.1.1 Fall-Through Behavior

With colon-style cases, execution jumps to the matching case label.

If there is no `break`, it continues into the next case until a `break`, `return`, or `throw` is encountered.

```
int n = 2;

switch (n) {
    case 1:
        System.out.println("1");
    case 2:
        System.out.println("2"); // printed
    case 3:
        System.out.println("3"); // printed (fall-through)
        break;
    default:
        System.out.println("message default");
}
```

Output:

```
2
3
```

> **Note**
>
> If in the previous example we remove the `break` on `case 3`, the message from the `default` branch will also be printed.

7.3.2 The Switch Expression

A **switch expression** always produces a single value as its result.

- Example:

```
int len = switch (s) { // switch expression
    case null -> 0;
    case "" -> 0;
    default -> s.length();
};
```

Key points:

- Each `case` clause includes one or more matching values separated by commas `,`, followed by the arrow operator `->`. Then an expression or a block (enclosed in `{}`) defines the result for that arm.
- When used with an assignment or a `return` statement, a switch expression requires a terminating semicolon `;` after the expression.
- There is no fall-through between arrow arms. Each matching arm executes exactly once.
- A switch expression must be **exhaustive**: all possible selector values must be covered (via explicit cases and/or `default`).
- The result type must be consistent across all branches. For example, if one arm yields an `int`, the other arms must yield values compatible with `int`.

7.3.2.1 `yield` in Switch Expression Blocks

When an arm of a switch expression uses a block instead of a single expression, you must use `yield` to provide the result of that arm.

```java
int len = switch (s) {
    case null -> 0;
    default -> {
        int l = s.trim().length();
        System.out.println("Length: " + l);
        yield l; // result of this arm
    }
};
```

> **Note**
>
> `yield` is used only in switch expressions. `break value;` is not allowed as a way to return a value from a switch expression.

7.3.2.2 Exhaustiveness for Switch Expressions

Because a switch expression must return a value, it must also be **exhaustive**; in other words, it must handle all possible selector values.

You can ensure this by:

- Providing a `default` case.
- For an enum selector: covering all enum constants explicitly.
- For sealed types or pattern switches: covering all permitted subtypes or providing a `default`.

Example, exhaustive via `default`:

```java
int val = switch (s) {
    case "one" -> 1;
    case "two" -> 2;
    default -> 0;
};
```

7.4 Null Handling

Classic switch (without patterns)

If the selector expression of a classic switch (without pattern matching) evaluates to `null`, a `NullPointerException` is thrown at runtime.

To avoid this, check for `null` before switching:

```java
if (s == null) {
    // handle null
} else {
    switch (s) {
        case "A" -> ...
        default -> ...
    }
}
```

Pattern switch (with `case null`)

With pattern matching, you can handle `null` directly inside the switch:

```java
int len = switch (s) {
    case null -> 0;
    default -> s.length();
};
```

Note

For switch expressions:

If you do not handle `null` and the selector is `null`, a `NullPointerException` is thrown.

Using `case null` makes the switch explicitly null-safe.

Warning

Any time `case null` is used in a switch, the switch is treated as a pattern switch, and all the rules for pattern switches (including exhaustiveness and dominance) apply.

◀ 6. Instantiating Types | ▲ Index | 8. Looping Constructs in Java ▶

8. Looping Constructs in Java

Table of Contents

- 8.1 The while Loop
- 8.2 The do-while Loop
- 8.3 The for Loop
- 8.4 The Enhanced for-each Loop
- 8.5 Nested Loops
- 8.6 Infinite Loops
- 8.7 break and continue
- 8.8 Labeled Loops
- 8.9 Loop Variable Scope
- 8.10 Unreachable Code After break continue and return
    - 8.10.1 Unreachable Code After break
    - 8.10.2 Unreachable Code After continue
    - 8.10.3 Unreachable Code After return

Java provides several **looping constructs** that allow repeated execution of a block of code as long as a condition holds.

Loops are essential for iteration, traversal of data structures, repeated computations, and implementing algorithms.

8.1 The `while` Loop

The `while` loop evaluates its **boolean condition before each iteration**.
If the condition is `false` from the beginning, the body is never executed.

Syntax

```
while (condition) {
    // loop body
}
```

- The condition must evaluate to a boolean.
- The loop may execute zero or more times.
- Common pitfalls include forgetting to update the loop variable, causing an infinite loop.
- Example:

```
int i = 0;
while (i < 3) {
    System.out.println(i);
    i++;
}
```

Output:

```
0
1
2
```

8.2 The `do-while` Loop

The `do-while` loop evaluates its condition *after* executing the body, ensuring the body runs at least once.

Syntax

```
do {
    // loop body
} while (condition);
```

> **Tip**
>
> `do-while` requires a semicolon after the closing parenthesis.

- Example:

```
int x = 5;
do {
    System.out.println(x);
    x--;
} while (x > 5); // body runs once even though condition is false
```

Output:

```
5
```

8.3 The `for` Loop

The traditional `for` loop is best suited for loops with a counter variable. It consists of three parts: initialization, condition, update.

Syntax

```
for (initialization; condition; update) {
    // loop body
}
```

- Initialization runs once before the loop starts.
- Condition is evaluated before each iteration.
- Update runs after each iteration.
- Initialization and update may contain multiple statements separated by commas.
- Variables in initialization must all be of the same type.
- Any component may be omitted, but semicolons remain.
- Example:

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

Omitting parts:

```
int j = 0;
for (; j < 3;) {  // valid
    j++;
}
```

Multiple statements:

```
int x = 0;
for (long i = 0, c = 3; x < 3 && i < 12; x++, i++) {
    System.out.println(i);
}
```

8.4 The Enhanced `for-each` Loop

The enhanced `for` simplifies iteration over arrays and collections.

Syntax

```
for (ElementType var : arrayOrCollection) {
    // loop body
}
```

- Loop variable is read-only relative to the underlying collection.
- Works with any `Iterable` or array.
- Cannot remove elements without an iterator.
- Example:

```
String[] names = {"A", "B", "C"};
for (String n : names) {
    System.out.println(n);
}
```

Output:

```
A
B
C
```

8.5 Nested Loops

Loops may be nested; each maintains its own variables and conditions.

```
for (int i = 1; i <= 2; i++) {
    for (int j = 1; j <= 3; j++) {
        System.out.println(i + "," + j);
    }
}
```

Output:

```
1,1
1,2
1,3
2,1
2,2
2,3
```

8.6 Infinite Loops

A loop is infinite when its condition always evaluates to `true` or is omitted.

```
while (true) { ... }
```

```
for (;;) { ... }
```

> **Tip**
>
> Infinite loops must contain `break`, `return`, or external control.

8.7 `break` and `continue`

break

Exits the innermost loop immediately.

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

continue

Skips the rest of the loop body and continues to next iteration.

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

> **Note**
>
> `break` and `continue` apply to the nearest loop unless labels are used.

8.8 Labeled Loops

A label (identifier + colon) may be applied to a loop to allow break/continue to affect outer loops.

```
labelName:
for (...) {
    for (...) {
        break labelName;
    }
}
```

- Example:

```
outer:
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
        if (j == 2) break outer;
        System.out.println(i + "," + j);
    }
}
```

8.9 Loop Variable Scope

- Variables declared in the loop header are scoped to that loop.
- Variables declared inside the body exist only inside that block.

```
for (int i = 0; i < 3; i++) {
    int x = i * 2;
}
// i and x are not accessible here
```

8.10 Unreachable code after `break`, `continue`, and `return`

Any statement placed **after** `break`, `continue`, or `return` in the same block is considered unreachable and will not compile.

8.10.1 Unreachable Code After `break`

```
for (int i = 0; i < 3; i++) {
    break;
    System.out.println("Unreachable"); // ❌ Compile-time error
}
```

8.10.2 Unreachable Code After `continue`

```
for (int i = 0; i < 3; i++) {
    continue;
    System.out.println("Unreachable"); // ❌ Compile-time error
}
```

> **Note**
>
> `continue` jumps to the next iteration, so following code is never executed.

8.10.3 Unreachable Code After `return`

```
int test() {
    return 5;
    System.out.println("Unreachable"); // ❌ Compile-time error
}
```

> **Note**
>
> `return` exits the method immediately; no statements can follow.

Core Standard APIs

9. Strings in Java

Table of Contents

- 9.1 Strings & Text Blocks
    - 9.1.1 Strings
        - 9.1.1.1 Initializing Strings
        - 9.1.1.2 The String Pool
        - 9.1.1.3 Special Characters and Escape Sequences
        - 9.1.1.4 Rules for String Concatenation
        - 9.1.1.5 Concatenation Rules
    - 9.1.2 Text Blocks since Java 15
        - 9.1.2.1 Formatting Essential vs Incidental Whitespace
        - 9.1.2.2 Line Count Blank Lines and Line Breaks
        - 9.1.2.3 Text Blocks and Escape Characters
        - 9.1.2.4 Common Errors with fixes
- 9.2 Core String Methods
    - 9.2.1 String Indexing
    - 9.2.2 length Method
    - 9.2.3 Boundary Rules Start Index vs End Index
    - 9.2.4 Methods Using Only Start Index Inclusive
    - 9.2.5 Methods with Start Inclusive End Exclusive
    - 9.2.6 Methods That Operate on Entire String
    - 9.2.7 Character Access
    - 9.2.8 Searching
    - 9.2.9 Replacement Methods
    - 9.2.10 Splitting and Joining
    - 9.2.11 Methods Returning Arrays
    - 9.2.12 Indentation
    - 9.2.13 Additional Examples

9.1 Strings & Text Blocks

9.1.1 Strings

9.1.1.1 Initializing Strings

In Java, a **String** is an object of the `java.lang.String` class, used to represent a sequence of characters.

Strings are **immutable**: once created, their content cannot be changed. Any operation that seems to modify a string actually creates a new one.

You can create and initialize strings in several ways:

```
String s1 = "Hello";                    // string literal
String s2 = new String("Hello");        // using constructor (not recommended)
String s3 = s1.toUpperCase();           // creates a new String ("HELLO")
```

> **Note**
>
> - String literals are stored in the `String pool`, a special memory area used to avoid creating duplicate string objects.
> - Using the `new` keyword always creates a new object outside the pool.

9.1.1.2 The String Pool

Because `String` objects are immutable and widely used, they could easily occupy a large amount of memory in a Java program.

To reduce duplication, Java reuses all strings that are declared as literals (see example above), storing them in a dedicated area of the JVM known as the **String Pool** or **Intern Pool**.

Please check the Paragraph: “**6.4.3 String Pool and Equality**” in Chapter: 6. Instantiating Types for a deeper explanation and examples.

9.1.1.3 Special Characters and Escape Sequences

Strings can contain escape characters, which allow you to include special symbols or control characters (characters with a special meaning in Java).
An escape sequence starts with a backslash `\`.

> **Note**
>
> **Table of Special Characters & Escape Sequences in Strings**

Escape	Meaning	Java Example	Result
`\"`	double quote	`“She said \"Hi\"”`	`She said “Hi”`
`\\`	backslash	`“C:\\Users\\Alex”`	`C:`
`\n`	newline (LF)	`“Hello\nWorld”`	`Hello` `World`
`\r`	carriage return (CR)	`“A\rB”`	`CR before B`
`\t`	tab	`“Name\tAge”`	`Name    Age`
`\'`	single quote	`“It\'s ok”`	`It’s ok`
`\b`	backspace	`“AB\bC”`	`AC` (the `B` is removed visually)
`\uXXXX`	Unicode code unit	`“\u00A9”`	`©`

9.1.1.4 Rules for String Concatenation

As introduced in the Chapter on 5. Java Operators, the symbol `+` normally represents **arithmetic addition** when used with numeric operands.

However, when applied to **Strings**, the same operator performs **string concatenation** — it creates a new string by joining operands together.

Since the operator `+` may appear in expressions where both numbers and strings are present, Java applies a specific set of rules to determine whether `+` means *numeric addition* or *string concatenation*.

9.1.1.5 Concatenation Rules

- If both operands are numeric, `+` performs **numeric addition**.
- If at least one operand is a `String`, the `+` operator performs **string concatenation**.
- Evaluation is strictly left-to-right, because `+` is **left-associative**.

This means that once a `String` appears on the left side of the expression, all subsequent `+` operations become concatenations.

> **Tip**
>
> Because evaluation is left-to-right, the position of the first `String` operand determines how the rest of the expression is evaluated.

- Examples

```
// *** Pure numeric addition

int a = 10 + 20;        // 30
double b = 1.5 + 2.3;   // 3.8



// *** String concatenation when at least one operand is a String

String s = "Hello" + " World";  // "Hello World"
String t = "Value: " + 10;      // "Value: 10"



// *** Left-to-right evaluation affects the result

System.out.println(1 + 2 + " apples");
// 3 + " apples"  → "3 apples"

System.out.println("apples: " + 1 + 2);
// "apples: 1" + 2 → "apples: 12"



// *** Adding parentheses changes the meaning

System.out.println("apples: " + (1 + 2));
// parentheses force numeric addition → "apples: 3"



// *** Mixed types with multiple operands

String result = 10 + 20 + "" + 30 + 40;
// (10 + 20) = 30
// 30 + ""  = "30"
// "30" + 30 = "3030"
String out = "3030" + 40; // "303040"

System.out.println(1 + 2 + "3" + 4 + 5);
// Step 1: 1 + 2 = 3
// Step 2: 3 + "3" = "33"
String r = "33" + 4;  // "334"
// Step 4: "334" + 5 = "3345"



// *** null is represented as a string when concatenated

System.out.println("AB" + null);
// ABnull
```

9.1.2 Text Blocks (since Java 15)

A text block is a multi-line string literal introduced to simplify writing large strings (such as HTML, JSON, or code) without the need for many escape sequences.

A text block starts and ends with three double quotes (`"""`).

You can use text blocks everywhere you would use strings.

```
String html = """
    <html>
        <body>
            <p>Hello, world!</p>
        </body>
    </html>
    """;
```

Note

- Text blocks automatically include line breaks and indentation for readability. Newlines are normalized to `\n`.
- Double quotes inside the block usually don't need escaping.
- The compiler interprets the content between the opening and closing triple quotes as the string's value.

9.1.2.1 Formatting: Essential vs Incidental Whitespace

- **Essential whitespace**: spaces and newlines that are part of the intended string content.
- **Incidental whitespace**: indentation in your source code that you don't conceptually consider part of the text.

```
String text = """
        Line 1
        Line 2
        Line 3
        """;
```

Important

- **Leftmost character (baseline)**: the position of the first non-space character across all lines (or the closing `"""`) defines the indentation baseline. Spaces to the left of this baseline are considered incidental and are removed.
- The line immediately following the opening `"""` is not included in the output if it's empty (typical formatting).
- The newline before the closing `"""` is included in the content.
  In the example above, the resulting string ends with a newline after `"Line 3"`: there are 4 lines in total.

Output with line numbers (showing the trailing blank line):

```
1: Line 1
2: Line 2
3: Line 3
4:
```

To suppress the trailing newline:

- Use a line-continuation backslash at the end of the last content line.
- Put the ending triple quotes on the same line as the last content.

```
String textNoTrail_1 = """
        Line 1
        Line 2
        Line 3\
        """;

// OR

String textNoTrail_2 = """
        Line 1
        Line 2
        Line 3""";
```

9.1.2.2 Line Count, Blank Lines, and Line Breaks

- Every visible line break inside the block becomes `\n`.
- Blank lines inside the block are preserved.

```
String textNoTrail_0 = """
        Line 1
        Line 2 \n
        Line 3

        Line 4
        """;
```

Output:

```
1: Line 1
2: Line 2
3:
4: Line 3
5:
6: Line 4
7:
```

9.1.2.3 Text Blocks & Escape Characters

Escape sequences still work inside text blocks when needed (for example, for backslashes or explicit control characters).

```
String json = """
    {
      "name": "Alice",
      "path": "C:\\\\Users\\\\Alice"
    }\
    """;
```

You can also format a text block using placeholders and `formatted()`:

```
String card = """
    Name: %s
    Age:  %d
    """.formatted("Alice", 30);
```

9.1.2.4 Common Errors (with fixes)

```
// ❌ Mismatched delimiters / missing closing triple quote
String bad = """
  Hello
World";      // ERROR — not a closing text block

// ✅ Fix
String ok = """
  Hello
  World
  """;
```

```java
// ❌ Text blocks require a line break after the opening """
String invalid = """Hello""";  // ERROR

// ✅ Fix
String valid = """
    Hello
    """;
```

```java
// ❌ Unescaped trailing backslash at end of a line inside the block
String wrong = """
    C:\Users\Alex\     // ERROR — backslash escapes the newline
    Documents
    """;

// ✅ Fix: escape backslashes, or avoid backslash at end of line
String correct = """
    C:\\Users\\Alex\\
    Documents\
    """;
```

9.2 Core String Methods

9.2.1 String Indexing

Strings in Java use **zero-based indexing**, meaning:

- The first character is at index `0`
- The last character is at index `length() - 1`
- Accessing any index outside this range causes a `StringIndexOutOfBoundsException`
- Example:

```java
String s = "Java";
// Indexes:  0    1    2    3
// Chars:    J    a    v    a

char c = s.charAt(2); // 'v'
```

9.2.2 `length()` Method

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

```java
String s = "hello";
System.out.println(s.length());  // 5
```

The last valid index is always `length() - 1`.

9.2.3 Boundary Rules: Start Index vs End Index

Many String methods use two indices:

- **Start index** — inclusive
- **End index** — exclusive

In other words, `substring(start, end)` includes characters from index `start` up to, but not including, index `end`.

- Start index must be `>= 0` and `<= length() - 1`
- End index may be equal to `length()` (the "virtual" position after the last character).
- End index must not exceed `length()`.
- Start index must never be greater than end index.
- Example:

```
String s = "abcdef";
s.substring(1, 4); // "bcd" (indexes 1,2,3)
```

This rule applies to most substring-based methods.

9.2.4 Methods Using Only Start Index (Inclusive)

Method	Description	Parameters	Index Rule	Example
substring(int start)	Returns substring from start to end	start	start inclusive	"abcdef".substring(2) → "cdef"
indexOf(String)	First occurrence	—	—	"Java".indexOf("a") → 1
indexOf(String, start)	Start searching at index	start	start inclusive	"banana".indexOf("a", 2) → 3
lastIndexOf(String)	Last occurrence	—	—	"banana".lastIndexOf("a") → 5
lastIndexOf(String, fromIndex)	Search backward from index	fromIndex	fromIndex inclusive	"banana".lastIndexOf("a", 3) → 3

9.2.5 Methods with Start Inclusive / End Exclusive

These methods follow the same slicing behavior: `start` included, `end` excluded.

Method	Description	Signature	Example
substring(start, end)	Extracts part of string	(int start, int end)	"abcdef".substring(1,4) → "bcd"
regionMatches	Compares substring regions	(toffset, other, ooffset, len)	"Hello".regionMatches(1, "ell", 0, 3) → true
getBytes(int srcBegin, int srcEnd, byte[] dst, int dstBegin)	Copies chars to byte array	start inclusive, end exclusive	Copies chars in [srcBegin, srcEnd)
copyValueOf(char[] data, int offset, int count)	Creates a new string	offset inclusive; offset+count exclusive	Same rule as substring

9.2.6 Methods That Operate on Entire String

Method	Description	Example
toUpperCase()	Uppercase version	"java".toUpperCase() → "JAVA"
toLowerCase()	Lowercase version	"JAVA".toLowerCase() → "java"
trim()	Removes leading/trailing whitespace	" hi ".trim() →"hi"
strip()	Unicode-aware trimming	" hi003".strip() → "hi"
stripLeading()	Removes leading whitespace	" hi".stripLeading() → "hi"
stripTrailing()	Removes trailing whitespace	"hi".stripTrailing() → "hi"
isBlank()	True if empty or whitespace only	" ".isBlank() → true
isEmpty()	True if length == 0	"".isEmpty() → true

9.2.7 Character Access

Method	Description	Example
charAt(int index)	Returns char at index	“Java”.charAt(2) → ‘v’
codePointAt(int index)	Returns Unicode code point	Useful for emojis or characters beyond BMP

9.2.8 Searching

Method	Description	Example
contains(CharSequence)	Substring test	“hello”.contains(“ell”) → true
startsWith(String)	Prefix	“abcdef”.startsWith(“abc”) → true
startsWith(String, offset)	Prefix at index	“abc”.startsWith(“b”, 1) → true
endsWith(String)	Suffix	“abcdef”.endsWith(“def”) → true

9.2.9 Replacement Methods

Method	Description	Example
replace(char old, char new)	Replace characters	“banana”.replace(‘a’,‘o’) → “bonono”
replace(CharSequence old, CharSequence new)	Replace substrings	“ababa”.replace(“aba”,“X”) → “Xba”
replaceAll(String regex, String replacement)	Regex replace all	“a1a2”.replaceAll(“\d”,“”) →”aa”
replaceFirst(String regex, String replacement)	First regex match only	“a1a2”.replaceFirst(“\d”,“”) →”aa2”

9.2.10 Splitting and Joining

Method	Description	Example
split(String regex)	Split by regex	“a,b,c”.split(“,”) → [“a”,“b”,“c”]
split(String regex, int limit)	Split with limit	limit < 0 keeps all trailing empty strings

9.2.11 Methods Returning Arrays

Method	Description	Example
toCharArray()	Returns char[]	“abc”.toCharArray()
getBytes()	Returns byte[] using platform/default encoding	“á”.getBytes()

9.2.12 Indentation

Method	Description	Example
indent(int numSpaces)	Adds (positive) or removes (negative) spaces from the beginning of each line; also adds a line break at the end if not already present	str.indent(-20)
stripIndent()	Removes all incidental leading whitespace from each line; does not add a final line break	str.stripIndent()

- Example:

```
var txtBlock = """

                    a
                      b
                     c""";

var conc = " a\n" + " b\n" + " c";

System.out.println("length: " + txtBlock.length());
System.out.println(txtBlock);
System.out.println("");
String stripped1 = txtBlock.stripIndent();
System.out.println(stripped1);
System.out.println("length: " + stripped1.length());

System.out.println("*********************");

System.out.println("length: " + conc.length());
System.out.println(conc);
System.out.println("");
String stripped2 = conc.stripIndent();
System.out.println(stripped2);
System.out.println("length: " + stripped2.length());
```

Output:

```
length: 9

a
  b
 c


a
  b
 c
length: 9
*********************
length: 8
 a
 b
 c

a
b
c
length: 5
```

9.2.13 Additional Examples

- Example 1 — Extract `[start, end)`

```
String s = "012345";
System.out.println(s.substring(2, 5));
// includes 2,3,4 → prints "234"
```

- Example 2 — Searching from a start index

```
String s = "hellohello";
int idx = s.indexOf("lo", 5); // search begins at index 5
```

- Example 3 — Common pitfalls

```
String s = "abcd";
System.out.println(s.substring(1,1)); // "" empty string
System.out.println(s.substring(3, 2)); // ❌ Exception: start index (3) > end index (2)

System.out.println("abcd".substring(2, 4)); // "cd" — includes indexes 2 and 3; 4 is excluded

System.out.println("abcd".substring(2, 5)); // ❌ StringIndexOutOfBoundsException (end index
```

◀ 8. Looping Constructs in Java | ▲ Index | 10. Arrays in Java ▶

10. Arrays in Java

Table of Contents

- 10.1 What an Array Is
  - 10.1.1 Declaring Arrays
  - 10.1.2 Creating Arrays Instantiation
  - 10.1.3 Default Values in Arrays
  - 10.1.4 Accessing Elements
  - 10.1.5 Array Initialization Shorthands
    - 10.1.5.1 Anonymous Array Creation
    - 10.1.5.2 Short Syntax Only at Declaration
- 10.2 Multidimensional Arrays Arrays of Arrays
  - 10.2.1 Creating a Rectangular Array
  - 10.2.2 Creating a Jagged Irregular Array
- 10.3 Array Length vs String Length
- 10.4 Array Reference Assignments
  - 10.4.1 Assigning Compatible References
  - 10.4.2 Incompatible Assignments Compile-Time Errors
  - 10.4.3 Covariance Runtime Danger ArrayStoreException
- 10.5 Comparing Arrays
- 10.6 Arrays Utility Methods
  - 10.6.1 Arrays.toString
  - 10.6.2 Arrays.deepToString for Nested Arrays
  - 10.6.3 Arrays.sort
  - 10.6.4 Arrays.binarySearch
  - 10.6.5 Arrays.compare
- 10.7 Enhanced for-loop with Arrays
- 10.8 Common Pitfalls
- 10.9 Summary

10.1 What an Array Is

Arrays in Java are **fixed-size**, **indexed**, **ordered** collections of elements of the same type.

They are **objects**, even when the elements are primitives.

10.1.1 Declaring Arrays

You can declare an array in two ways:

```
int[] a;      // preferred modern syntax
int b[];      // legal, older style
String[] names;
Person[] people;

// [] can be before or after the name: all the following declarations are equivalent.

int[] x;
int [] x1;
int []x2;
int x3[];
int x5 [];

// MULTIPLE ARRAY DECLARATIONS

int[] arr1, arr2;   // Declares two arrays of int

// WARNING:
// Here arr1 is an int[] and arr2 is just an int (NOT an array!)
int arr1[], arr2;
```

Declaring does NOT create the array — it only creates a variable capable of referencing one.

10.1.2 Creating Arrays (Instantiation)

An array is created using `new` followed by the element type and the array length:

```
int[] numbers = new int[5];
String[] words = new String[3];
```

Key rules - The length must be non-negative and specified at creation time. - The length cannot be changed later. - The array length can be any `int` expression.

```
int size = 4;
double[] values = new double[size];
```

- Illegal array creation examples:

```
// int length = -1;
// int[] arr = new int[-1];    // Runtime: NegativeArraySizeException

// int[] arr = new int[2.5];   // Compile error: size must be int
```

10.1.3 Default Values in Arrays

Arrays (because they are objects) always receive **default initialization**:

Element Type	Default Value
Numeric	0
boolean	false
char	'000'
Reference types	null

- Example:

```
int[] nums = new int[3];
System.out.println(nums[0]); // 0

String[] s = new String[3];
System.out.println(s[0]);     // null
```

10.1.4 Accessing Elements

Elements are accessed using zero-based indexing:

```
int[] a = new int[3];
a[0] = 10;
a[1] = 20;
System.out.println(a[1]); // 20
```

Common exception
- `ArrayIndexOutOfBoundsException` (runtime)

```
// int[] x = new int[2];
// System.out.println(x[2]); // ❌ index 2 out of bounds
```

10.1.5 Array Initialization Shorthands

10.1.5.1 Anonymous Array Creation

```
int[] a = new int[] {1,2,3};
```

10.1.5.2 Short Syntax (Only at Declaration)

```
int[] b = {1,2,3};
```

> The short syntax `{1,2,3}` can only be used at the point of declaration.

```
// int[] c;
// c = {1,2,3};   // ❌ does not compile
```

10.2 Multidimensional Arrays (Arrays of Arrays)

Java implements multi-dimensional arrays as **arrays of arrays**.

Declaration:

```
int[][] matrix;
String[][][] cube;
```

10.2.1 Creating a Rectangular Array

```
int[][] rect = new int[3][4]; // 3 rows, 4 columns each
```

10.2.2 Creating a Jagged (Irregular) Array

You can create rows with different lengths:

```
int[][] jagged = new int[3][];
jagged[0] = new int[2];
jagged[1] = new int[5];
jagged[2] = new int[1];
```

10.3 Array Length vs String Length

- Arrays use `.length` (public final field).
- Strings use `.length()` (method).

> **Tip**
>
> This is a classic trap: fields vs methods.

```java
// int x = arr.length;   // OK
// int y = s.length;     // ❌ does not compile: missing ()
int yOk = s.length();
```

10.4 Array Reference Assignments

10.4.1 Assigning Compatible References

```java
int[] a = {1,2,3};
int[] b = a; // both now point to the same array
```

Modifying one reference affects the other:

```java
b[0] = 99;
System.out.println(a[0]); // 99
```

10.4.2 Incompatible Assignments (Compile-Time Errors)

```java
// int[] x = new int[3];
// long[] y = x;      // ❌ incompatible types
```

Array references follow normal inheritance rules:

```java
String[] s = new String[3];
Object[] o = s;       // OK: arrays are covariant
```

10.4.3 Covariance Runtime Danger: ArrayStoreException

```java
Object[] objs = new String[3];
// objs[0] = Integer.valueOf(5); // ❌ ArrayStoreException at runtime
```

10.5 Comparing Arrays

`==` compares references (identity):

```java
int[] a = {1,2};
int[] b = {1,2};
System.out.println(a == b); // false
```

`equals()` on arrays does not compare contents (it behaves like `==`):

```java
System.out.println(a.equals(b)); // false
```

To compare contents, use methods from `java.util.Arrays`:

```java
Arrays.equals(a, b);         // shallow comparison
Arrays.deepEquals(o1, o2);   // deep comparison for nested arrays
```

10.6 `Arrays` Utility Methods

10.6.1 `Arrays.toString()`

```java
System.out.println(Arrays.toString(new int[]{1,2,3})); // [1, 2, 3]
```

10.6.2 `Arrays.deepToString()` (for nested arrays)

```
System.out.println(Arrays.deepToString(new int[][] {{1,2},{3,4}}));
// [[1, 2], [3, 4]]
```

10.6.3 `Arrays.sort()`

```
int[] a = {4,1,3};
Arrays.sort(a); // [1, 3, 4]
```

> **Tip**
>
> - Strings are sorted in natural (lexicographic) order.
> - Numbers sort before letters, and uppercase letters sort before lowercase letters (numbers < uppercase < lowercase).
> - For reference types, `null` is considered smaller than any non-null value.

```
String[] arr = {"AB", "ac", "Ba", "bA", "10", "99"};

Arrays.sort(arr);

System.out.println(Arrays.toString(arr));  // [10, 99, AB, Ba, ac, bA]
```

10.6.4 `Arrays.binarySearch()`

Requirements: the array must be sorted using the same ordering; otherwise the result is unpredictable.

```
int[] a = {1,3,5,7};
int idx = Arrays.binarySearch(a, 5); // returns 2
```

When the value is not found, `binarySearch` returns `-(insertionPoint) - 1`:

```
int pos = Arrays.binarySearch(a, 4); // returns -3
// Insertion point is index 2 → -(2) - 1 = -3
```

10.6.5 `Arrays.compare()`

The class `Arrays` offers an overloaded `equals()` that checks if two arrays contain the same elements (and have the same length):

```
System.out.println(Arrays.equals(new int[] {200}, new int[] {100}));        // false
System.out.println(Arrays.equals(new int[] {200}, new int[] {200}));        // true
System.out.println(Arrays.equals(new int[] {200}, new int[] {100, 200}));   // false
```

It also provides a `compare()` method with these rules:

- If the result `n < 0` → the first array is considered “smaller” than the second.
- If the result `n > 0` → the first array is considered “greater” than the second.
- If the result `n == 0` → the arrays are equal.
- Examples:

```
int[] arr1 = new int[] {200, 300};
int[] arr2 = new int[] {200, 300, 400};
System.out.println(Arrays.compare(arr1, arr2));  // -1

int[] arr3 = new int[] {200, 300, 400};
int[] arr4 = new int[] {200, 300};
System.out.println(Arrays.compare(arr3, arr4));  // 1

String[] arr5 = new String[] {"200", "300", "aBB"};
String[] arr6 = new String[] {"200", "300", "ABB"};
System.out.println(Arrays.compare(arr5, arr6));     // Positive: "aBB" > "ABB"

String[] arr7 = new String[] {"200", "300", "ABB"};
String[] arr8 = new String[] {"200", "300", "aBB"};
System.out.println(Arrays.compare(arr7, arr8));     // Negative: "ABB" < "aBB"

String[] arr9 = null;
String[] arr10 = new String[] {"200", "300", "ABB"};
System.out.println(Arrays.compare(arr9, arr10));    // -1 (null considered smaller)
```

10.7 Enhanced for-loop with Arrays

```
for (int value : new int[]{1,2,3}) {
    System.out.println(value);
}
```

Rules - The right side must be an array or an `Iterable`. - The loop variable type must be compatible with the element type (no primitive widening here).

Common error:

```
// for (long v : new int[]{1,2}) {} // ❌ not allowed: int elements cannot be assigned to long
```

10.8 Common Pitfalls

- **Accessing out of bounds** → throws `ArrayIndexOutOfBoundsException`.
- **Using short array initializer incorrectly**

```
// int[] x;
// x = {1,2}; // ❌ does not compile
```

- **Confusing `.length` and `.length()`**
- **Forgetting that arrays are objects** (they live on the heap and are referenced).
- **Mixing primitive arrays and wrapper arrays**

```
// int[] p = new Integer[3]; // ❌ incompatible
```

- **Using `binarySearch` on unsorted arrays** → unpredictable results.
- **Covariant array runtime exceptions** (`ArrayStoreException`).

10.9 Summary

Arrays in Java are:

- Objects (even if they hold primitives).
- Fixed-size, indexed collections.
- Always initialized with default values.

- Type-safe, but subject to covariance rules (which can cause runtime exceptions if misused).

◀ 9. Strings in Java | ▲ Index | 11. Math in Java ▶

11. Math in Java

Table of Contents

- 11.1 Math APIs
  - 11.1.1 Maximum and Minimum Between Two Values
  - 11.1.2 Math.round
  - 11.1.3 Math.ceil Ceiling
  - 11.1.4 Math.floor Floor
  - 11.1.5 Math.pow
  - 11.1.6 Math.random
  - 11.1.7 Math.abs
  - 11.1.8 Math.sqrt
  - 11.1.9 Summary Table
- 11.2 BigInteger and BigDecimal
  - 11.2.1 Why double and float Are Not Enough
  - 11.2.2 BigInteger — Arbitrary-Precision Integers
  - 11.2.3 Creating BigInteger
  - 11.2.4 Operations No Operators

11.1 Math APIs

The `java.lang.Math` class provides a set of static methods useful for numerical operations.

These methods work with primitive numeric types.

Below is a summary of the most frequently used ones, together with their overloaded forms.

11.1.1 Maximum and Minimum Between Two Values

`Math.max()` and `Math.min()` compare the two provided values and return the maximum or minimum between them.

There are four overloaded versions for each method:

```
public static int min(int x, int y);
public static float min(float x, float y);
public static long min(long x, long y);
public static double min(double x, double y);

public static int max(int x, int y);
public static float max(float x, float y);
public static long max(long x, long y);
public static double max(double x, double y);
```

- Example:

```
System.out.println(Math.max(10.50, 7.5));   // 10.5
System.out.println(Math.min(10, -20));      // -20
```

11.1.2 `Math.round()`

`round()` returns the nearest integer to its argument, following standard rounding rules:
values with fractional part 0.5 and above are rounded up; below 0.5 are rounded down (toward the nearest integer).

Overloads

- `long round(double value)`
- `int round(float value)`
- Examples:

```
Math.round(3.2);    // 3   (returns long)
Math.round(3.6);    // 4
Math.round(-3.5f);  // -3  (float version returns int)
```

> **Note**
>
> - The float version returns an `int`.
> - The double version returns a `long`.

11.1.3 `Math.ceil()` (Ceiling)

`ceil()` returns the smallest `double` value that is greater than or equal to the argument.

Overloads

- `double ceil(double value)`
- Examples:

```
Math.ceil(3.1);   // 4.0
Math.ceil(-3.1);  // -3.0
```

11.1.4 `Math.floor()` (Floor)

`floor()` returns the largest `double` value that is less than or equal to the argument.

Overloads

- `double floor(double value)`
- Examples:

```
Math.floor(3.9);   // 3.0
Math.floor(-3.1);  // -4.0
```

11.1.5 `Math.pow()`

`pow()` raises a value to a power.

Overloads

- `double pow(double base, double exponent)`
- Examples:

```
Math.pow(2, 3);      // 8.0
Math.pow(9, 0.5);    // 3.0  (square root)
Math.pow(10, -1);    // 0.1
```

11.1.6 `Math.random()`

`random()` returns a `double` in the range `[0.0, 1.0)` (0.0 inclusive, 1.0 exclusive).

Overloads

- `double random()`
- Examples:

```
double r = Math.random();   // 0.0 <= r < 1.0

// Example: random int 0-9
int x = (int)(Math.random() * 10);
```

11.1.7 `Math.abs()`

`abs()` returns the absolute value (distance from zero).

Overloads

- `int abs(int value)`
- `long abs(long value)`
- `float abs(float value)`
- `double abs(double value)`

11.1.8 `Math.sqrt()`

`sqrt()` computes the square root and returns a `double`.

```
Math.sqrt(9);     // 3.0
Math.sqrt(-1);    // NaN (not a number)
```

11.1.9 Summary Table

Method	Returns	Overloads	Notes
`round()`	int or long	float, double	Nearest integer
`ceil()`	double	double	Smallest value >= argument
`floor()`	double	double	Largest value <= argument
`pow()`	double	double, double	Exponentiation
`random()`	double	none	0.0 <= r < 1.0
`min()/max()`	same type	int, long, float, double	Compare two values
`abs()`	same type	int, long, float, double	Absolute value
`sqrt()`	double	double	Square root

11.2 BigInteger and BigDecimal

The classes `BigInteger` and `BigDecimal` (in `java.math`) provide arbitrary-precision number types.

They are used when:

- Primitive types (`int`, `long`, `double`, etc.) don't have enough range.
- Floating-point rounding errors of `float` / `double` are not acceptable (for example, in financial calculations).

Both are **immutable**: every operation returns a new instance.

11.2.1 Why `double` and `float` Are Not Enough

Floating-point types (`float`, `double`) use a binary representation. Many decimal fractions can't be represented exactly (like 0.1 or 0.2), so you get rounding errors:

```
System.out.println(0.1 + 0.2); // 0.30000000000000004
```

For tasks like financial calculations, this is unacceptable.

`BigDecimal` solves this by representing numbers using a decimal model with a configurable scale (number of digits after the decimal point).

11.2.2 BigInteger — Arbitrary-Precision Integers

`BigInteger` represents integer values of virtually any size, limited only by available memory.

11.2.3 Creating BigInteger

Common ways:

From a long

```
static BigInteger valueOf(long val);
```

From a String

```
BigInteger(String val);         // decimal by default
BigInteger(String val, int radix);
```

Random big value

```
BigInteger(int numBits, Random rnd);
```

- Examples:

```
import java.math.BigInteger;
import java.math.BigDecimal;
import java.util.Random;

BigInteger a = BigInteger.valueOf(10L);

// You can pass a long to both types, but a double only to BigDecimal

BigInteger g = BigInteger.valueOf(3000L);
BigDecimal p = BigDecimal.valueOf(3000L);
BigDecimal q = BigDecimal.valueOf(3000.00);

BigInteger b = new BigInteger("12345678901234567890"); // decimal string
BigInteger c = new BigInteger("FF", 16);               // 255 in base 16
BigInteger r = new BigInteger(128, new Random());      // random 128-bit number
```

11.2.4 Operations (No Operators!)

You cannot use the standard arithmetic operators (`+` , `-` , `*` , `/` , `%`) with `BigInteger` or `BigDecimal` .

Instead, you must call methods (all of which return new instances). Here are some common ones for `BigInteger` :

- `add(BigInteger val)`
- `subtract(BigInteger val)`
- `multiply(BigInteger val)`
- `divide(BigInteger val)` – integer division
- `remainder(BigInteger val)`
- `pow(int exponent)`
- `negate()`
- `abs()`
- `gcd(BigInteger val)`
- `compareTo(BigInteger val)` – ordering
- Example:

```java
BigInteger x = new BigInteger("100000000000000000000");
BigInteger y = new BigInteger("3");

BigInteger sum = x.add(y);         // x + y
BigInteger prod = x.multiply(y);  // x * y
BigInteger div = x.divide(y);      // integer division
BigInteger rem = x.remainder(y);  // modulus

if (x.compareTo(y) > 0) {
    System.out.println("x is larger");
}
```

◀ 10. Arrays in Java | ▲ Index | 12. Date and Time in Java ▶

12. Date and Time in Java

Table of Contents

- 12.1 Date and Time
    - 12.1.1 Creating Specific Dates and Times
    - 12.1.2 Date and Time Arithmetic plus and minus Methods
    - 12.1.3 Common Patterns
    - 12.1.4 LocalDate Arithmetic
    - 12.1.5 LocalTime Arithmetic
    - 12.1.6 LocalDateTime Arithmetic
    - 12.1.7 ZonedDateTime Arithmetic
    - 12.1.8 Summary Table
- 12.2 withXxx Methods
- 12.3 Conversion and at Methods Linking Date Time and Zone
- 12.4 Period Duration and Instant
- 12.5 Period — Human Date Amounts
- 12.6 Duration — Machine Time Amounts
- 12.7 Instant — Point on the UTC Timeline
- 12.8 Summary Table Period vs Duration vs Instant
- 12.9 TemporalUnit and TemporalAmount
    - 12.9.1 TemporalUnit
    - 12.9.2 ChronoUnit enum
    - 12.9.3 TemporalAmount
    - 12.9.4 Period as a TemporalAmount
    - 12.9.5 Duration as a TemporalAmount
    - 12.9.6 Using TemporalAmount vs TemporalUnit
    - 12.9.7 between Methods
    - 12.9.8 Common Pitfalls
    - 12.9.9 Summary

12.1 Date and Time

Java provides a modern, consistent, immutable date/time API in the package `java.time.*`.

This API replaces the old `java.util.Date` and `java.util.Calendar` classes.

Depending on the level of detail required, Java offers four main classes:

- `LocalDate` → represents a date only (year–month–day)
- `LocalTime` → represents a time only (hour–minute–second–nanosecond)
- `LocalDateTime` → combines date + time, but no time zone
- `ZonedDateTime` → full date + time + offset + time zone

> **Note**
>
> - A **time zone** defines rules such as daylight saving changes (for example, `Europe/Paris`).
> - A **zone offset** is a fixed shift from UTC/GMT (for example, `+01:00`, `-07:00`).
> - To compare two instants from different time zones, convert them to UTC (GMT) by applying the offset.

Getting the Current Date/Time

You can retrieve the current system values using the static `now()` methods:

```
System.out.println(LocalDate.now());
System.out.println(LocalTime.now());
System.out.println(LocalDateTime.now());
System.out.println(ZonedDateTime.now());
```

- Example output (your system may differ):

```
2025-12-01
19:11:53.213856300
2025-12-01T19:11:53.213856300
2025-12-01T19:11:53.214856900+01:00[Europe/Paris]
```

- Example: converting `ZonedDateTime` to GMT (UTC)

```
// Conceptual examples (not real code, just illustrating offsets):
// 2024-07-01T12:00+09:00[Asia/Tokyo]        ---> 12:00 minus 9 hours ---> 03:00 UTC
// 2024-07-01T20:00-07:00[America/Los_Angeles] ---> 20:00 plus 7 hours ---> 03:00 UTC
```

Both represent the same instant in time, simply expressed in different time zones.

12.1.1 Creating Specific Dates and Times

You can build precise date/time objects using the `of()` factory methods.

All classes include multiple overloaded versions of `of()` (only the most common are listed here).

LocalDate — overloaded `of()` forms

- `of(int year, int month, int dayOfMonth)`
- `of(int year, Month month, int dayOfMonth)`

LocalTime — overloaded `of()` forms

- `of(int hour, int minute)`
- `of(int hour, int minute, int second)`
- `of(int hour, int minute, int second, int nanoOfSecond)`

LocalDateTime — overloaded `of()` forms

- `of(int year, int month, int day, int hour, int minute)`
- `of(int year, int month, int day, int hour, int minute, int second)`
- `of(int year, int month, int day, int hour, int minute, int second, int nano)`
- `of(LocalDate date, LocalTime time)`

ZonedDateTime — overloaded `of()` forms

- `of(LocalDate date, LocalTime time, ZoneId zone)`
- `of(int y, int m, int d, int h, int min, int s, int nano, ZoneId zone)`
- Examples

```java
// Creating specific dates

var localDate1 = LocalDate.of(2025, 7, 31);
var localDate2 = LocalDate.of(2025, Month.JULY, 31);

// Creating specific times

var localTime1 = LocalTime.of(13, 21);
System.out.println(localTime1);                      // 13:21
System.out.println(LocalTime.of(13, 21, 52));        // 13:21:52
System.out.println(LocalTime.of(13, 21, 52, 200));   // 13:21:52.000000200

// Creating LocalDateTime

var localDateTime1 = LocalDateTime.of(2025, 7, 31, 13, 55, 22);
var localDateTime2 = LocalDateTime.of(localDate1, localTime1);

// Creating a ZonedDateTime

var zoned = ZonedDateTime.of(2025, 7, 31, 13, 55, 22, 0, ZoneId.of("Europe/Paris"));
```

12.1.2 Date and Time Arithmetic: `plus` and `minus` Methods

All classes in the `java.time` package (such as `LocalDate`, `LocalTime`, `LocalDateTime`, `ZonedDateTime`, etc.) are **immutable**.

This means that methods like `plusXxx()` and `minusXxx()` never modify the original object — instead, they return a new instance with the adjusted value.

12.1.3 Common Patterns

Most date/time classes support three kinds of arithmetic methods:

- **Type-specific shortcuts**
- `plusDays(long daysToAdd)`
- `plusHours(long hoursToAdd)`
- etc.
- **Generic amount-based methods**
- `plus(TemporalAmount amount)` → for example `Period`, `Duration`
- `minus(TemporalAmount amount)`
- **Generic unit-based methods**
- `plus(long amountToAdd, TemporalUnit unit)`
- `minus(long amountToSubtract, TemporalUnit unit)`

These allow flexible and readable date/time arithmetic.

12.1.4 `LocalDate` Arithmetic

`LocalDate` represents a date only (no time, no zone).

Main `plus` / `minus` methods (overloads)

Method	Description
`plusDays(long days)`	Add days
`plusWeeks(long weeks)`	Add weeks
`plusMonths(long months)`	Add months
`plusYears(long years)`	Add years
`minusDays(long days)`	Subtract days
`minusWeeks(long weeks)`	Subtract weeks
`minusMonths(long months)`	Subtract months
`minusYears(long years)`	Subtract years
`plus(TemporalAmount amount)`	Add a Period
`minus(TemporalAmount amount)`	Subtract a Period
`plus(long amountToAdd, TemporalUnit unit)`	Add using ChronoUnit (e.g., DAYS, MONTHS)
`minus(long amountToSubtract, TemporalUnit unit)`	Subtract using ChronoUnit

- Examples:

```
LocalDate date = LocalDate.of(2025, 3, 10);

LocalDate d1 = date.plusDays(5);           // 2025-03-15
LocalDate d2 = date.minusWeeks(2);         // 2025-02-24
LocalDate d3 = date.plusMonths(1);         // 2025-04-10
LocalDate d4 = date.plusYears(2);          // 2027-03-10

// Using ChronoUnit
LocalDate d5 = date.plus(10, ChronoUnit.DAYS);   // 2025-03-20

// Using Period
Period p = Period.of(1, 2, 3);  // 1 year, 2 months, 3 days
LocalDate d6 = date.plus(p);
```

12.1.5 `LocalTime` Arithmetic

`LocalTime` represents time only (no date, no zone).

Main `plus` / `minus` methods (overloads)

Method	Description
`plusNanos(long nanos)`	Add nanoseconds
`plusSeconds(long seconds)`	Add seconds
`plusMinutes(long minutes)`	Add minutes
`plusHours(long hours)`	Add hours
`minusNanos(long nanos)`	Subtract nanoseconds
`minusSeconds(long seconds)`	Subtract seconds
`minusMinutes(long minutes)`	Subtract minutes
`minusHours(long hours)`	Subtract hours
`plus(TemporalAmount amount)`	Add a Duration
`minus(TemporalAmount amount)`	Subtract a Duration
`plus(long amountToAdd, TemporalUnit unit)`	Add using ChronoUnit
`minus(long amountToSubtract, TemporalUnit unit)`	Subtract using ChronoUnit

- Examples

```
LocalTime time = LocalTime.of(13, 30);       // 13:30

LocalTime t1 = time.plusHours(2);            // 15:30
LocalTime t2 = time.minusMinutes(45);        // 12:45
LocalTime t3 = time.plusSeconds(90);         // 13:31:30

// Using ChronoUnit
LocalTime t4 = time.plus(3, ChronoUnit.HOURS);    // 16:30

// Using Duration
Duration d = Duration.ofMinutes(90);
LocalTime t5 = time.plus(d);                // 15:00
```

> **Note**
>
> When time arithmetic crosses midnight, the date is ignored with `LocalTime`. For example, 23:30 + 2 hours = 01:30 (with no date involved).

12.1.6 `LocalDateTime` Arithmetic

`LocalDateTime` combines date + time, but still no time zone.

It supports both the date-related and time-related shortcut methods.

Main `plus` / `minus` methods (overloads)

Method	Description
`plusYears(long years)` / `minusYears(long years)`	Adjust years
`plusMonths(long months)` / `minusMonths(long months)`	Adjust months
`plusWeeks(long weeks)` / `minusWeeks(long weeks)`	Adjust weeks
`plusDays(long days)` / `minusDays(long days)`	Adjust days
`plusHours(long hours)` / `minusHours(long hours)`	Adjust hours
`plusMinutes(long minutes)` / `minusMinutes(long minutes)`	Adjust minutes
`plusSeconds(long seconds)` / `minusSeconds(long seconds)`	Adjust seconds
`plusNanos(long nanos)` / `minusNanos(long nanos)`	Adjust nanoseconds
`plus(TemporalAmount amount)` / `minus(TemporalAmount amount)`	Add/subtract Period or Duration
`plus(long amountToAdd, TemporalUnit unit)` / `minus(long amountToSubtract, TemporalUnit unit)`	Using ChronoUnit

- Examples

```
LocalDateTime ldt = LocalDateTime.of(2025, 3, 10, 13, 30); // 2025-03-10T13:30

LocalDateTime l1 = ldt.plusDays(1);           // 2025-03-11T13:30
LocalDateTime l2 = ldt.minusHours(3);         // 2025-03-10T10:30
LocalDateTime l3 = ldt.plusMinutes(90);       // 2025-03-10T15:00

// Using ChronoUnit
LocalDateTime l4 = ldt.plus(2, ChronoUnit.WEEKS); // 2025-03-24T13:30

// Using Period and Duration
Period p = Period.ofDays(10);
Duration d = Duration.ofHours(5);

LocalDateTime l5 = ldt.plus(p);    // 2025-03-20T13:30
LocalDateTime l6 = ldt.plus(d);    // 2025-03-10T18:30
```

12.1.7 `ZonedDateTime` Arithmetic

`ZonedDateTime` represents date + time + time zone + offset.

It supports the same `plus` / `minus` methods as `LocalDateTime`, but with extra attention to time zones and Daylight Saving Time (DST).

Main `plus` / `minus` methods (overloads)

Method	Description
`plusYears(long years)` / `minusYears(long years)`	Adjust years
`plusMonths(long months)` / `minusMonths(long months)`	Adjust months
`plusWeeks(long weeks)` / `minusWeeks(long weeks)`	Adjust weeks
`plusDays(long days)` / `minusDays(long days)`	Adjust days
`plusHours(long hours)` / `minusHours(long hours)`	Adjust hours
`plusMinutes(long minutes)` / `minusMinutes(long minutes)`	Adjust minutes
`plusSeconds(long seconds)` / `minusSeconds(long seconds)`	Adjust seconds
`plusNanos(long nanos)` / `minusNanos(long nanos)`	Adjust nanoseconds
`plus(TemporalAmount amount)` / `minus(TemporalAmount amount)`	Period / Duration
`plus(long amountToAdd, TemporalUnit unit)` / `minus(long amountToSubtract, TemporalUnit unit)`	Using ChronoUnit

- Examples (with time zones and DST):

```
ZonedDateTime zdt = ZonedDateTime.of(
    2025, 3, 30, 1, 30, 0, 0,
    ZoneId.of("Europe/Paris")
);

// Add 2 hours across a possible DST change
ZonedDateTime z1 = zdt.plusHours(2);
System.out.println(zdt);
System.out.println(z1);
```

Depending on Daylight Saving rules for that date:

- The local time might jump from 02:00 to 03:00 or similar.
- `ZonedDateTime` adjusts the offset and local time according to the zone rules, but still represents the correct instant on the timeline.

> **Important**
>
> For `ZonedDateTime`, arithmetic is defined in terms of the local timeline and time zone rules, which can cause hour shifts during DST transitions.

12.1.8 Summary Table

Class	Shortcut plus/minus methods	Generic methods
LocalDate	plusDays, plusWeeks, plusMonths, plusYears (and minus)	plus/minus(TemporalAmount), plus/minus(long, TemporalUnit)
LocalTime	plusNanos, plusSeconds, plusMinutes, plusHours (and minus)	plus/minus(TemporalAmount), plus/minus(long, TemporalUnit)
LocalDateTime	All LocalDate + LocalTime shortcuts	plus/minus(TemporalAmount), plus/minus(long, TemporalUnit)
ZonedDateTime	Same as LocalDateTime, but zone-aware	plus/minus(TemporalAmount), plus/minus(long, TemporalUnit)

12.2 `withXxx(...)` Methods

The `with...` methods return a copy of the object with one field changed.
They never mutate the original instance.

Class	Common with... methods (not exhaustive)	Description
`LocalDate`	withYear(int year)	Same date, but with a different year
	LocalDate.withMonth(int month)	Same date, different month (1–12)
	LocalDate.withDayOfMonth(int dayOfMonth)	Same date, different day of month
	LocalDate.with(TemporalField field, long newValue)	Generic field-based adjustment
	LocalDate.with(TemporalAdjuster adjuster)	Uses an adjuster (e.g. firstDayOfMonth())
`LocalTime`	withHour(int hour)	Same time, different hour
	LocalTime.withMinute(int minute)	Same time, different minute
	LocalTime.withSecond(int second)	Same time, different second
	LocalTime.withNano(int nanoOfSecond)	Same time, different nanosecond
	LocalTime.with(TemporalField field, long newValue)	Generic field-based adjustment
	LocalTime.with(TemporalAdjuster adjuster)	Adjust using a temporal adjuster
`LocalDateTime`	withYear(int year), withMonth(int month), withDayOfMonth(int day)	Change date part only
	withHour(int hour), withMinute(int minute), withSecond(int second)	Change time part only
	withNano(int nanoOfSecond)	Change nanosecond
	with(TemporalField field, long newValue)	Generic field-based adjustment
	with(TemporalAdjuster adjuster)	Adjust using a temporal adjuster
`ZonedDateTime`	all the withXxx(...) of LocalDateTime	Change local date/time components
	withZoneSameInstant(ZoneId zone)	Same instant, different zone (changes local time)
	withZoneSameLocal(ZoneId zone)	Same local date/time, different zone (changes instant)

12.3 Conversion & `at...` Methods (Linking Date, Time, and Zone)

These methods are used to combine or convert between `LocalDate`, `LocalTime`, `LocalDateTime`, and `ZonedDateTime`.

From	Method	Result	Description
`LocalDate`	`atTime(LocalTime time)`	LocalDateTime	Combines this date with a given time
LocalDate	`atTime(int hour, int minute)`	LocalDateTime	Convenience overloads with numeric time components
LocalDate	`atTime(int hour, int minute, int second)`	LocalDateTime	—
LocalDate	`atTime(int hour, int minute, int second, int nano)`	LocalDateTime	—
LocalDate	`atStartOfDay()`	LocalDateTime	This date at time 00:00
LocalDate	`atStartOfDay(ZoneId zone)`	ZonedDateTime	This date at start of day in a specific zone
`LocalTime`	`atDate(LocalDate date)`	LocalDateTime	Combines this time with a given date
`LocalDateTime`	`atZone(ZoneId zone)`	ZonedDateTime	Adds a time zone to a local date-time
LocalDateTime	`toLocalDate()`	LocalDate	Extracts the date component
LocalDateTime	`toLocalTime()`	LocalTime	Extracts the time component
`ZonedDateTime`	`toLocalDate()`	LocalDate	Drops zone/offset, keeps local date
ZonedDateTime	`toLocalTime()`	LocalTime	Drops zone/offset, keeps local time
ZonedDateTime	`toLocalDateTime()`	LocalDateTime	Drops zone/offset, keeps local date-time

---

12.4 Period, Duration, and Instant

The `java.time` package provides three essential temporal classes that represent amounts of time or points on the timeline:

- **Period** → human-based date amounts (years, months, days)
- **Duration** → machine-based time amounts (seconds, nanoseconds)
- **Instant** → a point on the UTC timeline

12.5 `Period` — Human Date Amounts

`Period` represents a date-based amount of time, such as “3 years, 2 months, and 5 days”.

It is used with `LocalDate` and `LocalDateTime` (because they contain date parts).

Creation Methods

Method	Description
Period.ofYears(int years)	Only years
Period.ofMonths(int months)	Only months
Period.ofWeeks(int weeks)	Converts weeks to days
Period.ofDays(int days)	Only days
Period.of(int years, int months, int days)	Full period
Period.parse(CharSequence text)	ISO-8601 format: “P1Y2M3D”, “P7D”, “P1W”, …

Key properties

- Does not support hours, minutes, seconds, nanoseconds.
- Can be negative.
- Immutable.
- Examples

```
Period p1 = Period.ofYears(1);              // P1Y
Period p2 = Period.of(1, 2, 3);             // P1Y2M3D
Period p3 = Period.ofWeeks(2);              // P14D (converted to days)

LocalDate base = LocalDate.of(2025, 1, 10);
LocalDate result = base.plus(p2);           // 2026-03-13
```

> **Note**
>
> Period.parse(“P1W”) is allowed and represents a period of 7 days (equivalent to “P7D”).

> **Tip**
>
> Period is calendar-based: adding a period of months/years respects month lengths and leap years.

12.6 `Duration` — Machine Time Amounts

`Duration` represents a time-based amount in seconds and nanoseconds.

It is used with `LocalTime`, `LocalDateTime`, `ZonedDateTime`, and `Instant`.

Creation Methods

Method	Description
Duration.ofDays(long days)	Converts days to seconds
Duration.ofHours(long hours)	Converts hours to seconds
Duration.ofMinutes(long minutes)	Converts minutes to seconds
Duration.ofSeconds(long seconds)	Base representation in seconds
Duration.ofSeconds(long seconds, long nanoAdjustment)	Seconds plus additional nanos
Duration.ofMillis(long millis)	Converts milliseconds to nanos
Duration.ofNanos(long nanos)	Nanoseconds only
Duration.between(Temporal start, Temporal end)	Compute duration between two instants
Duration.parse(CharSequence text)	ISO: “PT20H”, “PT15M”, “PT10S”

Key characteristics

- Supports hours down to nanoseconds, but not years/months/weeks directly.
- Ideal for machine-level time computations.
- Immutable.
- Examples

```
Duration d1 = Duration.ofHours(5);            // PT5H
Duration d2 = Duration.ofMinutes(90);         // PT1H30M

LocalTime t = LocalTime.of(10, 0);
LocalTime t2 = t.plus(d2);                    // 11:30

ZonedDateTime z1 = ZonedDateTime.of(
    2024, 3, 30, 1, 0, 0, 0,
    ZoneId.of("Europe/Paris")
);

ZonedDateTime z2 = z1.plusHours(2);           // DST-aware
ZonedDateTime z3 = z1.plus(d2);               // Duration-based
```

> **Note**
>
> Duration.ofDays(1) represents exactly 24 hours of machine time. In a zone with DST, 24 hours may not align with "the same local time tomorrow".

12.7 `Instant` — Point on the UTC Timeline

`Instant` represents a single moment in time relative to UTC, with nanosecond precision.

It contains:

- Seconds since the epoch (1970-01-01T00:00Z).
- A nanoseconds adjustment.

Creation Methods

Method	Description
Instant.now()	Current moment in UTC
Instant.ofEpochSecond(long seconds)	From epoch seconds
Instant.ofEpochSecond(long seconds, long nanos)	From seconds plus nanos
Instant.ofEpochMilli(long millis)	From epoch milliseconds
Instant.parse(CharSequence text)	ISO: "2024-01-01T10:15:30Z"

Conversions

Action	Method
Instant → zoned time	instant.atZone(zoneId)
ZonedDateTime → Instant	zdt.toInstant()
LocalDateTime → Instant	Not allowed directly (needs a zone)

- Example

```java
Instant i = Instant.now();

ZonedDateTime z = i.atZone(ZoneId.of("Europe/Paris"));
Instant back = z.toInstant();  // same moment

// Duration between instants
Instant start = Instant.parse("2024-01-01T10:00:00Z");
Instant end   = Instant.parse("2024-01-01T12:30:00Z");

Duration between = Duration.between(start, end); // PT2H30M
```

> **Important**
>
> Instant is always UTC, with no time zone information attached. It cannot be combined with a Period; use Duration instead.

12.8 Summary Table (Period vs Duration vs Instant)

Concept	Represents	Good For	Works With	Notes
Period	Years, months, days	Calendar arithmetic	LocalDate, LocalDateTime	Human-based units
Duration	Hours to nanoseconds	Precise time calculations	LocalTime, LocalDateTime, ZonedDateTime, Instant	Machine-based
Instant	Exact point on UTC timeline	Timestamp representation	Convertible to/from ZonedDateTime	Cannot combine with Period

Common Traps

- `Period.of(1, 0, 0)` is not the same as `Duration.ofDays(365)` (leap years!).
- `Duration.ofDays(1)` may not equal a full "calendar day" in a DST zone.
- `LocalDateTime` cannot be converted to an `Instant` without a time zone.
- `Period.parse("P1W")` is valid and results in a period of 7 days.

12.9 TemporalUnit and TemporalAmount

The `java.time` API is built on two key interfaces that define how dates, times, and durations are manipulated:

- `TemporalUnit` → represents a unit of time (for example, DAYS, HOURS, MINUTES).
- `TemporalAmount` → represents an amount of time (for example, `Period`, `Duration`).

Both are essential for understanding how the `plus`, `minus`, and `with` methods work.

12.9.1 `TemporalUnit`

`TemporalUnit` represents a single unit of date/time measurement.
The main implementation used in Java is:

12.9.2 `ChronoUnit` enum

This enum provides the standard units used in the ISO-8601 chronology:

Category	Units
Date units	DAYS, WEEKS, MONTHS, YEARS, DECADES, CENTURIES, MILLENNIA, ERAS
Time units	NANOS, MICROS, MILLIS, SECONDS, MINUTES, HOURS, HALF_DAYS
Special	FOREVER

A `TemporalUnit` can be used directly with `plus()` and `minus()` methods.

- Examples using `ChronoUnit`:

```
LocalDate date = LocalDate.of(2025, 3, 10);

LocalDate d1 = date.plus(10, ChronoUnit.DAYS);     // 2025-03-20
LocalDate d2 = date.minus(2, ChronoUnit.MONTHS);   // 2025-01-10

LocalTime time = LocalTime.of(10, 0);
LocalTime t1 = time.plus(90, ChronoUnit.MINUTES);  // 11:30
```

Important

You cannot use time-based units with LocalDate, nor date-based units with LocalTime.

- Examples:

```
// ❌ UnsupportedTemporalTypeException
LocalDate d = LocalDate.now().plus(5, ChronoUnit.HOURS);

// ❌ UnsupportedTemporalTypeException
LocalTime t = LocalTime.now().plus(1, ChronoUnit.DAYS);
```

12.9.3 `TemporalAmount`

`TemporalAmount` represents a multiple-unit amount of time (for example, “2 years, 3 months”, or “90 minutes”).
It is implemented by:

- `Period` → years, months, days (date-based)
- `Duration` → seconds, nanoseconds (time-based)

Both can be passed to date/time objects to adjust them using `plus()` and `minus()`.

12.9.4 `Period` as a `TemporalAmount`

`Period` represents a human-based amount: years, months, days.

- Examples:

```
Period p = Period.of(1, 2, 3);  // 1 year, 2 months, 3 days

LocalDate base = LocalDate.of(2025, 3, 10);
LocalDate result = base.plus(p); // 2026-05-13
```

Notes

- `Period` cannot be used with `LocalTime` (no date component).
- `Period.ofWeeks(n)` is converted internally to days (n × 7).

12.9.5 `Duration` as a `TemporalAmount`

`Duration` represents machine-based time: seconds + nanoseconds.

- Examples:

```
Duration d = Duration.ofHours(5).plusMinutes(30); // PT5H30M

LocalDateTime ldt = LocalDateTime.of(2025, 3, 10, 10, 0);
LocalDateTime result = ldt.plus(d); // 2025-03-10T15:30
```

Notes

- `Duration` can be used with classes that have time components (`LocalTime` , `LocalDateTime` , `ZonedDateTime` , `Instant`).
- `Duration` cannot be applied to `LocalDate` → it will throw `UnsupportedTemporalTypeException` .
- `Duration` interacts with zones and DST transitions when applied to `ZonedDateTime` .

12.9.6 Using `TemporalAmount` vs `TemporalUnit`

Using a `TemporalUnit` :

```
LocalDate d1 = LocalDate.now().plus(5, ChronoUnit.DAYS);
```

Using a `TemporalAmount` :

```
Period p = Period.ofDays(5);
LocalDate d2 = LocalDate.now().plus(p);
```

Both produce the same result when supported.

Differences

Aspect	TemporalUnit	TemporalAmount
Represents	A single unit (e.g., DAYS)	A structured quantity (e.g. 2Y, 5M, 3D)
Examples	ChronoUnit.DAYS	Period.of(2,5,3)
Supports multiple fields	No	Yes
Good for	Simple increments	Complex increments
Common with	All date/time classes	Restricted by type

12.9.7 `between(...)` Methods

Many classes provide a `between` method from `ChronoUnit` , `Duration` , or `Period` .

Using `Duration.between` (for time-based classes)

```
Duration d = Duration.between(
    LocalTime.of(10, 0),
    LocalTime.of(13, 30)
);
// PT3H30M
```

Using `Period.between` (only for dates)

```
Period p = Period.between(
    LocalDate.of(2025, 3, 1),
    LocalDate.of(2025, 5, 10)
);
// P2M9D
```

Using `ChronoUnit` `between`

```java
long days = ChronoUnit.DAYS.between(
    LocalDate.of(2025, 3, 1),
    LocalDate.of(2025, 3, 10)
);
// 9
```

> **Important**
>
> ChronoUnit.between(...) always returns a long, while Period.between returns a Period, and Duration.between returns a Duration.

12.9.8 Common Pitfalls

- Applying the wrong `TemporalAmount`:

```java
// LocalTime.plus(Period.ofDays(1))    // ❌ compile-time error
// LocalDate.plus(Duration.ofHours(1)) // ❌ runtime error: UnsupportedTemporalTypeException
```

- DST changes with Duration: adding 24 hours is not always "tomorrow" in a zone with DST changes.
- `Period.ofWeeks(1)` is exactly 7 days; DST effects show up when applied to zone-aware types.
- `Instant.plus(Period)` → runtime `UnsupportedTemporalTypeException`; use `Duration` instead.
- `Instant` cannot be created directly from a `LocalDateTime`; you must first apply a time zone: `ldt.atZone(zone).toInstant()`.

12.9.9 Summary

Feature	TemporalUnit	TemporalAmount	ChronoUnit	Period	Duration
Represents	A unit	An amount	enum of units	Y/M/D	S + nanos
Multi-field	No	Yes	No	Yes	No
Works with	plus/minus	plus/minus	date/time	LocalDate/LocalDateTime	Time/time-zone
Human-based	No	Yes	No	Yes	No
Machine-based	Yes	Yes	Yes	No	Yes

◀ 11. Math in Java | ▲ Index | 13. Formatting and Localizing in Java ▶

13. Formatting and Localizing in Java

Table of Contents

- 13.1 String Formatting
  - 13.1.1 String.format and formatted
    - 13.1.1.1 Floating-point Flags
    - 13.1.1.2 Precision n
    - 13.1.1.3 Width m
    - 13.1.1.4 Zero Padding 0 Flag
    - 13.1.1.5 Left Justification - Flag
    - 13.1.1.6 Explicit Sign + Flag
    - 13.1.1.7 Parentheses for Negatives (Flag
    - 13.1.1.8 Combining Flags
    - 13.1.1.9 Locale Effects
    - 13.1.1.10 Common Pitfalls
  - 13.1.2 Custom Text Values and Escaping
- 13.2 Number Formatting
  - 13.2.1 NumberFormat
  - 13.2.2 Localizing Numbers
  - 13.2.3 DecimalFormat and NumberFormat
  - 13.2.4 DecimalFormat Pattern Structure
  - 13.2.5 The 0 Symbol Mandatory Digit
  - 13.2.6 The # Symbol Optional Digit
  - 13.2.7 Combining 0 and #
  - 13.2.8 Decimal and Grouping Separators
  - 13.2.9 DecimalFormatSymbols Locale-Specific Formatting Symbols
  - 13.2.10 Special DecimalFormat Patterns
  - 13.2.11 Common Rules and Pitfalls
- 13.3 Parsing Numbers
  - 13.3.1 Parsing with DecimalFormat
  - 13.3.2 CompactNumberFormat
- 13.4 Date and Time Formatting
  - 13.4.1 DateTimeFormatter
  - 13.4.2 Standard Date Time Symbols
  - 13.4.3 datetime.format vs formatter.format
  - 13.4.4 Localizing Dates
- 13.5 Internationalization i18n and Localization l10n
  - 13.5.1 Locales
  - 13.5.2 Locale Categories
  - 13.5.3 Real-world Example
- 13.6 Properties and Resource Bundles
  - 13.6.1 Resource Bundle Resolution Rules
- 13.7 Common Rules and Pitfalls

This chapter delivers a deep and practical treatment of formatting in Java 21.

13.1 String Formatting

13.1.1 String.format and formatted

`String.format()` creates formatted strings using printf-style placeholders.

It is locale-sensitive and returns a new immutable `String`.

```
String result = String.format("The User: %s | Score: %d", "Bob", 42);
System.out.println(result);

// Or

System.out.println("The User: %s | Score: %d".formatted("Bob", 42));
```

Output:

```
The User: Bob | Score: 42
```

Key characteristics:

- Uses format specifiers like `%s` (any type, commonly String values), `%d` (integral values), `%f` (floating-point values).
- Does not modify existing strings.
- Throws `IllegalFormatException` if arguments mismatch the format.
- Is locale-sensitive when a `Locale` is provided.

```
String price = String.format(Locale.GERMANY, "%.2f", 1234.5);
// Output (German locale): 1234,50
```

13.1.1.1 Floating-point Flags

`%f` is used to format floating-point numbers (`float` , `double` , `BigDecimal`) using decimal notation.

```
System.out.printf("%f", 12.345);
```

```
12.345000
```

- **Always prints 6 digits after the decimal point** by default.
- Uses rounding (not truncation).
- Is locale-sensitive for the decimal separator.

13.1.1.2 Precision (.n)

Precision defines the number of digits printed **after** the decimal point.

```
System.out.printf("%.2f", 12.345);
```

```
12.35
```

- `%.0f` prints no decimal digits.
- Rounding is applied.
- Precision is applied before width padding.

13.1.1.3 Width (m)

Width defines the minimum total number of characters in the output.

```
System.out.printf("%8.2f", 12.34);
```

```
   12.34
```

- **Pads with spaces** by default.

- If the value is longer, width is ignored (it never truncates).
- Padding is applied on the left by default.

13.1.1.4 Zero Padding `0` Flag

The `0` flag replaces space padding with zeros.

```
System.out.printf("%08.2f", 12.34);
```

```
00012.34
```

- Requires a width.
- Zeros are inserted after the sign.
- Ignored if left-justified (`-` flag).

13.1.1.5 Left Justification `-` Flag

The `-` flag left-aligns the value within the width.

```
System.out.printf("%-8.2f", 12.34);
```

```
12.34
```

- Padding is moved to the right.
- Overrides zero padding.

13.1.1.6 Explicit Sign `+` Flag

The `+` flag forces display of the sign for positive numbers.

```
System.out.printf("%+8.2f", 12.34);
```

```
  +12.34
```

- Negative numbers already show `-`.
- Overrides the space flag (which prints a leading space for positive values).

13.1.1.7 Parentheses for Negatives `(` Flag

The `(` flag formats negative numbers using parentheses.

```
System.out.printf("%(8.2f", -12.34);
```

```
 (12.34)
```

- Only affects negative values.
- Rarely used in practice.

13.1.1.8 Combining Flags

```
System.out.printf("%+010.2f", 12.34);
```

```
+000012.34
```

Evaluation order (semplificato):

- Precision is applied.
- Sign is handled.
- Width is enforced.
- Padding (spaces or zeros) is applied.

13.1.1.9 Locale Effects

```
System.out.printf(Locale.FRANCE, "%,.2f", 12345.67);
```

```
12 345,67
```

Decimal and grouping separators depend on the active `Locale`.

13.1.1.10 Common Pitfalls

- `%f` defaults to 6 decimal places if no precision is specified.
- Width never truncates output, it only pads if needed.
- `0` flag is ignored when `-` is present.
- `+` overrides the space flag.
- Grouping and separators are Locale-dependent.

13.1.2 Custom Text Values and Escaping

Certain characters have special meaning in format strings and must be escaped.

- `%%` → literal percent sign.
- `\n`, `\t` → standard Java escapes.

```
String msg = String.format("Completion: %d%%%nStatus: OK", 100);
System.out.println(msg);
```

Output:

```
Completion: 100%
Status: OK
```

> **Note**
>
> A single % without a valid specifier causes an IllegalFormatException at runtime.

13.2 Number Formatting

13.2.1 NumberFormat

`NumberFormat` is an abstract class used to format and parse numbers in a locale-aware manner.

```
NumberFormat nf = NumberFormat.getInstance(Locale.FRANCE);
System.out.println(nf.format(1234567.89));
```

> **Important**
>
> - Factory methods determine formatting style (general, integer, currency, percent, compact, ...).
> - Formatting depends on the provided `Locale`.
> - `NumberFormat` (and `DecimalFormat`) are not thread-safe.

13.2.2 Localizing Numbers

Number localization affects decimal separators, grouping separators, and currency symbols.

```java
NumberFormat nfUS = NumberFormat.getInstance(Locale.US);
NumberFormat nfIT = NumberFormat.getInstance(Locale.ITALY);

System.out.println(nfUS.format(1234.56)); // 1,234.56
System.out.println(nfIT.format(1234.56)); // 1.234,56
```

13.2.3 DecimalFormat and NumberFormat

`DecimalFormat` is a concrete subclass of `NumberFormat` that provides fine-grained control over numeric formatting using patterns.

`NumberFormat` defines locale-aware formatting via factory methods, while `DecimalFormat` allows explicit pattern-based control.

```java
NumberFormat nf = NumberFormat.getInstance(Locale.US);
DecimalFormat df = (DecimalFormat) nf;
```

Or directly:

```java
DecimalFormat df = new DecimalFormat("#,##0.00");
```

> **Note**
>
> - `DecimalFormat` is mutable (you can change pattern, symbols, etc.).
> - `DecimalFormat` is **not** thread-safe.
> - Formatting is locale-sensitive via `DecimalFormatSymbols`.

13.2.4 DecimalFormat Pattern Structure

A pattern may contain a positive and an optional negative subpattern, separated by `;`.

```
#,##0.00;(#,##0.00)
```

> **Note**
>
> - First part → positive numbers.
> - Second part → negative numbers.
> - If the negative part is omitted, negative numbers use a leading `-` automatically.

13.2.5 The `0` Symbol (Mandatory Digit)

The `0` symbol forces a digit to appear, padding with zeros if necessary.

```java
DecimalFormat df = new DecimalFormat("0000.00");
System.out.println(df.format(12.3));
```

```
0012.30
```

- Controls the minimum number of digits.
- Pads with zeros if the number has fewer digits.
- Useful for fixed-width or aligned output.

13.2.6 The `#` Symbol (Optional Digit)

The `` symbol displays a digit only if it exists.

```java
DecimalFormat df = new DecimalFormat("####.##");
System.out.println(df.format(12.3));
```

```
12.3
```

- Suppresses leading zeros.
- Suppresses unnecessary trailing zeros.
- Good for “human-friendly” formatting.

13.2.7 Combining `0` and `#`

Patterns often combine both symbols for flexibility.

```
DecimalFormat df = new DecimalFormat("#,##0.##");
System.out.println(df.format(12));
System.out.println(df.format(12.5));
System.out.println(df.format(12345.678));
```

```
12
12.5
12,345.68
```

Pattern explanation:

```
#,##0 . ##
 ^  ^    ^
 |  |    |
 |  |    └─ optional fractional digits (#)
 |  └───── mandatory integer digit (0)
 └──────── grouping pattern (,)
```

- At least one integer digit is guaranteed (the `0`).
- Digits are grouped by thousands using the grouping separator.
- Fractional digits are optional (up to two).

13.2.8 Decimal and Grouping Separators

In patterns:

- `.` → decimal separator.
- `,` → grouping separator.

The actual symbols used at runtime depend on the `Locale` (for example, comma vs dot).

13.2.9 DecimalFormatSymbols Locale-Specific Formatting Symbols

```
DecimalFormatSymbols symbols =
        DecimalFormatSymbols.getInstance(Locale.FRANCE);

DecimalFormat df =
        new DecimalFormat("#,##0.00", symbols);

System.out.println(df.format(1234.5));
```

```
1 234,50
```

- Controls decimal and grouping separators.
- Controls minus sign and currency symbol.
- Controls NaN and Infinity strings.

13.2.10 Special DecimalFormat Patterns

```
0.###E0   scientific notation
###%      percent
¤#,##0.00 currency (¤ is the currency sign)
```

13.2.11 Common Rules and Pitfalls

- `DecimalFormat` is a `NumberFormat` subclass.
- `0` forces digits, `#` does not.
- Patterns control formatting, not the rounding mode itself (use `setRoundingMode()`).
- Grouping only works if the grouping separator (usually `,`) is present in the pattern.
- Parsing may succeed partially without error if trailing characters appear after a valid number.
- `DecimalFormat` is mutable and not thread-safe.

13.3 Parsing Numbers

Parsing converts localized text into numeric values.
By default, parsing is lenient.

```
NumberFormat nf = NumberFormat.getInstance(Locale.FRANCE);
Number n = nf.parse("12 345,67abc"); // parses 12345.67
```

- Parsing stops at the first invalid character.
- Trailing text is ignored unless explicitly checked.

13.3.1 Parsing with DecimalFormat

`DecimalFormat` can also parse numbers. Parsing is lenient by default.

```
DecimalFormat df = new DecimalFormat("#,##0.##");
Number n = df.parse("1,234.56abc");
```

- Parsing stops at the first invalid character.
- Trailing text is ignored if present.

To enforce strict parsing:

```
df.setParseStrict(true);
```

13.3.2 CompactNumberFormat

Compact formatting shortens large numbers for human readability.

- Supports SHORT vs LONG styles.
- Uses locale-dependent abbreviations (for example, K, M, “million”).

```
NumberFormat cnf =
        NumberFormat.getCompactNumberInstance(
                Locale.US, NumberFormat.Style.SHORT);

System.out.println(cnf.format(1_200));        // 1.2K
System.out.println(cnf.format(5_000_000));    // 5M

NumberFormat cnf1 =
        NumberFormat.getCompactNumberInstance(
                Locale.US, NumberFormat.Style.SHORT);

NumberFormat cnf2 =
        NumberFormat.getCompactNumberInstance(
                Locale.US, NumberFormat.Style.LONG);

System.out.println(cnf1.format(315_000_000));   // 315M
System.out.println(cnf2.format(315_000_000));   // 315 million
```

13.4 Date and Time Formatting

13.4.1 DateTimeFormatter

Java 21 relies on `java.time` and `DateTimeFormatter` for modern date/time formatting.

```
DateTimeFormatter f =
        DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm");
System.out.println(LocalDateTime.now().format(f));
```

Core properties:

- Immutable.
- Thread-safe.
- Locale-aware.

13.4.2 Standard Date/Time Symbols

```
y   year
M   month number (or name with more letters)
d   day of month
E   day name
H   hour (0-23)
h   hour (1-12)
m   minute
s   second
a   AM/PM marker
z   time zone
```

13.4.3 datetime.format vs formatter.format

Both methods are functionally identical:

```
date.format(formatter);
formatter.format(date);
```

- `date.format(formatter)` → preferred for readability (data first, then formatting).
- `formatter.format(date)` → sometimes convenient in functional or reusable formatter code.

13.4.4 Localizing Dates

Localized styles adapt date output to cultural norms.

```
DateTimeFormatter fullIt =
        DateTimeFormatter
                .ofLocalizedDate(FormatStyle.FULL)
                .withLocale(Locale.ITALY);

DateTimeFormatter shortIt =
        DateTimeFormatter
                .ofLocalizedDate(FormatStyle.SHORT)
                .withLocale(Locale.ITALY);

LocalDate today = LocalDate.of(2025, 12, 17);

System.out.println(today.format(fullIt));
System.out.println(today.format(shortIt));
```

Possible output:

```
mercoledì 17 dicembre 2025
17/12/25
```

13.5 Internationalization (i18n) and Localization (l10n)

13.5.1 Locales

A `Locale` defines language, country, and optional variant.

```
Locale l1 = Locale.US;
Locale l2 = Locale.of("fr", "FR");
Locale l3 = new Locale.Builder()
        .setLanguage("en")
        .setRegion("US")
        .build();
```

Locale formats:

- `en` (it, fr, etc.): lowercase language code.
- `en_US` (fr_CA, it_IT, etc.): lowercase language code + underscore + uppercase country code.

13.5.2 Locale Categories

Locale categories separate formatting from UI language.

`Locale.Category` lets Java use different default locales for different purposes.

There are two categories:

Category	Used for
FORMAT	Numbers, dates, currency, other formatting
DISPLAY	Human-readable text (UI, names, messages)

13.5.3 Real-world Example

A French user living in Germany might want:

- Numbers and dates → German format.
- UI language → French.

Before Java 7, this was not possible.

```
Locale.setDefault(Locale.Category.FORMAT, Locale.GERMANY);
Locale.setDefault(Locale.Category.DISPLAY, Locale.FRANCE);
```

Sample effects:

Aspect	Result (example)
Numbers	1.234,56
Dates	31.12.2025
Currency	€
UI text	French
Month names	décembre
Country names	Allemagne

13.6 Properties and Resource Bundles

Resource bundles externalize text and allow localization without code changes.

```
ResourceBundle rb =
        ResourceBundle.getBundle("messages", Locale.GERMAN);

String msg = rb.getString("welcome");
```

13.6.1 Resource Bundle Resolution Rules

Java searches bundles in a strict fallback order. For example, with base name `messages` and locale `de_DE`:

- messages_de_DE.properties
- messages_de.properties
- messages.properties

If none is found → `MissingResourceException`.

> **Note**
>
> Traditional .properties files are specified as ISO-8859-1; non-ASCII characters must be encoded as Unicode escapes (for example, 0E9 for é) unless you use alternate loading mechanisms.

13.7 Common Rules and Pitfalls

- `DateTimeFormatter` is immutable and thread-safe.
- `NumberFormat` / `DecimalFormat` are mutable and not thread-safe.
- Changing the `Locale` affects how values are formatted and parsed, not the underlying numeric or temporal values.
- Parsing with `NumberFormat` or `DecimalFormat` may succeed partially without throwing if extra text follows a valid number.
- `java.time` replaces most uses of the old `java.util.Date` / `Calendar` APIs in modern code and in the exam.

◀ 12. Date and Time in Java | ▲ Index | 14. Methods, Attributes and Variables ▶

Object-Oriented Fundamentals

14. Methods, Attributes and Variables

Table of Contents

- 14.1 Methods
  - 14.1.1 Mandatory Components of a Method
    - 14.1.1.1 Access Modifiers
    - 14.1.1.2 Return Type
    - 14.1.1.3 Method Name
    - 14.1.1.4 Method Signature
    - 14.1.1.5 Method Body
  - 14.1.2 Optional Modifiers
  - 14.1.3 Declaring Methods
- 14.2 Java Is a Pass-by-Value Language
- 14.3 Overloading Methods
  - 14.3.1 Calling overloaded methods
    - 14.3.1.1 Exact match wins
    - 14.3.1.2 If no exact match exists Java picks the most specific compatible type
    - 14.3.1.3 Primitive widening beats boxing
    - 14.3.1.4 Boxing beats varargs
    - 14.3.1.5 For references Java picks the most specific reference type
    - 14.3.1.6 When there is no unambiguous most specific the-call-is-a-compile-error
    - 14.3.1.7 Mixed primitive + wrapper overloads
    - 14.3.1.8 When primitives mix with reference types
    - 14.3.1.9 When Object wins
    - 14.3.1.10 Summary Table Overload Resolution
- 14.4 Local and Instance Variables
  - 14.4.1 Instance Variables
  - 14.4.2 Local Variables
    - 14.4.2.1 Effectively Final Local Variables
    - 14.4.2.2 Parameters as Effectively Final
- 14.5 Varargs Variable-Length Argument Lists
- 14.6 Static Methods Static Variables and Static Initializers
  - 14.6.1 Static Variables Class Variables
  - 14.6.2 Static Methods
  - 14.6.3 Static Initializer Blocks
  - 14.6.4 Initialization Order Static vs Instance
  - 14.6.5 Accessing Static Members
    - 14.6.5.1 Recommended use class name
    - 14.6.5.2 Also legal via instance reference
  - 14.6.6 Static and Inheritance
  - 14.6.7 Common Pitfalls

This chapter introduces fundamental Object-Oriented Programming (OOP) concepts in Java, starting with **methods**, **parameter passing**, **overloading**, **local vs. instance variables**, and **varargs**.

14.1 Methods

`Methods` represent the **operations/behaviors** that can be performed by a particular data type (a **class**).

They describe *what the object can do* and how it interacts with its internal state and the outside world.

A `method declaration` is composed of **mandatory** and **optional** components.

14.1.1 Mandatory Components of a Method

14.1.1.1 Access Modifiers

`Access Modifiers` control *visibility*, not behavior.

(Please refer to Paragraph "**Access Modifiers**" in Chapter: 2. Basic Language Java Building Blocks)

14.1.1.2 Return Type

Appears **immediately before** the method name.

- If the method returns a value → the return type specifies the value's type.
- If the method does *not* return a value → the keyword `void` **must** be used.
- A method with a non-void return type **must** contain at least one `return value;` statement.
- A `void` method may:
  - omit a return statement
  - include `return;` (with **no** value)

14.1.1.3 Method Name

Follows the same rules as identifiers (Please refer to Chapter: 3. Java Naming Rules).

14.1.1.4 Method Signature

The **method signature** in Java includes:

- the *method name*
- the *parameter type list* (types + order)

> **Note**
>
> **Parameter names do NOT belong to the signature**, only types and order matter.

- Example of distinct signatures:

```
void process(int x)
void process(int x, int y)
void process(int x, String y)
```

- Example of *same* signature (illegal overloading):

```
// ❌ same signature: only parameter names differ
void m(int a)
void m(int b)
```

14.1.1.5 Method Body

A block `{ }` containing **zero or more statements**.

If the method is `abstract`, the body must be omitted.

14.1.2 Optional Modifiers

Optional method modifiers include:

- `static`
- `abstract`
- `final`

- `default` (interface methods)
- `synchronized`
- `native`
- `strictfp`

Rules:

- Optional modifiers may appear in **any order**.
- All modifiers must appear **before the return type**.
- Example:

```
public static final int compute() {
    return 10;
}
```

14.1.3 Declaring Methods

```
public final synchronized String formatValue(int x, double y) throws IOException {
    return "Result: " + x + ", " + y;
}
```

Breakdown:

Part	Meaning
`public`	access modifier
`final`	cannot be overridden
`synchronized`	thread control modifier
`String`	return type
`formatValue`	method name
`(int x, double y)`	parameter list
`throws IOException`	exception list
method body	implementation

14.2 Java Is a “Pass-by-Value” Language

Java uses **only pass-by-value**, no exceptions.

This means:

- For **primitive types** → the method receives a *copy of the value*.
- For **reference types** → the method receives a *copy of the reference*, meaning:
    - the reference itself cannot be changed by the method
    - the *object* **can** be modified through that reference
- Example:

```
void modify(int a, StringBuilder b) {
    a = 50;           // modifying the *copy* → no effect outside
    b.append("!");    // modifying the *object* → visible outside
}
```

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

    int num1 = 11;
    methodTryModif(num1);
    System.out.println(num1);

}

public static void methodTryModif(int num1){
    num1 = 10;  // this new assignement affects only the method parameter which, accidentally,
}
```

14.3 Overloading Methods

Method overloading means **same method name**, **different signature**.

Two methods are considered overloaded if they differ in:

- number of parameters
- parameter types
- parameter order

Overloading **does NOT depend on**:

- return type
- access modifier
- exceptions
- Example:

```java
void print(int x)
void print(double x)
void print(int x, int y)
```

Illegal overloaded method:

```java
// ❌ Return type does not count in overloading
int compute(int x)
double compute(int x)
```

14.3.1 Calling overloaded methods

When multiple overloaded methods are available, Java applies **overload resolution** to decide which method to call.

The compiler selects the method whose parameter types are the **most specific** and **compatible** with the provided arguments.

Overload resolution happens **at compile time** (unlike overriding, which is run-time based).

Java follows these rules:

14.3.1.1 Exact match wins

If an argument matches a method parameter exactly, that method is chosen.

```java
void call(int x)    { System.out.println("int"); }
void call(long x)   { System.out.println("long"); }

call(5); // prints: int (exact match for int)
```

14.3.1.2 — If no exact match exists, Java picks the *most specific* compatible type

Java prefers:

1. **widening** over autoboxing

2. **autoboxing** over varargs
3. **wider reference** only if more specific type is not available

- Example with numeric primitives:

```
void test(long x)   { System.out.println("long"); }
void test(float x)  { System.out.println("float"); }

test(5);  // int literal: can widen to long or float
          // but long is more specific than float for integer types
          // Output: long
```

14.3.1.3 — Primitive widening beats boxing

If a primitive argument can either widen or autobox, Java chooses widening.

```
void m(int x)        { System.out.println("int"); }
void m(Integer x)    { System.out.println("Integer"); }

byte b = 10;
m(b);                // byte → int (widening) wins
                     // Output: int
```

14.3.1.4 — Boxing beats varargs

```
void show(Integer x)     { System.out.println("Integer"); }
void show(int... x)      { System.out.println("varargs"); }

show(5);                 // int → Integer (boxing) preferred
                         // Output: Integer
```

14.3.1.5 — For references, Java picks the most specific reference type

```
void ref(Object o)       { System.out.println("Object"); }
void ref(String s)       { System.out.println("String"); }

ref("abc");              // "abc" is a String → more specific than Object
                         // Output: String
```

More specific means *lower in the inheritance hierarchy*.

14.3.1.6 — When there is no unambiguous “most specific”, the call is a compile error

Example with sibling classes:

```
void check(Number n)      { System.out.println("Number"); }
void check(String s)      { System.out.println("String"); }

check(null);    // Both String and Number can accept null
                // String is more specific because it is a concrete class
                // Output: String
```

But if two unrelated classes compete:

```
void run(String s)    { }
void run(Integer i)   { }

run(null);  // ❌ Compile-time error: ambiguous method call
```

14.3.1.7 — Mixed primitive + wrapper overloads

Java evaluates widening, boxing, and varargs in this order:

`widening → boxing → varargs`

Example:

```java
void mix(long x)        { System.out.println("long"); }
void mix(Integer x)     { System.out.println("Integer"); }
void mix(int... x)      { System.out.println("varargs"); }

short s = 5;
mix(s);    // short → int → long  (widening)
           // Boxing and varargs ignored
           // Output: long
```

14.3.1.8 — When primitives mix with reference types

```java
void fun(Object o)      { System.out.println("Object"); }
void fun(int x)         { System.out.println("int"); }

fun(10);                 // exact primitive match wins
                         // Output: int

Integer i = 10;
fun(i);                  // reference accepted → Object
                         // Output: Object
```

14.3.1.9 — When Object wins

```java
void fun(List<String> o)     { System.out.println("O"); }
void fun(CharSequence x)     { System.out.println("X"); }
void fun(Object y)           { System.out.println("Y"); }

fun(LocalDate.now());        // Output: Y
```

14.3.1.10 Summary Table (Overload Resolution)

Situation	Rule
Exact match	Always chosen
Primitive widening vs boxing	Widening wins
Boxing vs varargs	Boxing wins
Most specific reference type	Wins
Unrelated reference types	Ambiguous → compile error
Mixed primitive + wrapper	Widening → boxing → varargs

14.4 Local and Instance Variables

14.4.1 Instance Variables

Instance variables are:

- declared as members of a class
- created when an object is instantiated
- accessible by all methods of the instance

Possible modifiers for instance variables:

- access modifiers (`public`, `protected`, `private`)
- `final`
- `volatile`
- `transient`
- Example:

```
public class Person {
    private String name;          // instance variable
    protected final int age = 0; // final means cannot be reassigned
}
```

14.4.2 Local Variables

Local variables:

- are declared *inside* a method, constructor, or block
- have **no default values** → must be explicitly initialized before use
- only modifier allowed: **final**
- Example:

```
void calculate() {
    int x;         // declared
    x = 10;        // must be initialized before use

    final int y = 5;   // legal
}
```

Two special cases:

14.4.2.1 Effectively Final Local Variables

A local variable is *effectively final* if it is **assigned once**, even without `final`.

Effectively final variables can be used in:

- lambda expressions
- local/anonymous classes

14.4.2.2 Parameters as Effectively Final

Method parameters behave as local variables and follow the same rules.

14.5 Varargs (Variable-Length Argument Lists)

Varargs allow a method to accept **zero or more** parameters of the same type.

Syntax:

```
void printNames(String... names)
```

Rules:

- A method may have **only one** varargs parameter.
- It must be the **last** parameter in the list.
- Varargs are treated as an **array** inside the method.
- Example:

```
void show(int x, String... values) {
    System.out.println(values.length);
}

show(10);                        // length = 0
show(10, "A");                   // length = 1
show(10, "A", "B", "C");         // length = 3
```

> **Important**
>
> Varargs and arrays participate in method overloading. Overload resolution may become ambiguous.

14.6 Static Methods, Static Variables, and Static Initializers

In Java, the keyword `static` marks elements that **belong to the class itself**, not to individual instances. This means:

- They are **loaded once** into memory when the class is first loaded by the JVM.
- They are shared among **all instances**.
- They can be accessed **without creating an object** of the class.

Static members are stored in the JVM **method area** (class-level memory), while instance members live in the **heap**.

14.6.1 Static Variables (Class Variables)

A **static variable** is a variable defined at class level and shared by all instances.

Characteristics:

- Created when the class is loaded.
- Exists **even if no instance** of the class is created.
- All objects see the **same value**.
- May be marked `final`, `volatile`, or `transient`.
- Example:

```
public class Counter {
    static int count = 0;     // shared by all instances
    int id;                   // instance variable

    public Counter() {
        count++;
        id = count;           // each instance gets a unique id
    }
}
```

14.6.2 Static Methods

A **static method** belongs to the class, not to any object instance.

Rules:

- They can be called using the class name: `ClassName.method()`.
- They **cannot** access instance variables or instance methods directly, but only through an instance of the class.
- They **cannot** use `this` or `super`.
- They are commonly used for:
  - utility methods (e.g., `Math.sqrt()`)
  - factory methods
  - global behaviors that do not depend on instance state
- Example:

```
public class MathUtil {

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

    void instanceMethod() {
        // System.out.println(count);   // OK: accessing static variable
        // square(5);                   // OK: static method accessible
    }
}
```

Common errors:

```java
// ❌ Compile error: instance method cannot be accessed directly in static context
static void go() {
    run();        // run() is instance method!
}

void run() { }
```

14.6.3 Static Initializer Blocks

Static initializer blocks allow executing code **once**, when the class is loaded.

Syntax:

```java
static {
    // initialization logic
}
```

Usage:

- initializing complex static variables
- performing class-level setup
- running code that must execute exactly once
- Example:

```java
public class Config {

    static final Map<String, String> settings = new HashMap<>();

    static {
        settings.put("mode", "production");
        settings.put("version", "1.0");
        System.out.println("Static initializer executed");
    }
}
```

> **Important**
>
> Static initializer blocks run **once**, in the order they appear, before `main()` and before any static method is called.

14.6.4 Initialization Order (Static vs. Instance)

(Please refer to Chapter: 15. Class Loading, Initialization, and Object Construction)

14.6.5 Accessing Static Members

14.6.5.1 Recommended: use class name

```java
Math.sqrt(16);
MyClass.staticMethod();
```

14.6.5.2 Also legal: via instance reference

```java
MyClass obj = new MyClass();
obj.staticMethod();
```

14.6.6 Static and Inheritance

Static methods:

- **can be hidden**, not overridden
- binding is **compile-time**, not runtime
- accessed based on **reference type**, not object type
- Example:

```
class A {
    static void test() { System.out.println("A"); }
}

class B extends A {
    static void test() { System.out.println("B"); }
}

A ref = new B();
ref.test();    // prints "A" — static binding!
```

> **Note**
>
> Key rule: static methods use **reference type**, not object type.

14.6.7 Common Pitfalls

- Attempting to reference an instance variable/method from a static context.
- Assuming static methods are overridden → they are **hidden**.
- Calling a static method from an instance reference (legal but confusing).
- Confusing initialization order of static elements vs. instance elements.
- Forgetting that static variables are shared across all objects.
- Not knowing that static initializers run *once*, in declaration order.

◀ 13. Formatting and Localizing in Java | ▲ Index | 15. Class Loading, Initialization, and Object Construction ▶

15. Class Loading, Initialization, and Object Construction

Table of Contents

- 15.1 Java Memory Areas Relevant to Class and Object Initialization
- 15.2 Class Loading with Inheritance
    - 15.2.1 Class Loading Order
    - 15.2.2 What Happens During Class Loading
- 15.3 Object Creation with Inheritance
    - 15.3.1 Full Instance Creation Order
- 15.4 A Complete Example Static + Instance Initialization Across Inheritance
- 15.5 Visualization Diagram
- 15.6 Key Rules
- 15.7 Summary Table

In Java, understanding **how classes are loaded**, **how static and instance members are initialized**, and **how constructors run — especially with inheritance** — is essential for mastering the language.

This chapter provides a unified, clear explanation of:

- How a class is loaded into memory
- How static variables and static initializers are executed
- How objects are created step-by-step
- How constructors run in an inheritance chain
- How different memory areas (Heap, Stack, Method Area) participate

15.1 Java Memory Areas Relevant to Class and Object Initialization

Before understanding initialization order, it is useful to recall the three main memory areas involved:

- **Method Area (a.k.a. Class Area)** — stores class metadata, static variables, and static initializer blocks.
- **Heap** — stores all objects and instance fields.
- **Stack** — stores method calls, local variables, and references.

> **Note**
>
> Static members belong to the **class** and are created **once** in the Method Area.
>
> Instance members belong to **each object** and live in the **Heap**.

15.2 Class Loading (with Inheritance)

When a Java program starts, the JVM loads classes *on demand*.

When a class is referenced for the first time (e.g., by calling `new` or accessing a static member), **its entire inheritance chain must be loaded in memory first**.

15.2.1 Class Loading Order

Given a class hierarchy:

```
class A { ... }
class B extends A { ... }
class C extends B { ... }
```

If the code executes:

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

Then class loading proceeds in this strict order:

- Load class A
- Initialize A's static variables (default → explicit)
- Run A's static initializer blocks (top → bottom)
- Load class B and repeat the same logic
- Load class C and repeat the same logic

15.2.2 What Happens During Class Loading

- **Step 1: Static variables are allocated** (default values first).
- **Step 2: Explicit static initializations run**.
- **Step 3: Static initializer blocks** run in source order.

Note

After these steps, the class is fully prepared and may now be used (instantiated or referenced).

Warning

Accessing a static field triggers the initialization only of the class or interface that directly declares that field.

This is true even if the field is accessed using the name of a subclass, a subinterface, or a class that implements the interface.

15.3 Object Creation (with Inheritance)

When the `new` keyword is used, **instance creation follows a strict and predictable sequence** involving all parent classes.

15.3.1 Full Instance Creation Order

- **1. Memory is allocated on the Heap for the new object** (fields get default values).
- **2. The constructor chain runs (not yet the bodies) from parent to child** — the top of the hierarchy runs first, then each subclass.
- **3. Instance variables receive explicit initializations**.
- **4. Instance initializer blocks execute**.
- **5. The constructor body runs**: for each class in the inheritance chain, steps 3–5 (field initialization, instance blocks, constructor body) are applied from parent to child.

15.4 A Complete Example: Static + Instance Initialization Across Inheritance

Consider the following three-level hierarchy:

```
class A {
    static int sa = init("A static var");

    static {
        System.out.println("A static block");
    }

    int ia = init("A instance var");

    {
        System.out.println("A instance block");
    }

    A() {
        System.out.println("A constructor");
    }

    static int init(String msg) {
        System.out.println(msg);
        return 0;
    }
}

class B extends A {
    static int sb = init("B static var");   // call to the inherited static method init(String

    static {
        System.out.println("B static block");
    }

    int ib = init("B instance var");    // call to the inherited static method init(String)

    {
        System.out.println("B instance block");
    }

    B() {
        System.out.println("B constructor");
    }
}

class C extends B {
    static int sc = init("C static var");   // call to the inherited static method init(String

    static {
        System.out.println("C static block");
    }

    int ic = init("C instance var");    // call to the inherited static method init(String)

    {
        System.out.println("C instance block");
    }

    C() {
        System.out.println("C constructor");
    }
}

public class Test {
    public static void main(String[] args) {
        new C();
    }
}
```

Output

```
A static var
A static block
B static var
B static block
C static var
C static block
A instance var
A instance block
A constructor
B instance var
B instance block
B constructor
C instance var
C instance block
C constructor
```

15.5 Visualization Diagram

```
            CLASS LOADING (top to bottom)

                A  --->  B  --->  C
                |         |         |
      static vars + static blocks executed in order

-------------------------------------------------------

            OBJECT CREATION (bottom to top)

 new C()
    |
    +--> allocate memory for C (default values)
    +--> call B() constructor
            |
            +--> call A() constructor
                    |
                    +--> init A instance vars
                    +--> run A instance blocks
                    +--> run A constructor
            +--> init B instance vars
            +--> run B instance blocks
            +--> run B constructor
    +--> init C instance vars
    +--> run C instance blocks
    +--> run C constructor
```

15.6 Key Rules

- Static initialization happens **once** per class.
- Static initializers run in parent → child order.
- Instance initialization runs every time an object is created.
- For each class in the inheritance chain, instance fields and instance blocks run before that class's constructor body.
- Overall, both field/instance initialization and constructors execute from parent to child.
- Constructors always call the parent constructor (explicitly or implicitly).

15.7 Summary Table

STATIC (Class Level)	INSTANCE (Object Level)
One-time-only	Happens at every 'new'
Executed parent → child	Instance initialization and constructors parent → child
static vars (default → explicit)	instance vars (default → explicit)
static blocks	instance blocks + constructor

◀ 14. Methods, Attributes and Variables | ▲ Index | 16. Inheritance in Java ▶

16. Inheritance in Java

Table of Contents

- 16.1 General Definition of Inheritance
- 16.2 Single Inheritance and java.lang.Object
- 16.3 Transitive Inheritance
- 16.4 What Gets Inherited Short Reminder
- 16.5 Class Modifiers Affecting Inheritance
- 16.6 this and super References
    - 16.6.1 The this Reference
    - 16.6.2 The super Reference
- 16.7 Declaring Constructors in an Inheritance Chain
- 16.8 Default no-arg Constructor
- 16.9 Using this and Constructor Overloading
- 16.10 Calling the Parent Constructor Using super
- 16.11 Default Constructor Tips and Traps
- 16.12 super Always Refers to the Most Direct Parent
- 16.13 Inheriting Members
    - 16.13.1 Method Overriding
        - 16.13.1.1 Definition and Role in Inheritance
        - 16.13.1.2 Using super to Call the Parent Implementation
        - 16.13.1.3 Overriding Rules Instance Methods
        - 16.13.1.4 Hiding Static Methods Method Hiding
    - 16.13.2 Abstract Classes
        - 16.13.2.1 Definition and Purpose
        - 16.13.2.2 Rules for Abstract Classes
    - 16.13.3 Creating Immutable Objects
        - 16.13.3.1 What Is an Immutable Object
        - 16.13.3.2 Guidelines for Designing Immutable Classes

---

`Inheritance` is one of the core pillars of Object-Oriented Programming.

It allows a class (the *subclass*) to acquire the state and behavior of another class (the *superclass*), creating hierarchical relationships that promote code reuse, specialization, and polymorphism.

16.1 General Definition of Inheritance

Inheritance enables a class to extend another class, automatically gaining its accessible fields and methods.

The extending class may add new features or override existing behaviors, creating more specialized versions of its parent.

> **Note**
>
> Inheritance expresses an *"is-a"* relationship: a Dog **is a** Animal.

16.2 Single Inheritance and java.lang.Object

Java supports **single inheritance**, meaning every class may extend **only one** direct superclass.

All classes ultimately inherit from `java.lang.Object`, which sits at the top of the hierarchy.

This ensures all Java objects share a minimal common behavior (e.g., `toString()`, `equals()`, `hashCode()`).

```
class Animal { }
class Dog extends Animal { }

// All classes implicitly extend Object
System.out.println(new Dog() instanceof Object); // true
```

16.3 Transitive Inheritance

`Inheritance` is **transitive**.

If class `C` extends `B` and `B` extends `A`, then `C` effectively inherits accessible members from both `B` *and* `A`.

```
class A { }
class B extends A { }
class C extends B { } // C inherits from both A and B
```

16.4 What Gets Inherited? (Short Reminder)

A subclass inherits all **accessible** members of the superclass.

However, this depends on `access modifiers`:

- **public** → always inherited
- **protected** → inherited if accessible through package or subclass rules
- **default (package-private)** → inherited only within the same package
- **private** → **NOT** inherited

> **Note**
>
> (Please refer to Paragraph "**Access Modifiers**" in Chapter: 2. Basic Language Java Building Blocks)

16.5 Class Modifiers Affecting Inheritance

Some class-level modifiers affect whether a class may be extended.

Modifier	Meaning	Effect on Inheritance
`final`	Class cannot be extended	Inheritance STOP
`abstract`	Class cannot be instantiated	Must be extended
`sealed`	Only allows a fixed list of subclasses	Restricts inheritance
`non-sealed`	Subclass of sealed class that reopens inheritance	Inheritance allowed
`static`	Applies only to nested classes	Behaves like a top-level class inside its enclosing class

Note

A `static` class in Java can exist only as a **static nested class**.

16.6 `this` and `super` References

16.6.1 The `this` Reference

The `this` reference refers to the current object instance and helps in disambiguing access to current and inherited members.

Java uses a **granular scope** rule:

- If a method/local variable has the same name as an instance field, the local one "shadows" the field.
- `this.fieldName` is required to access the instance attribute.

```java
public class Person {
    String name;

    public Person(String name) {
        // Without "this", we would reassign the parameter to itself
        this.name = name;
    }
}
```

If names differ, `this` is optional:

```java
public class Person {
    String name;

    public Person(String n) {
        name = n; // fine, the names of the variables differ: there is no ambiguity, no shadow
    }
}
```

Warning

`this` cannot be used inside static methods because, in that context, no instance exists.

16.6.2 The `super` Reference

The `super` reference gives access to members of the direct parent class.

Useful when:

- The parent and child define a field/method with the same name: See below: Inheriting Members
- Parent and child define a field with the same name → variable hiding (two copies)
- Parent and child define a method with the same signature → method overriding
- You want to explicitly call the inherited implementation

```
class Parent { int value = 10; }

class Child extends Parent {
    int value = 20;

    void printBoth() {
        System.out.println(value);       // child value
        System.out.println(super.value); // parent value
    }
}
```

> **Note**
>
> `super` cannot be used inside static contexts.

16.7 Declaring Constructors in an Inheritance Chain

A `constructor` initializes a newly created object.

Constructors are **never inherited**, but each subclass constructor must ensure that the superclass is initialized.

`Constructors` are special methods with a name that matches the name of the class and that does not declare any return type.

A class may define multiple constructors (constructor overloading), each with a unique `signature`.

You can explicitly declare a `no-arg` or a specific constructor or, if you don't, Java will implicitly create a `default no-arg constructor`.

If you explicitly declare a constructor, the Java compiler will not include any `default no-arg constructor`: this rule applies independently to every class in the hierarchy.

A parent class still gets its own default constructor unless it also defines one.

16.8 Default `no-arg` Constructor

If a class does not declare any constructor, Java automatically inserts a **default no-argument constructor**.

This constructor calls `super()` implicitly: the Java compiler implicitly insert a call to the no-arg constructor super().

```
class Parent { }

class Child extends Parent {
    // Compiler inserts:
    // Child() { super(); }
}
```

16.9 Using `this()` and Constructor Overloading

this() calls another constructor in the same class.

Rules:

- Must be the **first** statement in the constructor
- Cannot be combined with `super()`
- Only one call to this() is allowed in a constructor
- Used to centralize initialization

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

    Car() {
        this(2020, "Unknown");
    }

    Car(int year, String model) {
        this.year = year;
        this.model = model;
    }
}
```

16.10 Calling the Parent Constructor Using `super()`

Every constructor must call a superclass constructor, either explicitly or implicitly.

`super()` must appear as the **first** line in the constructor (unless replaced by `this()`).

```
class Parent {
    Parent() { System.out.println("Parent constructor"); }
}

class Child extends Parent {
    Child() {
        super(); // optional, compiler would insert it
        System.out.println("Child constructor");
    }
}
```

16.11 Default Constructor — Tips and Traps

- **If the superclass does not have a no-arg constructor, the subclass MUST call `super(args)` explicitly.**
- If the subclass defines any constructor, Java does NOT create a default constructor automatically for that subclass.
- If you forget to call an existing parent constructor explicitly, the compiler inserts `super()` — which may not exist.

```
class Parent {
    Parent(int x) { }
}

class Child extends Parent {
    // ERROR → compiler inserts super(), but Parent() does not exist
    Child() { }
}
```

16.12 `super()` Always Refers to the Most Direct Parent

Even in long inheritance chains, `super()` always calls the constructor of the **immediate** superclass, not any higher ancestor.

```java
class A {
    A() { System.out.println("A"); }
}
class B extends A {
    B() { System.out.println("B"); }
}
class C extends B {
    C() {
        super(); // Calls B(), not A()
        System.out.println("C");
    }
}
```

Output:

```
A
B
C
```

16.13 Inheriting Members

In Java, **field access and static method calls** are resolved at compile time,
while **instance method calls** are resolved at runtime.

The key distinction is:

- The `variable` or `static method` that is used depends on the **declared type of the reference**.
- The `instance method` that is executed depends on the **actual runtime type of the object**.

Example: `Field Access` (Not Polymorphic)

Fields are resolved based on the **declared type of the reference**, not the actual object.

```java
class Parent {
    String name = "Parent";
}

class Child extends Parent {
    String name = "Child";
}

Parent p = new Child();
System.out.println(p.name);    // Output: Parent
```

Explanation:

- The reference `p` is declared as type `Parent`.
- Field access is determined at compile time.
- Therefore, `Parent.name` is used, even though the object is a `Child`.

> **Important**
>
> - Fields are **not polymorphic**.

Example: `Static Methods` (Not Polymorphic)

Static methods are also resolved using the **declared type of the reference**.

```java
class Parent {
    static void print() {
        System.out.println("Parent static");
    }
}

class Child extends Parent {
    static void print() {
        System.out.println("Child static");
    }
}

Parent p = new Child();
p.print();   // Output: Parent static
```

Explanation:

- Static methods are bound at compile time.
- The method chosen depends on the reference type (`Parent`), not the object type.

> **Important**
>
> - This is called **method hiding**, not overriding.

Example: `Instance Methods` (Polymorphic)

Instance methods are resolved at runtime based on the **actual object type**.

```java
class Parent {
    void print() {
        System.out.println("Parent instance");
    }
}

class Child extends Parent {
    @Override
    void print() {
        System.out.println("Child instance");
    }
}

Parent p = new Child();
p.print();   // Output: Child instance
```

Explanation:

- The reference type is `Parent` .
- The actual object is `Child` .
- Java uses dynamic dispatch.
- Therefore, `Child.print()` is executed.

> **Important**
>
> - Instance methods are **polymorphic**.

16.13.1 Method Overriding

Method overriding is a core concept of inheritance: it allows a subclass to provide a **new implementation** for a method that is already defined in its superclass.

At runtime, the version of the method that is executed depends on the **actual object type**, not on the reference type.

This is called **dynamic dispatch** and it is what enables polymorphism in Java.

16.13.1.1 Definition and Role in Inheritance

A method in a subclass **overrides** a method in its superclass if:

- the superclass method is `instance` (non static).
- the subclass method has the same name, the same parameter list and a return type which is the same type or a subtype of the return type in the inherited method.
- when the return type of the overridden method (i.e. the method in the base/super class) is a **primitive**, the return type of the overriding method (i.e. the method in the sub class) must match the return type of the overridden method.
- both methods are accessible (not private) and the subclass method is NOT less visible than the superclass one.
- The overriding method cannot declare new or broader checked exceptions.

Overriding is used to specialize behavior: a subclass can adapt or refine what the parent class does while still being used through a reference of the parent type.

```
class Animal {
    void speak() {
        System.out.println("Some generic animal sound");
    }
}

class Dog extends Animal {

    @Override
    void speak() {
        System.out.println("Woof!");
    }
}

public class TestOverride {
    public static void main(String[] args) {
        Animal a = new Dog(); // reference type = Animal, object type = Dog
        a.speak(); // prints "Woof!" (Dog implementation)
    }
}
```

You cannot override an instance method with a static method, nor override a static method with an instance method.

However, a subinterface is allowed to redeclare a static method from a superinterface as a `default` method.

- Example:

```
class Alpha {
    static void a() { }
    void b() { }
    static void c() { }
    void d() { }
}

class Beta extends Alpha {
    void a() { }        // DOES NOT COMPILE (cannot override static with instance)
    static void b() { } // DOES NOT COMPILE (cannot override instance with static)
    static void c() { } // VALID, c() in Alpha is hidden
    void d() { }        // VALID, d() in Alpha is overridden
}
```

16.13.1.2 Using `super` to Call the Parent Implementation

When a subclass overrides a method, it can still access the superclass implementation via the `super` reference.

This is useful if you want to reuse or extend the behavior defined in the parent class.

```java
class Person {
    void introduce() {
        System.out.println("I am a person.");
    }
}

class Student extends Person {
    @Override
    void introduce() {
        super.introduce(); // calls Person.introduce()
        System.out.println("I am also a student.");
    }
}
```

If both parent and child declare a member (field or method) with the same name, The child can access both:

- the overridden version (default)
- the parent version via `super`

```java
class Base {
    int value = 10;

    void show() {
        System.out.println("Base value = " + value);
    }
}

class Derived extends Base {
    int value = 20; // hides Base.value

    @Override
    void show() {
        System.out.println("Derived value = " + value);           // 20
        System.out.println("Base value via super = " + super.value); // 10
    }
}
```

16.13.1.3 Overriding Rules (Instance Methods)

- **Same signature**: same method name, same parameter types and order.
- **Covariant return type**: the overriding method can return the same type as the parent, or a **subtype** of the parent return type.
- **Accessibility**: the overriding method cannot be less accessible than the overridden one (for example, cannot change from public to protected or private). It can keep the same visibility or increase it.
- **Checked exceptions**: the overriding method cannot declare new or broader checked exceptions than the parent method; it may declare fewer or more specific checked exceptions, or remove them entirely.
- **Unchecked exceptions**: can be added or removed without restriction.
- **final methods**: cannot be overridden.

```java
class Parent {
    Number getValue() throws Exception {
        return 42;
    }
}

class Child extends Parent {
@Override
    // Covariant return type: Integer is a subclass of Number
    Integer getValue() throws RuntimeException {
        return 100;
    }
}
```

16.13.1.4 Hiding Static Methods (Method Hiding)

Static methods are **not overridden**; they are **hidden**.

If a subclass defines a static method with the same signature as a static method in the parent, the subclass method **hides** the parent method.

If one of the methods is marked as `static` and the other is not, the code will NOT compile.

Method selection for static methods happens at **compile time**, based on the reference type, not the object type.

```
class A {
    static void show() {
        System.out.println("A.show()");
    }
}

class B extends A {
    static void show() {
        System.out.println("B.show()");
    }
}

public class TestStatic {
    public static void main(String[] args) {
        A a = new B();
        B b = new B();

        a.show(); // A.show()   (reference type A)
        b.show(); // B.show()   (reference type B)
    }
}
```

> **Important**
>
> - **final** static methods cannot be hidden, and instance methods declared **final** cannot be overridden.
> - If you try to redefine them in a subclass, the code will not compile.

16.13.2 Abstract Classes

16.13.2.1 Definition and Purpose

An **abstract class** is a class that cannot be instantiated directly and is intended to be extended.

It may contain:

- abstract methods (declared without a body);
- concrete methods (with implementation);
- fields, constructors, static members, and even static initializers.

Abstract classes are used when you want to define a common **base behavior** and contract, but leave some details to be implemented by concrete subclasses.

16.13.2.2 Rules for Abstract Classes

- A class with at least one abstract method **must** be declared abstract.
- An abstract class **cannot** be instantiated directly.
- Abstract methods have no body and end with a semicolon.
- **Abstract methods cannot be `final`, `static`, or `private`**, because they must be overridable.
- The first concrete (non-abstract) subclass in the hierarchy must implement all inherited abstract methods, otherwise it must itself be declared abstract.

```java
abstract class Shape {

    abstract double area(); // must be implemented by concrete subclasses

    void describe() {
        System.out.println("I am a shape.");
    }

    Shape() {
        System.out.println("Shape constructor");
    }
}

class Circle extends Shape {
    private final double radius;

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

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

> **Note**
>
> - Although an abstract class cannot be instantiated, its constructors are still called when creating instances of concrete subclasses.
> - The chain always starts from the top of the hierarchy and moves down.

16.13.3 Creating Immutable Objects

16.13.3.1 What Is an Immutable Object?

An object is **immutable** if, after it has been created, its state **cannot change**.

All fields that represent the state remain constant for the lifetime of that object.

Immutable objects are simpler to reason about, inherently thread safe (if properly designed), and widely used in the Java Standard Library (for example, `String`, wrapper classes like `Integer`, and many classes in `java.time`).

16.13.3.2 Guidelines for Designing Immutable Classes

- Declare the class **final** so it cannot be subclassed (or make all constructors private and provide controlled factory methods).
- Make all fields that represent state **private** and **final**.
- Do not provide any mutator (setter) methods.
- Initialize all fields in constructors (or factory methods) and never expose them in a mutable way.
- If a field refers to a mutable object, make **defensive copies** on construction and when returning it via getters.

```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public final class Person {
    private final String name; // String is immutable
    private final int age;
    private final List<String> hobbies; // List is mutable, we must protect it

    public Person(String name, int age, List<String> hobbies) {
        this.name = name;
        this.age = age;
        // Defensive copy on input
        this.hobbies = new ArrayList<>(hobbies);
    }

    public String getName() {
        return name; // safe (String is immutable)
    }

    public int getAge() {
        return age;
    }

    public List<String> getHobbies() {
        // Defensive copy or unmodifiable view on output
        return Collections.unmodifiableList(hobbies);
    }
}
```

In this example:

- `Person` is final: it cannot be subclassed and its behavior cannot be altered through inheritance.
- All fields are `private` and `final`, set only once in the constructor.
- The list of hobbies is defensively copied on construction and wrapped as unmodifiable in the getter, so external code cannot modify the internal state.

Designing immutable objects is especially important in multithread contexts and when passing objects across layers of an application.

◀ 15. Class Loading, Initialization, and Object Construction | ▲ Index | 17. Beyond Classes ▶

17. Beyond Classes

Table of Contents

- 17.1 Interfaces
  - 17.1.1 What Interfaces Can Contain
  - 17.1.2 Implementing an Interface
  - 17.1.3 Multiple Inheritance
  - 17.1.4 Interface Inheritance and Conflicts
  - 17.1.5 Default methods
  - 17.1.6 Static methods
  - 17.1.7 Private interface methods
- 17.2 Sealed, non-sealed, and final Types
  - 17.2.1 Rules
- 17.3 Enums
  - 17.3.1 Simple Enum Definition
  - 17.3.2 Complex Enums with State and Behavior
  - 17.3.3 Enum Methods
  - 17.3.4 Rules
  - 17.4 Records Java 16+
  - 17.4.1 Summary of Basic Rules for Records
  - 17.4.2 Long Constructor
  - 17.4.3 Compact Constructor
  - 17.4.4 Pattern Matching for Records
  - 17.4.5 Nested Record Patterns and Matching Records with var and Generics
    - 17.4.5.1 Basic Nested Record Pattern
    - 17.4.5.2 Nested Record Patterns with var
    - 17.4.5.3 Nested Record Patterns and Generics
    - 17.4.5.4 Common Errors with Nested Record Patterns
- 17.5 Nested Classes in Java
  - 17.5.1 Static Nested Classes
    - 17.5.1.1 Syntax and Access Rules
    - 17.5.1.2 Common Pitfalls
  - 17.5.2 Inner Classes Non-Static Nested Classes
    - 17.5.2.1 Syntax and Access Rules
    - 17.5.2.2 Common Pitfalls
  - 17.5.3 Local Classes
    - 17.5.3.1 Characteristics
    - 17.5.3.2 Common Pitfalls
  - 17.5.4 Anonymous Classes
    - 17.5.4.1 Syntax and Usage
    - 17.5.4.2 Anonymous Class Extending a Class
  - 17.5.5 Comparison of Nested Class Types
- 17.6 Nesting of Interfaces in Java
  - 17.6.1 Where an Interface Can Be Declared
  - 17.6.2 Nested Interfaces

        - 17.6.2.1 Interface Nested Inside a Class
        - 17.6.2.2 Interface Nested Inside Another Interface
    - 17.6.3 Access Rules
    - 17.6.4 Nested Types Inside Interfaces
    - 17.6.5 Essential Summary

This chapter presents several advanced type mechanisms beyond the Java Class design: **interfaces**, **enums**, **sealed / non-sealed classes**, **records**, and **nested classes**.

17.1 Interfaces

An **interface** in Java is a reference type that defines a contract of methods that a class agrees to implement.

An `interface` is implicitly `abstract` and cannot be marked as `final`: as with top-level classes, an interface can declare visibility as `public` or `default` (package-private).

A Java class may implement any number of interfaces through the `implements` keyword.

An `interface` may in turn extend multiple interfaces using the `extends` keyword.

Interfaces enable abstraction, loose coupling, and multiple inheritance of type.

17.1.1 What Interfaces Can Contain

- **Abstract methods** (implicitly `public` and `abstract`)
- **Concrete methods**
    - **Default methods** (include code and are implicitly `public`)
    - **Static methods** (declared as `static`, include code and are implicitly `public`)
    - **Private methods** (Java 9+) for internal reuse
- **Constants** → implicitly `public static final` and initialized at declaration

```
interface Calculator {
    int add(int a, int b);                    // abstract
    default int mult(int a, int b) {          // default method
        return a * b;
    }
    static double pi() { return 3.14; }       // static method
}
```

Warning

Because interface abstract methods are implicitly `public`, you **cannot** reduce the access level on an implementing method.

17.1.2 Implementing an Interface

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

Note

Every abstract method must be implemented unless the class is abstract.

17.1.3 Multiple Inheritance

A class may implement multiple interfaces.

```
interface A { void a(); }
interface B { void b(); }

class C implements A, B {
    public void a() {}
    public void b() {}
}
```

17.1.4 Interface Inheritance and Conflicts

If two interfaces provide `default` methods with the same signature, the implementing class must override the method.

```
interface X { default void run() { } }
interface Y { default void run() { } }

class Z implements X, Y {
    public void run() { } // mandatory
}
```

If you still want to access a particular implementation of the inherited `default` method, you can use the following syntax:

```
interface X { default int run() { return 1; } }
interface Y { default int run() { return 2; } }

class Z implements X, Y {
    public int useARun(){
        return Y.super.run();
    }
}
```

17.1.5 `Default` methods

A `default` method (declared with the `default` keyword) is a method that defines an implementation and can be overridden by a class implementing the interface.

- A default method includes code and is implicitly `public`;
- A default method cannot be `abstract`, `static`, or `final`;
- An interface can redeclare a default method and provide a different implementation ;
- A subinterface is allowed to redeclare a static method from a superinterface as a `default` method.
- As we saw just above, if two interfaces provide default methods with the same signature, the implementing class must override the method;
- An implementing class may of course rely on the provided implementation of the `default` method without overriding it;
- The `default` method can be invoked on an instance of the implementing class and NOT as a `static` method of the containing interface;
- A class (or an interface) can explicitly invoke a default method of an interface that is directly listed in its `implements` clause (or `extends` clause) by using the syntax `InterfaceName.super.methodName()`; this is typically used to disambiguate multiple inherited default methods;
- This syntax can only be used if the interface is explicitly mentioned in the `implements` (or `extends`) clause; it cannot be used to invoke a default method from an indirectly inherited interface;
- The `InterfaceName.super.methodName()` syntax applies only to `default` methods and cannot be used for abstract methods, static methods, private interface methods, or fields.

Example: Valid Usage

```java
interface A {
    default void hello() {
        System.out.println("Hello from A");
    }
}

class B implements A {

    @Override
    public void hello() {
        A.super.hello();  // ✅ allowed
        System.out.println("Hello from B");
    }
}
```

Example: Invalid Usage

```java
interface A {
    default void hello() {
        System.out.println("Hello from A");
    }
}

interface B extends A {
}

class C implements B {

    public void test() {
        A.super.hello();  // ❌ compilation error
    }
}
```

> **Note**
>
> - A subinterface is allowed to redeclare a static method from a superinterface as a `default` method.

Example:

```java
interface Parent {
    static void p() { }
}

interface Child extends Parent {
    default void p() { } // VALID, static method redeclared as default
}
```

> **Note**
>
> - An interface is allowed to redeclare a `default` method inherited from a superinterface and turn it into an `abstract` method.
>
> When this happens, the `default` implementation from the superinterface is effectively removed in the subinterface. As a consequence, any class that implements the subinterface will NOT inherit the original `default` implementation and must provide its own implementation.

Example:

```java
interface Parent {
    default void greet() {
        System.out.println("Hello from Parent");
    }
}

interface Child extends Parent {
    void greet();   // redeclared as abstract
}

class Demo implements Child {
    public void greet() {   // mandatory
        System.out.println("Hello from Demo");
    }
}
```

Explanation:

- `Parent` provides a default implementation of `greet()`.
- `Child` redeclares `greet()` without `default`, making it abstract again.
- `Demo` cannot inherit the default implementation from `Parent`.
- Therefore, `Demo` must implement `greet()` explicitly.

17.1.6 `Static` methods

- An interface can provide `static methods` (through the keyword `static`) which are implicitly `public`;
- Static methods must include a method body and are accessed using the interface name;
- Static methods cannot be `abstract` or `final`;

17.1.7 `Private` interface methods

Among all the concrete methods that an interface can implement, we also have:

- **`private` methods**: visible only inside the declaring interface and which can only be invoked from a `non-static` context (`default` methods or other `non-static private methods`).
- **`private static` methods**: visible only inside the declaring interface and which can be invoked by any method of the enclosing interface.

17.2 `Sealed`, `non-sealed`, and `final` Types

`Sealed` classes and interfaces (Java 17+) restrict which other classes (or interfaces) can extend or implement them.

A `sealed type` is declared by placing the `sealed` modifier right before the class (or interface) keyword, and adding, after the Type name, the `permits` keyword followed by the list of types that can extend (or implement) it.

```java
public sealed class Shape permits Circle, Rectangle { }

final class Circle extends Shape { }

non-sealed class Rectangle extends Shape { }
```

17.2.1 Rules

- A sealed Type must declare all permitted subtypes.
- A permitted subtype must be **final**, **sealed**, or **non-sealed**; because interfaces cannot be final, they can only be marked `sealed` or `non-sealed` when extending a sealed interface.
- If a sealed class belongs to a `named` module, then all the classes listed in its permits clause must also belong to that `same module`.
- If a sealed class belongs to an `unnamed module`, then all the classes listed in its permits clause must be declared in the `same package`.

17.3 Enums

Enums define a fixed set of constant values.

`Enums` can declare `fields`, `constructors`, and `methods` as regular classes do but **they can't be extended**.

The list of enum values must end with a semicolon `(;)` in case of `Complex Enums`, but this is not mandatory for `Simple Enums`.

17.3.1 `Simple` Enum Definition

```
enum Day { MON, TUE, WED, THU, FRI, SAT, SUN } // semicolon not present
```

17.3.2 `Complex` Enums with State and Behavior

```
enum Level {
    LOW(1), MEDIUM(5), HIGH(10); // mandatory semicolon

    private int code;

    Level(int code) { this.code = code; }

    public int getCode() { return code; }
}

public static void main(String[] args) {
    Level.MEDIUM.getCode();      // invoking a method
}
```

17.3.3 Enum Methods

- `values()` – returns an array of all the constant values that can be used, for example, in a `for-each` loop
- `valueOf(String)` – returns constant by name
- `ordinal()` – index (int) of the constant

17.3.4 Rules

- **Enum constructors are implicitly `private`**;
- Enums can contain `static` and `instance` methods;
- Enums can implement `interfaces`;
- Enums cannot be extended.

17.4 Records (Java 16+)

A **record** is a special class designed to model immutable data: they are, in fact, implicitly **final**.

You can't extend a record, but it is allowed to implement a `regular` or `sealed interface`.

It automatically provides:

- **private final fields** for each component;
- **constructor** with parameters in the same order as in the record declaration;
- **getters** (named like fields);
- **`equals()`, `hashCode()`, `toString()`**: you are also permitted to override those methods;
- **Records** can include `nested classes`, `interfaces`, `records`, `enums` and `annotations`.

```
public record Point(int x, int y) { }

var element = new Point(11, 22);

System.out.println(element.x);
System.out.println(element.y);
```

If you need additional validation or transformation of the provided fields, you can define a `long constructor` or a `compact constructor`.

17.4.1 Summary of Basic Rules for Records

A record may be declared in three locations:

- As a **top-level record** (directly in a package)
- As a **member record** (inside a class or interface)
- As a **local record** (inside a method)

All `member` and `local` record classes are implicitly `static`.

- A member record may redundantly declare `static`.
- A local record must not declare `static` explicitly.

Every record class is implicitly `final`.

- Declaring `final` explicitly is permitted but redundant.
- A record cannot be declared `abstract`, `sealed`, or `non-sealed`.

The direct superclass of every record is `java.lang.Record`.

- A record cannot declare an `extends` clause.
- A record cannot extend any other class.

Serialization of records differs from ordinary serializable classes.

- During deserialization, the canonical constructor is invoked.

The body of a record may contain:

- Constructors
- Methods
- Static fields
- Static initializer blocks

The body of a record must NOT contain:

- Instance field declarations
- Instance initializer blocks
- `abstract` methods
- `native` methods

17.4.2 Long Constructor

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

    public Person (String name, int age){
        if (age < 0) throw new IllegalArgumentException();
        this.name = name;
        this.age = age;
    }
}
```

You can still define overloaded constructors, as long as they ultimately delegate to the canonical one using `this(...)`:

```java
public record Point(int x, int y) {

    // Overloaded constructor (NOT canonical)
    public Point(int value) {
        this(value, value); // must call, in the first line, another overloaded constructor ar
    }
}
```

> **Note**
>
> - The compiler will not insert a constructor if you manually provide one with the same list of parameters in the defined order;
> - In this case, you must explicitly set every field manually;

17.4.3 Compact Constructor

You can define a `compact constructor` which implicitly sets all fields, while letting you perform validations and transformations on selected fields.

Java will execute the full constructor, setting all fields, after the compact constructor has completed.

```java
public record Person(String name, int age) {

    public Person {
        if (age < 0) throw new IllegalArgumentException();

        name = name.toUpperCase(); // This transformation is still (at this level of initializ

        // this.name = name; // ❌ Does not compile.
    }
}
```

> **Warning**
>
> - If you try to modify a Record field inside a Compact Constructor, your code will not compile

17.4.4 Pattern Matching for Records

When you use pattern matching with `instanceof` or with `switch`, a record pattern must specify:

- The record type;
- A pattern for each field of the record (matching the correct number of components, and compatible types);

Example record:

```java
Object obj = new Point(3, 5);

if (obj instanceof Point(int a, int b)) {
    System.out.println(a + b);    // 8
}
```

17.4.5 Nested Record Patterns and Matching Records with `var` and Generics

Nested record patterns allow you to destructure records that contain other records or complex types, extracting values recursively directly within the pattern itself.

They combine the power of `record` deconstruction with pattern matching, giving you a concise and expressive way to navigate hierarchical data structures.

17.4.5.1 Basic Nested Record Pattern

If a record contains another record, you can destructure both at once:

```java
record Address(String city, String country) {}
record Person(String name, Address address) {}

void printInfo(Object obj) {

    switch (obj) {
        case Person(String n, Address(String c, String co)) -> System.out.println(n + " lives
        default -> System.out.println("Unknown");
    }
}
```

In the example above, the `Person` pattern includes a nested `Address` pattern.

Both are matched structurally.

17.4.5.2 Nested Record Patterns with `var`

Instead of specifying exact types for each field, you can use `var` inside the pattern to let the compiler infer the type.

```java
    switch (obj) {
        case Person(var name, Address(var city, var country)) -> System.out.println(name + " -
    }
```

`var` in patterns works like `var` in local variables: it means "infer the type".

> **Warning**
>
> - You still need the enclosing record type (Person, Address);
> - only the field types can be replaced with `var`.

17.4.5.3 Nested Record Patterns and Generics

Record patterns also work with generic records.

```java
record Box<T>(T value) {}
record Wrapper(Box<String> box) {}

static void test(Object o) {
    switch (o) {
        case Wrapper(Box<String>(var v)) -> System.out.println("Boxed string: " + v);
        default -> System.out.println("Something else");
    }
}
```

In this example:

- The pattern requires exactly `Box<String>`, not `Box<Integer>`.
- Inside the pattern, `var v` captures the unboxed generic value.

17.4.5.4 Common Errors with Nested Record Patterns

Mismatched record structure

```java
// ❌ ERROR: pattern does not match record structure
case Person(var n, var city) -> ...
```

`Person` has 2 fields, but one of them is a record. You must destructure correctly.

Wrong number of components

```java
// ❌ ERROR: Address has 2 components, not 1
case Person(var n, Address(var onlyCity)) -> ...
```

Generic mismatch

```
// ❌ ERROR: expecting Box<String> but found Box<Integer>
case Wrapper(Box<Integer>(var v)) -> ...
```

Illegal placement of `var`

```
// ❌ var cannot replace the record type itself
case var(Person(var n, var a)) -> ...
```

> **Note**
>
> - `var` cannot stand in for the whole pattern, only for individual components.

17.5 Nested Classes in Java

Java supports several kinds of **nested classes** — classes declared inside another class.

They are a fundamental tool for encapsulation, code organization, event-handling patterns, and representing logical hierarchies.

A nested class always belongs to an **enclosing class** and has special accessibility and instantiation rules depending on its category.

Java defines four kinds of nested classes:

- **Static Nested Classes** – declared with `static` inside another class.
- **Inner Classes** (non-static **nested** classes).
- **Local Classes** – declared inside a block (method, constructor, or initializer).
- **Anonymous Classes** – unnamed classes created inline, usually to override a method or implement an interface.

> **Warning**
>
> - `static` applies only to **nested member** classes
> - `Top-level` classes → cannot be static
> - `Local` classes (inside methods) → cannot be static
> - `Anonymous` classes → cannot be static
> - A `static nested` class cannot access instance members without an explicit outer object reference.

17.5.1 Static Nested Classes

A **static nested class** behaves like a top-level class that is namespaced inside its enclosing class.
It **cannot** access instance members of the outer class but **can** access static members.
It does **not** hold a reference to an instance of the enclosing class. A static nested class can contain non-static member variables.

17.5.1.1 Syntax and Access Rules

- Declared using `static class` inside another class.
- Can access only **static** members of the outer class.
- Does not have an implicit reference to the enclosing instance.
- Can be instantiated without an outer instance.
- Can contain non-static member variables.

```
class Outer {
    static int version = 1;

    static class Nested {
        void print() {
            System.out.println("Version: " + version); // OK: accessing static member
        }
    }
}

class Test {
    public static void main(String[] args) {
        Outer.Nested n = new Outer.Nested(); // No Outer instance required
        n.print();
    }
}
```

17.5.1.2 Common Pitfalls

- Static nested classes **cannot access instance variables**:

```
class Outer {
    int x = 10;
    static class Nested {
        void test() {
            // System.out.println(x); // ❌ Compile error
        }
    }
}
```

17.5.2 Inner Classes (Non-Static Nested Classes)

An **inner class** is associated with an instance of the outer class and can access **all members** of the outer class, including **private** ones.

17.5.2.1 Syntax and Access Rules

- Declared without `static`.
- Has an implicit reference to the enclosing instance.
- Can access both static and instance members of the outer class.
- Since it is not static, it must be created through an instance of the enclosing class.

```
class Outer {
    private int value = 100;

    class Inner {
        void print() {
            System.out.println("Value = " + value); // OK: accessing private
        }
    }

    void make() {
        Inner i = new Inner(); // OK inside the outer class
        i.print();
    }
}

class Test {
    public static void main(String[] args) {
        Outer o = new Outer();
        Outer.Inner i = o.new Inner(); // MUST be created from an instance
        i.print();
    }
}
```

Inside the `non-static` inner class, you can refer to the enclosing object using `OuterClass.this` and `InnerClass.this`, which is equivalent to `this`, refers to the current Inner object:

- Example:

```
class Outer {
    int x = 10;

    class Inner {
        int x = 20;

        void print() {
            System.out.println(x);              // 20 (Inner.this.x)
            System.out.println(this.x);         // 20
            System.out.println(Outer.this.x);   // 10
        }
    }
}
```

17.5.2.2 Common Pitfalls

- Inner classes **cannot declare static members** except **static final constants**.

```
class Outer {
    class Inner {
        // static int x = 10;     // ❌ Compile error
        static final int OK = 10; // ✓ Allowed (constant)
    }
}
```

> **Warning**
>
> - Instantiating an inner class WITHOUT an outer instance is illegal.

17.5.3 Local Classes

A **local class** is a nested class defined inside a block — most commonly a method.

It has no access modifier and is visible only within the block where it is declared.

17.5.3.1 Characteristics

- Declared inside a method, constructor, or initializer.
- Can access members of the outer class.
- Can access local variables if they are **effectively final**.
- Cannot declare static members (except static final constants).

```
class Outer {
    void compute() {
        int base = 5; // must be effectively final

        class Local {
            void show() {
                System.out.println(base); // OK
            }
        }

        new Local().show();
    }
}
```

A local class, just like in a member inner class, has an implicit reference to the enclosing instance using `OuterClass.this` and also `LocalClass.this`, equivalent to `this`, is valid inside the local class body.

- Example:

```java
class Outer {
    int x = 10;

    void method() {
        class Local {
            void print() {
                System.out.println(Outer.this.x);  // ✓ valid

                System.out.println(Local.this);   // ✓ valid
            }
        }
    }
}
```

17.5.3.2 Common Pitfalls

- `base` must be effectively final; changing it breaks compilation.

```java
void compute() {
    int base = 5;
    base++; // ❌ Now base is NOT effectively final
    class Local {}
}
```

17.5.4 Anonymous Classes

An **anonymous class** is a one-off class created inline, usually to implement an interface or override a method without naming a new class.

17.5.4.1 Syntax and Usage

- Created using `new` + type + body.
- Cannot have constructors (no name).
- Often used for event handling, callbacks, comparators.

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

17.5.4.2 Anonymous Class Extending a Class

```java
Button b = new Button("Click");
b.onClick(new ClickHandler() {
    @Override
    public void handle() {
        System.out.println("Handled!");
    }
});
```

17.5.5 Comparison of Nested Class Types

A quick table summarizing all kinds of nested classes.

Type	Has Outer Instance?	Can Access Outer Instance Members?	Can Have Static Members?	Typical Use
Static Nested	No	No	Yes	Namespacing, helpers
Inner Class	Yes	Yes	No (except constants)	Object-bound behavior
Local Class	Yes	Yes	No	Temporary scoped classes
Anonymous Class	Yes	Yes	No	Inline customization

17.6 Nesting of Interfaces in Java

In Java, an interface can be declared in different locations and follows specific rules regarding nesting and permitted members.

17.6.1 Where an Interface Can Be Declared

An interface can be:

- **Top-level** (directly inside a package)
- **Nested member interface** (declared inside a class or another interface)
- **Local interface** ❌ (not allowed)
- **Anonymous interface** ❌ (does not exist as a declaration, only anonymous implementations exist)

In Java, it is **not permitted to declare a local interface** (that is, inside a method or block). Interfaces can only be `top-level` or `member`.

17.6.2 Nested Interfaces

A Nested Interface can be declared inside:

17.6.2.1 Interface Nested Inside a Class

- It is implicitly `static`
- It cannot be declared `non-static`
- It may be declared `public`, `protected`, `private`, or `package-private`
- Example:

```
class Outer {
    interface InnerInterface {
        void test();
    }
}
```

The `static` keyword is implicit:

```
class Outer {
    static interface InnerInterface {   // allowed but redundant
        void test();
    }
}
```

17.6.2.2 Interface Nested Inside Another Interface

- It is implicitly `public` and `static`
- It cannot be `private` or `protected`

```
interface A {
    interface B {
        void test();
    }
}
```

17.6.3 Access Rules

A `nested interface`:

- Does not have an implicit reference to an instance of the enclosing class
- Cannot directly access instance members of the enclosing class
- **Can access only `static` members of the enclosing class**

17.6.4 Nested Types Inside Interfaces

An interface may contain:

- Nested classes (implicitly `public static`)
- Nested records (implicitly `public static`)
- Nested enums (implicitly `public static`)
- Other nested interfaces (implicitly `public static`)

17.6.5 Essential Summary

- Nested interfaces are always `static`
- Local interfaces do not exist
- Fields are always `public static final`
- Methods are implicitly `public abstract` (except default/static/private)
- They may contain other nested types

◀ 16. Inheritance in Java | ▲ Index | 18. Generics in Java ▶

18. Generics in Java

Table of Contents

- 18.1 Generic Type Basics
- 18.2 Why Generics Exist
- 18.3 Generic Methods
- 18.4 Type Erasure
  - 18.4.1 How Type Erasure Works
  - 18.4.2 Erasure of Unbounded Type Parameters
  - 18.4.3 Erasure of Bounded Type Parameters
  - 18.4.4 Multiple Bounds The First Bound Determines Erasure
  - 18.4.5 Why Only the First Bound Becomes the Runtime Type
  - 18.4.6 A More Complex Example
  - 18.4.7 Overriding and Generics
    - 18.4.7.1 How the Compiler Validates an Override
    - 18.4.7.2 Generic Parameters and Overriding
    - 18.4.7.3 Valid Override - Erasing Generic Specificity
    - 18.4.7.4 Invalid Override - Adding Generic Specificity
    - 18.4.7.5 Valid Override - Matching Parameterization
    - 18.4.7.6 Invalid Override - Changing Generic Argument
    - 18.4.7.7 Why This Rule Exists
    - 18.4.7.8 Mental Model
    - 18.4.7.9 Covariant returns and Generics
    - 18.4.7.10 Summary Rules
  - 18.4.8 Overloading a Generic Method Why Some Overloads Are Impossible
  - 18.4.9 Overloading a Generic Method Inherited from a Parent Class
  - 18.4.10 Returning Generic Types Rules and Restrictions
  - 18.4.11 Summary of Erasure Rules
- 18.5 Bounds on Type Parameters
  - 18.5.1 Upper Bounds extends
  - 18.5.2 Multiple Bounds
  - 18.5.3 Wildcards: ?, ? extends, ? super
    - 18.5.3.1 Unbounded Wildcard
    - 18.5.3.2 Upper-Bounded Wildcard extends
    - 18.5.3.3 Lower-Bounded Wildcard super
- 18.6 Generics and Inheritance
- 18.7 Type Inference Diamond Operator
- 18.8 Raw Types Legacy Compatibility
- 18.9 Generic Arrays Not Allowed
- 18.10 Bounded Type Inference
- 18.11 Wildcards vs Type Parameters
- 18.12 PECS Rule Producer Extends Consumer Super
- 18.13 Common Pitfalls
- 18.14 Summary Table of Wildcards
- 18.15 Summary of Concepts
- 18.16 Complete Example

Java `Generics` allow you to create classes, interfaces, and methods that work with user-specified types, ensuring that only objects of the correct type are used.

All type checks are performed by the compiler at compile time.

During compilation, the compiler verifies type correctness and then removes the generic type information, replacing it with concrete types (a process known as **type erasure**) or with Object when necessary.

The resulting bytecode does not contain generics: it only contains concrete types and, when needed, casts automatically inserted by the compiler.

In this way, type errors are caught before execution, making the code safer, more readable, and more reusable.

Generics apply to:

- `Classes`
- `Interfaces`
- `Methods` (generic methods)
- `Constructors`

18.1 Generic Type Basics

A generic class or interface introduces one or more **type parameters**, enclosed in angle brackets.

```
class Box<T> {
    private T value;
    void set(T v) { value = v; }
    T get()       { return value; }
}

Box<String> b = new Box<>();
b.set("hello");
String x = b.get(); // no cast needed
```

Multiple type parameters are allowed:

```
class Pair<K, V> {
    K key;
    V value;
}
```

18.2 Why Generics Exist

```
List list = new ArrayList();          // pre-generics
list.add("hi");
Integer x = (Integer) list.get(0);    // ClassCastException at runtime
```

With generics:

```
List<String> list = new ArrayList<>();
list.add("hi");
String x = list.get(0);               // type-safe, no cast
```

18.3 Generic Methods

A **generic method** introduces its own type parameter(s), independent of the class.

```java
class Util {
    static <T> T pick(T a, T b) { return a; }
}

String s = Util.<String>pick("A", "B"); // explicit
String t = Util.pick("A", "B");         // inference works
```

18.4 Type Erasure

`Type erasure` is the process by which the Java compiler removes all generic type information before generating bytecode.

This ensures backward compatibility with pre-Java-5 JVMs.

At compile time, generics are fully checked: type bounds, variance, method overloading with generics, etc. However, at runtime, all generic information disappears.

18.4.1 How Type Erasure Works

- Replace all type variables (like `T`) with their erasure.
- Insert casts where needed.
- Remove all generic type arguments (e.g., `List<String>` → `List`).

18.4.2 Erasure of Unbounded Type Parameters

If a type variable has no bound:

```java
class Box<T> {
    T value;
    T get() { return value; }
}
```

The erasure of `T` is `Object`.

```java
class Box {
    Object value;
    Object get() { return value; }
}
```

18.4.3 Erasure of Bounded Type Parameters

If the type parameter has bounds:

```java
class TaskRunner<T extends Runnable> {
    void run(T task) { task.run(); }
}
```

Then the erasure of `T` is the first bound: `Runnable`.

```java
class TaskRunner {
    void run(Runnable task) { task.run(); }
}
```

18.4.4 Multiple Bounds: The First Bound Determines Erasure

Java allows multiple bounds:

```java
<T extends Runnable & Serializable & Cloneable>
```

The critical rule:

> **Important**
>
> The erasure of `T` is always the **first bound**, which must be a class or interface.

Because `Runnable` is the first bound, the compiler erases `T` to `Runnable`.

-Example with Multiple Bounds (Fully Expanded)

```
public static <T extends Runnable & Serializable & Cloneable>
void runAll(List<T> list) {
    for (T t : list) {
        t.run();
    }
}
```

Erased Version

```
public static void runAll(List list) {
    for (Object obj : list) {
        Runnable t = (Runnable) obj;    // cast set by the compiler
        t.run();
    }
}
```

What happens to the other bounds (Serializable, Cloneable)?

- They are enforced only at compile time.
- They do NOT appear in bytecode.
- No additional interfaces are attached to the erased type.

18.4.5 Why Only the First Bound Becomes the Runtime Type?

Because the JVM must operate using a single, concrete reference type for each variable or parameter.

Runtime bytecode instructions like `invokevirtual` require a single class or interface, not a composite type such as “Runnable & Serializable & Cloneable”.

Thus:

> **Note**
>
> Java selects the **first bound** as the runtime type, and uses the remaining bounds for **compile-time validation only**.

18.4.6 A More Complex Example

```
interface A { void a(); }
interface B { void b(); }

class C implements A, B {
    public void a() {}
    public void b() {}
}

class Demo<T extends A & B> {
    void test(T value) {
        value.a();
        value.b();
    }
}
```

Erased Version:

```
class Demo {
    void test(A value) {
        value.a();
        // value.b();   // ❌ not available after erasure: type is A, not B
    }
}
```

> **Note**
>
> The compiler may insert additional casts or bridge methods in more complex inheritance scenarios, but erasure always uses only the first bound (A in this case).

18.4.7 Overriding and Generics

When generics interact with inheritance, two fundamental rules must be clearly understood:

> **Important**
>
> **Override is checked after type erasure.**
>
> **Type compatibility is checked before type erasure.**

These two steps explain why some methods override correctly while others produce compile-time errors.

18.4.7.1 How the Compiler Validates an Override

When a subclass declares a method that *might* override a superclass method, the compiler performs two checks:

1. **Before erasure**
    - The method must be type-compatible with the parent method:
        - Same method name
        - Same parameter types (including generic arguments)
        - Compatible return type (covariant allowed)
2. **After erasure**
    - The erased signatures must match exactly.
        - Both conditions must be satisfied.

18.4.7.2 Generic Parameters and Overriding

Generic type arguments are part of the method signature *at compile time*, but disappear after erasure.

Because of this:

- You are allowed to **erase generic information in the overriding method**
- You are NOT allowed to **introduce new generic specificity**
- If both methods declare parameterized types, they must match exactly

18.4.7.3 Valid Override - Erasing Generic Specificity

```
class Parent {
    void process(Set<Integer> data) {}
}

class Child extends Parent {
    @Override
    void process(Set data) {}   // ✓ allowed (raw type)
}
```

Explanation:

- Before erasure: `Set` is assignment-compatible with `Set<Integer>`
- After erasure: both become `Set`

✓ Valid override.

18.4.7.4 Invalid Override - Adding Generic Specificity

```
class Parent {
    void process(Set data) {}
}

class Child extends Parent {
    void process(Set<Integer> data) {}    // ❌ compile error
}
```

Explanation:

- Before erasure: `Set<Integer>` is NOT assignment-compatible with `Set`
- The compiler rejects it before even considering erasure

18.4.7.5 Valid Override - Matching Parameterization

```
class Parent {
    void process(Set<Integer> data) {}
}

class Child extends Parent {
    @Override
    void process(Set<Integer> data) {}    // ✓ exact match
}
```

Both checks pass: - Compatible before erasure - Identical after erasure

18.4.7.6 Invalid Override - Changing Generic Argument

```
class Parent {
    void process(Set<Integer> data) {}
}

class Child extends Parent {
    void process(Set<String> data) {}    // ❌ compile error
}
```

Explanation:

- Before erasure: `Set<String>` is not compatible with `Set<Integer>`
- After erasure: both would become `Set`
- Collision + incompatibility → compile error

18.4.7.7 Why This Rule Exists

Java must guarantee:

- **Compile-time type safety**
- **Runtime polymorphism after erasure**

Since generics disappear at runtime, the JVM sees only erased signatures. The compiler must therefore ensure compatibility before erasure, and consistency after erasure.

18.4.7.8 Mental Model

Think of overriding with generics as a two-phase check:

```
Phase 1 → Are the source-level types compatible?
Phase 2 → Do the erased signatures match?
```

If either phase fails → compilation error.

18.4.7.9 Covariant Returns and Generics

An overriding method (i.e., a method declared in a subclass) is allowed to return a **subtype** of the return type declared in the overridden method (i.e., the superclass method).

This is known as the **rule of covariant returns**.

The first step when validating an override is therefore:

- Check whether the return type of the overriding method is a subtype of the return type declared in the superclass.

> **Important**
>
> - If the overridden method returns `List`, the overriding method may return `ArrayList`.
> - It may **not** return `Object`, because `Object` is a supertype, not a subtype.

When generics are involved, return type validation becomes more subtle.

You must evaluate subtype relationships using the generic type hierarchy rules.

Assume that `S` is a subtype of `T`,

There are two important generic hierarchies to remember.

Hierarchy 1 (upper bounded wildcards):

`A<S>` is a subtype of `A<? extends S>` which is a subtype of `A<? extends T>`

- Example:

Since `Integer` is a subtype of `Number`:

- `List<Integer> <<< List<? extends Integer>`
- `List<? extends Integer> <<< List<? extends Number>`

Therefore, if an overridden method returns:

`List<? extends Integer>`

the overriding method may return:

- `List<Integer>`

but may **not** return:

- `List<Number>`
- `List<? extends Number>`

Hierarchy 2 (lower bounded wildcards):

`A<T>` is a subtype of `A<? super T>` which is a subtype of `A<? super S>`

- Example:
- `List<Number> <<< List<? super Number>`
- `List<? super Number> <<< List<? super Integer>`

Therefore, if an overridden method returns:

`List<? super Number>`

the overriding method may return:

- `List<Number>`

but may **not** return:

- `List<Integer>`
- `List<? super Integer>`

A crucial point to remember:

Even though `Integer` is a subtype of `Number`,
`List<Integer>` is **not** a subtype of `List<Number>`.

Generic types in Java are invariant unless wildcards are involved.

These rules explain why certain methods that appear compatible are rejected by the compiler.
Generic specificity must respect formal subtype hierarchies before override validation proceeds to erasure-based signature checks (see 18.4.7.10).

18.4.7.10 Summary Rules

- Override is validated **after erasure**
- Compatibility is validated **before erasure**
- You may erase generic information in the subclass
- You may NOT introduce new generic specificity
- If both methods are parameterized, arguments must match exactly
- After erasure, signatures must be identical
- Covariant return types require the overriding return type to be a true subtype
- With generics, subtype relationships must follow wildcard hierarchy rules
- Apparent logical relationships between type arguments (e.g., `Integer` and `Number`) do not automatically translate into subtype relationships between parameterized types

This explains why some methods that *look* like overloads are rejected: after erasure they collide, and if they are not valid overrides, the compiler blocks them.

18.4.8 Overloading a Generic Method — Why Some Overloads Are Impossible

When Java compiles generic code, it applies type erasure: type parameters such as T are removed, and the compiler substitutes them with their erased type (usually Object or the first bound).

Because of this, two methods that look different at the source level may become identical after erasure.

If the erased signatures are the same, Java cannot distinguish between them, therefore the code does not compile.

- Example: Two Methods That Collapse to the Same Signature

```
public class Demo {
    public void testInput(List<Object> inputParam) {}

    // public void testInput(List<String> inputParam) {}   // ❌ Compile error: after erasure,
}
```

Explanation

```
List<Object> and List<String> are both erased to List.
```

At runtime both methods would appear as:

```
void testInput(List inputParam)
```

Java does not allow two methods with identical signatures in the same class, so the overload is rejected at compile time.

18.4.9 Overloading a Generic Method Inherited from a Parent Class

The same rule applies when a subclass tries to introduce a method that erases to the same signature as one in its superclass.

```
public class SubDemo extends Demo {
    public void testInput(List<Integer> inputParam) {}
    // ❌ Compile error: erases to testInput(List), same as parent
}
```

Again, the compiler rejects the overload because the erased signatures collide.

When Overloading Does Work

Erasure only removes type parameters, not the actual class used in the method parameter.

Therefore, if two method parameters differ in their raw (non-generic) type, the overload is legal, even if one is a generic parameterized type.

```
public class Demo {
    public void testInput(List<Object> inputParam) {}
    public void testInput(ArrayList<String> inputParam) {}  // ✓ Compiles
}
```

Why this works

Even though `ArrayList<String>` erases to `ArrayList`, and `List<Object>` erases to `List`, these are different classes (ArrayList vs. List), so the signatures remain distinct:

```
void testInput(List inputParam)
void testInput(ArrayList inputParam)
```

No collision → legal overloading.

18.4.10 Returning Generic Types — Rules and Restrictions

When returning a value from a method, Java follows a strict rule:

The return type of an overriding method must be a subtype of the parent's return type, and any generic arguments must remain type-compatible (even though they are erased at runtime).

This often confuses developers, because generics on return types cause similar erasure-based conflicts as parameter types.

Key Points:

- Return type covariance applies only to the raw type, not the generic arguments.
- Generic arguments must remain compatible after erasure (they must match).
- Two methods cannot differ only by generic parameter on the return type.
- Example: Illegal Return Type Change Due to Generic Mismatch

```
class A {
    List<String> getData() { return null; }
}

class B extends A {
    // List<Integer> is not a covariant return type of List<String>
    // ❌ Compile error
    List<Integer> getData() { return null; }
}
```

Explanation:

Even though generics are erased, Java still enforces source-level type safety:

`List<Integer>` is not a subtype of `List<String>`.

Both erase to `List`, but Java rejects overriding that breaks type compatibility.

- Example: Legal Covariant Return Type

```
class A {
    Collection<String> getData() { return null; }
}

class B extends A {
    List<String> getData() { return null; }  // ✓ List is a subtype of Collection
}
```

This is allowed because:

- The raw types are covariant (List extends Collection).
- The generic arguments match (String vs. String).
- Example: Illegal Overload on Return Type Alone

Two methods differing only by the generic argument in the return type cannot coexist:

```
class Demo {
    List<String> getList() { return null; }

    // List<Integer> getList() { return null; }
    // ❌ Compile error: return type alone does not distinguish methods
}
```

Java does not use the return type when distinguishing overloaded methods.

18.4.11 Summary of Erasure Rules

- `Unbounded T` → erased to Object.
- `T extends X` → erased to X.
- `T extends X & Y & Z` → erased to X.
- All generic parameters are erased in method signatures.
- Casts are inserted to preserve compile-time typing.
- Bridge methods may be generated to preserve polymorphism.

18.5 Bounds on Type Parameters

This section introduces **bounds on type parameters and wildcards** in Java generics.
Bounds restrict the set of types that can be used with a generic type parameter or wildcard.

They are used to enforce **type constraints** and to express relationships between types in generic code.

Bounds appear in two main forms:

- **Type parameter bounds** using `extends`
- **Wildcard bounds** using `?`, `? extends`, and `? super`

These mechanisms allow generic APIs to specify what kinds of types are acceptable and what operations are type-safe.

Rules

- `T extends Type` → the type parameter must be `Type` or a subclass.
- `T extends Class & Interface1 & Interface2` → multiple bounds are allowed.
- In multiple bounds, the class must appear first.
- `?` represents an unknown type.
- `? extends Type` → accepts types that are `Type` or subclasses.
- `? super Type` → accepts types that are `Type` or superclasses.
- `? extends` allows **reading (extraction)** but forbids insertion.
- `? super` allows **writing (insertion)** but reading returns `Object`.

Summary Table

Syntax	Meaning	Assignment Compatibility	Read	Write
`<T extends Number>`	Type parameter must be `Number` or subclass	Generic declaration bound	`T`	`T`
`<T extends Class & Interface>`	Multiple bounds	Generic declaration bound	`T`	`T`
`List<?>`	Unknown element type	Any `List<T>`	`Object`	❌
`List<? extends Number>`	Unknown subtype of `Number`	`List<Integer>`, `List<Double>`, etc.	`Number`	❌
`List<? super Integer>`	`Integer` or supertype	`List<Integer>`, `List<Number>`, `List<Object>`	`Object`	`Integer`

18.5.1 Upper Bounds: extends

`<T extends Number>` means **T must be Number or a subclass**.

```
class Stats<T extends Number> {
    T num;
    Stats(T num) { this.num = num; }
}
```

18.5.2 Multiple Bounds

Syntax: `T extends Class & Interface1 & Interface2 ...`

The class must come first.

```
class C<T extends Number & Comparable<T>> { }
```

18.5.3 Wildcards: `?`, `? extends`, `? super`

18.5.3.1 Unbounded Wildcard `?`

Use when you want to accept a list of unknown type:

```
void printAll(List<?> list) { ... }
```

18.5.3.2 Upper-Bounded Wildcard `? extends`

```
List<? extends Number> nums = List.of(1, 2, 3);
Number n = nums.get(0);    // OK
// nums.add(5);            // ❌ cannot add: type safety
```

> **You cannot add elements (except null) to ? extends** because you don't know the exact subtype.

18.5.3.3 Lower-Bounded Wildcard `? super`

`<? super Integer>` means **the type must be Integer or a superclass of Integer**.

```
List<? super Integer> list = new ArrayList<Number>();
list.add(10);     // OK
Object o = list.get(0); // returns Object (lowest common supertype)
```

> **Important**
>
> - `Super` accepts **insertion**
> - `extends` accepts **extraction**.

18.6 Generics and Inheritance

> **Generics do NOT participate in inheritance.**
>
> A `List<String>` is not a subtype of `List<Object>`; parameterized types are invariant.

```
List<String> ls = new ArrayList<>();
List<Object> lo = ls;       // ❌ compile error
```

Instead:

```
List<? extends Object> ok = ls;    // works
```

18.7 Type Inference (Diamond Operator)

```
Map<String, List<Integer>> map = new HashMap<>();
```

The compiler infers generic arguments from the assignment.

18.8 Raw Types (Legacy Compatibility)

A **raw type** disables generics, re-introducing unsafe behavior.

```
List raw = new ArrayList();
raw.add("x");
raw.add(10);    // allowed, but unsafe
```

> Raw types should be avoided.

18.9 Generic Arrays (Not Allowed)

You cannot create arrays of parameterized types:

```
List<String>[] arr = new List<String>[10];    // ❌ compile error
```

Because arrays enforce runtime type safety while generics rely on compile-time checks only.

18.10 Bounded Type Inference

```
static <T extends Number> T identity(T x) { return x; }

int v = identity(10);    // OK
// String s = identity("x"); // ❌ not a Number
```

18.11 Wildcards vs. Type Parameters

Use **wildcards** when you need flexibility in parameters.
Use **type parameters** when the method must return or maintain type information.

Example — wildcard too weak:

```
List<?> copy(List<?> list) {
   return list;  // loses type information
}
```

Better:

```
<T> List<T> copy(List<T> list) {
    return list;
}
```

18.12 PECS Rule (Producer Extends, Consumer Super)

Use **? extends** when the parameter **produces** values.
Use **? super** when the parameter **consumes** values.

```
List<? extends Number> listExtends = List.of(1, 2, 3);
List<? super Integer> listSuper = new ArrayList<Number>();

// ? extends → safe read
Number n = listExtends.get(0);

// ? super → safe write
listSuper.add(10);
```

18.13 Common Pitfalls

- Sorting lists with wildcards: `List<? extends Number>` cannot accept insertions.
- Misunderstanding that `List<Object>` is NOT a supertype of `List<String>`.
- Forgetting generic arrays are illegal.
- Thinking generic types are preserved at runtime (they are erased).
- Trying to overload methods using only different type parameters.

18.14 Summary Table of Wildcards

Syntax	Meaning
`?`	unknown type (read-only except Object methods)
`? extends T`	read T safely, cannot add (except null)
`? super T`	can add T, retrieving gives Object

18.15 Summary of Concepts

```
Generics = compile-time type safety
Bounds = restrict legal types
Wildcards = flexibility in parameters
Type Inference = compiler deduces types
Type Erasure = generics disappear at runtime
Bridge Methods = maintain polymorphism
```

18.16 Complete Example

```
class Repository<T extends Number> {
    private final List<T> store = new ArrayList<>();

    void add(T value) { store.add(value); }

    T first() { return store.isEmpty() ? null : store.get(0); }

    // generic method with wildcard
    static double sum(List<? extends Number> list) {
        double total = 0;
        for (Number n : list) total += n.doubleValue();
        return total;
    }
}
```

◀ 17. Beyond Classes | ▲ Index | 19. Exceptions and Error Handling ▶

19. Exceptions and Error Handling

Table of Contents

- 19.1 Exception hierarchy and types
  - 19.1.1 Throwable
  - 19.1.2 Error (unchecked)
  - 19.1.3 Checked Exceptions (`Exception`)
  - 19.1.4 Unchecked Exceptions (`RuntimeException`)
- 19.2 Declaring and throwing exceptions
  - 19.2.1 Declaring exceptions with throws
  - 19.2.2 Throwing exceptions
- 19.3 Overriding methods and exception rules
- 19.4 Handling exceptions: try, catch, finally
  - 19.4.1 Basic try-catch syntax
  - 19.4.2 Multiple catch blocks
  - 19.4.3 Multi-catch Java-7
  - 19.4.4 finally block
- 19.5 Automatic Resource Management try-with-resources
  - 19.5.1 Basic syntax
  - 19.5.2 Declaring multiple resources
  - 19.5.3 Scope of resources
- 19.6 Suppressed exceptions
- 19.7 Exceptions summary

`Exceptions` are Java's structured mechanism for handling abnormal conditions at runtime.

They allow programs to separate normal execution flow from error-handling logic, improving robustness, readability, and correctness.

19.1 Exception hierarchy and types

All exceptions derive from `Throwable`.

The hierarchy defines which conditions are recoverable, which must be declared, and which represent fatal system failures.

```
java.lang.Object
└── java.lang.Throwable
    ├── java.lang.Error
    └── java.lang.Exception
        └── java.lang.RuntimeException
```

19.1.1 Throwable

- Base class for all errors and exceptions
- Supports message, cause, and stack trace
- Only `Throwable` (and subclasses) can be thrown or caught

19.1.2 Error (unchecked)

- Represents serious JVM or system problems
- Not intended to be caught or handled

- Examples: `OutOfMemoryError`, `StackOverflowError`

> **Note**
>
> Errors indicate conditions from which the application is generally not expected to recover.

19.1.3 Checked Exceptions (`Exception`)

- Subclasses of `Exception` **excluding** `RuntimeException`
- Represent conditions that applications may want to handle
- Must be either **caught** or **declared**
- Examples: `IOException`, `SQLException`

19.1.4 Unchecked Exceptions (`RuntimeException`)

- Subclasses of `RuntimeException`
- Not required to be declared or caught
- Usually represent programming errors
- Examples: `NullPointerException`, `IllegalArgumentException`

19.2 Declaring and throwing exceptions

19.2.1 Declaring exceptions with throws

A method declares checked exceptions using the `throws` clause. This is part of the method's API contract.

```
void readFile(Path p) throws IOException {
    Files.readString(p);
}
```

> **Note**
>
> - Only **checked exceptions** must be declared.
> - Unchecked exceptions may be declared, but are usually omitted.

19.2.2 Throwing exceptions

Exceptions are created with `new` and thrown explicitly using `throw`.

```
if (value < 0) {
    throw new IllegalArgumentException("value must be >= 0");
}
```

- `throw` throws exactly one exception instance
- `throws` declares possible exceptions in the method signature

19.3 Overriding methods and exception rules

When overriding a method, exception rules are strictly enforced.

- An overriding method may throw **fewer** or **narrower** checked exceptions
- It may throw any unchecked exceptions
- It may throw **no new or broader** checked exceptions

```java
class Parent {
    void work() throws IOException {}
}

class Child extends Parent {
    @Override
    void work() throws FileNotFoundException {} // OK (subclass)
}
```

> **Note**
>
> Changing only the **unchecked** exceptions never breaks the override contract.

> **Important**
>
> Remember: `constructors` follow a different rule.
>
> A `Constructor` must declare all the checked exceptions declared in the base constructor (or the superclasses of those checked exceptions).
>
> It may also declare additional checked exceptions. This behavior is the opposite of method overriding.
>
> An `overriding method` cannot throw any checked exception other than those declared by the overridden method. It may only throw subclasses of those exceptions.

19.4 Handling exceptions: try, catch, finally

19.4.1 Basic try-catch syntax

```java
try {
    riskyOperation();
} catch (IOException e) {
    handle(e);
}
```

- A `try` block must be followed by at least one `catch` or a `finally`
- Catches are checked top-down

19.4.2 Multiple catch blocks

Multiple catch blocks allow different handling for different exception types.

```java
try {
    process();
} catch (FileNotFoundException e) {
    recover();
} catch (IOException e) {
    log();
}
```

> **Note**
>
> More specific exceptions must come before more general ones, otherwise compilation fails. If you place a catch for a superclass (e.g. `IOException`) before a catch for a subclass (e.g. `FileNotFoundException`), the subclass catch becomes unreachable.

19.4.3 Multi-catch (Java 7+)

```
try {
    process();
} catch (IOException | SQLException e) {
    log(e);
}
```

- Exception types must be unrelated (no parent/child)
- The caught variable is implicitly `final`

19.4.4 finally block

The `finally` block executes regardless of whether an exception is thrown, except in extreme JVM termination cases.

```
try {
    open();
} finally {
    close();
}
```

- Used for cleanup logic
- Executes even if `return` is used in try and/or catch block

> **Note**
>
> A `finally` block can override a return value or swallow an exception. This is generally discouraged because it makes the control flow harder to reason about.

> **Important**
>
> When both a `catch` block and a `finally` block throw exceptions, the exception thrown in the `finally` block is the one that is propagated from the method.
>
> The exception thrown in the `catch` block is lost and is **not** added to the suppressed exceptions list.

```
try {
    throw new RuntimeException("try");
} catch (RuntimeException e) {
    throw new RuntimeException("catch");
} finally {
    throw new RuntimeException("finally");
}
```

In this case, only the `"finally"` exception is thrown.

19.5 Automatic Resource Management (try-with-resources)

Try-with-resources provides automatic closing of resources that implement `AutoCloseable`.

It eliminates the need for explicit `finally` cleanup in most cases.

19.5.1 Basic syntax

```
try (BufferedReader br = Files.newBufferedReader(path)) {
    return br.readLine();
}
```

- Resources are closed automatically
- Closure happens even if an exception is thrown
- Resources are closed before any catch or finally block executes.

```
try (Resource a = new Resource()) {
    a.read();
} finally {
    a.close();  // ❌ Compile-time error: a is out of scope here
}
```

19.5.2 Declaring multiple resources

```
try (InputStream in = Files.newInputStream(p);
        OutputStream out = Files.newOutputStream(q)) {
    in.transferTo(out);
}
```

- Resources are closed in **reverse order** of declaration

19.5.3 Scope of resources

- Resources are in scope only inside the `try` block
- They are implicitly `final`
- Since Java 9, you can declare resources ahead of time, outside the `try-with-resources`, provided they are declared as `final` or are effectively final.

```
final var firstWriter = Files.newBufferedWriter(filePath);

try (firstWriter; var secondWriter = Files.newBufferedWriter(filePath)) {
    // CODE
}
```

> **Note**
>
> Attempting to reassign a resource variable causes a compilation error.

```
Resource a = new Resource();
try(a){ // since Java 9
  ...
}finally{
   a.close(); // this code will compile but the resource referred to by the reference 'a', has
}
```

19.6 Suppressed exceptions

When both the `try` block and the resource's `close()` method throw exceptions, Java preserves the primary exception and **suppresses** the others.

```
try (BadResource r = new BadResource()) {
    throw new RuntimeException("main");
}
```

If `close()` also throws an exception, it becomes **suppressed**.

```
catch (Exception e) {
    for (Throwable t : e.getSuppressed()) {
        System.out.println(t);
    }
}
```

- Primary exception is thrown
- Secondary exceptions are accessible via `getSuppressed()`

Important

Suppressed exceptions are generated only by the **implicit** `finally` block created by `try-with-resources`.

In contrast, exceptions thrown in an **explicit** `finally` block are not suppressed: they replace any previous exception and become the only exception propagated.

19.7 Exceptions summary

- Checked exceptions must be caught or declared
- Overriding methods may not widen checked exceptions
- Use multi-catch for shared handling logic
- Prefer try-with-resources over finally cleanup
- Resources close in reverse order
- Suppressed exceptions preserve full failure context

◀ 18. Generics in Java | ▲ Index | 20. Functional Programming in Java ▶

Module 05

Functional Programming

20. Functional Programming in Java

Table of Contents

- 20.1 Functional Interfaces
  - 20.1.1 Rules for Functional Interfaces
  - 20.1.2 Common Functional Interfaces (java.util.function)
  - 20.1.3 Convenience Methods on Functional Interfaces
  - 20.1.4 Primitive Functional Interfaces
  - 20.1.5 Summary
- 20.2 Lambda Expressions
  - 20.2.1 Syntax of Lambda Expressions
  - 20.2.2 Examples of Lambda Syntax
  - 20.2.3 Rules for Lambda Expressions
  - 20.2.4 Type Inference
  - 20.2.5 Restrictions in Lambda Bodies
  - 20.2.6 Return Type Rules
  - 20.2.7 Lambdas vs Anonymous Classes
  - 20.2.8 Common Lambda Errors
- 20.3 Method References
  - 20.3.1 Reference to a Static Method
  - 20.3.2 Reference to an Instance Method of a Particular Object
  - 20.3.3 Reference to an Instance Method of an Arbitrary Object of a Given Type
  - 20.3.4 Reference to a Constructor
  - 20.3.5 Summary Table of Method Reference Types
  - 20.3.6 Common Pitfalls

`Functional programming` is a programming paradigm that focuses on describing what should be done rather than how it should be done.

Starting from Java 8, the language added several features that enable functional-style programming: `lambda expressions`, `functional interfaces`, and `method references`.

These features allow developers to write more expressive, concise, and reusable code, especially when working with collections, concurrency APIs, and event-driven systems.

20.1 Functional Interfaces

In Java, a **functional interface** is an interface that contains **exactly one** abstract method.

Functional interfaces enable **Lambda Expressions** and **Method References**, forming the core of Java's functional programming model.

> **Note**
>
> Java automatically treats any interface with a single abstract method as a functional interface. The `@FunctionalInterface` annotation is optional but recommended.

20.1.1 Rules for Functional Interfaces

- **Exactly one abstract method** (SAM = Single Abstract Method).
- Interfaces may declare any number of **default**, **static** or **private** methods.

- They may override `Object` methods (`toString()` , `equals(Object)` , `hashCode()`) without affecting SAM count.
- The functional method may come from a **superinterface**.

Example:

```
@FunctionalInterface
interface Adder {
    int add(int a, int b);   // single abstract method
    static void info() {}
    default void log() {}
}
```

20.1.2 Common Functional Interfaces (java.util.function)

Below is a summary of the most important functional interfaces.

Functional Interface	Returns	Method	Parameters
`Supplier<T>`	T	get()	0
`Consumer<T>`	void	accept(T)	1
`BiConsumer<T,U>`	void	accept(T,U)	2
`Function<T,R>`	R	apply(T)	1
`BiFunction<T,U,R>`	R	apply(T,U)	2
`UnaryOperator<T>`	T	apply(T)	1 (same types)
`BinaryOperator<T>`	T	apply(T,T)	2 (same types)
`Predicate<T>`	boolean	test(T)	1
`BiPredicate<T,U>`	boolean	test(T,U)	2

- Examples

```
Supplier<String> sup = () -> "Hello!";

Consumer<String> printer = s -> System.out.println(s);

Function<String, Integer> length = s -> s.length();

UnaryOperator<Integer> square = x -> x * x;

Predicate<Integer> positive = x -> x > 0;
```

20.1.3 Convenience Methods on Functional Interfaces

Many functional interfaces come with helper methods that allow chaining and composition.

Interface	Method	Description
Function	andThen()	applies the function, then the other one specified in this additional method
Function	compose()	applies the other function specified in the additional method, then the function
Function	identity()	returns a function x -> x
Predicate	and()	logical AND
Predicate	or()	logical OR
Predicate	negate()	logical NOT
Consumer	andThen()	chains consumers
BinaryOperator	minBy()	comparator-based minimum
BinaryOperator	maxBy()	comparator-based maximum

- Examples

```
Function<Integer, Integer> times2 = x -> x * 2;
Function<Integer, Integer> plus3  = x -> x + 3;

var result1 = times2.andThen(plus3).apply(5);   // (5*2)+3 = 13
var result2 = times2.compose(plus3).apply(5);   // (5+3)*2 = 16

Predicate<String> longString = s -> s.length() > 5;
Predicate<String> startsWithA = s -> s.startsWith("A");

boolean ok = longString.and(startsWithA).test("Amazing");  // true
```

20.1.4 Primitive Functional Interfaces

Java provides specialized versions of functional interfaces for primitives to avoid boxing/unboxing overhead.

Functional Interface	Return Type	Single Abstract Method	# Parameters
IntSupplier	int	getAsInt()	0
LongSupplier	long	getAsLong()	0
DoubleSupplier	double	getAsDouble()	0
BooleanSupplier	boolean	getAsBoolean()	0
IntConsumer	void	accept(int)	1 (int)
LongConsumer	void	accept(long)	1 (long)
DoubleConsumer	void	accept(double)	1 (double)
IntPredicate	boolean	test(int)	1 (int)
LongPredicate	boolean	test(long)	1 (long)
DoublePredicate	boolean	test(double)	1 (double)
IntUnaryOperator	int	applyAsInt(int)	1 (int)
LongUnaryOperator	long	applyAsLong(long)	1 (long)
DoubleUnaryOperator	double	applyAsDouble(double)	1 (double)
IntBinaryOperator	int	applyAsInt(int, int)	2 (int,int)
LongBinaryOperator	long	applyAsLong(long, long)	2 (long,long)
DoubleBinaryOperator	double	applyAsDouble(double,double)	2
IntFunction	R	apply(int)	1 (int)
LongFunction	R	apply(long)	1 (long)
DoubleFunction	R	apply(double)	1 (double)
ToIntFunction	int	applyAsInt(T)	1 (T)
ToLongFunction	long	applyAsLong(T)	1 (T)
ToDoubleFunction	double	applyAsDouble(T)	1 (T)
ToIntBiFunction<T,U>	int	applyAsInt(T,U)	2 (T,U)
ToLongBiFunction<T,U>	long	applyAsLong(T,U)	2 (T,U)
ToDoubleBiFunction<T,U>	double	applyAsDouble(T,U)	2 (T,U)
ObjIntConsumer	void	accept(T,int)	2 (T,int)
ObjLongConsumer	void	accept(T,long)	2 (T,long)
ObjDoubleConsumer	void	accept(T,double)	2 (T,double)
DoubleToIntFunction	int	applyAsInt(double)	1
DoubleToLongFunction	long	applyAsLong(double)	1
IntToDoubleFunction	double	applyAsDouble(int)	1
IntToLongFunction	long	applyAsLong(int)	1
LongToDoubleFunction	double	applyAsDouble(long)	1
LongToIntFunction	int	applyAsInt(long)	1

- Example

```
IntSupplier dice = () -> (int)(Math.random() * 6) + 1;

IntPredicate even = x -> x % 2 == 0;

IntUnaryOperator doubleIt = x -> x * 2;
```

20.1.5 Summary

- Functional interfaces contain exactly one abstract method (SAM).
- They power Lambdas and Method References.
- Java offers many built-in FIs in java.util.function.
- Primitive variants improve performance by removing boxing.

20.2 Lambda Expressions

A lambda expression is a compact way of writing a function.

Lambda expressions provide a concise way to define implementations of functional interfaces.

A lambda is essentially a short block of code that takes parameters and returns a value, without requiring a full method declaration.

They represent behavior as data and are a key element of Java's functional programming model.

20.2.1 Syntax of Lambda Expressions

The general syntax is:

`(parameters) -> expression` or `(parameters) -> { statements }`

> **Important**
>
> A lambda expression does not introduce a new variable scope. As a result, variable names that already exist in the surrounding context cannot be redeclared as parameters in the lambda expression.

20.2.2 Examples of Lambda Syntax

Zero parameters

```
Runnable r = () -> System.out.println("Hello");
```

One parameter (parentheses optional)

```
Consumer<String> c = s -> System.out.println(s);
```

Multiple parameters

```
BinaryOperator<Integer> add = (a, b) -> a + b;
```

With a block body

```
Function<Integer, String> f = (x) -> {
    int doubled = x * 2;
    return "Value: " + doubled;
};
```

20.2.3 Rules for Lambda Expressions

- Parameter types may be omitted (type inference).
- If a parameter has a type, all parameters must specify the type.
- A single parameter does not require parentheses.

- Multiple parameters require parentheses.
- If the body is a single expression (no `{ }`), `return` is not allowed; the expression itself is the return value.
- If the body uses `{ }` (a block), `return` must appear if a value is returned.
- Lambda expressions can only be assigned to functional interfaces (SAM types).

20.2.4 Type Inference

The compiler infers the lambda's type from the target functional interface context.

```
Predicate<String> p = s -> s.isEmpty();  // s inferred as String
```

If the compiler cannot infer the type, you must specify it explicitly.

```
BiFunction<Integer, Integer, Integer> f = (Integer a, Integer b) -> a * b;
```

20.2.5 Restrictions in Lambda Bodies

Lambdas can only capture local variables that are final or effectively final (not reassigned).

```
int x = 10;
Runnable r = () -> {
    // x++;   // ❌ compile error — x must be effectively final
    System.out.println(x);
};
```

They CAN modify object state (only references must be effectively final).

```
var list = new ArrayList<>();
Runnable r2 = () -> list.add("OK");  // allowed
```

20.2.6 Return Type Rules

If the body is an expression: the expression is the return value.

```
Function<Integer, Integer> f = x -> x * 2;
```

If the body is a block: you must include `return`.

```
Function<Integer, Integer> g = x -> {
    return x * 2;
};
```

20.2.7 Lambdas vs Anonymous Classes

- Lambdas do NOT create a new scope — they share the enclosing scope.
- `this` inside a lambda refers to the enclosing object, not the lambda.

```
class Test {
    void run() {
        Runnable r = () -> System.out.println(this.toString());
    }
}
```

In anonymous classes, `this` refers to the anonymous class instance.

20.2.8 Common Lambda Errors

Inconsistent return types

```
x -> { if (x > 0) return 1; }  // ❌ missing return for negative case
```

Mixing typed and untyped parameters

```
(a, int b) -> a + b   // ❌ illegal
```

Returning a value from a void-target lambda

```
Runnable r = () -> 5;  // ❌ Runnable.run() returns void
```

Ambiguous overload resolution

```
void m(IntFunction<Integer> f) {}
void m(Function<Integer, Integer> f) {}

m(x -> x + 1);  // ❌ ambiguous
```

20.3 Method References

Method references provide a shorthand syntax for using an existing method as a functional interface implementation.

They are equivalent to lambda expressions, but more concise, readable, and often preferred when the target method already exists.

There are four categories of method references in Java:

- Reference to a static method (`ClassName::staticMethod`)
- Reference to an instance method of a particular object (`instance::method`)
- Reference to an instance method of an arbitrary object of a given type (`ClassName::instanceMethod`)
- Reference to a constructor (`ClassName::new`)

20.3.1 Reference to a Static Method

A static method reference replaces a lambda that calls a static method.

```
class Utils {
    static int square(int x) { return x * x; }
}

Function<Integer, Integer> f1 = x -> Utils.square(x);
Function<Integer, Integer> f2 = Utils::square;  // method reference
```

Both `f1` and `f2` behave identically.

20.3.2 Reference to an Instance Method of a Particular Object

Used when you already have an object instance, and want to refer to one of its methods.

```
String prefix = "Hello, ";

UnaryOperator<String> op1 = s -> prefix.concat(s);
UnaryOperator<String> op2 = prefix::concat;   // method reference

System.out.println(op2.apply("World"));
```

The reference `prefix::concat` binds `concat` to **that specific object**.

20.3.3 Reference to an Instance Method of an Arbitrary Object of a Given Type

This is the trickiest form.

The functional interface's first parameter becomes the method's receiver (`this`).

```java
BiPredicate<String, String> p1 = (s1, s2) -> s1.equals(s2);
BiPredicate<String, String> p2 = String::equals;   // method reference

System.out.println(p2.test("abc", "abc"));  // true
```

> **Note**
>
> This form applies the method to the *first argument* of the lambda.

20.3.4 Reference to a Constructor

Constructor references replace lambdas that call `new`.

```java
Supplier<ArrayList<String>> sup1 = () -> new ArrayList<>();
Supplier<ArrayList<String>> sup2 = ArrayList::new; // method reference

Function<Integer, ArrayList<String>> sup3 = ArrayList::new;
// calls the constructor ArrayList(int capacity)
```

20.3.5 Summary Table of Method Reference Types

The table below summarizes all method reference categories.

Type	Syntax Example	Equivalent Lambda
Static method	Class::staticMethod	x -> Class.staticMethod(x)
Instance method of specific object	instance::method	x -> instance.method(x)
Instance method of arbitrary object	Class::method	(obj, x) -> obj.method(x)
Constructor	Class::new	() -> new Class()

20.3.6 Common Pitfalls

- A method reference must match *exactly* the functional interface signature.
- Method overloads can make method references ambiguous.
- Instance-method reference (`Class::method`) shifts the receiver to parameter 1.
- Constructor reference fails if there is no matching constructor.

```java
// ❌ Ambiguous: which println()? (println(int), println(String)...)
Consumer<String> c = System.out::println; // OK only because FI parameter is String

// ❌ No matching constructor: wrong functional interface
Supplier<Integer> s = Integer::new;          // ✓ OK: calls Integer()
Function<String, Long> f = Integer::new;     // ❌ ERROR: constructor returns Integer, not Lo
```

When in doubt, rewrite the method reference as a lambda — if the lambda works but the method reference does not, the problem is usually signature matching.

---

21. Java Optionals and Streams

Table of Contents

- 21.1 Optionals Optional OptionalInt OptionalLong OptionalDouble
  - 21.1.1 Creating Optionals
  - 21.1.2 Reading values safely
  - 21.1.3 Transforming Optionals
  - 21.1.4 Optionals and Streams
  - 21.1.5 Primitive Optionals
  - 21.1.6 Common pitfalls
- 21.2 What Is a Stream And What It Is Not
- 21.3 Stream Pipeline Architecture
  - 21.3.1 Stream Sources
  - 21.3.2 Intermediate Operations
    - 21.3.2.1 Table of Common intermediate operations
  - 21.3.3 Terminal Operations
    - 21.3.3.1 Table of terminal operations
- 21.4 Lazy Evaluation and Short-Circuiting
- 21.5 Stateless vs Stateful Operations
  - 21.5.1 Stateless Operations
  - 21.5.2 Stateful Operations
- 21.6 Stream Ordering and Determinism
- 21.7 Parallel Streams
- 21.8 Reduction Operations
  - 21.8.1 reduce combining a stream into a single object
    - 21.8.1.1 Correct mental model
  - 21.8.2 collect
  - 21.8.3 Why collect is different from reduce
- 21.9 Common Streams Pitfalls
- 21.10 Primitive Streams
  - 21.10.1 Why primitive streams matter
  - 21.10.2 Common creation methods
  - 21.10.3 Primitive-specialized mapping methods
  - 21.10.4 Mapping table among StreamT and primitive streams
  - 21.10.5 Terminal operations and their result types
  - 21.10.6 Common pitfalls and gotchas
- 21.11 Collectors collect Collector and the Collectors Factory Methods
  - 21.11.1 collect vs Collector
  - 21.11.2 Core collectors quick-reference
  - 21.11.3 Grouping collectors
  - 21.11.4 partitioningBy
  - 21.11.5 toMap and merge rules
  - 21.11.6 collectingAndThen
  - 21.11.7 How collectors relate to parallel streams

21.1 Optionals (Optional, OptionalInt, OptionalLong, OptionalDouble)

`Optional<T>` is a container object that may or may not hold a non-null value.

It is designed to make "absence of a value" explicit and to reduce `NullPointerException` risk by forcing callers to handle the empty case.

> **Note**
>
> - `Optional` is intended primarily for **return types**.
> - It is generally discouraged for fields, method parameters, and serialization boundaries (unless a specific API contract requires it).

21.1.1 Creating Optionals

There are three core factory methods.

- `Optional.of(value)` → value must be non-null; otherwise `NullPointerException` is thrown
- `Optional.ofNullable(value)` → returns empty if value is null
- `Optional.empty()` → an explicitly empty Optional

```
Optional<String> a = Optional.of("x");
Optional<String> b = Optional.ofNullable(null); // Optional.empty
Optional<String> c = Optional.empty();
```

21.1.2 Reading values safely

Optionals provide multiple ways to access the wrapped value.

- `isPresent()` / `isEmpty()` → test presence
- `get()` → returns the value or throws `NoSuchElementException` if empty (discouraged)
- `orElse(defaultValue)` → returns value or default (default evaluated immediately)
- `orElseGet(supplier)` → returns value or supplier result (supplier evaluated lazily)
- `orElseThrow()` → returns value or throws `NoSuchElementException`
- `orElseThrow(exceptionSupplier)` → returns value or throws custom exception

```
Optional<String> opt = Optional.of("java");

String v1 = opt.orElse("default");
String v2 = opt.orElseGet(() -> "computed");
String v3 = opt.orElseThrow(); // ok because opt is present
```

> **Note**
>
> - A common trap: `orElse(...)` evaluates its argument even if the Optional is present.
> - Use `orElseGet(...)` when the default is expensive to compute.

21.1.3 Transforming Optionals

Optionals support functional transformations similar to streams, but with "0 or 1 element" semantics.

- `map(fn)` → transforms the value if present
- `flatMap(fn)` → transforms to an Optional without nesting
- `filter(predicate)` → keeps value only if predicate is true

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

Optional<Integer> len =
    name.map(String::length); // Optional[5]

Optional<String> filtered =
    name.filter(n -> n.startsWith("A")); // Optional[Alice]

System.out.println(len.orElse(0));
System.out.println(filtered.orElseGet(() -> "11"));
```

Output:

```
5
Alice
```

Note

- `map` wraps the result in an Optional.
- If your mapping function already returns an Optional, use `flatMap` to avoid `Optional<Optional<T>>` nesting.

21.1.4 Optionals and Streams

A very common pipeline pattern is to map to an Optional and then remove empties.

Since Java 9, `Optional` provides `stream()` to convert "present → one element" and "empty → zero elements".

```java
        Stream<String> words = Stream.of("a", "bb", "ccc");

        words.map(w -> w.length() > 1 ? Optional.of(w.length()) : Optional.<Integer>empty()).f
                .forEach(System.out::println);
```

Output:

```
2
3
```

Note

Before Java 9, this pattern required `filter(Optional::isPresent)` plus `map(Optional::get)`.

21.1.5 Primitive Optionals

Primitive streams use primitive optionals to avoid boxing: `OptionalInt`, `OptionalLong`, `OptionalDouble`.

They mirror the main Optional API with primitive getters such as `getAsInt()`.

- `OptionalInt.getAsInt()` / `OptionalLong.getAsLong()` / `OptionalDouble.getAsDouble()`
- `orElse(...)` / `orElseGet(...)` / `orElseThrow(...)`

```java
OptionalInt m = IntStream.of(3, 1, 2).min(); // OptionalInt[1]
int value = m.orElse(0); // 1
```

21.1.6 Common pitfalls

- Do not call `get()` without checking presence; prefer `orElseThrow` or transformations
- Avoid returning `null` instead of `Optional.empty()`; an Optional reference itself should not be null

- Remember: `average()` on primitive streams always returns `OptionalDouble` (even for `IntStream` and `LongStream`)
- Use `orElseGet` when computing the default is expensive

21.2 What Is a Stream (And What It Is Not)

A `Java Stream` represents a sequence of elements supporting functional-style operations.

Streams are designed for data processing, not data storage.

Key characteristics:

- A stream does not store data
- A stream is lazy — nothing happens until a terminal operation is invoked
- A stream can be consumed only once
- Streams encourage side-effect-free operations

> **Note**
>
> Streams are conceptually similar to database queries: they describe what to compute, not how to iterate.

21.3 Stream Pipeline Architecture

Every stream pipeline consists of three distinct phases:

- 1 **Source**
- 2 Zero or more **Intermediate Operations**
- 3 Exactly one **Terminal Operation**

21.3.1 Stream Sources

Common stream sources include:

- Collections: `collection.stream()`
- Arrays: `Arrays.stream(array)`
- I/O channels and files
- Infinite streams: `Stream.iterate`, `Stream.generate`

```
List<String> names = List.of("Ana", "Bob", "Carla");
Stream<String> s = names.stream();
```

21.3.2 Intermediate Operations

Intermediate operations:

- Return a new stream
- Are lazy
- Do not trigger execution

21.3.2.1 Table of Common intermediate operations:

Method	Common input Params	Return value	Desctiption
`filter`	Predicate	`Stream<T>`	filter the stream according to a predicate match
`map`	Function	`Stream<R>`	transform a stream through a one to one mapping input/output
`flatMap`	Function	`Stream<R>`	flatten nested streams into a single stream
`sorted`	(none) or Comparator	`Stream<T>`	sort by natural order or by the provided Comparator
`distinct`	(none)	`Stream<T>`	remove duplicate elements
`limit` / `skip`	long	`Stream<T>`	limit size or skip elements
`peek`	Consumer	`Stream<T>`	run side-effect action for each element (debugging)

- Example:

```
List<String> names = List.of("Ana", "Bob", "Carla", "Mario");

names.stream().filter(n -> n.length() > 3).map(String::toUpperCase).forEach(System.out
```

Output:

```
CARLA
MARIO
```

> **Note**
>
> Intermediate operations only describe the computation. No element is processed yet.

21.3.3 Terminal Operations

Terminal operations:

- Trigger execution
- Consume the stream
- Produce a result or side effect

21.3.3.1 Table of terminal operations:

Method	Return value	behaviour for infinite streams
`forEach`	**void**	does not terminate
`collect`	varies	does not terminate
`reduce`	varies	does not terminate
`findFirst` / `findAny`	`Optional<T>`	terminates
`anyMatch` / `allMatch` / `noneMatch`	**boolean**	may terminate early (short-circuit)
`min` / `max`	`Optional<T>`	does not terminate
`count`	**long**	does not terminate

21.4 Lazy Evaluation and Short-Circuiting

```
var newNames = new ArrayList<String>();

newNames.add("Bob");
newNames.add("Dan");

// Streams are lazily evaluated: this does not traverse the data yet,
// it only creates a pipeline description bound to the source.
var stream = newNames.stream();

newNames.add("Erin");

// Terminal operation triggers evaluation. The stream sees the updated source,
// so the count includes "Erin".
stream.count(); // 3
```

> **Note**
>
> A stream is bound to its *source* (`newNames`), and the pipeline is not executed until a terminal operation is invoked.
> For this reason, if you **modify the collection before the terminal operation**, the terminal operation "sees" the new elements (here, `Erin`).
> In general, however, **modifying the source while a stream pipeline is in use is bad practice** and can lead to non-deterministic behavior (or `ConcurrentModificationException` with some sources/operations). The practical rule is: *build the source, then create and execute the stream without mutating it.*

Streams process elements **one at a time**, flowing "vertically" through the pipeline rather than stage-by-stage.

Below we modify the example to use a **short-circuiting** terminal operation: `findFirst()` .

```
Stream.of("a", "bb", "ccc")
    .filter(s -> {
        System.out.println("filter " + s);
        return s.length() > 1;
    })
    .map(s -> {
        System.out.println("map " + s);
        return s.toUpperCase();
    })
    .findFirst()
    .ifPresent(System.out::println);
```

Execution order:

> **Note**
>
> Only the minimum number of elements required by the terminal operation are processed.

```
filter a
filter bb
map bb
BB
```

`findFirst()` is satisfied as soon as it finds the **first** element that successfully passes through the pipeline (here `"bb"`), therefore:

- `"ccc"` is never processed (neither `filter` nor `map`);
- lazy evaluation avoids unnecessary work compared to a terminal operation that consumes all elements (such as `forEach` or `count`).

> **Important**
>
> `allMatch`, `noneMatch`, `anyMatch`, `findFirst`, and `findAny` are **short-circuiting terminal operations**.
>
> This means that the given predicate is **not necessarily evaluated for every element of the stream**.
> The operation may stop as soon as the final result can already be determined.
>
> For example, with `allMatch`, if the predicate returns `false` for the **first element**, the overall result of the operation is already known to be `false`. Since `allMatch` requires every element to satisfy the predicate, finding a single element that does not match is sufficient to determine the result.
>
> Therefore, once such an element is encountered, **the remaining elements of the stream do not need to be tested**, and the stream processing stops immediately.

21.5 Stateless vs Stateful Operations

21.5.1 Stateless Operations

Operations like `map` and `filter` process each element independently.

21.5.2 Stateful Operations

Operations like `distinct`, `sorted`, and `limit` require maintaining internal state.

> **Note**
>
> Stateful operations can severely impact parallel stream performance.

21.6 Stream Ordering and Determinism

Streams may be:

- Ordered (e.g., `List.stream()`)
- Unordered (e.g., `HashSet.stream()`)

Some operations respect encounter order:

- `forEachOrdered`
- `findFirst`

> **Note**
>
> In parallel streams, `forEach` does not guarantee order.

21.7 Parallel Streams

Parallel streams divide work across threads using the ForkJoinPool.commonPool().

```
int sum =
IntStream.range(1, 1_000_000)
    .parallel()
        .sum();
```

Rules for safe parallel streams:

- No side effects
- No mutable shared state
- Associative operations only

> **Note**
>
> Parallel streams can be slower for small workloads.

21.8 Reduction Operations

21.8.1 `reduce()` : combining a stream into a single object

There are three method signatures for this operation:

- public `Optional<T>` **reduce(** `BinaryOperator<T> accumulator`);
- public `T` **reduce(** `T identity`, `BinaryOperator<T> accumulator`);
- public `<U> U` **reduce(** `U identity`, `BiFunction<U, ? super T, U> accumulator`, `BinaryOperator<U> combiner`)

```
int sum = Stream.of(1, 2, 3)
    .reduce(0, Integer::sum);
```

Reduction requires:

- **Identity**: Initial value for each partial reduction; must be a neutral element; Example: 0 for sum, 1 for multiplication, empty collection for collecting;
- **Accumulator**: Incorporates one stream element into a partial result;
- (Optional) **Combiner**: Merges two partial results; Used only when the stream is parallel; Ignored for sequential streams

> **Note**
>
> The accumulator must be associative and stateless.

21.8.1.1 Correct mental model

- Accumulator: result + element
- Combiner: result + result

Example 1: Correct use (sum of lengths)

```java
int totalLength =
    Stream.of("a", "bb", "ccc")
          .parallel()
          .reduce(
              0,                           // identity
              (sum, s) -> sum + s.length(), // accumulator
              (left, right) -> left + right // combiner
          );
```

What happens in parallel

Suppose the stream is split:

- Thread 1: “a”, “bb” → 0 + 1 + 2 = 3
- Thread 2: “ccc” → 0 + 3 = 3

Then the combiner merges the partial results:

```
3 + 3 = 6
```

Example 2: Combiner ignored in sequential streams

```java
int result =
    Stream.of("a", "bb", "ccc")
          .reduce(
              0,
              (sum, s) -> sum + s.length(),
              (x, y) -> {
                  throw new RuntimeException("Never called");
              }
          );
```

Example 3: Incorrect combiner

```java
int result =
    Stream.of(1, 2, 3, 4)
          .parallel()
          .reduce(
              0,
              (a, b) -> a - b,    // accumulator
              (x, y) -> x - y     // combiner
          );
```

Why this is wrong

Subtraction is not associative.

Possible execution:

- Thread 1: 0 - 1 - 2 = -3
- Thread 2: 0 - 3 - 4 = -7

Combiner:

```
-3 - (-7) = 4
```

Sequential result would be:

```
(((0 - 1) - 2) - 3) - 4 = -10
```

> **Warning**
>
> ❌ Parallel and sequential results differ → illegal reduction

21.8.2 `collect()`

`collect` is a mutable reduction optimized for grouping and aggregation.

It is the Stream API's standard tool for "mutable reduction": you accumulate elements into a mutable container (like a List, Set, Map, StringBuilder, custom result object), and then optionally merge partial containers when running in parallel.

The general form is:

- `public <R> R **collect**(Supplier<R> supplier, BiConsumer<R, ? super T> accumulator, BiConsumer<R, R> combiner)`;

And a common version used is:

- `public <R, A> R **collect**(Collector<? super T, A, R> collector)`

where Collectors.* provides prebuilt collectors (grouping, mapping, joining, counting, etc.).

Meaning:

- **supplier**: creates a new empty result container (e.g. new `ArrayList<>()`)
- **accumulator**: adds one element into that container (e.g. `list::add`)
- **combiner**: merges two containers (e.g. `list1.addAll(list2)`)

21.8.3 Why `collect()` is different from `reduce()`

- **Intent**: mutation vs immutability
  - `reduce()` is designed for immutable-style reduction: combine values into a new value (e.g. sum, min, max).
  - `collect()` is designed for mutable containers: build up a List, Map, StringBuilder, etc.
- **Correctness** in parallel
  - `reduce()` requires the operation to be:
    - associative
    - stateless
    - compatible with identity/combiner rules
  - `collect()` is built to support parallelism safely by:
    - creating one container per thread (supplier)
    - accumulating locally (accumulator)
    - merging at the end (combiner)
- **Performance**
  - `collect()` can be optimized because the stream runtime knows you are building containers:
    - it can avoid unnecessary copying
    - it can pre-size or use specialized implementations (depending on collector)
    - it's the idiomatic and expected approach
    - using reduce() to build a collection often creates extra objects or forces unsafe mutation.
- Example: "collect into a List" the right way

```
List<String> longNames =
    names.stream()
        .filter(s -> s.length() > 3)
          .collect(Collectors.toList());
```

- Example: groupingBy with explanation

```
Map<Integer, List<String>> byLength =
    names.stream()
        .collect(Collectors.groupingBy(String::length));
```

What happens conceptually:

- The collector creates an empty `Map<Integer, List<String>>`
- For each name:
  - compute the key (String::length)
  - put it in the correct bucket list
- In parallel:
  - each thread builds its own partial maps
  - the combiner merges maps by merging lists per key

21.9 Common Streams Pitfalls

- Reusing a consumed stream → `IllegalStateException`
- Modifying external variables inside lambdas
- Assuming execution order in parallel streams
- Using `peek` for logic instead of debugging

21.10 Primitive Streams

Java provides three specialized stream types to avoid boxing overhead and to enable numeric-focused operations:

- `IntStream` for `int`
- `LongStream` for `long`
- `DoubleStream` for `double`

Primitive streams are still streams (lazy pipelines, intermediate + terminal operations, single-use), but they are **not generic** and they use primitive-specialized functional interfaces (e.g., `IntPredicate`, `LongUnaryOperator`, `DoubleConsumer`).

> **Note**
>
> Use primitive streams when the data is naturally numeric or when performance matters: they avoid boxing/unboxing overhead and provide additional numeric terminal operations.

21.10.1 Why primitive streams matter

- Performance: avoid allocating wrapper objects and repeated boxing/unboxing in large pipelines
- Convenience: built-in numeric reductions such as `sum()`, `average()`, `summaryStatistics()`
- Common traps: understanding when results are primitives vs `OptionalInt` / `OptionalLong` / `OptionalDouble`

21.10.2 Common creation methods

The following are the most frequently used ways to create primitive streams. Many certification questions start by identifying the stream type created by a factory method.

Sources
IntStream.of(int...)
IntStream.range(int startInclusive, int endExclusive)
IntStream.rangeClosed(int startInclusive, int endInclusive)
IntStream.iterate(int seed, IntUnaryOperator f) // infinite unless limited
IntStream.iterate(int seed, IntPredicate hasNext, IntUnaryOperator f)
IntStream.generate(IntSupplier s) // infinite unless limited
LongStream.of(long...)
LongStream.range(long startInclusive, long endExclusive)
LongStream.rangeClosed(long startInclusive, long endInclusive)
LongStream.iterate(long seed, LongUnaryOperator f)
LongStream.iterate(long seed, LongPredicate hasNext, LongUnaryOperator f)
LongStream.generate(LongSupplier s)
DoubleStream.of(double...)
DoubleStream.iterate(double seed, DoubleUnaryOperator f)
DoubleStream.iterate(double seed, DoublePredicate hasNext, DoubleUnaryOperator f)
DoubleStream.generate(DoubleSupplier s)

> **Important**
>
> - Only `IntStream` and `LongStream` provide `range()` and `rangeClosed()`.
> - There is no `DoubleStream.range` because counting with doubles has rounding issues.

21.10.3 Primitive-specialized mapping methods

Primitive streams provide **primitive-only** mapping operations that avoid boxing:

- `IntStream.map(IntUnaryOperator)` → `IntStream`
- `IntStream.mapToLong(IntToLongFunction)` → `LongStream`
- `IntStream.mapToDouble(IntToDoubleFunction)` → `DoubleStream`
- `LongStream.map(LongUnaryOperator)` → `LongStream`
- `LongStream.mapToInt(LongToIntFunction)` → `IntStream`
- `LongStream.mapToDouble(LongToDoubleFunction)` → `DoubleStream`
- `DoubleStream.map(DoubleUnaryOperator)` → `DoubleStream`
- `DoubleStream.mapToInt(DoubleToIntFunction)` → `IntStream`
- `DoubleStream.mapToLong(DoubleToLongFunction)` → `LongStream`

21.10.4 Mapping table among `Stream<T>` and primitive streams

This table summarizes the main conversions among object streams and primitive streams.

The “From” column tells you which methods are available and the resulting target stream type.

From (source)	To (target)	Primary method(s)
`Stream<T>`	`Stream<R>`	`map(Function<? super T, ? extends R>)`
`Stream<T>`	`Stream<R> (flatten)`	`flatMap(Function<? super T, ? extends Stream<? extends R>>)`
`Stream<T>`	`IntStream`	`mapToInt(ToIntFunction<? super T>)`
`Stream<T>`	`LongStream`	`mapToLong(ToLongFunction<? super T>)`
`Stream<T>`	`DoubleStream`	`mapToDouble(ToDoubleFunction<? super T>)`
`Stream<T>`	`IntStream (flatten)`	`flatMapToInt(Function<? super T, ? extends IntStream>)`
`Stream<T>`	`LongStream (flatten)`	`flatMapToLong(Function<? super T, ? extends LongStream>)`
`Stream<T>`	`DoubleStream (flatten)`	`flatMapToDouble(Function<? super T, ? extends DoubleStream>)`
`IntStream`	`Stream<Integer>`	`boxed()`
`LongStream`	`Stream<Long>`	`boxed()`
`DoubleStream`	`Stream<Double>`	`boxed()`
`IntStream`	`Stream<U>`	`mapToObj(IntFunction<? extends U>)`
`LongStream`	`Stream<U>`	`mapToObj(LongFunction<? extends U>)`
`DoubleStream`	`Stream<U>`	`mapToObj(DoubleFunction<? extends U>)`
`IntStream`	`LongStream`	`asLongStream()`
`IntStream`	`DoubleStream`	`asDoubleStream()`
`LongStream`	`DoubleStream`	`asDoubleStream()`

Important

- There is no **`unboxed()`** operation.
- To go from wrappers to primitives you must start from `Stream<T>` and use `mapToInt` / `mapToLong` / `mapToDouble`.

21.10.5 Terminal operations and their result types

Primitive streams have several terminal operations that are either unique or have primitive-specific return types.

Terminal operation	IntStream returns	LongStream returns	DoubleStream returns
`count()`	long	long	long
`sum()`	int	long	double
`min()` / max()	OptionalInt	OptionalLong	OptionalDouble
`average()`	OptionalDouble	OptionalDouble	OptionalDouble
`findFirst()` / findAny()	OptionalInt	OptionalLong	OptionalDouble
`reduce(op)`	OptionalInt	OptionalLong	OptionalDouble
`reduce(identity, op)`	int	long	double
`summaryStatistics()`	IntSummaryStatistics	LongSummaryStatistics	DoubleSummaryStatistics

> **Warning**
>
> - Even for `IntStream` and `LongStream`, **`average()`** returns `OptionalDouble` (not `OptionalInt` or `OptionalLong`).

- Example 1: `Stream<String>` → `IntStream` → primitive terminal operations.

```
List<String> words = List.of("a", "bb", "ccc");

int totalLength = words.stream()
    .mapToInt(String::length) // IntStream
        .sum(); // int

// totalLength = 1 + 2 + 3 = 6
```

- Example 2: `IntStream` → boxed `Stream<Integer>` (boxing introduced).

```
Stream<Integer> boxed = IntStream.rangeClosed(1, 3) // 1,2,3
    .boxed(); // Stream<Integer>
```

- Example 3: primitive stream → object stream via `mapToObj`.

```
Stream<String> labels = IntStream.range(1, 4) // 1,2,3
    .mapToObj(i -> "N=" + i); // Stream<String>
```

21.10.6 Common pitfalls and gotchas

- Do not confuse `Stream<Integer>` with `IntStream`: their mapping methods and functional interfaces differ
- `IntStream.sum()` returns `int` but `IntStream.count()` returns `long`
- `average()` always returns `OptionalDouble` for all primitive stream types
- Using `boxed()` reintroduces boxing; only do it if the downstream API requires objects (e.g., collecting to `List<Integer>`)
- Be careful with narrowing conversions: `LongStream.mapToInt` and `DoubleStream.mapToInt` may truncate values

---

21.11 Collectors (collect(), Collector, and the Collectors Factory Methods)

A `Collector` describes how to accumulate stream elements into a final result.

The `collect(...)` terminal operation executes this recipe.

The `Collectors` utility class provides ready-made collectors for common aggregation tasks.

21.11.1 collect() vs Collector

There are two main ways to collect:

- `collect(Collector)` → the common form using `Collectors.*`
- `collect(supplier, accumulator, combiner)` → explicit mutable reduction (lower-level)

```
List<String> list =
Stream.of("a", "b")
    .collect(Collectors.toList());

StringBuilder sb =
Stream.of("a", "b")
    .collect(StringBuilder::new, StringBuilder::append, StringBuilder::append);
```

> **Note**
>
> Use `collect(supplier, accumulator, combiner)` when you need a custom mutable container and do not want to implement a full `Collector`.

21.11.2 Core collectors (quick reference)

These are the most frequently used collectors and the ones most likely to appear in exam questions.

- `toList()` → `List<T>` (no guarantees about mutability/implementation)
- `toSet()` → `Set<T>`
- `toCollection(supplier)` → specific collection type (e.g., `TreeSet`)
- `joining(delim, prefix, suffix)` → `String` from `CharSequence` elements
- `counting()` → `Long` count
- `summingInt` / `summingLong` / `summingDouble` → numeric sums
- `averagingInt` / `averagingLong` / `averagingDouble` → numeric averages
- `minBy(comparator)` / `maxBy(comparator)` → `Optional<T>`
- `mapping(mapper, downstream)` → transform then collect with downstream
- `filtering(predicate, downstream)` → filter inside collector (Java 9+)

21.11.3 Grouping collectors

`groupingBy` classifies elements into buckets keyed by a classifier function.

It produces a `Map<K, V>` where `V` depends on the downstream collector.

```
Map<Integer, List<String>> byLen =
Stream.of("a", "bb", "ccc", "dd")
    .collect(Collectors.groupingBy(String::length));
System.out.println("byLen: " + byLen.toString());
```

Output:

```
byLen: {1=[a], 2=[bb, dd], 3=[ccc]}
```

With a downstream collector you control what each bucket contains:

```
Map<Integer, Long> countByLen =
Stream.of("a", "bb", "ccc", "dd")
    .collect(Collectors.groupingBy(String::length, Collectors.counting()));
System.out.println("countByLen: " + countByLen.toString());

Map<Integer, Set<String>> setByLen =
Stream.of("a", "bb", "ccc", "dd")
    .collect(Collectors.groupingBy(String::length, Collectors.toSet()));
System.out.println("setByLen: " + setByLen.toString());
```

Output:

```
countByLen: {1=1, 2=2, 3=1}
setByLen: {1=[a], 2=[bb, dd], 3=[ccc]}
```

Warning

Pay attention to the resulting map value type. Example: `groupingBy(..., counting())` yields `Map<K, Long>` (not `int`).

21.11.4 partitioningBy

`partitioningBy` splits the stream into exactly two groups using a boolean predicate. It always returns a map with keys `true` and `false`.

```
Map<Boolean, List<String>> parts =
Stream.of("a", "bb", "ccc")
.collect(Collectors.partitioningBy(s -> s.length() > 1));
System.out.println("parts: " + parts.toString());
```

Output:

```
parts: {false=[a], true=[bb, ccc]}
```

Note

`partitioningBy` always creates two buckets, while `groupingBy` can create many. Both support downstream collectors.

21.11.5 toMap and merge rules

`toMap` throws an exception on duplicate keys unless you provide a merge function.

```
Map<Integer, String> m1 =
Stream.of("aa", "bb")
    .collect(Collectors.toMap(String::length, s -> s)); // ❌ Exception in thread "main" java.

Map<Integer, String> m2 =
Stream.of("aa", "bb", "cc")
    .collect(Collectors.toMap(String::length, s -> s, (oldV, newV) -> oldV + "," + newV)); //
```

Output:

```
m2: {2=aa,bb,cc}
```

21.11.6 collectingAndThen

`collectingAndThen(downstream, finisher)` lets you apply a final transformation after collecting (e.g., make the list unmodifiable).

```
List<String> unmodifiable =
Stream.of("a", "b", "c")
    .collect(Collectors.collectingAndThen(Collectors.toList(), List::copyOf));
```

21.11.7 How collectors relate to parallel streams

Collectors are designed to work with parallel streams by using supplier/accumulator/combiner internally. In parallel, each worker builds a partial result container and then merges containers.

- The accumulator mutates a per-thread container (no shared mutable state)
- The combiner merges containers (required for parallel execution)
- Some collectors are “concurrent” or have characteristics that affect performance and ordering

> **Note**
>
> prefer `collect(Collectors.toList())` over using `reduce` to build collections. `reduce` is for immutable-style reductions; `collect` is for mutable containers.

◀ 20. Functional Programming in Java | ▲ Index | 22. Introduction to the Collections Framework ▶

Module 06

Collections Framework

22. Introduction to the Collections Framework

Table of Contents

- 22.1 What Is the Collections Framework
- 22.2 The Core Interfaces
  - 22.2.1 Main Collection Interfaces
  - 22.2.2 Map Hierarchy
- 22.3 Sequenced Collections Java-21
- 22.4 Why the Collections Framework Exists
- 22.5 The Two Sides of the Framework Collections-vs-Maps
- 22.6 Generic Types in the Collections Framework
- 22.7 Mutability vs Immutability
- 22.8 Big-O Performance Expectations
- 22.9 Summary

The `Java Collections Framework (JCF)` is a set of **interfaces, classes, and algorithms** designed to store, manipulate, and process groups of data efficiently.

It provides a unified architecture for handling collections, allowing developers to write reusable, interoperable code with predictable behaviors and performance characteristics.

This chapter introduces the foundational concepts needed before studying Lists, Sets, Queues, Maps, and Sequenced Collections, explored in detail in subsequent chapters.

22.1 What Is the Collections Framework?

The Collections Framework provides:

- A **set of interfaces** (Collection, List, Set, Queue, Deque, Map...)
- A **set of implementations** (ArrayList, HashSet, TreeSet, LinkedList...)
- A **set of utility algorithms** (sorting, searching, copying, reversing...) in java.util.Collections and java.util.Arrays.
- A common language for performance expectations (Big-O complexity).

All major collection structures share a consistent design so that code working with one implementation can often be reused with another.

22.2 The Core Interfaces

At the heart of the Java Collections Framework is a small set of **root interfaces** that define generic data-handling behaviors.

- **List**: an `ordered` collection of elements that allows `duplicates`;
- **Set**: a collection that does not allow `duplicates`;
- **Queue**: a collection designed for holding elements prior to processing, typically FIFO (first-in-first-out), with variants like priority queues and deques.
- **Map**: a structure that maps keys to values, where duplicate keys are not allowed; each key can map to at most one value.

22.2.1 Main Collection Interfaces

Below is the conceptual hierarchy.

```
java.util
├─ Collection<E>
│ ├─ SequencedCollection<E> (Java 21+)
│ │ ├─ List<E>
│ │ │   ├─ ArrayList<E>
│ │ │   └─ LinkedList<E> (also implements Deque<E>)
│ │ └─ Deque<E> (also extends Queue<E>)
│ │     ├─ ArrayDeque<E>
│ │     └─ LinkedList<E>
│ ├─ Set<E>
│ │     ├─ SequencedSet<E> (Java 21+)
│ │     │       └─ LinkedHashSet<E>
│ │     ├─ SortedSet<E>
│ │     │       └─ NavigableSet<E>
│ │     │           └─ TreeSet<E>
│ │     ├─ HashSet<E>
│ │     └─ (other Set implementations)
│ ├─ Queue<E>
│ │     ├─ Deque<E> (already under SequencedCollection<E>)
│ │     ├─ PriorityQueue<E>
│ │     └─ (other Queue implementations)
│ └─ (other Collection implementations)
│
└─ Map<K,V> (not a Collection)
    ├─ SequencedMap<K,V> (Java 21+)
    │   └─ LinkedHashMap<K,V>
    ├─ SortedMap<K,V>
    │   └─ NavigableMap<K,V>
    │   └─ TreeMap<K,V>
    ├─ HashMap<K,V>
    ├─ Hashtable<K,V>
    └─ (other Map/ConcurrentMap implementations)
```

The **Map** interface does not extend Collection because a map stores key/value pairs rather than single values.

22.2.2 Map Hierarchy

```
java.util
└─ Map<K,V>
    ├─ SequencedMap<K,V> (Java 21+)
    │   └─ LinkedHashMap<K,V>
    ├─ SortedMap<K,V>
    │   └─ NavigableMap<K,V>
    │       └─ TreeMap<K,V>
    ├─ HashMap<K,V>
    ├─ Hashtable<K,V>
    └─ ConcurrentMap<K,V> (java.util.concurrent)
        └─ ConcurrentHashMap<K,V>
```

22.3 Sequenced Collections (Java 21+)

Java 21 introduces the new interface `SequencedCollection`, which formalizes the idea that a collection maintains a **defined encounter order**. This was already true for List, LinkedHashSet, LinkedHashMap, Deque, etc., but now the behavior is standardized.

- `SequencedCollection` defines methods like `getFirst()`, `getLast()`, `addFirst()`, `addLast()`, `removeFirst()`, `removeLast()`, and `reversed()`.
- SequencedSet, SequencedMap extend the idea for sets and maps.

This drastically simplifies the specification of ordering behaviors and will be used throughout the following chapters.

22.4 Why the Collections Framework Exists

- Avoid reinventing data structures
- Provide well-tested, high-performance algorithms

- Improve interoperability through shared interfaces
- Support generic types for type-safe collections

Before Java 1.2, data structures were ad-hoc, inconsistent, and untyped.

The Collections Framework unified all of this into a consistent API.

22.5 The Two Sides of the Framework: Collections vs. Maps

"Does Map extend Collection?" **No.** A Map stores **pairs**, while a Collection stores **single elements**.

- Collection = List, Set, Queue, Deque, SequencedCollection
- Map = Dictionary-like key/value store

22.6 Generic Types in the Collections Framework

Collections are almost always used with generics. Using raw types is discouraged.

```
List<String> names = new ArrayList<>();
Map<Integer, String> map = new HashMap<>();
```

> **Note**
>
> Generics in collections work through `type erasure`: Please check the Paragraph "**18.4 Type Erasure**" in Chapter: 18. Generics in Java.

22.7 Mutability vs. Immutability

Many methods in the Collections API return **unmodifiable** collections:

```
List<String> immutable = List.of("a", "b");
immutable.add("c"); // ❌ UnsupportedOperationException
```

Java provides several ways to create immutable collections:

- `List.of()`, `Set.of()`, `Map.of()`
- `List.copyOf(collection)`
- `Collections.unmodifiableList(...)` wrappers
- `Records` used as immutable value containers

> **Note**
>
> The method `Arrays.asList(varargs)`, which is backed by an array, behaves differently: see examples below.

```java
String[] vargs = new String[] {"u", "v", "z"};
List<String> fromAsList = Arrays.asList(vargs);

List<String> immutable1 = List.of(vargs);
immutable1.add("c"); // ❌ UnsupportedOperationException

List<String> immutable2 = List.copyOf(fromAsList);
immutable2.set(0, "k"); // ❌ UnsupportedOperationException


// We can't ADD or REMOVE elements from "fromAsList" but we can replace them,
// either by modifying the underlying array "vargs" or by mutating the list itself:


fromAsList.set(0, "k");  // the update will be reflected on the backing array as well.
```

> **Note**
>
> `Arrays.asList(...)` returns a fixed-size, but **mutable**, List view backed by the original array. You cannot add/remove elements, but you can replace existing ones.

22.8 Big-O Performance Expectations

Understanding complexity is essential. Here are common examples:

Type	Methods	Complexity
ArrayList	`get()`, `add()`, `remove()`	**`O(1)`, `amortized O(1)`, `O(n)`**
LinkedList	`get()`, `add/remove first/last`	**`O(n)`, `O(1)`**
HashSet	`add()`, `contains()`, `remove()`	~ **`O(1)`**
TreeSet	`add()`, `contains()`, `remove()`	**`O(log n)`**
HashMap	`get()/put()`	~ **`O(1) on average`**
TreeMap	`get()/put()`	**`O(log n)`**
Deque	`add/remove first/last`	**`O(1)`**

> **Note**
>
> These values are averages; worst-case may be different (especially for hash-based structures).

22.9 Summary

- The Collection Framework is built on a small set of core interfaces.
- Java 21 adds Sequenced Collections to unify ordering behavior.
- Maps are not Collections — they form a parallel hierarchy.
- Collections rely heavily on generics.
- Mutability matters — factory methods often return immutable collections.
- Performance characteristics are predictable.

---

23. Shared Collection Operations & Equality

Table of Contents

- 23.1 Core Collection Methods Available to Most Collections
  - 23.1.1 Mutating Operations
  - 23.1.2 Query Operations
- 23.2 Equality
- 23.3 Fail-Fast Behavior
- 23.4 Bulk Operations
- 23.5 Common Return Types and Exceptions
- 23.6 Summary Table — Shared Operations

This chapter covers the fundamental operations shared across the Java Collections API, including how equality is determined inside collections.

These concepts apply to all main collection families based on Collection (List, Set, Queue, Deque and their Sequenced variants).

Map shares several conceptual behaviors (iteration, equality) but does not inherit Collection.

Mastering these operations is essential, as they explain how collections behave when adding, searching, removing, comparing, iterating, and sorting elements.

23.1 Core Collection Methods (Available to Most Collections)

The following methods come from the `Collection<E>` interface and are inherited by **all** major collections except `Map` (which has its own family of operations).

> **Note**
>
> `Map` does not implement `Collection`, but its `keySet()`, `values()`, and `entrySet()` views **do**, and therefore expose these shared operations.

23.1.1 Mutating Operations

- `boolean add(E e)` — Adds an element (allowed to add duplicates in lists).
- `boolean remove(Object o)` — Removes the first matching element.
- `void clear()` — Removes all elements.
- `boolean addAll(Collection<? extends E> c)` — Bulk insertion.
- `boolean removeAll(Collection<?> c)` — Removes all elements contained in the given collection.
- `boolean retainAll(Collection<?> c)` — Keeps only matching elements.

23.1.2 Query Operations

- `int size()` — Number of elements.
- `boolean isEmpty()` — Whether collection contains zero elements.
- `boolean contains(Object o)` — Relies on element equality rules.
- `Iterator<E> iterator()` — Returns an iterator (fail-fast).
- `Object[] toArray()` and `<T> T[] toArray(T[] a)` — Copy into an array.

23.2 Equality

A custom implementation of the method `equals()` allows us to compare the type and content of two collections.

The implementation will differ depending if we are dealing with Lists or Sets.

- Example

```
List<Integer> firstList = List.of(10, 11, 22);
List<Integer> secondList = List.of(10, 11, 22);
List<Integer> thirdList = List.of(22, 11, 10);

System.out.println("firstList.equals(secondList): " + firstList.equals(secondList));
System.out.println("secondList.equals(thirdList): " + secondList.equals(thirdList));

Set<Integer> firstSet = Set.of(10, 11, 22);
Set<Integer> secondSet = Set.of(10, 11, 22);
Set<Integer> thirdSet = Set.of(22, 11, 10);

System.out.println("firstSet.equals(secondSet): " + firstSet.equals(secondSet));
System.out.println("secondSet.equals(thirdSet): " + secondSet.equals(thirdSet));
```

Output

```
firstList.equals(secondList): true
secondList.equals(thirdList): false
firstSet.equals(secondSet): true
secondSet.equals(thirdSet): true
```

> **Note**
>
> - Lists compare size, order, and element equality one-by-one.
> - Sets compare size and membership only — encounter order is irrelevant.
> - Two sets with the same logical elements are equal even if they maintain different iteration order internally.

23.3 Fail-Fast Behavior

Most collection iterators (except concurrent collections) are `fail-fast`: modifying a collection structurally while iterating triggers a `ConcurrentModificationException`.

```
List<Integer> list = new ArrayList<>(List.of(1,2,3));
for (Integer i : list) {
    list.add(99); // ❌ ConcurrentModificationException
}
```

> **Note**
>
> Use `Iterator.remove()` when you must remove elements during iteration. Fail-fast behavior is **not guaranteed** — the exception is thrown on a best-effort basis. You must not rely on catching it for program correctness.

23.4 Bulk Operations

- `removeIf(Predicate<? super E> filter)` — Removes all matching items.
- `replaceAll(UnaryOperator<E> op)` — Replaces every element.
- `forEach(Consumer<? super E> action)` — Applies action to each element.
- `stream()` — Returns a stream for pipeline operations.

23.5 Common Return Types and Exceptions

- `add(E)` returns **boolean** — always `true` for `ArrayList`, may be `false` for `Set` if no change occurs.
- `remove(Object)` returns boolean (not the removed element!).
- `get(int)` throws `IndexOutOfBoundsException`.
- `iterator().remove()` throws `IllegalStateException` if called twice without next().
- `toArray()` always returns a `Object[]` — not `T[]`.

23.6 Summary Table — Shared Operations

Operation	Applies To	Notes
`add(e)`	All collections except Map	Lists allow duplicates
`remove(o)`	All collections except Map	Removes first occurrence
`contains(o)`	All collections except Map	Uses equals()
`size(), isEmpty()`	All collections	Constant-time for most
`iterator()`	All collections	Fail-fast
`clear()`	All collections	Removes all elements
`stream()`	All collections	Returns sequential stream
`removeIf(), replaceAll()`	Lists only (most Sets do not support replaceAll)	Bulk operations
`toArray()`	All collections	Returns Object[]

◀ 22. Introduction to the Collections Framework | ▲ Index | 24. Comparable, Comparator & Sorting in Java ▶

24. Comparable, Comparator & Sorting in Java

Table of Contents

- 24.1 Comparable — Natural Ordering
  - 24.1.1 Comparable Method Contract
  - 24.1.2 Example Class Implementing Comparable
  - 24.1.3 Common Comparable Pitfalls
- 24.2 Comparator — Custom Ordering
  - 24.2.1 Comparator Core Methods
    - 24.2.1.1 Comparator Helper Static Methods
    - 24.2.1.2 Instance Methods on Comparator
  - 24.2.2 Comparator Example
- 24.3 Comparable vs Comparator
- 24.4 Sorting Arrays and Collections
  - 24.4.1 Arrays sort
  - 24.4.2 Collections sort
- 24.5 Multi-Level Sorting thenComparing
- 24.6 Comparing Primitives Efficiently
- 24.7 Common Traps
- 24.8 Full Example
- 24.9 Summary

Java provides two main strategies for sorting and comparing: `Comparable` (natural ordering) and `Comparator` (custom ordering).

Understanding their rules, constraints, and interactions with `generics` is essential.

- For **numeric types**, sorting follows natural numerical order, meaning smaller values come before larger ones.
- Sorting **strings** follows lexicographical (Unicode code point) order: character-by-character comparison; digits come before uppercase, uppercase before lowercase.

This ordering is based on each character's Unicode code point, not alphabetical intuition.

A **Unicode code point** is a unique numerical value assigned to a character in the Unicode standard.

More precisely: a `Unicode code point` is an integer (written in hexadecimal as U+XXXX) that represents a specific character, symbol, or control mark—independent of font, language, or platform.

- Examples:
  - U+0041 → A
  - U+0061 → a
  - U+0030 → 0
  - U+1F600 → 😀

A code point is not a byte sequence. It's an abstract number.

How a code point is stored in memory depends on the encoding (UTF-8, UTF-16, UTF-32).

Unicode defines code points from U+0000 to U+10FFFF.

In short: Unicode code points define what the character is; encodings define how it is represented in bytes.

- Example natural ordering

```
List<String> items = List.of("10", "2", "A", "Z", "a", "b");

List<String> sorted = new ArrayList<>(items);
Collections.sort(sorted);

System.out.println(sorted);
```

Output:

```
[10, 2, A, Z, a, b]
```

> **Note**
>
> Natural ordering is only defined for types that implement Comparable.

24.1 Comparable — Natural Ordering

The interface `Comparable<T>` defines the natural order of a type.

A class implements it when it wants to define its default sorting rule.

24.1.1 Comparable Method Contract

```
public interface Comparable<T> {
    int compareTo(T other);
}
```

Rules and expectations:

- Return **negative** → `this` < `other`
- Return **zero** → `this` == `other`
- Return **positive** → `this` > `other`

> **Important**
>
> - Natural ordering must be consistent with `equals()`, unless explicitly documented otherwise:
> - `compareTo()` is consistent with `equals()` if, and only if, `a.compareTo(b) == 0` and `a.equals(b) is true`.

> **Warning**
>
> compareTo may throw ClassCastException if given a non-comparable type — but this usually appears only with raw types.

24.1.2 Example: Class Implementing Comparable

```
public class Person implements Comparable<Person> {

    private String name;
    private int age;

    public Person(String n, int a) {
        this.name = n;
        this.age = a;
    }

    @Override
    public int compareTo(Person other) {
        return Integer.compare(this.age, other.age);
    }



}

var list = List.of(new Person("Bob", 40), new Person("Alice", 30));

list.stream().sorted().forEach(p -> System.out.println(p.getAge()));
```

The list sorts by age, because that is the natural numbering order.

24.1.3 Common Comparable Pitfalls

- Compare all relevant fields → inconsistent results if not
- Violating transitivity → leads to undefined behavior
- Throwing exceptions inside compareTo() breaks sorting
- Failing to implement the same logic as equals() → common trap

---

24.2 Comparator — Custom Ordering

The interface `Comparator<T>` allows defining multiple sorting strategies without modifying the class itself.

24.2.1 Comparator Core Methods

```
int compare(T a, T b);
```

Additional helper methods:

24.2.1.1 Comparator Helper Static Methods

Method	Static / Instance	Return Type	Parameters	Description
`Comparator.comparing(keyExtractor)`	static	Comparator	Function<? super T, ? extends U>	Builds a comparator comparing extracted keys using natural ordering.
`Comparator.comparing(keyExtractor, keyComparator)`	static	Comparator	Function<T,U>, Comparator	Builds comparator comparing extracted keys using a custom comparator.
`Comparator.comparingInt(keyExtractor)`	static	Comparator	ToIntFunction	Optimized comparator for int keys (avoids boxing).
`Comparator.comparingLong(keyExtractor)`	static	Comparator	ToLongFunction	Optimized comparator for long keys.
`Comparator.comparingDouble(keyExtractor)`	static	Comparator	ToDoubleFunction	Optimized comparator for double keys.
`Comparator.naturalOrder()`	static	Comparator	none	Comparator using natural ordering (Comparable).
`Comparator.reverseOrder()`	static	Comparator	none	Reverse natural ordering.
`Comparator.nullsFirst(comparator)`	static	Comparator	Comparator	Wraps comparator so nulls compare before non-nulls.
`Comparator.nullsLast(comparator)`	static	Comparator	Comparator	Wraps comparator so nulls compare after non-nulls.

24.2.1.2 Instance Methods on Comparator

Method	Static / Instance	Return Type	Parameters	Description
`thenComparing(otherComparator)`	instance	Comparator	Comparator	Adds a secondary comparator when the primary compares equal.
`thenComparing(keyExtractor)`	instance	Comparator	Function<T,U>	Secondary comparison using natural ordering of extracted key.
`thenComparing(keyExtractor, keyComparator)`	instance	Comparator	Function<T,U>, Comparator	Secondary comparison with custom comparator.
`thenComparingInt(keyExtractor)`	instance	Comparator	ToIntFunction	Secondary numeric comparison (optimized).
`thenComparingLong(keyExtractor)`	instance	Comparator	ToLongFunction	Secondary numeric comparison.
`thenComparingDouble(keyExtractor)`	instance	Comparator	ToDoubleFunction	Secondary numeric comparison.
`reversed()`	instance	Comparator	none	Returns a reversed comparator for the same comparison logic.

24.2.2 Comparator Example

```
var people = List.of(new Person("Bob",40), new Person("Ann",30));

Comparator<Person> byName = Comparator.comparing(Person::getName);

Comparator<Person> byAgeDesc = Comparator.comparingInt(Person::getAge).reversed();

var sorted = people.stream().sorted(byName.thenComparing(byAgeDesc)).toList();
```

24.3 Comparable vs Comparator

Feature	Comparable	Comparator
Package	java.lang	java.util
Method	compareTo(T)	compare(T,T)
Sorting Type	Natural (default)	Custom (multiple strategies)
Modifies Source Class	YES	NO
Useful For	Default ordering	External or alternate ordering
Allows Multiple Orders	NO	YES
Used By Collections.sort	YES	YES
Used By Arrays.sort	YES	YES

24.4 Sorting Arrays and Collections

24.4.1 Arrays sort()

```
int[] nums = {3,1,2};
Arrays.sort(nums); // natural order

Person[] arr = {...};
Arrays.sort(arr); // Person must implement Comparable
Arrays.sort(arr, byName); // using Comparator
```

24.4.2 Collections sort()

```
Collections.sort(list); // natural order
Collections.sort(list, byName); // comparator
```

> **Note**
>
> Collections.sort(list) delegates to list.sort(comparator) since Java 8.

24.5 Multi-Level Sorting (thenComparing)

```
var cmp = Comparator
    .comparing(Person::getLastName)
        .thenComparing(Person::getFirstName)
            .thenComparingInt(Person::getAge);
```

24.6 Comparing Primitives Efficiently

```
Comparator.comparingInt(Person::getAge)
Comparator.comparingLong(...)
Comparator.comparingDouble(...)
```

> **Note**
>
> These avoid boxing and are preferred in performance-sensitive code.

24.7 Common Traps

- Sorting a list of Objects without Comparable → runtime ClassCastException
- compareTo inconsistent with equals → unpredictable behavior
- Comparator that breaks transitivity → sorting becomes undefined
- Null elements → unless Comparator handles them, sorting throws NPE
- Comparator comparing fields of mixed types → ClassCastException
- Using subtraction to compare ints can overflow → always use `Integer.compare()`
- Sorting a list with null elements and natural order → NPE
- compareTo must never return inconsistent negative/zero/positive on same two objects (no randomness)

24.8 Full Example

```
record Book(String title, double price, int year) {}

var books = List.of(
new Book("Java 17", 40.0, 2021),
new Book("Algorithms", 55.0, 2019),
new Book("Java 21", 42.0, 2023)
);

Comparator<Book> cmp =
Comparator
    .comparingDouble(Book::price)
        .thenComparing(Book::year)
            .reversed();

books.stream().sorted(cmp)
    .forEach(System.out::println);
```

> **Note**
>
> `reversed()` applies to the entire composed comparator, not just the first comparison key.

24.9 Summary

- Use `Comparable` for natural ordering (1 default order).
- Use `Comparator` for flexible or multiple sorting strategies.
- Comparators can compose (reversed, thenComparing).
- Sorting requires consistent comparison logic.
- Arrays.sort and Collections.sort use both Comparable and Comparator.

◀ 23. Shared Collection Operations & Equality | ▲ Index | 25. The List API ▶

25. The List API

Table of Contents

- 25.1 Characteristics of Lists
- 25.2 Creating Lists Constructors
  - 25.2.1 ArrayList Constructors
  - 25.2.2 LinkedList Constructors
- 25.3 Factory Methods
  - 25.3.1 List of immutable
  - 25.3.2 List copyOf immutable-copy
  - 25.3.3 Arrays asList fixed-size-list
- 25.4 Core List Operations
  - 25.4.1 Adding Elements
  - 25.4.2 Accessing Elements
  - 25.4.3 Removing Elements
  - 25.4.4 Important Behaviors and Characteristics
- 25.5 contains, equals and hashCode
  - 25.5.1 contains
  - 25.5.2 Equality of Lists
  - 25.5.3 hashCode
- 25.6 Iterating Through a List
  - 25.6.1 Classic For Loop
  - 25.6.2 Enhanced For Loop
  - 25.6.3 Iterator–ListIterator
- 25.7 The subList Method
  - 25.7.1 Syntax
  - 25.7.2 Rules
  - 25.7.3 Examples
  - 25.7.4 Modifying the parent list invalidates the view
  - 25.7.5 Modifying the subList modifies the parent
  - 25.7.6 Clearing the subList clears part of the parent list
  - 25.7.7 Common Pitfalls
- 25.8 Summary Table of Important Operations

In the `Collections Framework`, a **List** represents an ordered, index-based, duplicate-allowing collection.

The List interface extends `Collection` and is implemented by:

```
List
├── ArrayList (Resizable array — fast random access, slower inserts/removals in the middle)
├── LinkedList (Doubly-linked list — fast inserts/removals, slower random access)
└── Vector (Legacy synchronized list — rarely used today)
```

> **Note**
>
> Vector is legacy and synchronized — avoid unless explicitly required.

25.1 Characteristics of Lists

- Ordered — elements preserve insertion order.
- Indexed — accessible via `get(int)` and `set(int,E)`.
- Allow duplicates — `List` does not enforce uniqueness.
- Can contain `null` — unless using special implementations.

25.2 Creating Lists (Constructors)

25.2.1 ArrayList Constructors

```
List<String> a1 = new ArrayList<>();
List<String> a2 = new ArrayList<>(50); // initial capacity
List<String> a3 = new ArrayList<>(List.of("A", "B"));
```

> **Note**
>
> Initial capacity is not a size. It just decides how many elements the internal array can hold before resizing.

25.2.2 LinkedList Constructors

```
List<String> l1 = new LinkedList<>();
List<String> l2 = new LinkedList<>(List.of("A", "B"));
```

> **Note**
>
> `LinkedList` also implements `Deque`.

25.3 Factory Methods

25.3.1 `List.of()` (immutable)

```
List<String> list1 = List.of("A", "B", "C");
list1.add("X"); // ❌ UnsupportedOperationException
list1.set(0, "Z"); // ❌ UnsupportedOperationException
```

> **Note**
>
> All `List.of()` lists: - reject `nulls` - are immutable - throw `UOE` on structural modification

25.3.2 `List.copyOf()` (immutable copy)

```
List<String> src = new ArrayList<>();
src.add("Hello");

List<String> copy = List.copyOf(src); // immutable snapshot
```

25.3.3 Arrays.asList() (fixed-size list)

```
String[] arr = {"A", "B"};
List<String> list = Arrays.asList(arr);

list.set(0, "Z"); // OK
list.add("X"); // ❌ UOE — size is fixed
```

> **Note**
>
> The list is backed by the array: modifying one affects the other.

25.4 Core List Operations

25.4.1 Adding Elements

```
list.add("A");
list.add(1, "B"); // insert at index
list.addAll(otherList);
list.addAll(2, otherList);
```

25.4.2 Accessing Elements

```
String x = list.get(0);
list.set(1, "NewValue");
```

> **Note**
>
> `get()` throws `IndexOutOfBoundsException` for invalid indices.

If you try to `update` an element in an empty List, even at index 0, you get an `IndexOutOfBoundsException`

```
List<Integer> list = new ArrayList<Integer>();
list.add(3);
list.add(5);
System.out.println(list.toString());
list.clear();
list.set(0, 2);
```

Output

```
[3, 5]
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index 0 out of bounds for leng
```

> **Warning**
>
> Calling get/set with an invalid index throws IndexOutOfBoundsException

25.4.3 Removing Elements

```
list.remove(0); // remove(int index) — removes by index;  remove(Object) — removes first equal
list.remove("A"); // removes first occurrence
list.removeIf(s -> s.startsWith("X"));
list.clear();
```

25.4.4 Important Behaviors and Characteristics

Operation	Behavior	Exception(s)
`add(E)`	always appends	—
`add(int,E)`	shifts elements right	IndexOutOfBoundsException
`get(int)`	constant-time for ArrayList, linear for LinkedList	IndexOutOfBoundsException
`set(int,E)`	replaces element	IndexOutOfBoundsException
`remove(int)`	shifts elements left	IndexOutOfBoundsException
`remove(Object)`	removes first equal element	—

25.5 `contains()`, `equals()`, and `hashCode()`

25.5.1 `contains()`

Method `contains()` uses `.equals()` on elements.

25.5.2 Equality of Lists

`List.equals()` performs element-wise comparison in order.

```
List<String> a = List.of("A", "B");
List<String> b = List.of("A", "B");

System.out.println(a.equals(b)); // true
```

Note

- Order matters.
- Type of list does NOT matter.

25.5.3 `hashCode()`

Computed based on the contents.

25.6 Iterating Through a List

25.6.1 Classic For Loop

```
for (int i = 0; i < list.size(); i++) {
    System.out.println(list.get(i));
}
```

25.6.2 Enhanced For Loop

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

25.6.3 Iterator & ListIterator

```
Iterator<String> it = list.iterator();
while (it.hasNext()) { System.out.println(it.next()); }

ListIterator<String> lit = list.listIterator();
while (lit.hasNext()) {
    if (lit.next().equals("A")) lit.set("Z");
}
```

Warning

All standard List iterators are fail-fast: structural modification outside iterator causes ConcurrentModificationException.

Note

Only `ListIterator` supports bidirectional traversal and modification.

25.7 The `subList()` Method

`subList()` creates a view of a portion of the list, not a copy. Modifying either list can modify the other.

25.7.1 Syntax

```
List<E> subList(int fromIndex, int toIndex);
```

25.7.2 Rules

Rule	Explanation
fromIndex inclusive	element at fromIndex is included
toIndex exclusive	element at toIndex is NOT included
The view is backed by original list	modifying one modifies the other
Structural modification of parent invalidates the subList	→ ConcurrentModificationException

25.7.3 Examples

```
List<String> list = new ArrayList<>(List.of("A", "B", "C", "D"));
List<String> view = list.subList(1, 3);
// view = ["B", "C"]

view.set(0, "X");
// list = ["A", "X", "C", "D"]
// view = ["X", "C"]
```

25.7.4 Modifying the parent list invalidates the view

```
List<String> list = new ArrayList<>(List.of("A","B","C","D"));
List<String> view = list.subList(1, 3);

list.add("E"); // structural change to parent list

view.get(0); // ❌ ConcurrentModificationException
```

25.7.5 Modifying the subList modifies the parent

```
view.remove(1);
// removes "C" from both view and parent list
```

25.7.6 Clearing the subList clears part of the parent list

```
view.clear();
// removes indices 1 and 2 from the parent
```

25.7.7 Common Pitfalls

- Assuming subList is independent: it is a view, not a copy
- Assuming subList allows resizing: works only on modifiable parent lists.
- Forgetting that parent modifications invalidate results in ConcurrentModificationException
- Incorrect index expectations: End index is exclusive

25.8 Summary Table of Important Operations

Operation	ArrayList	LinkedList	Immutable Lists
`add(E)`	fast	fast	❌ unsupported
`add(index,E)`	slow (shift)	fast	❌
`get(index)`	fast	slow	fast
`remove(index)`	slow	slow (unless removing first/last)	❌
`remove(Object)`	slower	slower	❌
`set(index,E)`	fast	slow	❌
`iterator()`	fast	fast	fast
`listIterator()`	fast	fast	fast
`contains(Object)`	O(n)	O(n)	O(n)

◀ 24. Comparable, Comparator & Sorting in Java | ▲ Index | 26. Set API ▶

26. Set API

Table of Contents

- 26.1 Set Hierarchy Java-Collections-Framework
- 26.2 Characteristics of Each Set Implementation
  - 26.2.1 HashSet
  - 26.2.2 LinkedHashSet
  - 26.2.3 TreeSet
- 26.3 Equality Rules in Sets
  - 26.3.1 HashSet–LinkedHashSet
  - 26.3.2 TreeSet
- 26.4 Creating Set Instances
  - 26.4.1 Using Constructors
  - 26.4.2 Copy Constructors
  - 26.4.3 Factory Methods
- 26.5 Main Operations on Sets
  - 26.5.1 Adding Elements
  - 26.5.2 Checking Membership
  - 26.5.3 Removing Elements
  - 26.5.4 Bulk Operations
- 26.6 Common Pitfalls
- 26.7 Summary Table

A **Set** in Java represents a collection that **contains no duplicate elements**.

It models the mathematical concept of a `set`: unordered (unless using an ordered implementation) and composed of unique values.

All Set implementations rely on **equality semantics** (either `equals()` or `comparator` logic.

26.1 Set Hierarchy (Java Collections Framework)

```
Set<E>
 ├── SequencedSet<E> (Java 21+)
 │     └── LinkedHashSet<E>    (ordered)
 ├── HashSet<E>                (unordered)
 └── SortedSet<E>
       └── NavigableSet<E>
             └── TreeSet<E>    (sorted)
```

All `Set` implementations require:

- uniqueness of elements
- predictable equality and hashing (depending on implementation)

> **Note**
>
> `LinkedHashSet` is now formally a `SequencedSet` since Java 21.

26.2 Characteristics of Each Set Implementation

26.2.1 HashSet

- Fastest general-purpose Set
- Unordered (no iteration order guarantee)
- Uses `hashCode()` and `equals()`
- Allows one `null` element

```
Set<String> set = new HashSet<>();
set.add("A");
set.add("B");
set.add("A");   // duplicate ignored
System.out.println(set); // order not guaranteed
```

26.2.2 LinkedHashSet

- Maintains **insertion order**
- Slightly slower than HashSet
- Useful when predictable iteration order is required

```
Set<String> set = new LinkedHashSet<>();
set.add("A");
set.add("C");
set.add("B");
System.out.println(set);  // [A, C, B]
```

26.2.3 TreeSet

A **sorted** Set whose order is determined by:

- Natural ordering (`Comparable`)
- A provided `Comparator`

TreeSet:

- No `null` elements allowed (NullPointerException at runtime)
- Guarantees sorted iteration
- Supports range views: `headSet()` , `tailSet()` , `subSet()`

```
TreeSet<Integer> tree = new TreeSet<>();
tree.add(10);
tree.add(1);
tree.add(5);

System.out.println(tree); // [1, 5, 10]
```

> **Note**
>
> `TreeSet` requires all elements to be mutually comparable — mixing non-comparable types produces `ClassCastException` . Operations (add, remove, contains) are O(log n).

26.3 Equality Rules in Sets

The rules differ depending on implementation.

26.3.1 HashSet & LinkedHashSet

`Uniqueness` is determined by two methods:

- `hashCode()`
- `equals()`

Two objects are considered the same element if:

- Their hash codes match
- Their `equals()` method returns `true`

> **Warning**
>
> If you mutate an object after adding it to a HashSet or LinkedHashSet, its hashCode may change and the set may lose track of it.

26.3.2 TreeSet

Uniqueness is based on `compareTo()` or the provided `Comparator`.

If `compare(a, b) == 0` then the objects are considered duplicates, even if `equals()` is false.

```
Comparator<String> comp = (a, b) -> a.length() - b.length();
Set<String> set = new TreeSet<>(comp);

set.add("Hi");
set.add("Yo"); // same length → treated as duplicate

System.out.println(set);  // ["Hi"]
```

26.4 Creating Set Instances

26.4.1 Using Constructors

```
Set<String> s1 = new HashSet<>();
Set<String> s2 = new LinkedHashSet<>();
Set<String> s3 = new TreeSet<>();
```

26.4.2 Copy Constructors

```
List<String> list = List.of("A", "B", "C");

Set<String> copy = new HashSet<>(list); // order lost
System.out.println(copy);

Set<String> ordered = new LinkedHashSet<>(list); // maintains the order from the list
System.out.println(ordered);
```

26.4.3 Factory Methods

```
Set<String> s1 = Set.of("A", "B", "C");   // immutable
Set<String> empty = Set.of();             // empty immutable set
```

> **Note**
>
> Factory-created sets are **immutable**: adding or removing elements throws `UnsupportedOperationException`. `Set.of(...)` rejects duplicates at creation time → IllegalArgumentException and rejects null → NullPointerException

---

26.5 Main Operations on Sets

26.5.1 Adding Elements

```
set.add("A");           // returns true if added
set.add("A");           // returns false if duplicate
```

26.5.2 Checking Membership

```
set.contains("A");
```

26.5.3 Removing Elements

```
set.remove("A");
set.clear();
```

26.5.4 Bulk Operations

```
set.addAll(otherSet);
set.removeAll(otherSet);
set.retainAll(otherSet); // intersection
```

26.6 Common Pitfalls

- Using TreeSet with non-comparable objects → `ClassCastException`
- TreeSet does not use `equals()` at all: only comparator/compareTo decides uniqueness.
- Using mutable objects as Set keys → breaks hashing rules
- Factory Set.of() is immutable — modification fails
- HashSet does not guarantee iteration order
- TreeSet treats objects with compare()==0 as duplicates even if not equal

---

26.7 Summary Table

Implementation	Keeps Order?	Allows Null?	Sorted?	Underlying Logic
HashSet	No	Yes (1 null)	No	hashCode + equals
LinkedHashSet	Yes (insertion order)	Yes (1 null)	No	hash table + linked list
TreeSet	Yes (sorted)	No	Yes (natural/comparator)	compareTo / Comparator

---

◀ 25. The List API | ▲ Index | 27. Queue & Deque API ▶

27. Queue & Deque API

Table of Contents

- 27.1 Queue — Overview
  - 27.1.1 Queue Core Methods
  - 27.1.2 Queue Implementations
- 27.2 Deque — Overview
  - 27.2.1 Deque Core Methods
  - 27.2.2 Deque Implementations
- 27.3 Using a Queue
- 27.4 Using a Deque as-Queue and as-Stack
  - 27.4.1 FIFO Example Queue-Behavior
  - 27.4.2 LIFO Example Stack-Behavior
- 27.5 PriorityQueue — Special Queue
- 27.6 Blocking Queues Basics
- 27.7 Common Pitfalls
- 27.8 Summary Table

Java's `Queue` and `Deque` interfaces model ordered collections designed for processing elements in a particular sequence.

- A **Queue** typically models a **FIFO** (First-In, First-Out) structure.
- A **Deque** (`double-ended queue`) allows insertion and removal from both ends, enabling **FIFO** and **LIFO** behavior in a single API.

27.1 Queue — Overview

The `Queue` interface extends `Collection` and is commonly used in asynchronous programming, work distribution, algorithms, and buffering.

Two families of methods exist: ones that **throw exceptions** and ones that **return special values** (usually `null`).

27.1.1 Queue Core Methods

Operation	Throws Exception	Returns Special Value	Description
Insert	`add(e)`	`offer(e)`	Adds an element; offer preferred for bounded queues
Remove	`E remove()`	`E poll()`	Removes and returns head. `remove()` throws NoSuchElementException if queue is empty, `poll()` returns null
Read	`E element()`	`E peek()`	Returns head without removing. `element()` throws NoSuchElementException if queue is empty, `peek()` returns null

27.1.2 Queue Implementations

Common classes implementing `Queue`:

- `LinkedList` — unbounded, also implements `Deque` and `List`.
- `ArrayDeque` — fast, resizable array-based queue; cannot store `null`.
- `PriorityQueue` — orders elements by natural order or comparator; not FIFO.
- `ConcurrentLinkedQueue` — thread-safe, lock-free.

> **Note**
>
> `PriorityQueue` does not guarantee traversal order matching priority sorting.

> **Warning**
>
> Most Queue implementations reject `null` because `null` is used as a return value for “empty”.

27.2 Deque — Overview

`Deque` (double-ended queue) supports insertion, removal, and inspection from both the head and the tail.

It is more versatile than a Queue:

- FIFO (queue-like)
- LIFO (stack-like)
- Bidirectional algorithms

27.2.1 Deque Core Methods

Operation	Front	End
Insert	addFirst(e), offerFirst(e)	addLast(e), offerLast(e)
Remove	removeFirst(), pollFirst()	removeLast(), pollLast()
Examine	getFirst(), peekFirst()	getLast(), peekLast()

27.2.2 Deque Implementations

- `ArrayDeque` — recommended general-purpose implementation (fast, no capacity limit).
- `LinkedList` — full-featured but slower due to node-based structure.
- `ConcurrentLinkedDeque` — non-blocking concurrent deque.

> **Note**
>
> `Stack` is legacy; use `Deque` for stack behavior (push/pop). ArrayDeque, LinkedList queue operations (add/remove/peek) are O(1) amortized

---

27.3 Using a Queue

```
Queue<String> q = new LinkedList<>();

q.offer("A");
q.offer("B");
q.offer("C");

System.out.println(q.peek());   // A
System.out.println(q.poll());   // A
System.out.println(q.poll());   // B
System.out.println(q.poll());   // C
System.out.println(q.poll());   // null (empty queue)
```

27.4 Using a Deque (as Queue and as Stack)

27.4.1 FIFO Example (Queue Behavior)

```
Deque<String> dq = new ArrayDeque<>();

dq.offerLast("A"); // enqueue
dq.offerLast("B");
dq.offerLast("C");

System.out.println(dq.pollFirst()); // A
System.out.println(dq.pollFirst()); // B
System.out.println(dq.pollFirst()); // C
```

27.4.2 LIFO Example (Stack Behavior)

```
Deque<String> stack = new ArrayDeque<>();

stack.push("A");
stack.push("B");
stack.push("C");

System.out.println(stack.pop()); // C
System.out.println(stack.pop()); // B
System.out.println(stack.pop()); // A
```

27.5 PriorityQueue — Special Queue

`PriorityQueue` orders elements by **natural order** or by a provided `Comparator`.

Important characteristics:

- Not FIFO — head is the "smallest" element.
- Order is only guaranteed during removal, not iteration.
- Null elements not permitted.

```
PriorityQueue<Integer> pq = new PriorityQueue<>();

pq.offer(50);
pq.offer(10);
pq.offer(30);

System.out.println(pq.poll()); // 10
System.out.println(pq.poll()); // 30
System.out.println(pq.poll()); // 50
```

27.6 Blocking Queues (Basics)

In concurrent environments, the `java.util.concurrent` package provides blocking queue types.

- `ArrayBlockingQueue` — fixed-size backing array.
- `LinkedBlockingQueue` — optionally bounded.
- `PriorityBlockingQueue` — thread-safe priority queue.
- `DelayQueue` — elements released after delays.

> **Note**
>
> - BlockingQueue never allows `null`.
> - put(e) — blocks until space available
> - take() — blocks until element available
> - BlockingQueue also supports timed operations: offer(e, timeout), poll(timeout)

27.7 Common Pitfalls

- `Queue` and `Deque` methods come in "exception" and "special-value" variants — memorize which is which.
- `ArrayDeque` cannot store `null` — `null` is used internally.
- `PriorityQueue` iteration order is NOT sorted.
- Using `Stack` is discouraged; use `Deque` instead.
- Deque enables both FIFO and LIFO and has the **most complete** API.

27.8 Summary Table

Interface	Typical Behavior	Null Allowed?	Common Implementations	Notes
Queue	FIFO	Depends	LinkedList, ArrayDeque, PriorityQueue	PriorityQueue not FIFO
Deque	FIFO + LIFO	No (ArrayDeque)	ArrayDeque, LinkedList	Full double-ended operations
PriorityQueue	Ordered by priority	No	PriorityQueue	Removes smallest element first
BlockingQueue	Thread-safe FIFO	No	ArrayBlockingQueue, LinkedBlockingQueue	add/offer vs put differences
ConcurrentLinkedQueue	Lock-free FIFO	No	ConcurrentLinkedQueue	Very fast for multi-threading

---

◀ 26. Set API | ▲ Index | 28. Map API ▶

28. Map API

Table of Contents

- 28.1 Core Map Characteristics
- 28.2 Main Map Implementations
- 28.3 Creating Maps
- 28.4 Basic Map Operations
- 28.5 Iterating Over a Map
- 28.6 Determining Equality in Maps
- 28.7 Special Behavior of TreeMap
- 28.8 Null Handling
- 28.9 Common Pitfalls
- 28.10 Summary

The `Map` interface represents a collection of **key–value pairs**, where each key maps to at most one value.

Unlike other collection types, `Map` does **not** extend `Collection` and therefore has its own hierarchy and rules.

28.1 Core Map Characteristics

- Each key is unique; **duplicate keys overwrite the previous value**
- Values may be duplicated
- Maps do not support positional (index-based) access
- Iteration is performed over `keySet()`, `values()`, or `entrySet()`

> **Note**
>
> A `Map` is not a `Collection`, but its views (keySet, values, entrySet) are collections.

28.2 Main Map Implementations

Implementation	Ordering	Null Keys	Null Values	Thread-Safe	Notes
`HashMap`	No ordering	1	Many	No	Fast, most common
`LinkedHashMap`	Insertion order	1	Many	No	Predictable iteration
`TreeMap`	Sorted by key	No	Many	No	Keys must be comparable
`Hashtable`	No ordering	No	No	Yes	Legacy
`ConcurrentHashMap`	No ordering	No	No	Yes	Concurrent-friendly

> **Note**
>
> `TreeMap` ordering is determined either by `Comparable` or by a `Comparator` provided at construction.

28.3 Creating Maps

`Maps` can be created using constructors or factory methods.

```
Map<String, Integer> map1 = new HashMap<>();
Map<String, Integer> map2 = new LinkedHashMap<>();
Map<String, Integer> map3 = new TreeMap<>();

Map<String, Integer> map4 = Map.of("A", 1, "B", 2);
Map<String, Integer> map5 = Map.ofEntries(
    Map.entry("X", 10),
    Map.entry("Y", 20)
);
```

> **Note**
>
> Maps created with `Map.of(...)` and `Map.ofEntries(...)` are **immutable**. Any modification attempt throws `UnsupportedOperationException`.

28.4 Basic Map Operations

Method	Description	Return
`put(k, v)`	Adds or replaces a mapping	Return prev. value or null
`putIfAbsent(k,v)`	Adds only if key not present	Returns existing or null
`get(k)`	Returns value or null	Return specific value or null
`getOrDefault(k, default)`	Returns value or default	Return specific value or default
`remove(k)`	Removes mapping	Remove and return specific value or null
`containsKey(k)`	Checks key presence	boolean
`containsValue(v)`	Checks value presence	boolean
`size()`	Number of entries	int
`isEmpty()`	Empty check	boolean
`clear()`	Removes all entries	void
`V merge(k, v, BiFunction(V, V, V))`	merge(k, v, remappingFunction)	if key absent → sets value; if key present → function(oldValue, newValue); if function returns null → mapping removed

```
Map<String, String> map = new HashMap<>();
map.put("A", "Apple");
map.put("B", "Banana");

map.put("A", "Avocado"); // overwrites value

String v = map.get("B"); // Banana
```

28.5 Iterating Over a Map

Maps are iterated via views:

- `keySet()` → Set of keys
- `values()` → Collection of values
- `entrySet()` → Set of Map.Entry

```
for (String key : map.keySet()) {
    System.out.println(key);
}

for (String value : map.values()) {
    System.out.println(value);
}

for (Map.Entry<String, String> e : map.entrySet()) {
    System.out.println(e.getKey() + " = " + e.getValue());
}
```

> **Note**
>
> Modifying the map while iterating over these views may throw `ConcurrentModificationException` (except for concurrent maps).

28.6 Determining Equality in Maps

Map equality is defined as follows:

- Two maps are equal if they contain the same key–value mappings
- Key comparison uses `equals()`
- Value comparison uses `equals()`

```
Map<String, Integer> m1 = Map.of("A", 1, "B", 2);
Map<String, Integer> m2 = Map.of("B", 2, "A", 1);

System.out.println(m1.equals(m2)); // true
```

> **Note**
>
> Iteration order does not affect map equality.

28.7 Special Behavior of TreeMap

TreeMap maintains entries in sorted order based on keys.

```
Map<Integer, String> tm = new TreeMap<>();
tm.put(3, "C");
tm.put(1, "A");
tm.put(2, "B");

System.out.println(tm); // {1=A, 2=B, 3=C}
```

> **Warning**
>
> All keys in a `TreeMap` must be mutually comparable. Mixing incompatible types causes `ClassCastException` at runtime.

28.8 Null Handling

Implementation	Null Key	Null Value
HashMap	Yes (1)	Yes
LinkedHashMap	Yes (1)	Yes
TreeMap	No	Yes
Hashtable	No	No
ConcurrentHashMap	No	No

> **Note**
>
> TreeMap accepts null values only when they do not participate in key comparison. In practice this is rare, because null keys are banned and comparators may reject nulls.
>
> HashMap/LinkedHashMap allow only ONE null key — inserting another replaces the existing one.

28.9 Common Pitfalls

- Assuming Map is a Collection
- Forgetting that duplicate keys overwrite values
- Using null keys in TreeMap or ConcurrentHashMap
- Confusing iteration order with equality
- Trying to modify immutable maps created via Map.of

28.10 Summary

- Maps store unique keys mapped to values
- Ordering depends on implementation
- Equality is based on key–value pairs
- TreeMap requires comparable keys
- Immutable maps throw exceptions on modification

◀ 27. Queue & Deque API | ▲ Index | 29. Sequenced Collections & Sequenced Maps ▶

29. Sequenced Collections & Sequenced Maps

Table of Contents

- 29.1 Motivation and Background
- 29.2 SequencedCollection Interface
  - 29.2.1 Core Methods of SequencedCollection
  - 29.2.2 Implementations of SequencedCollection
  - 29.2.3 Reversed Views
- 29.3 SequencedMap Interface
  - 29.3.1 Core Methods of SequencedMap
  - 29.3.2 Implementations of SequencedMap
  - 29.3.3 Reversed Maps
- 29.4 Relationship with Existing APIs
  - 29.4.1 Which Built-in Types Are Sequenced
- 29.5 Common Pitfalls
- 29.6 Summary

Java 21 introduces `Sequenced Collections` and `Sequenced Maps` to unify and formalize access to elements based on their encounter order.

This addition solves long-standing inconsistencies between lists, sets, queues, deques, and maps, providing a common API to work with the first and last elements, as well as reversed views.

29.1 Motivation and Background

Before Java 21, ordered collections (such as List, LinkedHashSet, Deque, or LinkedHashMap) exposed ordering operations through different methods or not at all.

Developers had to rely on implementation-specific APIs or indirect workarounds.

Sequenced interfaces introduce a consistent contract for all ordered collections and maps, making order-based operations explicit, safe, and uniform.

29.2 SequencedCollection Interface

`SequencedCollection<E>` is a new interface that extends Collection and represents collections with a well-defined encounter order.

Implemented by `List`, `Deque`, and `LinkedHashSet` (TreeSet is ordered but does not implement SequencedCollection directly).

29.2.1 Core Methods of SequencedCollection

The interface defines methods to access and manipulate elements at both ends of the collection.

Method	Description
`E getFirst()`	Returns the first element
`E getLast()`	Returns the last element
`void addFirst(E e)`	Inserts element at the beginning
`void addLast(E e)`	Inserts element at the end
`E removeFirst()`	Removes and returns the first element
`E removeLast()`	Removes and returns the last element
`SequencedCollection<E> reversed()`	Returns a reversed view

29.2.2 Implementations of SequencedCollection

The following standard types implement SequencedCollection:

Type	Notes
List	Ordered by index
Deque	Double-ended queue
LinkedHashSet	Maintains insertion order

29.2.3 Reversed Views

Calling reversed() does not create a copy.

It returns a live view of the same collection with inverted order.

```
List<Integer> list = new ArrayList<>(List.of(1, 2, 3));
SequencedCollection<Integer> rev = list.reversed();

rev.removeFirst(); // removes 3
System.out.println(list); // [1, 2]
```

> **Note**
>
> Reversed views share the same backing collection. Structural changes in either view affect the other: modifying either the original collection or the reversed view affects the other.

29.3 SequencedMap Interface

`SequencedMap<K,V>` extends `Map<K,V>` and represents maps with a defined encounter order of entries.

It standardizes operations that previously existed only in specific implementations such as `LinkedHashMap`.

29.3.1 Core Methods of SequencedMap

Method	Description
`Entry<K,V> firstEntry()`	First map entry
`Entry<K,V> lastEntry()`	Last map entry
`Entry<K,V> pollFirstEntry()`	Removes and returns the first entry, or null if empty
`Entry<K,V> pollLastEntry()`	Removes and returns last entry, or null if empty
`SequencedMap<K,V> reversed()`	Reversed view of the map

29.3.2 Implementations of SequencedMap

Currently, the primary standard implementation is:

Type	Ordering
LinkedHashMap	Insertion order (or access order if configured)

> **Note**
>
> LinkedHashMap can reorder entries on read if constructed with accessOrder=true.
>
> In that case, “first” and “last” reflect most-recent-access order.

29.3.3 Reversed Maps

As with collections, reversed() on a sequenced map returns a view, not a copy.

```
SequencedMap<String, Integer> map =
new LinkedHashMap<>(Map.of("A", 1, "B", 2, "C", 3));

SequencedMap<String, Integer> rev = map.reversed();

rev.pollFirstEntry(); // removes C=3
System.out.println(map); // {A=1, B=2}
```

> **Note**
>
> Like SequencedCollection, `reversed()` returns a live view — mutations apply to both maps.

29.4 Relationship with Existing APIs

Sequenced interfaces do not replace existing collection types.

They sit above them in the hierarchy and unify common behaviors.

All existing ordered collections automatically benefit from these APIs without breaking backward compatibility.

29.4.1 Which Built-in Types Are Sequenced?

The following table summarizes whether standard collection types are ordered, and whether they implement the new Sequenced interfaces.

Type	Ordered?	SequencedCollection?	SequencedMap?
`List`	✓ Yes	✓ Yes	✗ No
`Deque`	✓ Yes	✓ Yes	✗ No
`LinkedHashSet`	✓ Yes	✓ Yes	✗ No
`TreeSet`	✓ Yes (sorted)	✗ No*	✗ No
`HashSet`	✗ No	✗ No	✗ No
`LinkedHashMap`	✓ Yes	✗ No	✓ Yes
`HashMap`	✗ No	✗ No	✗ No
`TreeMap`	✓ Yes (sorted)	✗ No	✗ No

> **Note**
>
> `TreeSet` is ordered, but implements `SortedSet` / `NavigableSet`, not `SequencedCollection`.

29.5 Common Pitfalls

- Sequenced interfaces define views, not copies
- `reversed()` reflects changes bidirectionally
- Not all Set or Map implementations are sequenced
- HashSet and HashMap do not implement sequenced interfaces
- Order is guaranteed only when explicitly defined
- Removing elements via iterator on reversed view impacts original order immediately.

29.6 Summary

- Sequenced interfaces formalize encounter order
- They provide first/last access and reversal
- They work via live views, not copies
- They unify APIs across lists, deques, sets, and maps

◀ 28. Map API | ▲ Index | 30. Java Threads – Fundamentals and Execution Model ▶

Concurrency and Threads

30. Java Threads – Fundamentals and Execution Model

Table of Contents

- 30.1 Threads Processes and the Operating System
- 30.2 Memory Model Stack and Heap
- 30.3 Context and Context Switching
- 30.4 Concurrency vs Parallelism
- 30.5 Threads in Java Conceptual Model
- 30.6 Thread Categories in Java 21
- 30.7 Creating Threads in Java
- 30.8 Thread Lifecycle and Execution
- 30.9 Starting vs Running a Thread Synchronous-or-Asynchronous
- 30.10 Thread Priority and Scheduling
- 30.11 Thread Deferring and Yielding
- 30.12 Thread Interruption and Cooperative Cancellation
  - 30.12.1 What Interrupting a Thread Means
  - 30.12.2 Interrupting Blocking Operations
  - 30.12.3 Checking the Interruption Status
  - 30.12.4 Example Interrupting-a-Sleeping-Thread
  - 30.12.5 Key Observations
- 30.13 Threads and the Main Thread
- 30.14 Thread Concurrency and Shared State
- 30.15 Summary

This chapter introduces **threads** from first principles and explains how they are modeled and used in Java 21.

It builds the conceptual foundation required for understanding `concurrency`, `synchronization`, and the `Java Concurrency API` covered in the next chapter.

30.1 Threads, Processes, and the Operating System

To understand threads, we must start from the operating system execution model. Modern operating systems execute programs using **processes** and **threads**.

- **Process**: An executing program instance managed by the operating system. A process owns its own virtual memory space, system resources (files, sockets), and at least one thread.
- **Thread**: A lightweight execution unit within a process. Threads share the process memory and resources but execute independently.
- **Task**: A logical unit of work to be executed. A task may be executed by a thread but is not itself a thread.
- **CPU Core**: A physical or logical execution unit capable of running one thread at a time. Multiple cores allow true parallel execution.

A single process can contain many threads, all operating within the same shared environment. This shared environment is both the source of concurrency power and concurrency risk.

30.2 Memory Model: Stack and Heap

Threads interact with memory in two fundamentally different ways.

- **Thread Stack**: Private memory area for each thread. It stores method call frames, local variables, and execution state. Each thread has exactly one stack.
- **Heap**: Shared memory area used for objects and class instances. All threads within the same process can access the heap.

Because `stacks are isolated` and the `heap is shared`, concurrency problems arise when multiple threads access the same heap objects without proper coordination.

30.3 Context and Context Switching

The operating system schedules threads onto CPU cores.

Since the number of runnable threads often exceeds the number of available cores, the OS performs **context switching**.

- **Context**: The complete execution state of a thread, including registers, program counter, and stack pointer.
- **Context Switch**: The act of suspending one thread and resuming another by saving and restoring their contexts.

Context switching enables concurrency but has a cost: CPU cycles are consumed without executing application logic.

Java developers must design systems that balance concurrency and efficiency.

30.4 Concurrency vs Parallelism

These two terms are often confused but describe different concepts.

- **Concurrency**: Multiple threads are in progress during the same time interval, possibly interleaved on a single CPU core.
- **Parallelism**: Multiple threads execute simultaneously on different CPU cores.

Java supports concurrency independently of hardware parallelism.

Even on a single-core system, Java threads can be concurrent through time slicing.

30.5 Threads in Java: Conceptual Model

In Java, a **thread** represents an independent path of execution within a single JVM process. All Java threads run within the same heap and class loader context unless explicitly isolated by advanced mechanisms.

- **Java Thread**: An object of type `java.lang.Thread` that maps to an underlying execution unit.
- **Runnable**: A functional interface representing a task whose `run()` method contains executable logic.

A thread executes code by invoking its `run()` method, either directly or indirectly through the JVM thread scheduler: please see [Starting vs Running a Thread]

30.6 Thread Categories in Java 21

Java 21 defines multiple kinds of threads, differing in lifecycle, scheduling, and intended use.

- **Platform Thread**: A traditional Java thread mapped one-to-one to an operating system thread.
- **Virtual Thread**: A lightweight thread managed by the JVM and scheduled onto carrier threads. Introduced to enable massive concurrency with minimal overhead.
- **Carrier Thread**: A platform thread used internally by the JVM to execute virtual threads.
- **Daemon Thread**: A background thread that does not prevent JVM termination. When only daemon threads remain, the JVM exits.
- **User Thread**: Any non-daemon thread. The JVM waits for all user threads to complete before exiting.
- **System Thread**: Threads created internally by the JVM for garbage collection, JIT compilation, and other runtime services.

> **Note**
>
> - `Virtual threads` are lightweight user threads; they are **not** daemon by default;
> - A VirtualThread (created directly via `Thread.startVirtualThread()` or `Thread.ofVirtual().start(...)`) accepts a Runnable as its task. It does not directly accept a Callable: If you need to run a Callable with virtual threads and retrieve a result, you must use an ExecutorService;
> - Virtual threads are implemented by the `java.lang.VirtualThread` class. This class extends `BaseVirtualThread`, which itself extends Thread. Therefore, a virtual thread is technically a subclass of Thread. However, it is not accurate to describe a virtual thread as a direct instance of the Thread class, since it is actually an instance of a specialized subclass designed specifically for virtual thread behavior.

30.7 Creating Threads in Java

Threads can be created in multiple ways, all conceptually centered around providing executable logic.

- Extending `Thread` and overriding `run()`.
- Passing a `Runnable` to a `Thread` constructor.
- Using thread factories and executors (covered in the Concurrency API section).

```
Runnable runnable = ...

  // Create a platform thread through constructor
  Thread thread = new Thread(runnable);
  thread.start();

  // Start a daemon thread to run a task
  Thread thread = Thread.ofPlatform().daemon().start(runnable);

  // Create an unstarted thread with name "duke", its start() method
  // must be invoked to schedule it to execute.
  Thread thread = Thread.ofPlatform().name("duke").unstarted(runnable);

  // A ThreadFactory that creates daemon threads named "worker-0", "worker-1", ...
  ThreadFactory factory = Thread.ofPlatform().daemon().name("worker-", 0).factory();

  // Start a virtual thread to run a task
  Thread thread = Thread.ofVirtual().start(runnable);

  // A ThreadFactory that creates virtual threads
  ThreadFactory factory = Thread.ofVirtual().factory();
```

> **Warning**
>
> - Thread creation alone does not start execution.
> - Execution begins only when the JVM scheduler is engaged.

30.8 Thread Lifecycle and Execution

A Java thread progresses through well-defined states during its lifetime.

- **New**: Thread object created but not yet started.
- **Runnable**: Eligible for execution by the scheduler.
- **Running**: Actively executing on a CPU core.
- **Blocked / Waiting**: Temporarily unable to proceed due to synchronization or coordination.
- **Terminated**: Execution completed or aborted.

The JVM and operating system cooperate to move threads between these states.

Threads in `BLOCKED`, `WAITING` or `TIMED_WAITING` state are **not using any CPU resources**

30.9 Starting vs Running a Thread: Synchronous or Asynchronous

A critical conceptual distinction exists between invoking `run()` and invoking `start()`.

- Calling `run()` directly executes the method synchronously in the current thread, like a normal method call.
- Calling `start()` requests the JVM to create a new call stack and execute `run()` asynchronously in a separate thread.

Therefore, code such as `new Thread(r).run();` does NOT create concurrency. The execution remains synchronous and blocks the calling thread until completion.

> **Note**
>
> `Asynchronous execution` means the caller continues immediately while the new thread progresses independently, subject to scheduling.
>
> `Synchronous execution` means the caller waits for the operation to complete.

> **Important**
>
> Concurrency starts **only** when `start()` is invoked.

30.10 Thread Priority and Scheduling

Java threads have an associated priority hint that influences scheduling.

- `Thread Priority`: An integer value indicating relative importance, ranging from minimum to maximum.
- `Scheduling`: The JVM delegates scheduling decisions to the operating system, which may or may not honor priorities strictly.

Thread priority affects scheduling probability but never guarantees execution order. Portable Java code must never rely on priorities for correctness.

You can set **priority** on `platform threads`; for `virtual threads` the **priority** is always set to **5** (Thread.NORM_PRIORITY) and trying to change it has no effect.

30.11 Thread Deferring and Yielding

Threads can voluntarily influence scheduling behavior.

Calling Thread.yield() signals willingness to pause execution.

- `Yielding` : A thread hints that it is willing to pause execution to allow other runnable threads to proceed.
- `Sleeping` : A thread suspends execution for a fixed duration, entering a timed waiting state.

These mechanisms do not guarantee immediate execution of other threads; they merely provide scheduling hints.

30.12 Thread Interruption and Cooperative Cancellation

Java threads cannot be stopped forcibly from the outside.

Instead, Java provides a cooperative mechanism called **thread interruption**, which allows one thread to request that another thread stop what it is doing.

The target thread decides how and when to respond.

30.12.1 What Interrupting a Thread Means

Interrupting a thread does **not** terminate it. Calling `interrupt()` sets an internal **interruption flag** on the target thread. It is the responsibility of the running thread to observe this flag and react appropriately.

- `Interrupt Request` : A signal sent to a thread indicating that it should stop or change its current activity.
- `Interruption Flag` : A boolean status associated with each thread, set when `interrupt()` is invoked.
- `Cooperative Cancellation` : A design pattern where threads periodically check for interruption and terminate themselves cleanly.

30.12.2 Interrupting Blocking Operations

Some blocking methods in Java respond immediately to interruption by throwing `InterruptedException` and clearing the interruption flag. These methods include `sleep()` , `wait()` , and `join()` .

When a thread is blocked in one of these methods and another thread interrupts it, the blocked thread is awakened and an exception is thrown. This provides a safe escape point from blocking operations.

30.12.3 Checking the Interruption Status

Threads that are not blocked must explicitly check whether they have been interrupted. Java provides two ways to do this.

- `Thread.currentThread().isInterrupted()` : Returns the interruption status without clearing it.
- `Thread.interrupted()` : Returns the interruption status and clears it. This is subtle: the next call will return false.

Failing to check the interruption status may cause threads to ignore cancellation requests and run indefinitely.

30.12.4 Example: Interrupting a Sleeping Thread

The following example demonstrates cooperative cancellation using interruption.

A worker thread repeatedly sleeps while performing work. The main thread interrupts it, causing a clean shutdown.

```
class Main {

    static class Task implements Runnable {
        public void run() {
            try {
                while (true) {
                    System.out.println("Working...");
                    Thread.sleep(1000);
                }
            } catch (InterruptedException e) {
                System.out.println("Task interrupted, shutting down");
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Thread worker = new Thread(new Task());
        worker.start();
        System.out.println("main before sleep...");
        Thread.sleep(3000);
        System.out.println("main after sleep...");
        worker.interrupt();
        System.out.println("main reached END");
    }
}
```

Output:

```
main before sleep...
Working...
Working...
Working...
main after sleep...
main reached END
Task interrupted, shutting down
```

> **Note**
>
> Output order may vary slightly due to scheduling.

30.12.5 Key Observations

- Calling `interrupt()` does not stop the thread directly.
- The interruption is detected because `sleep()` throws `InterruptedException`.
- The worker thread terminates itself in a controlled manner.
- Proper interruption handling allows threads to release resources and maintain program correctness.

> **Note**
>
> Swallowing `InterruptedException` without terminating or restoring the interruption status is considered bad practice and may lead to unresponsive threads.

30.13 Threads and the Main Thread

Every Java application starts with a **main thread**. This thread executes the `main(String[])` method.

- The main thread is a user thread.
- The JVM remains alive as long as at least one user thread is running.
- If the main thread terminates but other user threads exist, the JVM continues execution waiting for the user threads to be done.
- Daemon threads do not keep JVM alive.

Understanding the role of the main thread is essential for reasoning about program termination and background processing.

30.14 Thread Concurrency and Shared State

`Concurrency` arises when multiple threads access shared mutable state.

- `Shared State` : Any heap-based data accessible by more than one thread.
- `Race Condition` : A correctness error caused by unsynchronized access to shared state.
- `Visibility Problem` : A thread observes stale data due to lack of proper memory synchronization.

Java solves these with synchronization, volatile, locks, atomics, and high-level frameworks (Executors, futures).

Synchronization, volatile variables, and higher-level concurrency utilities will be studied in subsequent sections.

30.15 Summary

- `Threads` are the fundamental building block of concurrent execution in Java.
- They exist within processes, share memory, and are scheduled by the JVM in cooperation with the operating system.
- Correct thread management avoids leaks, deadlocks, and wasted CPU.

◀ 29. Sequenced Collections & Sequenced Maps | ▲ Index | 31. Java Concurrency APIs ▶

31. Java Concurrency APIs

Table of Contents

- 31.1 Goals and Scope of the Concurrency API
- 31.2 Fundamental Threading Problems
  - 31.2.1 Race Conditions
  - 31.2.2 Deadlock
  - 31.2.3 Starvation
  - 31.2.4 Livelock
- 31.3 From Threads to Tasks
- 31.4 Executor Framework
  - 31.4.1 Submitting Tasks and Futures
  - 31.4.2 Callable vs Runnable
- 31.5 Thread Pools and Scheduling
- 31.6 Executor Lifecycle and Termination
- 31.7 Thread Safety Strategies
  - 31.7.1 Synchronization
  - 31.7.2 Atomic Variables
    - 31.7.2.1 Atomic classes
    - 31.7.2.2 Atomic methods
  - 31.7.3 Lock Framework
    - 31.7.3.1 Lock implementations
    - 31.7.3.2 Common Lock methods
  - 31.7.4 Coordination Utilities
- 31.8 Concurrent Collections
- 31.9 Parallel Streams
- 31.10 Relation to Virtual Threads
- 31.11 Summary

This chapter introduces the **Java Concurrency API**, which provides high-level abstractions for managing concurrent execution safely, efficiently, and scalably.

Unlike low-level thread manipulation, the Concurrency API focuses on **tasks**, **executors**, and **coordination mechanisms**, allowing developers to reason about what should be done rather than how threads are scheduled.

31.1 Goals and Scope of the Concurrency API

The `Java Concurrency API`, primarily located in the `java.util.concurrent` package, was introduced to address fundamental problems inherent in manual thread management.

- Decouple task submission from thread management.
- Reduce error-prone low-level synchronization.
- Improve scalability and performance on multi-core systems.
- Provide structured mechanisms for coordination, cancellation, and shutdown.

The API does not eliminate concurrency problems but provides disciplined tools to manage them safely and predictably.

Instead of explicitly creating and controlling threads, developers submit tasks and let the framework manage **thread allocation, reuse, and synchronization**.

```java
ExecutorService executor = Executors.newSingleThreadExecutor();
executor.execute(() -> System.out.println("Task executed"));
executor.shutdown();
```

31.2 Fundamental Threading Problems

Before understanding the `Concurrency API`, it is essential to understand the concurrency problems it is designed to mitigate.

These problems arise from `shared mutable state`, `scheduling unpredictability`, and `improper coordination`.

31.2.1 Race Conditions

A **race condition** occurs when multiple threads access shared mutable state and the program's correctness depends on the timing or interleaving of their execution.

- Caused by unsynchronized access to shared data.
- Leads to inconsistent or incorrect program state.

```java
class Counter {
    int count = 0;
    void increment() {
        count++;
    }
}
```

If multiple threads invoke `increment()` concurrently, increments may be lost because the operation is not atomic.

31.2.2 Deadlock

A **deadlock** occurs when two or more threads are permanently blocked, each waiting for a resource held by another thread.

- Typically caused by circular lock dependencies.
- No thread involved can make progress.

```java
synchronized (lockA) {
    synchronized (lockB) {
    }
}
```

If another thread acquires `lockB` first and then waits for `lockA`, a deadlock may occur.

> **Note**
>
> Real-world deadlocks typically involve multiple locks and order inversion.

31.2.3 Starvation

Starvation happens when a thread is indefinitely denied access to resources, even though those resources are available.

- Often caused by unfair locking or scheduling policies.
- Thread remains runnable but never executes.

```java
ReentrantLock lock = new ReentrantLock(false); // unfair lock
```

Threads may repeatedly acquire the lock while others wait indefinitely.

31.2.4 Livelock

In a **livelock**, threads are not blocked but continuously react to each other in a way that prevents progress.

- Threads remain active but ineffective.

- Often caused by aggressive retry or avoidance logic.

```
while (!tryLock()) {
    Thread.sleep(10);
}
```

Both threads may repeatedly retry, preventing forward progress.

31.3 From Threads to Tasks

The Concurrency API shifts the programming model from managing **threads** directly to submitting **tasks**.

A **task** represents a logical unit of work independent of the thread that executes it.

- **Runnable**: Represents a task that does not return a result.
- **Callable**: Represents a task that returns a result and may throw checked exceptions.

```
Runnable task = () -> System.out.println("Runnable task");
Callable<Integer> callable = () -> 42;
```

This abstraction allows tasks to be reused, scheduled flexibly, and executed by different execution strategies.

31.4 Executor Framework

The **Executor Framework** is the core of the Concurrency API.

It manages thread creation, reuse, and task execution behind a simple interface.

- **Executor**: Basic interface for executing tasks.
- **ExecutorService**: Extends Executor with lifecycle control and result handling.
- **ScheduledExecutorService**: Supports delayed and periodic task execution.

```
ExecutorService executor = Executors.newFixedThreadPool(2);
executor.execute(() -> System.out.println("Task 1"));
executor.execute(() -> System.out.println("Task 2"));
executor.shutdown();
```

31.4.1 Submitting Tasks and Futures

Tasks submitted using `execute()` return `void: it is a “fire-and-forget” method which does not give back any information about the result of the task.

Tasks submitted using `submit()` return a **Future**, which represents the result of an asynchronous computation.

Both methods are used to submit work for asynchronous execution.

```
Future<Integer> future = executor.submit(() -> 10 + 20);
Integer result = future.get();
```

Method	Description
void **execute(Runnable task)**	Executes a task asynchronously with no return value and no `Future`.
Future<?> **submit(Runnable task)**	Executes a task asynchronously; no result is produced (`Future.get()` returns `null`).
Future **submit(Callable task)**	Executes a task asynchronously and returns a result of type `T`.
List<Future> **invokeAll(Collection<? extends Callable> tasks)**	Executes all tasks and returns a `Future` for each, after all complete.
T **invokeAny(Collection<? extends Callable> tasks)**	Executes tasks and returns the result of one that completes successfully; others are cancelled.

Method	Description
boolean **isDone()**	Returns `true` if the task has completed (normally, exceptionally, or via cancellation).
boolean **isCancelled()**	Returns `true` if the task was cancelled before normal completion.
boolean **cancel(boolean mayInterruptIfRunning)**	Attempts to cancel execution. If `true`, interrupts the running thread if possible.
T **get()**	Blocks until completion and returns the result, or throws an exception if failed or cancelled.
T **get(long timeout, TimeUnit unit)**	Blocks up to the given timeout and returns the result, or throws `TimeoutException` if not completed.

> **Warning**
>
> `execute()` will drop exceptions silently unless handled inside the task.

31.4.2 Callable vs Runnable

Both interfaces represent tasks, but with different capabilities.

- `Runnable`: No return value, cannot throw checked exceptions.
- `Callable`: Returns a value and supports checked exceptions.

```
Callable<String> c = () -> "done";
Runnable r = () -> System.out.println("done");
```

For result-oriented asynchronous computation, `Callable` is generally preferred.

31.5 Thread Pools and Scheduling

Executors manage **thread pools**, which reuse a fixed or dynamic number of threads to execute tasks efficiently.

- **Fixed thread pool**: Limits concurrency to a fixed number of threads.
- **Cached thread pool**: Dynamically grows and shrinks based on demand: creates new threads as needed but reuses available threads.
- **Single-thread executor**: Ensures sequential task execution.
- **Scheduled executor**: Supports delayed and periodic tasks.

Factory Method	Return Type	Description
`Executors.newFixedThreadPool(int nThreads)`	ExecutorService	Creates a thread pool with a fixed number of threads.
`Executors.newFixedThreadPool(int nThreads, ThreadFactory threadFactory)`	ExecutorService	Same as newFixedThreadPool but with a custom ThreadFactory.
`Executors.newSingleThreadExecutor()`	ExecutorService	Creates a single-worker thread pool that executes tasks sequentially.
`Executors.newSingleThreadExecutor(ThreadFactory threadFactory)`	ExecutorService	Single-thread executor with a custom ThreadFactory.
`Executors.newCachedThreadPool()`	ExecutorService	Creates a thread pool that creates new threads as needed and reuses idle ones.
`Executors.newCachedThreadPool(ThreadFactory threadFactory)`	ExecutorService	Cached thread pool with a custom ThreadFactory.
`Executors.newSingleThreadScheduledExecutor()`	ScheduledExecutorService	Creates a single-thread scheduled executor.
`Executors.newSingleThreadScheduledExecutor(ThreadFactory threadFactory)`	ScheduledExecutorService	Single-thread scheduled executor with a custom ThreadFactory.
`Executors.newScheduledThreadPool(int corePoolSize)`	ScheduledExecutorService	Creates a scheduled thread pool with the given core size.
`Executors.newScheduledThreadPool(int corePoolSize, ThreadFactory threadFactory)`	ScheduledExecutorService	Scheduled thread pool with a custom ThreadFactory.
`Executors.newWorkStealingPool()`	ExecutorService	Creates a work-stealing pool using available processors as parallelism level.
`Executors.newWorkStealingPool(int parallelism)`	ExecutorService	Creates a work-stealing pool with the specified parallelism level.
`Executors.newThreadPerTaskExecutor(ThreadFactory threadFactory)`	ExecutorService	Creates an executor that starts a new thread for each task.
`Executors.newVirtualThreadPerTaskExecutor()`	ExecutorService	Creates an executor that starts a new virtual thread for each task.

`Task scheduling`: tasks submitted to an executor are enqueued and picked by pool threads; the execution order depends on the executor implementation, queue policy, and thread availability. For scheduled executors, tasks are ordered by trigger time; periodic tasks are re-queued after each run.

```
ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(1);

scheduler.schedule(
    () -> System.out.println("Delayed"),
    2, TimeUnit.SECONDS);
```

Method	Description
ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit)	Schedules a one-shot action that becomes enabled after the given delay.
ScheduledFuture schedule(Callable callable, long delay, TimeUnit unit)	Schedules a one-shot task that returns a value after the given delay.
ScheduledFuture<?> scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)	Schedules periodic execution at a fixed rate: each execution is scheduled relative to the initial start time; if a run is delayed, subsequent runs may attempt to "catch up".
ScheduledFuture<?> scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit)	Schedules periodic execution with fixed delay: each execution is scheduled relative to the completion time of the previous run; no catch-up behavior.

Important

Never create threads manually in a loop: use pools or virtual threads instead.

31.6 Executor Lifecycle and Termination

Executors must be shut down explicitly to release resources and allow JVM termination.

- **shutdown()**: Initiates orderly shutdown: completes waiting tasks but doesn't accept additionals ones.
- **close()**: (Java 19+) calls shutdown() and waits for tasks to finish, behaving like try-with-resources support for ExecutorService.
- **shutdownNow()**: Attempts immediate shutdown and interrupts running tasks.
- **awaitTermination()**: Waits for completion or timeout.

```
executor.shutdown();
executor.awaitTermination(5, TimeUnit.SECONDS);
```

31.7 Thread Safety Strategies

The Concurrency API provides multiple complementary strategies for achieving thread safety.

31.7.1 Synchronization

Synchronization enforces mutual exclusion and memory visibility by using an intrinsic lock (monitor) associated with an object or a class.

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

When a thread enters a synchronized method:

- It **acquires the intrinsic lock** of the target object (`this` for instance methods).
- Only one thread at a time can hold the same lock, preventing concurrent execution.
- When the method exits, the lock is released automatically.

Synchronization establishes a **happens-before relationship** in the Java Memory Model:

- All writes made inside the synchronized region are flushed to main memory when the lock is released.
- A thread acquiring the same lock later is guaranteed to see those updates.

The synchronized keyword can be applied to:

- **Instance methods** (lock on `this`)
- **Static methods** (lock on the `Class` object)
- **Blocks** (lock on a specific object, allowing finer-grained control)

> **Important**
>
> Synchronization is simple but may hurt scalability under contention.

31.7.2 Atomic Variables

Atomic classes provide lock-free, thread-safe operations implemented using low-level CPU primitives such as Compare-And-Swap (CAS).

```
AtomicInteger count = new AtomicInteger();
count.incrementAndGet();
```

31.7.2.1 Atomic classes

Atomic Class	Description
AtomicBoolean	Atomically updates and reads a `boolean` value.
AtomicInteger	Atomically updates and reads an `int` value.
AtomicLong	Atomically updates and reads a `long` value.
AtomicReference	Atomically updates and reads an object reference.
AtomicIntegerArray	Provides atomic operations on elements of an `int` array.
AtomicLongArray	Provides atomic operations on elements of a `long` array.
AtomicReferenceArray	Provides atomic operations on elements of a reference array.
AtomicStampedReference	Atomically updates a reference with an integer stamp to avoid ABA problems.
AtomicMarkableReference	Atomically updates a reference with a boolean mark.

31.7.2.2 Atomic methods

Method	Description
get()	Returns the current value with volatile-read semantics.
set(value)	Sets the value with volatile-write semantics.
lazySet(value)	Eventually sets the value with weaker ordering guarantees.
compareAndSet(expect, update)	Atomically sets the value if the current value equals the expected value.
getAndSet(value)	Atomically sets the value and returns the previous value.
incrementAndGet()	Atomically increments the value and returns the updated result.
getAndIncrement()	Atomically increments the value and returns the previous result.
decrementAndGet()	Atomically decrements the value and returns the updated result.
getAndDecrement()	Atomically decrements the value and returns the previous result.
addAndGet(delta)	Atomically adds the given delta and returns the updated result.
getAndAdd(delta)	Atomically adds the given delta and returns the previous result.

Atomic variables

:

- Perform single operations **atomically**
- Provide **memory visibility guarantees** similar to `volatile`
- Avoid thread blocking, making them highly scalable under contention

However, atomic variables only guarantee atomicity for **individual operations**.

Composing multiple operations still requires external synchronization.

Atomic variables are typically used for

:

- Counters and sequence generators
- Flags and state indicators
- High-throughput, low-latency updates

31.7.3 Lock Framework

The `java.util.concurrent.locks` package provides explicit locking mechanisms that offer greater flexibility and control than synchronized.

```
ReentrantLock lock = new ReentrantLock();
lock.lock();
try {
    // critical section
} finally {
    lock.unlock();
}
```

Key characteristics of the Lock framework:

- Locks must be explicitly acquired and released
- Lock acquisition can be interruptible or time-bounded
- Locks may be configured with fairness policies (parameter) when ordering is required (when you need to control the order in which threads run)
- Multiple Condition objects can be associated with a single lock

31.7.3.1 Lock implementations

Lock Implementation	Description
Lock	Core interface defining explicit lock operations.
ReentrantLock	Reentrant mutual exclusion lock with optional fairness policy.
ReadWriteLock	Interface defining separate read and write locks.
ReentrantReadWriteLock	Provides separate reentrant read and write locks to improve read scalability.
StampedLock	Lock supporting optimistic, read, and write locking modes (non-reentrant).

> **Warning**
>
> Unlike other locks, StampedLock is **not reentrant** — re-acquiring it from the same thread causes deadlock.

31.7.3.2 Common Lock methods

Method	Description
lock()	Acquires the lock, blocking indefinitely until available.
unlock()	Releases the lock; must be called by the owning thread.
tryLock()	Attempts to acquire the lock immediately without blocking: returns boolean indicating if lock has been succesfully acquired
tryLock(long, TimeUnit)	Attempts to acquire the lock within the given timeout.
lockInterruptibly()	Acquires the lock unless the thread is interrupted.
newCondition()	Creates a `Condition` instance for fine-grained thread coordination.

Unlike synchronized, locks do not release automatically, making proper try/finally usage essential to avoid deadlocks.

31.7.4 Coordination Utilities

Coordination utilities allow threads to coordinate execution phases without protecting shared data via mutual exclusion.

Other coordination primitives include: - `CountDownLatch` - `Semaphore` - `Phaser`

```java
import java.util.concurrent.CyclicBarrier;

public class BarrierExample {

    private static final int THREAD_COUNT = 3;

    public static void main(String[] args) {

        CyclicBarrier barrier = new CyclicBarrier(
            THREAD_COUNT,
            () -> System.out.println("All threads reached the barrier. Proceeding...")
        );

        Runnable task = () -> {
            String name = Thread.currentThread().getName();
            try {
                System.out.println(name + " performing initial work");
                Thread.sleep((long) (Math.random() * 2000));

                // Wait for other threads
                System.out.println(name + " waiting at barrier");
                barrier.await();

                // Executed only after all threads reach the barrier
                System.out.println(name + " performing next phase");

            } catch (Exception e) {
                e.printStackTrace();
            }
        };

        for (int i = 1; i <= THREAD_COUNT; i++) {
            new Thread(task, "Worker-" + i).start();
        }
    }
}
```

Sample Output:

```
Worker-1 performing initial work
Worker-2 performing initial work
Worker-3 performing initial work
Worker-3 waiting at barrier
Worker-1 waiting at barrier
Worker-2 waiting at barrier
All threads reached the barrier. Proceeding...
Worker-3 performing next phase
Worker-1 performing next phase
Worker-2 performing next phase
```

A `CyclicBarrier`:

- Blocks threads until a predefined number of threads reach the barrier
- Releases all waiting threads simultaneously once the barrier is tripped
- Can be reused for multiple coordination cycles

These utilities focus on execution ordering and synchronization, not data protection.

31.8 Concurrent Collections

Concurrent collections are **thread-safe data structures** designed to support **high levels of concurrency** without requiring external synchronization.

Unlike synchronized wrappers (e.g. `Collections.synchronizedMap`), concurrent collections: - Use **fine-grained locking** or **lock-free techniques** - Allow multiple threads to access and modify the collection simultaneously - Scale better under contention

Common examples include:

- **ConcurrentHashMap**
  A high-performance concurrent map that allows concurrent reads and updates by partitioning internal state and minimizing lock contention.

- **CopyOnWriteArrayList**
  A thread-safe list optimized for scenarios with **many reads and few writes**. Write operations create a new internal array, allowing reads to proceed without locking.
- **BlockingQueue**
  A queue designed for **producer-consumer patterns**, where threads can block while waiting for elements or available capacity.

```
BlockingQueue<String> queue = new LinkedBlockingQueue<>();
queue.put("item");    // blocks if the queue is full
queue.take();         // blocks if the queue is empty
```

Blocking queues handle synchronization internally, simplifying coordination between producer and consumer threads.

> **Caution**
>
> CopyOnWrite collections are memory-expensive; avoid in write-heavy workloads.

31.9 Parallel Streams

`Parallel streams` provide **declarative data parallelism**, allowing stream operations to be executed concurrently across multiple threads with minimal code changes.

Key characteristics: - Activated via `parallelStream()` or `stream().parallel()` - Internally executed using the **common ForkJoinPool** - Automatically splits data into chunks processed in parallel

Parallel streams work best when: - Operations are **CPU-bound** - Functions are **stateless and non-blocking** - The data source is large enough to amortize parallelization overhead

```
list.parallelStream()
    .map(x -> x * x)
    .forEach(System.out::println);
```

Because execution order is not guaranteed, parallel streams should avoid: - Shared mutable state - Blocking I/O - Order-dependent side effects

> **Note**
>
> Use `forEachOrdered()` if deterministic output is required.

31.10 Relation to Virtual Threads

In Java 21, the `Executor framework` integrates seamlessly with **virtual threads**, enabling massive concurrency with minimal resource usage.

```
ExecutorService executor =
Executors.newVirtualThreadPerTaskExecutor();

executor.submit(() -> blockingIO());
executor.close();
```

This allows blocking code to scale efficiently without redesigning APIs.

31.11 Summary

- The `Java Concurrency API` provides a robust, scalable, and safer alternative to manual thread management.
- By abstracting execution, coordinating tasks, and offering thread-safe utilities, it enables developers to build concurrent systems that are both performant and maintainable.

- Choose the right tool: synchronized → locks → atomics → executors → virtual threads.

◀ 30. Java Threads – Fundamentals and Execution Model | ▲ Index | 32. Files and Paths Fundamentals ▶

Module 08

Java I/O and NIO

32. Files and Paths Fundamentals

Table of Contents

- 32.1 Conceptual Model Filesystem, Files, Directories, Links and IO-Targets
- 32.2 Filesystem – The Global Abstraction
- 32.3 Path – Locating an Entry in a Filesystem
- 32.4 Files – Persistent Data Containers
- 32.5 Directories – Structural Containers
- 32.6 Links – Indirection Mechanisms
    - 32.6.1 Hard Links
    - 32.6.2 Symbolic Soft Links
- 32.7 Other Filesystem Entry Types
- 32.8 How Java IO Interacts with These Concepts
- 32.9 Core Conceptual Pitfalls
- 32.10 Why Path and Files Exist IO-Context
- 32.11 Is File legacy-apis both a path and a file-operations api
    - 32.11.1 What File Really Is
    - 32.11.2 Path-like Responsibilities
    - 32.11.3 Filesystem Operation Responsibilities
    - 32.11.4 What File Is NOT
    - 32.11.5 The Old dual role
    - 32.11.6 How NIO Fixed This
    - 32.11.7 Summary
- 32.12 Path Is a Description Not a Resource
- 32.13 Absolute vs Relative Paths
    - 32.13.1 Absolute Paths
    - 32.13.2 Relative Paths
- 32.14 Filesystem Awareness and Separators
    - 32.14.1 FileSystem
    - 32.14.2 Path Separators
- 32.15 What Files Actually Do and What They Don't
    - 32.15.1 Files DO
    - 32.15.2 Files DO NOT
- 32.16 Error Handling Philosophy Old-vs-NIO
- 32.17 Common Misconceptions

This section focuses on `Path`, `File`, `Files`, and related classes, explaining why they exist, what problems they solve, and which are the differences between legacy `java.io APIs` and `NIO v.2` (new I/O APIs), with special attention to filesystem semantics, path resolution, and common misconceptions.

32.1 Conceptual Model: Filesystem, Files, Directories, Links, and I/O Targets

Before understanding Java I/O APIs, it is essential to understand what they interact with.

Java I/O does not operate in a vacuum: it interacts with filesystem abstractions provided by the operating system.

This section defines those concepts independently of Java, then explains how Java I/O maps onto them and what problems are being solved.

32.2 Filesystem – The Global Abstraction

A `filesystem` is a structured mechanism provided by an operating system to organize, store, retrieve, and manage data on persistent storage devices.

At a conceptual level, a filesystem solves several fundamental problems:

- Persistent storage beyond program execution
- Hierarchical organization of data
- Naming and locating data
- Access control and permissions
- Concurrency and consistency guarantees

In Java NIO, a filesystem is represented by the `FileSystem` abstraction, typically obtained via `FileSystems.getDefault()` for the OS filesystem.

Aspect	Meaning
Persistence	Data survives JVM termination
Scope	OS-managed, not JVM-managed
Multiplicity	Multiple filesystems may exist
Examples	Disk FS, ZIP FS, in-memory FS

> **Note**
>
> Java does not implement filesystems; it adapts to filesystem implementations provided by the OS or custom providers.

32.3 Path – Locating an Entry in a Filesystem

A `path` is a logical locator, not a resource.

It describes where something would be in a filesystem, not what it is or whether it exists.

A `path` solves the problem of `addressing`:

- Identifies a location
- Is interpreted within a specific filesystem
- May or may not correspond to an existing entry

Property	Path
Existence-aware	No
Type-aware	No
Immutable	Yes
OS resource	No

> **Note**
>
> In Java, `Path` represents potential filesystem entries, not actual ones.

32.4 Files – Persistent Data Containers

A `file` is a filesystem entry whose primary role is to store data.

The filesystem treats files as opaque byte sequences.

Problems solved by files:

- Durable storage of information
- Sequential and random access to data
- Sharing data between processes

From the filesystem perspective, a file has:

- Content (bytes)
- Metadata (size, timestamps, permissions)
- A location (path)

Aspect	Description
Content	Byte-oriented
Interpretation	Application-defined
Lifetime	Independent of processes
Java access	Streams, channels, Files methods

> **Note**
>
> `Text` vs `binary` is not a filesystem concept; it is an application-level interpretation.

32.5 Directories – Structural Containers

A `directory (or folder)` is a filesystem entry whose purpose is to organize other entries.

`Directories` solve the problem of scalability and organization:

- Group related entries
- Enable hierarchical naming
- Support efficient lookup

Aspect	Directory
Stores data	No (stores references)
Contains	Files, directories, links
Read/write	Structural, not content-based
Java access	Files.list, Files.walk

> **Note**
>
> A directory is not a file with content, even if both share common metadata.

32.6 Links – Indirection Mechanisms

A `link` is a filesystem entry that refers to another entry.

Links solve the problem of indirection and reuse.

32.6.1 Hard Links

A `hard link` is an additional name for the same underlying data.

- Multiple paths point to the same file data
- Deletion occurs only when all links are removed

32.6.2 Symbolic (Soft) Links

A `symbolic link` is a special file containing a path to another entry:

- May point to non-existing targets
- Resolved at access time

Link Type	Refers To	Can Dangle	Java Handling
Hard	Data	No	Transparent
Symbolic	Path	Yes	Explicit control

> **Note**
>
> Java NIO exposes link behavior explicitly via `LinkOption`.
>
> In many common filesystems, Java code cannot create hard links in a fully portable way, while symbolic links are directly supported via `Files.createSymbolicLink(...)` (where permitted by the OS / permissions).

32.7 Other Filesystem Entry Types

Some filesystem entries are not data containers but interaction endpoints.

Type	Purpose
Device file	Interface to hardware
FIFO / Pipe	Inter-process communication
Socket file	Network communication

> **Note**
>
> Java I/O may interact with these entries, but behavior is platform-dependent.

32.8 How Java I/O Interacts with These Concepts

Java I/O APIs operate at different abstraction layers:

- `Path` / `File` (legacy API) → describes a filesystem entry
- `File` (legacy API) / `Files` → queries or modifies filesystem state
- `Streams` / `Channels` → move bytes or characters

Java API	Role
`Path`	Addressing
`File` (legacy APIs)	Addressing / filesystem operations
`Files`	Filesystem operations
`InputStream` / `Reader`	Reading data
`OutputStream` / `Writer`	Writing data
`Channel` / `SeekableByteChannel`	Advanced / random access

> **Note**
>
> No Java API "is" a file; APIs mediate access to filesystem-managed resources.

32.9 Core Conceptual Pitfalls

- Confusing paths with files
- Assuming paths imply existence
- Assuming directories store file data
- Assuming links are always resolved automatically

> **Note**
>
> Always separate location, structure, and data flow when reasoning about I/O.

32.10 Why Path and Files Exist (I/O Context)

Classic `java.io` mixed three different concerns into poorly separated APIs:

- Path representation (where is the resource?)
- Filesystem interaction (does it exist? what type?)
- Data access (reading/writing bytes or characters)

The NIO.2 design (Java 7+) deliberately separates these concerns:

- `Path` → describes a location
- `Files` → performs filesystem operations
- `Streams / Channels` → move data

> **Note**
>
> A `Path` never opens a file and never touches the disk by itself.

32.11 Is `File` (legacy APIs) both a `path` and a `file-operations` API?

Yes — in the old I/O API, `java.io.File` confusingly plays two roles at the same time, and this design is exactly one of the reasons `java.nio.file` was introduced.

Short Answer

- `File` represents a filesystem path
- `File` also exposes filesystem operations

- It does **not** represent an open file, nor file contents

> **Note**
>
> This mixing of responsibilities is considered a design flaw in hindsight.

32.11.1 What `File` Really Is

Conceptually, `File` is closer to what we now call a `Path`, but with added operational methods.

Aspect	java.io.File
Represents a location	Yes
Opens a file	No
Reads / writes data	No
Queries filesystem	Yes
Modifies filesystem	Yes
Holds OS handle	No

> **Note**
>
> A `File` object can exist even if the file does not.

32.11.2 Path-like Responsibilities

`File` behaves like a path abstraction because it:

- Encapsulates a filesystem pathname (absolute or relative)
- Can be resolved against the working directory
- Can be converted to absolute or canonical form

Examples:

```
File f = new File("data.txt"); // relative path
File abs = f.getAbsoluteFile(); // absolute path
File canon = f.getCanonicalFile(); // normalized + resolved
```

32.11.3 Filesystem Operation Responsibilities

At the same time, `File` exposes methods that touch the filesystem:

- exists()
- isFile(), isDirectory()
- length()
- delete()
- mkdir(), mkdirs()
- list(), listFiles()

> **Note**
>
> Most of these methods return `boolean` instead of throwing `IOException`, which hides failure causes.

32.11.4 What `File` Is NOT

- Not an open file descriptor
- Not a stream
- Not a channel

- Not a container of file data

You must still use streams or readers/writers to access contents.

32.11.5 The Old dual role

The dual role of `File` caused several issues:

- Mixed concerns (path + operations)
- Poor error handling (boolean instead of exceptions)
- Weak support for links and multiple filesystems
- Platform-dependent behavior

32.11.6 How NIO Fixed This

NIO.2 explicitly separates responsibilities:

Responsibility	Old API	NIO API
`Path representation`	`File`	`Path`
`Filesystem operations`	`File`	`Files`
`Data access`	Streams	Streams / Channels

> **Note**
>
> This separation is one of the most important conceptual improvements in Java I/O.

32.11.7 Summary

- `File` represents a path AND performs filesystem operations
- It never reads or writes file contents
- It never opens a file
- `Path` + `Files` is the modern replacement

32.12 Path Is a Description, Not a Resource

A `Path` is a pure abstraction representing a sequence of name elements in a filesystem.

- It does NOT imply existence
- It does NOT imply accessibility
- It does NOT hold a file descriptor

This is fundamentally different from streams or channels.

Concept	Path	Stream / Channel
`Opens resource`	No	Yes
`Touches disk`	No	Yes
`Holds OS handle`	No	Yes
`Immutable`	Yes	No

> **Note**
>
> Creating a Path cannot throw `IOException` because no I/O happens.

32.13 Absolute vs Relative Paths

Understanding path resolution is essential.

32.13.1 Absolute Paths

An absolute path fully identifies a location from the filesystem root.

- Platform-dependent root
- Independent of JVM working directory

Platform	Example Absolute Path
Unix	`/home/user/file.txt`
Windows	`C:\Users\User\file.txt`

Important

- A path starting with a forward slash `(/)` (Unix-like) or with a drive letter such as `C:` (Windows) is **typically** considered an absolute path.
- The symbol `.` is a reference to the current directory while `..` is a reference to its parent directory. On Windows, a path like `\dir\file.txt` (without drive letter) is *rooted* on the current drive, not fully qualified with drive + path.

Example:

```
/dirA/dirB/../dirC/./content.txt

is equivalent to:

/dirA/dirC/content.txt

// in this example the symbols were redundant and unnecessary
```

32.13.2 Relative Paths

A relative path is resolved against the JVM current working directory.

- Depends on where JVM was launched
- Common source of bugs

Note

The working directory is typically available via `System.getProperty("user.dir")`.

Example:

```
dirB/dirC/content.txt
```

32.14 Filesystem Awareness and Separators

NIO introduces filesystem abstraction, which was mostly absent in java.io.

32.14.1 FileSystem

A `FileSystem` represents a concrete filesystem implementation.

- Default filesystem corresponds to OS filesystem
- Other filesystems possible (ZIP, memory, network)

> **Note**
>
> Paths are always associated with exactly ONE FileSystem.

32.14.2 Path Separators

Separators differ across platforms, but `Path` abstracts them.

Aspect	java.io.File	java.nio.file.Path
Separator	String-based	Filesystem-aware
Portability	Manual handling	Automatic
Comparison	Error-prone	Safer

> **Note**
>
> Hardcoding `"/"` or `"\\"` is discouraged; `Path` handles this automatically.

32.15 What Files Actually Do (and What They Don't)

The `Files` class performs real I/O operations.

32.15.1 Files DO

- Open files indirectly (via streams / channels returned by its methods)
- Create and delete filesystem entries
- Throw checked exceptions on failure
- Respect filesystem permissions

32.15.2 Files DO NOT

- Maintain open resources after method returns (except streams)
- Store file contents internally
- Guarantee atomicity unless specified
- Maintain a persistent handle to open files (streams/channels own the handle instead)

> **Note**
>
> Methods returning streams (e.g. `Files.lines()`) DO keep the file open until the stream is closed.

32.16 Error Handling Philosophy: Old vs NIO

A major conceptual difference lies in error reporting.

Aspect	`java.io.File`	`java.nio.file.Files`
Error signaling	boolean / `null`	`IOException`
Diagnostics	Poor	Rich
Race awareness	Weak	Improved
Preference	Discouraged	Preferred

32.17 Common Misconceptions

- “Path represents a file” → false
- “normalize checks existence” → false
- “Files.readAllLines streams data” → false
- “Relative paths are portable” → false
- “Creating a Path may fail due to permissions” → false

> **Note**
>
> Many NIO methods that sound “safe” are purely syntactic (like `normalize` or `resolve`): they do **not** touch the filesystem and cannot detect missing files.

◀ 31. Java Concurrency APIs | ▲ Index | 33. Files and Paths APIs ▶

33. Files and Paths APIs

Table of Contents

- 33.1 Legacy File and NIO Path Creation-and-Conversion
  - 33.1.1 Creating a File Legacy
  - 33.1.2 Creating a Path NIO-v2
  - 33.1.3 Absolute vs Relative What Relative Means
  - 33.1.4 Joining–Building-Paths
    - 33.1.4.1 resolve
    - 33.1.4.2 relativize
  - 33.1.5 Converting Between File and Path
  - 33.1.6 URI Conversion When-Needed
  - 33.1.7 Canonical vs Absolute vs Normalized Core-Differences
    - 33.1.7.1 normalize
  - 33.1.8 Quick Comparison Table Creation–Conversion
- 33.2 Managing Files and Directories Create-Copy-Move-Replace-Compare-Delete
  - 33.2.1 Mental Model Path-Locator-vs-Operations
  - 33.2.2 Creating Files and Directories
    - 33.2.2.1 Create a File
    - 33.2.2.2 Create Directories
  - 33.2.3 Copying Files and Directories
    - 33.2.3.1 Copy a File NIO
    - 33.2.3.2 Manual Copy Legacy-Stream-Based
  - 33.2.4 Moving–Renaming-and-Replacing
    - 33.2.4.1 Legacy Rename Common-Pitfall
    - 33.2.4.2 NIO Move Preferred
  - 33.2.5 Comparing Paths and Files
    - 33.2.5.1 Equality-vs-Same-File
  - 33.2.6 Deleting Files and Directories
    - 33.2.6.1 Legacy Delete
    - 33.2.6.2 NIO Delete and Delete-If-Exists
  - 33.2.7 Recursively Copying–Deleting-Directory-Trees NIO-Pattern
  - 33.2.8 Summary Checklist

This section focuses on how to create filesystem locators using the legacy `java.io.File` API and the modern `java.nio.file.Path` API: how to convert between them and understanding overloads, defaults, and common pitfalls.

33.1 Legacy `File` and NIO `Path`: Creation and Conversion

33.1.1 Creating a `File` (Legacy)

A `File` instance represents a filesystem pathname (absolute or relative).

Creating one does **not** access the filesystem and does not throw `IOException`.

Core constructors (most common):

- `new File(String pathname)`
- `new File(String parent, String child)`
- `new File(File parent, String child)`
- `new File(URI uri)` (typically `file:...`)

```
import java.io.File;
import java.net.URI;

File f1 = new File("data.txt"); // relative
File f2 = new File("/tmp", "data.txt"); // parent + child
File f3 = new File(new File("/tmp"), "data.txt");

File f4 = new File(URI.create("file:///tmp/data.txt"));
```

> **Note**
>
> - `new File(...)` never opens the file.
> - Existence/permissions are checked only when you call methods like `exists()`, `length()`, or when you open a stream/channel.

33.1.2 Creating a `Path` (NIO v.2)

A `Path` is also just a locator.

Like `File`, creating a `Path` does not access the filesystem.

Core factories:

- `Path.of(String first, String... more)` (Java 11+)
- `Paths.get(String first, String... more)` (older style; still valid)
- `Path.of(URI uri)` (e.g., `file:///...`)

```
import java.net.URI;
import java.nio.file.Path;
import java.nio.file.Paths;

Path p1 = Path.of("data.txt"); // relative
Path p2 = Path.of("/tmp", "data.txt"); // parent + child

Path p3 = Paths.get("data.txt"); // legacy factory style
Path p4 = Path.of(URI.create("file:///tmp/data.txt"));
```

> **Note**
>
> - `Path.of(...)` and `Paths.get(...)` are equivalent for the default filesystem.
> - Prefer `Path.of` in modern code.

33.1.3 Absolute vs Relative: What “Relative” Means

Both `File` and `Path` can be created as relative paths.

Relative paths are resolved against the process working directory (typically `System.getProperty("user.dir")`).

```java
import java.io.File;
import java.nio.file.Path;

File rf = new File("data.txt");
Path rp = Path.of("data.txt");

System.out.println(rf.isAbsolute()); // false
System.out.println(rp.isAbsolute()); // false

System.out.println(rf.getAbsolutePath());
System.out.println(rp.toAbsolutePath());
```

> **Note**
>
> Relative paths are a common source of "works on my machine" bugs because `user.dir` depends on how/where the JVM was launched.

33.1.4 Joining / Building Paths

- Legacy `File` uses constructors (parent + child).
- NIO uses `resolve` and related methods.

Task	Legacy (File)	NIO (Path)
Join parent + child	`new File(parent, child)`	`parent.resolve(child)`
Join many segments	Repeated constructors	`Path.of(a, b, c)` or chained `resolve()`

```java
import java.io.File;
import java.nio.file.Path;

File f = new File("/tmp", "a.txt");

Path base = Path.of("/tmp");
Path p = base.resolve("a.txt"); // /tmp/a.txt
Path p2 = base.resolve("dir").resolve("a.txt"); // /tmp/dir/a.txt
```

33.1.4.1 `resolve()`

Combines paths in a filesystem-aware way.

- Relative paths are appended
- Absolute argument replaces base path

> **Note**
>
> `Path.resolve(...)` has a rule: if the argument is absolute, it returns the argument and discards the base (you cannot combine two absolute paths using `resolve`).

33.1.4.2 `relativize()`

`Path.relativize` computes a **relative path** from one path to another. The resulting path, when `resolved` against the source path, yields the target path.

In other words:

- It answers the question: "How do I go from path A to path B?"
- The result is always a **relative** path
- No filesystem access occurs

Fundamental Rules

`relativize` has strict preconditions. Violating them throws an exception.

Rule	Explanation
Both paths must be absolute	or both relative
Both paths must belong to the same filesystem	same provider
Root components must match	same root (on Windows, same drive)
Result is never absolute	always relative

> **Note**
>
> If one path is absolute and the other relative, `IllegalArgumentException` is thrown.

Simple Relative Example:

Both paths are relative, so relativization is allowed.

```
Path p1 = Path.of("docs/manual");
Path p2 = Path.of("docs/images/logo.png");

Path relative = p1.relativize(p2);
System.out.println(relative);
```

```
../images/logo.png
```

Interpretation: from `docs/manual`, go up one level, then into `images/logo.png`.

Absolute Paths Example:

Absolute paths work exactly the same way.

```
Path base = Path.of("/home/user/projects");
Path target = Path.of("/home/user/docs/readme.txt");

Path relative = base.relativize(target);
System.out.println(relative);
```

```
../docs/readme.txt
```

Using `resolve` to Verify the Result

A key property of `relativize` is this identity:

```
base.resolve(base.relativize(target)).equals(target)
```

```
Path base = Path.of("/a/b/c");
Path target = Path.of("/a/d/e");

Path r = base.relativize(target);
System.out.println(r); // ../../d/e
System.out.println(base.resolve(r)); // /a/d/e
```

Example: Mixing Absolute and Relative Paths (ERROR CASE)

This is one of the most common mistakes.

```
Path abs = Path.of("/a/b");
Path rel = Path.of("c/d");

abs.relativize(rel); // throws exception
```

```
Exception in thread "main" java.lang.IllegalArgumentException
```

> **Note**
>
> `relativize` does NOT attempt to convert paths to absolute automatically.

Example: Different Roots (Windows-Specific Trap)

On Windows, paths with different drive letters cannot be relativized.

```
Path p1 = Path.of("C:\\data\\a");
Path p2 = Path.of("D:\\data\\b");

p1.relativize(p2); // IllegalArgumentException
```

> **Note**
>
> On Unix-like systems, the root is always `/`, so this issue does not occur.

33.1.5 Converting Between `File` and `Path`

Conversion is straightforward and lossless for normal local filesystem paths.

Conversion	How
File → Path	`file.toPath()`
Path → File	`path.toFile()`

```
import java.io.File;
import java.nio.file.Path;

File f = new File("data.txt");
Path p = f.toPath();

File back = p.toFile();
```

> **Note**
>
> Conversion does not validate existence. It only converts representations.

33.1.6 URI Conversion (When Needed)

`URIs` are useful when paths must be represented in a standard, absolute form (e.g., interoperating with networked resources or configuration).

Both APIs support URI conversion.

Direction	Legacy (File)	NIO (Path)
From URI	`new File(uri)`	`Path.of(uri)`
To URI	`file.toURI()`	`path.toUri()`

```
import java.io.File;
import java.net.URI;
import java.nio.file.Path;

File f = new File("/tmp/data.txt");
URI u1 = f.toURI();

Path p = Path.of("/tmp/data.txt");
URI u2 = p.toUri();

Path pFromUri = Path.of(u2);
File fFromUri = new File(u1);
```

> **Note**
>
> `new File(URI)` requires a `file:` URI and throws `IllegalArgumentException` if the URI is not hierarchical or not a file URI.

33.1.7 Canonical vs Absolute vs Normalized (Core Differences)

These terms are often mixed up. They are not the same.

Concept	Legacy (File)	NIO (Path)	Touches filesystem
Absolute	`getAbsoluteFile()`	`toAbsolutePath()`	No
Normalized	(no pure normalize, use canonical)*	`normalize()`	`normalize()` : No
Canonical / Real	`getCanonicalFile()`	`toRealPath()`	Yes

> **Note**
>
> `File.getCanonicalFile()` and `Path.toRealPath()` may resolve symlinks and require the path to exist, so they can throw `IOException`.
>
> File does not provide a method for purely syntactic normalization: historically many developers used getCanonicalFile(), but this accesses the filesystem and can fail.

```java
import java.io.File;
import java.io.IOException;
import java.nio.file.Path;

File f = new File("a/../data.txt");
System.out.println(f.getAbsolutePath()); // absolute, may still contain ".."

try {
    System.out.println(f.getCanonicalPath()); // resolves "..", may touch filesystem
} catch (IOException e) {
    System.out.println("Canonical failed: " + e.getMessage());
}

Path p = Path.of("a/../data.txt");
System.out.println(p.toAbsolutePath()); // absolute, may still contain ".."
System.out.println(p.normalize()); // purely syntactic

try {
    System.out.println(p.toRealPath()); // resolves symlinks, requires existence
} catch (IOException e) {
    System.out.println("RealPath failed: " + e.getMessage());
}
```

33.1.7.1 `normalize()`

Removes **redundant** name elements like `.` and `..`.

- Purely syntactic
- Does not check if path exists

> **Note**
>
> `normalize()` is purely syntactic, does not check existence, and can produce invalid paths if misused.

33.1.8 Quick Comparison Table (Creation + Conversion)

Need	Legacy (File)	NIO (Path)	Preferred today
Create from string	`new File("x")`	`Path.of("x")`	Path
Parent + child	`new File(p, c)`	`Path.of(p, c)` or `resolve()`	Path
Convert between APIs	`toPath()`	`toFile()`	Path-centric
Normalize	`getCanonicalFile()` (filesystem-based)	`normalize()` (syntactic only)	Path
Resolve symlinks	Canonical	`toRealPath()`	Path

33.2 Managing Files and Directories: Create, Copy, Move, Replace, Compare, Delete (Legacy vs NIO)

This section covers the operations you perform on filesystem entries (files/directories): creating, copying, moving/renaming, replacing, comparing, and deleting.

It contrasts legacy `java.io.File` (and related legacy helpers) with modern `java.nio.file` (NIO.2).

33.2.1 Mental Model: "Path/Locator" vs "Operations"

Both APIs use objects that represent a path, but operations differ:

- Legacy: `File` is both a path wrapper and an operations API (mixed responsibility)
- NIO: `Path` is the path; `Files` performs operations (separation of concerns)

Responsibility	Legacy	NIO
Path representation	`File`	`Path`
Filesystem operations	`File`	`Files`
Rich error reporting	Weak (booleans)	Strong (exceptions)

> **Note**
>
> legacy methods often return `boolean` (silent failure), while NIO throws `IOException` with cause.

33.2.2 Creating Files and Directories

Creating is where the old API is most awkward and the NIO API is most expressive.

Task	Legacy approach	NIO approach	Notes
Create empty file	open+close stream	`Files.createFile`	NIO fails if exists
Create one directory	`mkdir`	`Files.createDirectory`	Parent must exist
Create directories recursively	`mkdirs`	`Files.createDirectories`	Creates parents

33.2.2.1 Create a File

Legacy has no “create empty file” method, so you typically create a file by opening an output stream (side effect).

```
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;

File f = new File("created-legacy.txt");
try (FileOutputStream out = new FileOutputStream(f)) {
    // file is created (or truncated) as a side effect
}
```

NIO provides an explicit creation method.

```
import java.nio.file.Files;
import java.nio.file.Path;
import java.io.IOException;

Path p = Path.of("created-nio.txt");
Files.createFile(p);
```

> **Note**
>
> `Files.createFile` throws `FileAlreadyExistsException` if the entry exists.

33.2.2.2 Create Directories

```
import java.io.File;

File dir1 = new File("a/b");
boolean ok1 = dir1.mkdir(); // fails if parent "a" does not exist
boolean ok2 = dir1.mkdirs(); // creates parents
```

```
import java.nio.file.Files;
import java.nio.file.Path;
import java.io.IOException;

Path d = Path.of("a/b");
Files.createDirectory(d); // parent must exist
Files.createDirectories(d); // creates parents, ok if already exists
```

> **Note**
>
> Legacy `mkdir()/mkdirs()` return `false` on failure without telling why. NIO throws `IOException`.

33.2.3 Copying Files and Directories

Legacy copy is usually manual stream-copy (or external libs). NIO has a single, explicit operation.

Capability	Legacy	NIO
Copy file contents	Manual streams	`Files.copy`
Copy into existing target	Manual	`REPLACE_EXISTING` option
Copy directory tree	Manual recursion	Manual recursion (but better tools: `Files.walk` + `Files.copy`)

33.2.3.1 Copy a File (NIO)

```
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.StandardCopyOption;
import java.io.IOException;

Path src = Path.of("src.txt");
Path dst = Path.of("dst.txt");

Files.copy(src, dst); // fails if dst exists
Files.copy(src, dst, StandardCopyOption.REPLACE_EXISTING);
```

> **Note**
>
> `Files.copy` throws `FileAlreadyExistsException` if the target exists and you did not use `REPLACE_EXISTING`.

33.2.3.2 Manual Copy (Legacy, Stream-Based)

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

try (FileInputStream in = new FileInputStream("src.bin");
FileOutputStream out = new FileOutputStream("dst.bin")) {

    byte[] buf = new byte[8192];
    int n;
    while ((n = in.read(buf)) != -1) {
        out.write(buf, 0, n);
    }
}
```

> **Note**
>
> Remember `read(byte[])` returns the number of bytes read; you must write only that count, not the full buffer.

33.2.4 Moving / Renaming and Replacing

In both APIs, rename/move is "metadata-level" when possible, but can behave like copy+delete across filesystems. NIO makes this explicit via options.

Operation	Legacy	NIO
Rename/move	`File.renameTo`	`Files.move`
Replace existing	Unreliable	`REPLACE_EXISTING`
Atomic move	Not supported	`ATOMIC_MOVE` (if supported)

33.2.4.1 Legacy Rename (Common Pitfall)

```
import java.io.File;

File from = new File("old.txt");
File to = new File("new.txt");

boolean ok = from.renameTo(to); // may fail silently
System.out.println(ok);
```

> **Note**
>
> - `renameTo` is notoriously platform-dependent and returns only `boolean`.
> - It may fail because target exists, file is open, permissions, or cross-filesystem move.

33.2.4.2 NIO Move (Preferred)

```
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.StandardCopyOption;
import java.io.IOException;

Path from = Path.of("old.txt");
Path to = Path.of("new.txt");

Files.move(from, to); // fails if target exists
Files.move(from, to, StandardCopyOption.REPLACE_EXISTING);
```

> **Note**
>
> `Files.move` throws `FileAlreadyExistsException` when the target exists and `REPLACE_EXISTING` is not specified.

33.2.5 Comparing Paths and Files

Comparing locators can mean: string/path equality, normalized/canonical equality, or "same file on disk".

The APIs differ here significantly.

Comparison goal	Legacy	NIO
Same path text	`File.equals`	`Path.equals`
Normalize path	`getCanonicalFile`	`normalize`
Same file/resource on disk	weak (canonical heuristic)	`Files.isSameFile`

33.2.5.1 Equality vs Same File

Two different path strings can refer to the same file.

```
import java.nio.file.Files;
import java.nio.file.Path;
import java.io.IOException;

Path p1 = Path.of("a/../data.txt");
Path p2 = Path.of("data.txt");

System.out.println(p1.equals(p2)); // false (different path text)
System.out.println(p1.normalize().equals(p2.normalize())); // might still be false if relative

try {
    System.out.println(Files.isSameFile(p1, p2)); // may be true, may throw if not accessible
} catch (IOException e) {
    System.out.println("isSameFile failed: " + e.getMessage());
}
```

> **Note**
>
> `Files.isSameFile` may access the filesystem and can throw `IOException` (permission issues, missing files, etc.).

33.2.6 Deleting Files and Directories

Deletion is simple in concept but has important edge cases: non-empty directories, missing targets, and error reporting differences.

Task	Legacy	NIO	Behavior if missing
Delete file/dir	`File.delete`	`Files.delete`	Legacy false, NIO exception
Delete if exists	No direct (check+delete)	`Files.deleteIfExists`	returns boolean
Delete non-empty dir	Manual recursion	Manual recursion (walk)	Both require recursion

33.2.6.1 Legacy Delete

```
import java.io.File;

File f = new File("x.txt");
boolean ok = f.delete(); // false if not deleted
System.out.println(ok);
```

> **Note**
>
> Legacy `delete()` fails (returns false) for a non-empty directory and often provides no reason.

33.2.6.2 NIO Delete and Delete-If-Exists

```
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.NoSuchFileException;
import java.nio.file.DirectoryNotEmptyException;
import java.io.IOException;

Path p = Path.of("x.txt");

try {
    Files.delete(p);
} catch (NoSuchFileException e) {
    System.out.println("Missing: " + e.getFile());
} catch (DirectoryNotEmptyException e) {
    System.out.println("Directory not empty: " + e.getFile());
} catch (IOException e) {
    System.out.println("Delete failed: " + e.getMessage());
}

boolean deleted = Files.deleteIfExists(p);
System.out.println(deleted);
```

> **Note**
>
> Certification tip: `Files.delete` throws `NoSuchFileException` if missing, while `deleteIfExists` returns `false`.

33.2.7 Recursively Copying / Deleting Directory Trees (NIO Pattern)

NIO doesn't provide a single "copyTree/deleteTree" method, but the standard approach uses `Files.walk` or `Files.walkFileTree`.

```java
import java.io.IOException;
import java.nio.file.*;
import java.nio.file.attribute.BasicFileAttributes;

Path root = Path.of("dirToDelete");

Files.walkFileTree(root, new SimpleFileVisitor<Path>() {
    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws IOException
        Files.delete(file);
        return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult postVisitDirectory(Path dir, IOException exc) throws IOException {
        if (exc != null) throw exc;
        Files.delete(dir);
        return FileVisitResult.CONTINUE;
    }
});
```

> **Note**
>
> Deleting a directory tree requires deleting files first, then directories (post-order). This is a common reasoning question.

33.2.8 Summary Checklist

- Prefer `Files.createFile/createDirectory/createDirectories` over legacy workarounds
- `File.renameTo` is unreliable; prefer `Files.move` with options
- `Files.copy/move` throw `FileAlreadyExistsException` unless `REPLACE_EXISTING` is used
- `Files.delete` throws; `Files.deleteIfExists` returns boolean
- `Files.isSameFile` can throw `IOException` and may touch the filesystem
- Non-empty directory deletion requires recursion (both APIs)

◀ 32. Files and Paths Fundamentals | ▲ Index | 34. Java I/O Streams ▶

34. Java I/O Streams

Table of Contents

- 34.1 What Is an IO Stream in Java
- 34.2 Byte Streams vs Character Streams
    - 34.2.1 Byte Streams
    - 34.2.2 Character Streams
    - 34.2.3 Summary Table
- 34.3 Low-Level vs High-Level Streams
    - 34.3.1 Low-Level Streams Node-Streams
    - 34.3.2 Common Low-Level Streams
    - 34.3.3 High-Level Streams Filter–Processing-Streams
    - 34.3.4 Common High-Level Streams
    - 34.3.5 Stream Chaining Rules and Common Errors
        - 34.3.5.1 Fundamental Chaining Rule
        - 34.3.5.2 Byte vs Character Stream Incompatibility
        - 34.3.5.3 Invalid Chaining Compile-Time-Error
        - 34.3.5.4 Bridging Byte Streams to Character Streams
        - 34.3.5.5 Correct Conversion Pattern
        - 34.3.5.6 Ordering Rules in Stream Chains
        - 34.3.5.7 Correct Logical Order
        - 34.3.5.8 Resource Management Rule
        - 34.3.5.9 Common Pitfalls
- 34.4 Core javaio Base Classes and Key Methods
    - 34.4.1 InputStream
        - 34.4.1.1 Key Methods
        - 34.4.1.2 Typical Usage Example
    - 34.4.2 OutputStream
        - 34.4.2.1 Key Methods
        - 34.4.2.2 Typical Usage Example
    - 34.4.3 Reader and Writer
        - 34.4.3.1 Charset Handling
- 34.5 Buffered Streams and Performance
    - 34.5.1 Why Buffering Matters
    - 34.5.2 How Unbuffered Reading Works
    - 34.5.3 How BufferedInputStream Works
    - 34.5.4 Buffered Output Example
    - 34.5.5 BufferedReader vs Reader
    - 34.5.6 BufferedWriter Example
- 34.6 java io vs java nio and java nio file
    - 34.6.1 Conceptual Differences
    - 34.6.2 java-nio Modern-File-IO
- 34.7 When to Use Which API
- 34.8 Common Traps and Tips

This chapter provides a detailed explanation of `Java I/O Streams`.

It covers classic **java.io** streams, contrasts them with **java.nio / java.nio.file**, and explains design principles, APIs, edge cases, and relevant distinctions.

34.1 What Is an I/O Stream in Java?

An `I/O Stream` represents a flow of data between a Java program and an external source or destination.

The data flows sequentially, like water in a pipe.

- A stream is not a data structure; it does not store data permanently
- Streams are unidirectional (input OR output)
- Streams abstract the underlying source (file, network, memory, device)
- Streams operate in a blocking, synchronous manner (classic I/O)

In Java, streams are organized around two major dimensions:

- `Direction`: Input vs Output
- `Data type`: Byte vs Character

34.2 Byte Streams vs Character Streams

Java distinguishes streams based on the unit of data they process.

34.2.1 Byte Streams

- Work with raw 8-bit bytes
- Used for binary data (images, audio, PDFs, ZIPs)
- Base classes: `InputStream` and `OutputStream`

34.2.2 Character Streams

- Work with 16-bit Unicode characters
- Handle character encoding automatically
- Base classes: `Reader` and `Writer`

34.2.3 Summary Table

Aspect	Byte Streams	Character Streams
`Unit of data`	byte (8 bits)	char (16 bits)
`Encoding handling`	None	Yes (Charset aware)
`Base classes`	InputStream / OutputStream	Reader / Writer
`Typical usage`	Binary files	Text files
`Focus`	Low-level I/O	Text processing

34.3 Low-Level vs High-Level Streams

Streams in `java.io` follow a decorator pattern. Streams are stacked to add functionality.

34.3.1 Low-Level Streams (Node Streams)

Low-level streams connect directly to a data source or sink.

- They know how to read/write bytes or characters

- They do NOT provide buffering, formatting, or object handling

34.3.2 Common Low-Level Streams

Stream Class	Purpose
`FileInputStream`	Read bytes from file
`FileOutputStream`	Write bytes to file
`FileReader`	Read characters from file
`FileWriter`	Write characters to file

- Example: Low-Level Byte Stream

```
try (InputStream in = new FileInputStream("data.bin")) {
    int b;
    while ((b = in.read()) != -1) {
        System.out.println(b);
    }
}
```

> **Note**
>
> Low-level streams are rarely used alone in real applications due to poor performance and limited features.

34.3.3 High-Level Streams (Filter / Processing Streams)

High-level streams wrap other streams to add functionality.

- Buffering
- Data type conversion
- Object serialization
- Primitive reading/writing

34.3.4 Common High-Level Streams

Stream Class	Adds Functionality
`BufferedInputStream`	Buffering
`BufferedReader`	Line-based reading
`DataInputStream`	Primitive types
`ObjectInputStream`	Object serialization
`PrintWriter`	Formatted text output

- Example: Stream Chaining

```
try (BufferedReader reader =
    new BufferedReader(
        new InputStreamReader(
            new FileInputStream("text.txt")))) {

    String line;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
}
```

34.3.5 Stream Chaining Rules and Common Errors

The previous example illustrates stream chaining, a core concept in `java.io` based on the decorator pattern.

Each stream wraps another stream, adding functionality while preserving a strict type hierarchy.

34.3.5.1 Fundamental Chaining Rule

A stream can only wrap another stream of a compatible abstraction level.

- Byte streams can only wrap byte streams
- Character streams can only wrap character streams
- High-level streams require an underlying low-level stream

> **Note**
>
> You cannot arbitrarily mix `InputStream` with `Reader` or `OutputStream` with `Writer`.

34.3.5.2 Byte vs Character Stream Incompatibility

A very common error is attempting to wrap a byte stream directly with a character-based class (or vice versa).

34.3.5.3 Invalid Chaining (Compile-Time Error)

```
BufferedReader reader =
    new BufferedReader(new FileInputStream("text.txt"));
```

> **Note**
>
> This fails because `BufferedReader` expects a `Reader`, not an `InputStream`.

34.3.5.4 Bridging Byte Streams to Character Streams

To convert between byte-based and character-based streams, Java provides bridge classes that perform explicit charset decoding/encoding.

- `InputStreamReader` converts bytes → characters
- `OutputStreamWriter` converts characters → bytes

34.3.5.5 Correct Conversion Pattern

```
BufferedReader reader =
    new BufferedReader(
        new InputStreamReader(new FileInputStream("text.txt")));
```

> **Note**
>
> The bridge handles character decoding using a charset (default or explicit).

34.3.5.6 Ordering Rules in Stream Chains

The order of wrapping is not arbitrary.

- Low-level stream must be innermost
- Bridges (if needed) come next
- Buffered or processing streams come last

34.3.5.7 Correct Logical Order

```
FileInputStream → InputStreamReader → BufferedReader
```

34.3.5.8 Resource Management Rule

Closing the outermost stream automatically closes all wrapped streams.

> **Note**
>
> This is why try-with-resources should reference only the highest-level stream.

34.3.5.9 Common Pitfalls

- Trying to buffer a stream of the wrong type
- Forgetting the bridge between byte and char streams
- Assuming `Reader` works with binary data
- Using default charset unintentionally
- Closing inner streams manually (risking double-close): `close()` on the outer wrapper is enough and recommended

---

34.4 Core `java.io` Base Classes and Key Methods

The `java.io` package is built around a small set of **abstract base classes**. Understanding these classes and their contracts is essential, because all concrete I/O classes build on them.

34.4.1 InputStream

Abstract base class for byte-oriented input. All input streams read raw bytes (8-bit values) from a source such as a file, network socket, or memory buffer.

34.4.1.1 Key Methods

Method	Description
int `read()`	Reads one byte (0–255); returns -1 at end of stream
int `read(byte[])`	Reads bytes into buffer; returns number of bytes read or -1
int `read(byte[], int, int)`	Reads up to length bytes into a buffer slice
int `available()`	Bytes readable without blocking (hint, not guarantee)
void `close()`	Releases the underlying resource

> **Note**
>
> The `read()` methods are blocking by default.
>
> They suspend the calling thread until data is available, end-of-stream is reached, or an I/O error occurs.

The `single-byte read()` method is primarily a low-level primitive.

In practice, reading one byte at a time is inefficient and should almost always be avoided in favor of buffered reads.

34.4.1.2 Typical Usage Example

```
try (InputStream in = new FileInputStream("data.bin")) {
    byte[] buffer = new byte[1024];
    int count;
    while ((count = in.read(buffer)) != -1) {
        // process buffer[0..count-1]
    }
}
```

34.4.2 OutputStream

Abstract base class for byte-oriented output.

It represents a destination where raw bytes can be written.

34.4.2.1 Key Methods

Method	Description
void `write(int b)`	Writes the low 8 bits of the integer
void `write(byte[])`	Writes an entire byte array
void `write(byte[], int, int)`	Writes a slice of a byte array
void `flush()`	Forces buffered data to be written
void `close()`	Flushes and releases the resource

> **Note**
>
> Calling `close()` implicitly calls `flush()`.
>
> Failing to flush or close an OutputStream may result in data loss.

34.4.2.2 Typical Usage Example

```
try (OutputStream out = new FileOutputStream("out.bin")) {
    out.write(new byte[] {1, 2, 3, 4});
    out.flush();
}
```

34.4.3 Reader and Writer

`Reader` and `Writer` are the `character-oriented` counterparts of InputStream and OutputStream.

They operate on 16-bit Unicode characters instead of raw bytes.

Class	Direction	Character-based	Encoding aware
`Reader`	Input	Yes	Yes
`Writer`	Output	Yes	Yes

Readers and Writers always involve a `charset`, either explicitly or implicitly.

This makes them the correct abstraction for text processing.

34.4.3.1 Charset Handling

```
Reader reader = new InputStreamReader(
    new FileInputStream("file.txt"),
    StandardCharsets.UTF_8
);
```

> **Note**
>
> `InputStreamReader` and `OutputStreamWriter` are bridge classes.
>
> They convert between `byte streams` and `character streams` using a `charset`.

34.5 Buffered Streams and Performance

`Buffered streams` wrap another stream and add an in-memory buffer.

Instead of interacting with the operating system on every read or write, data is accumulated in memory and transferred in larger chunks.

- `BufferedInputStream` / `BufferedOutputStream` for byte streams
- `BufferedReader` / `BufferedWriter` for character streams

Note

`Buffered streams` are `decorators` : they do not replace the underlying stream, they enhance it by adding buffering behavior.

34.5.1 Why Buffering Matters

Aspect	Unbuffered	Buffered
`System calls`	Frequent	Reduced
`Performance`	Poor	High
`Memory usage`	Minimal	Slightly higher

System calls are expensive operations.

Buffering minimizes them by grouping multiple logical reads or writes into fewer physical I/O operations.

34.5.2 How Unbuffered Reading Works

In an unbuffered stream, each call to read() may result in a native system call.

This is especially inefficient when reading large amounts of data.

```
try (InputStream in = new FileInputStream("data.bin")) {
    int b;
    while ((b = in.read()) != -1) {
        // each read() may trigger a system call
    }
}
```

Note

Reading byte-by-byte without buffering is almost always a performance anti-pattern.

34.5.3 How BufferedInputStream Works

`BufferedInputStream` internally reads a large block of bytes into a buffer.

Subsequent `read()` calls are served directly from memory until the buffer is empty.

```
try (InputStream in =
    new BufferedInputStream(new FileInputStream("data.bin"))) {
        int b;
        while ((b = in.read()) != -1) {
            // most reads are served from memory, not the OS
        }
}
```

Note

The program still calls `read()` repeatedly, but the operating system is accessed only when the internal buffer needs refilling.

34.5.4 Buffered Output Example

Buffered output accumulates data in memory and writes it in larger chunks.

The `flush()` operation forces the buffer to be written immediately.

```java
try (OutputStream out =
    new BufferedOutputStream(new FileOutputStream("out.bin"))) {
        for (int i = 0; i < 1_000; i++) {
            out.write(i);
        }
        out.flush(); // forces buffered data to disk
}
```

> **Note**
>
> `close()` automatically calls flush().
>
> Calling `flush()` explicitly is useful when data must be visible immediately.

34.5.5 BufferedReader vs Reader

`BufferedReader` adds efficient `**line-based reading**` on top of a Reader.

Without buffering, each character read may involve a system call.

```java
try (BufferedReader reader =
    new BufferedReader(new FileReader("file.txt"))) {

        String line;
        while ((line = reader.readLine()) != null) {
            System.out.println(line);
        }
}
```

> **Note**
>
> The `readLine()` method is only available on `BufferedReader` (not `Reader`), because it relies on buffering to efficiently detect line boundaries.

34.5.6 BufferedWriter Example

```java
try (BufferedWriter writer =
    new BufferedWriter(new FileWriter("file.txt"))) {

        writer.write("Hello");
        writer.newLine();
        writer.write("World");
}
```

`BufferedWriter` minimizes disk access and provides convenience methods such as newLine().

> **Note**
>
> Always wrap file streams with buffering unless there is a strong reason not to
>
> Prefer BufferedReader / BufferedWriter for text
>
> Prefer BufferedInputStream / BufferedOutputStream for binary data

---

34.6 java.io vs java.nio (and java.nio.file)

Modern Java applications increasingly favor NIO and NIO.2 APIs, but java.io remains fundamental and widely used.

34.6.1 Conceptual Differences

Aspect	java.io	java.nio / nio.2
`Programming model`	Stream-based	Buffer / Channel-based
`Blocking I/O`	blocking by default	Non-blocking capable
`File API`	File	Path + Files
`Scalability`	Limited	High
`Introduced`	Java 1.0	Java 4 / Java 7

> **Note**
>
> `java.nio` does not replace `java.io`.
>
> Many NIO classes internally rely on streams or coexist with them.

34.6.2 java.nio (Modern File I/O)

The `java.nio.file` package (NIO.2) provides a high-level, expressive, and safer file API. It is the preferred approach for file operations in Java 11+.

Example: Reading a File (NIO)

```
Path path = Path.of("file.txt");
List<String> lines = Files.readAllLines(path);
```

Equivalent java.io Code

```
try (BufferedReader reader = new BufferedReader(new FileReader("file.txt"))) {
    String line;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
}
```

34.7 When to Use Which API

Scenario	Recommended API
`Simple file read/write`	java.nio.file.Files
`Binary streaming`	InputStream / OutputStream
`Character text processing`	Reader / Writer
`High-performance servers`	java.nio.channels
`Legacy APIs`	java.io

34.8 Common Traps and Tips

- End-of-file is indicated by -1, not by an exception
- Closing a wrapper stream closes the wrapped stream automatically
- `BufferedReader.readLine()` strips line separators
- `InputStreamReader` always involves a charset
- Files utility methods throw checked IOException
- `available()` must not be used to detect EOF

Note

Most I/O bugs come from incorrect assumptions about blocking, buffering, or character encoding.

◀ 33. Files and Paths APIs | ▲ Index | 35. Java I/O APIs (Legacy and NIO) ▶

35. Java I/O APIs (Legacy and NIO)

Table of Contents

- 35.1 Legacy java.io — Design, Behavior, and Subtleties
    - 35.1.1 The Stream Abstraction
    - 35.1.2 Stream Chaining and the Decorator Pattern
    - 35.1.3 Blocking I/O: What It Means
    - 35.1.4 Resource Management: close(), flush(), and Why They Exist
    - 35.1.5 finalize(): Why It Exists and Why It Fails
    - 35.1.6 available(): Purpose and Misuse
    - 35.1.7 mark() and reset(): Controlled Backtracking
    - 35.1.8 Readers, Writers, and Character Encoding
    - 35.1.9 File vs FileDescriptor
- 35.2 java.nio — Buffers, Channels, and Non-Blocking IO
    - 35.2.1 From Streams to Buffers: A Conceptual Shift
    - 35.2.2 Buffers: Purpose and Structure
    - 35.2.3 Buffer Lifecycle: Write → Flip → Read
    - 35.2.4 clear() vs compact()
    - 35.2.5 Heap Buffers vs Direct Buffers
    - 35.2.6 Channels: What They Are
    - 35.2.7 Blocking vs Non-Blocking Channels
    - 35.2.8 Scatter/Gather I/O
    - 35.2.9 Selectors: Multiplexing Non-Blocking I/O
    - 35.2.10 When to Use java.nio
- 35.3 java.nio.file (NIO.2) — File and Directory Operations (Legacy vs Modern)
    - 35.3.1 Existence and Accessibility Checks
    - 35.3.2 Creating Files and Directories
    - 35.3.3 Deleting Files and Directories
    - 35.3.4 Copying Files and Directories
    - 35.3.5 Moving and Renaming
    - 35.3.6 Reading and Writing Text and Bytes (Files Enhancements)
    - 35.3.7 newInputStream/newOutputStream and newBufferedReader/newBufferedWriter
    - 35.3.8 Listing Directories and Traversing Trees
    - 35.3.9 Searching and Filtering
    - 35.3.10 Attributes: Reading, Writing, and Views
    - 35.3.11 Symbolic Links and Link Following
    - 35.3.12 Summary: Why Files Is an Enhancement
- 35.4 Serialization — Object Streams, Compatibility, and Traps
    - 35.4.1 What Serialization Does (and What It Does Not)
    - 35.4.2 The Two Main Marker Interfaces
    - 35.4.3 Basic Example: Writing and Reading an Object
    - 35.4.4 Object Graphs, References, and Identity
    - 35.4.5 serialVersionUID: The Versioning Key
    - 35.4.6 transient and static Fields
    - 35.4.7 Non-Serializable Fields and NotSerializableException
    - 35.4.8 Constructors and Serialization
    - 35.4.9 Custom Serialization Hooks: writeObject and readObject
    - 35.4.10 Example Use Case: Restoring a transient Derived Field

- 35.4.11 Externalizable: Full Control (and Full Responsibility)
- 35.4.12 readObject() Security Considerations
- 35.4.13 Common Traps and Practical Tips
- 35.4.14 When to Use (or Avoid) Java Serialization

35.1 Legacy java.io — Design, Behavior, and Subtleties

The legacy `java.io` API is the original I/O abstraction introduced in Java 1.0.

It is stream-oriented, blocking, and closely mapped to operating system I/O concepts.

Although newer APIs exist, `java.io` remains fundamental: many higher-level APIs build on it, and it is still heavily used.

35.1.1 The Stream Abstraction

A `stream` represents a continuous flow of data between a source and a destination.

In `java.io`, streams are **unidirectional**: they are either **input** or **output**.

Stream	Direction	Data unit	Category
`InputStream`	Input	Bytes (8-bit)	Byte stream
`OutputStream`	Output	Bytes (8-bit)	Byte stream
`Reader`	Input	Characters	Character stream
`Writer`	Output	Characters	Character stream

`Streams` hide the concrete origin of data (file, network, memory) and expose a uniform read/write interface.

35.1.2 Stream Chaining and the Decorator Pattern

Most java.io streams are designed to be combined.

Each wrapper adds behavior without changing the underlying data source.

```
InputStream in =
    new BufferedInputStream(
        new FileInputStream("data.bin"));
```

In this example:

- `FileInputStream` performs the actual file access
- `BufferedInputStream` adds an in-memory buffer

> **Note**
>
> This design is known as the Decorator Pattern.
>
> It allows features to be layered dynamically.

35.1.3 Blocking I/O: What It Means

All legacy `java.io` streams are **blocking**.

This means a thread performing I/O may be suspended by the operating system.

For example, when calling `read()`:

- If data is available, it is returned immediately
- If no data is available, the thread waits
- If end-of-stream is reached, -1 is returned

> **Note**
>
> Blocking behavior simplifies programming but limits scalability.

35.1.4 Resource Management: `close()`, `flush()`, and Why They Exist

Streams often encapsulate native operating system resources such as `file descriptors` or `socket handles`.

These resources are limited and must be released explicitly.

Method	Purpose
`flush()`	Writes buffered data to the destination
`close()`	Flushes and releases the resource

```
try (OutputStream out = new FileOutputStream("file.bin")) {
    out.write(42);
} // close() called automatically
```

> **Note**
>
> Failing to close streams may cause data loss or resource exhaustion.

35.1.5 `finalize()`: Why It Exists and Why It Fails

Early Java attempted to automate resource cleanup using finalization.

The `finalize()` method was called by the garbage collector before reclaiming memory.

However, garbage collection timing is unpredictable.

Aspect	finalize()
Execution time	Unspecified
Reliability	Low
Current status	Deprecated

> **Note**
>
> `finalize()` must never be used for I/O cleanup; it is deprecated and unsafe.

35.1.6 `available()`: Purpose and Misuse

The `available()` method estimates how many bytes can be read without blocking.

It does not report total remaining data.

Typical use cases include:

- Avoiding blocking in UI or protocol parsing
- Sizing temporary buffers

```
if (in.available() > 0) {
    in.read(buffer);
}
```

> **Note**
>
> `available()` must not be used to detect end-of-file. Only `read()` returning -1 signals EOF.

35.1.7 `mark()` and `reset()`: Controlled Backtracking

Some input streams allow marking a position and returning to it later.

```
BufferedInputStream in = new BufferedInputStream(...);
in.mark(1024);
// read ahead
in.reset();
```

Stream	markSupported()
`FileInputStream`	No
`BufferedInputStream`	Yes
`ByteArrayInputStream`	Yes

35.1.8 Readers, Writers, and Character Encoding

`Reader` and `Writer` operate on `characters`, not bytes.

This requires a `character encoding`.

If no charset is specified, the platform default is used.

```
new FileReader("file.txt"); // platform default encoding
```

> **Note**
>
> Relying on the platform default charset leads to non-portable bugs.
>
> Always specify a charset explicitly.

35.1.9 File vs FileDescriptor

`File` represents a `path` in the filesystem.

It does not represent an open resource.

FileDescriptor represents a native OS handle to an open file or stream.

Class	Represents	Owns OS handle?
`File`	Filesystem path	No
`FileDescriptor`	Native OS file handle	Yes

> **Note**
>
> Multiple streams may share the same FileDescriptor.
>
> Closing one closes the underlying resource for all.

35.2 `java.nio` — Buffers, Channels, and Non-Blocking I/O

The `java.nio` API (New I/O) was introduced to address limitations of legacy `java.io`.

It provides a lower-level, more explicit model of I/O that maps closely to modern operating systems.

At its core, `java.nio` is built around three concepts:

- `Buffers` — explicit memory containers
- `Channels` — bidirectional data connections
- `Selectors` — multiplexing non-blocking I/O

35.2.1 From Streams to Buffers: A Conceptual Shift

Legacy streams hide memory management from the programmer.

In contrast, `NIO` makes memory explicit through buffers.

Aspect	java.io	java.nio
Data model	Stream-based (push)	Buffer-based (pull from buffers)
Memory	Hidden inside streams	Explicit via buffers
Control	Simple, coarse-grained	More granular and configurable

With NIO, the application controls when data is read into memory and how it is consumed.

35.2.2 Buffers: Purpose and Structure

A `buffer` is a fixed-size, typed container for data.

All NIO I/O operations read from or write to buffers.

The most common buffer is ByteBuffer.

```
ByteBuffer buffer = ByteBuffer.allocate(1024);
```

Property	Meaning
`capacity`	Total size of the buffer
`position`	Current read/write index
`limit`	Boundary of readable or writable data

35.2.3 Buffer Lifecycle: Write → Flip → Read

`Buffers` have a strict `usage lifecycle`.

Misunderstanding it is a common source of bugs.

Typical sequence:

- Write data into the buffer
- `flip()` to switch to read mode
- Read data from the buffer
- `clear()` or `compact()` to reuse

```
ByteBuffer buffer = ByteBuffer.allocate(16);

buffer.put((byte) 1);
buffer.put((byte) 2);

buffer.flip(); // switch to read mode

while (buffer.hasRemaining()) {
    byte b = buffer.get();
}

buffer.clear(); // ready for writing again
```

> **Note**
>
> `flip()` does not erase data: it adjusts position and limit.

35.2.4 `clear()` vs `compact()`

After reading, a buffer can be reused in two ways.

Method	Behavior
`clear()`	Discards unread data
`compact()`	Preserves unread data

`compact()` is useful in streaming protocols where partial messages may remain in the buffer.

35.2.5 Heap Buffers vs Direct Buffers

Buffers can be allocated in two different memory regions.

```
ByteBuffer heap = ByteBuffer.allocate(1024);
ByteBuffer direct = ByteBuffer.allocateDirect(1024);
```

Type	Memory location	Characteristics
`Heap`	JVM heap	Garbage collected, cheap to allocate
`Direct`	Native memory	Better I/O throughput, more expensive to allocate

> **Note**
>
> Direct buffers reduce copying between JVM and OS but must be used carefully to avoid memory pressure.

35.2.6 Channels: What They Are

A `channel` represents a connection to an I/O entity such as a file, socket, or device.

Unlike streams, **channels are bidirectional**.

Channel	Type	Purpose
`FileChannel`	File	File I/O
`SocketChannel`	TCP	Stream (TCP) networking
`DatagramChannel`	UDP	Datagram (UDP) networking

```
try (FileChannel channel =
    FileChannel.open(Path.of("file.txt"))) {

        ByteBuffer buffer = ByteBuffer.allocate(128);
        channel.read(buffer);
}
```

35.2.7 Blocking vs Non-Blocking Channels

Channels can operate in blocking or non-blocking mode.

```
SocketChannel channel = SocketChannel.open();
channel.configureBlocking(false);
```

In `**non-blocking mode**`: - `read()` may return immediately with 0 bytes - `write()` may write only part of the data

> **Note**
>
> Non-blocking I/O shifts complexity from the OS to the application.

35.2.8 Scatter/Gather I/O

NIO supports reading into or writing from multiple buffers in a single operation.

```java
ByteBuffer header = ByteBuffer.allocate(128);
ByteBuffer body = ByteBuffer.allocate(1024);

ByteBuffer[] buffers = { header, body };
channel.read(buffers);
```

This is useful for structured protocols (headers + payload).

35.2.9 Selectors: Multiplexing Non-Blocking I/O

`Selectors` allow a single thread to monitor multiple channels.

They are the foundation of scalable servers.

Component	Role
`Selector`	Monitors multiple channels
`SelectionKey`	Represents channel registration and state
`Interest set`	Operations the selector watches for (read, write, etc.)

35.2.10 When to Use `java.nio`

`NIO` is appropriate when:

- High concurrency is required
- You need fine-grained memory control
- You are implementing protocols or servers

For simple file operations, `java.nio.file.Files` is usually sufficient.

35.3 `java.nio.file` (NIO.2) — File and Directory Operations (Legacy vs Modern)

This section focuses on practical operations on files and directories.

We compare the legacy approaches (java.io.File + java.io streams) with modern NIO.2 approaches (Path + Files).

The goal is not only to know the method names, but to understand:

- what each method really does
- what it returns and how it reports errors
- what pitfalls exist (race conditions, links, permissions, portability)
- when a Files method is a safe enhancement over the old approach

35.3.1 Existence and Accessibility Checks

A very common operation is to check whether a file exists and whether it can be accessed (read, written, executed).

Both the legacy API (java.io.File) and the modern NIO.2 API (java.nio.file.Files) provide methods for these checks.

However, it is important to understand that these checks are intentionally imprecise in both APIs.

They are best-effort hints, not reliable guarantees.

35.3.1.1 Legacy API (File)

```
File f = new File("data.txt");

boolean exists = f.exists();
boolean canRead = f.canRead();
boolean canWrite = f.canWrite();
boolean canExec = f.canExecute();
```

These methods return a boolean and do not explain why an operation failed.

For example, exists() may return false when:

- the file truly does not exist
- the file exists but access is denied
- a symbolic link is broken
- an I/O error occurs

The API provides no way to distinguish between these cases.

35.3.1.2 Modern API (Files)

```
Path p = Path.of("data.txt");

boolean exists = Files.exists(p);
boolean readable = Files.isReadable(p);
boolean writable = Files.isWritable(p);
boolean executable = Files.isExecutable(p);
```

Despite being newer, these methods also return booleans and also hide the reason for failure.

NIO.2 adds an explicit method to express uncertainty:

```
boolean notExists = Files.notExists(p);
```

> **Note**
>
> `exists()` and `notExists()` can both be `false` when the status cannot be determined (for example, due to permissions).

This does not make the check more accurate — it merely makes the uncertainty explicit.

35.3.1.2.1 Symbolic Link Awareness (Real Improvement)

One genuine enhancement of NIO.2 is control over symbolic link handling:

```
Files.exists(p, LinkOption.NOFOLLOW_LINKS);
```

Legacy File cannot reliably distinguish:

- a missing file
- a broken symbolic link
- a link pointing to an inaccessible target

NIO.2 allows link-aware checks and explicit link inspection.

35.3.1.2.2 Correct Usage Pattern (Critical)

Neither API provides reliable diagnostics through boolean checks alone.

Correct NIO.2 code does not “check first”.

Instead, it attempts the operation and handles the exception:

```
try {
    Files.delete(p);
} catch (NoSuchFileException e) {
    // file truly does not exist
} catch (AccessDeniedException e) {
    // permission problem
} catch (IOException e) {
    // other I/O error
}
```

> **Note**
>
> NIO.2's real advantage is exception-based diagnostics during operations, not more accurate existence or accessibility checks.

35.3.1.2.3 Summary Table

Goal	Legacy (File)	Modern (Files)	Key detail
Check existence	`exists()`	`exists()` / `notExists()`	notExists() may be false when the status cannot be determined
Check read/write	`canRead()` / `canWrite()`	`isReadable()` / `isWritable()`	Files methods can use LinkOption.NOFOLLOW_LINKS where supported
Error details	Not available	Available via exceptions on actions	Boolean checks themselves still do not explain the reason for failure

35.3.2 Creating Files and Directories

Creation is a major weakness of legacy File.

Legacy often uses createNewFile() and mkdir/mkdirs(), which return boolean and provide little diagnostic information.

35.3.2.1 Legacy API (File)

```
File f = new File("a.txt");
boolean created = f.createNewFile(); // may throw IOException

File dir = new File("dir");
boolean ok1 = dir.mkdir();
boolean ok2 = new File("a/b/c").mkdirs();
```

`mkdir()` creates only one directory level, `mkdirs()` creates parents too.

Both return false on failure but do not tell you why.

35.3.2.2 Modern API (Files)

```
Path file = Path.of("a.txt");
Files.createFile(file);

Path dir1 = Path.of("dir");
Files.createDirectory(dir1);

Path dirDeep = Path.of("a/b/c");
Files.createDirectories(dirDeep);
```

> **Note**
>
> `Files.createFile` throws `FileAlreadyExistsException` if the file exists.
>
> This is often preferred over boolean checks because it is race-safe.

Goal	Legacy (File)	Modern (Files)	Key detail
Create file	`createNewFile()`	`createFile()`	NIO throws FileAlreadyExistsException if the file exists
Create directory	`mkdir()`	`createDirectory()`	NIO throws detailed exceptions on failure
Create parents	`mkdirs()`	`createDirectories()`	Atomicity is not guaranteed for deep directory creation

35.3.3 Deleting Files and Directories

Deletion semantics differ strongly between legacy and NIO.2.

Legacy `delete()` returns boolean; NIO.2 offers methods that throw meaningful exceptions.

35.3.3.1 Legacy API (File)

```
File f = new File("a.txt");
boolean deleted = f.delete();
```

If deletion fails (permission denied, file does not exist, directory not empty), `delete()` usually returns false without explanation.

35.3.3.2 Modern API (Files)

```
Files.delete(Path.of("a.txt"));
```

If you want “delete if present” semantics, use deleteIfExists().

```
Files.deleteIfExists(Path.of("a.txt"));
```

Goal	Legacy (File)	Modern (Files)	Key detail
Delete	`delete()`	`delete()`	`Files.delete()` throws an exception with the failure reason
Delete if present	`exists() + delete()`	`deleteIfExists()`	Avoids TOCTOU (check-then-act) race conditions

35.3.4 Copying Files and Directories

Legacy copying typically requires manually reading and writing bytes via streams.

NIO.2 provides high-level copy operations with options.

35.3.4.1 Legacy technique (manual streams)

```
try (InputStream in = new FileInputStream("src.bin"); OutputStream out = new FileOutputStream(

    byte[] buf = new byte[8192];
    int n;
    while ((n = in.read(buf)) != -1) {
        out.write(buf, 0, n);
    }
}
```

This is verbose and easy to get wrong (missing buffering, missing close, etc.).

35.3.4.2 Modern API (Files.copy)

```
Files.copy(Path.of("src.bin"), Path.of("dst.bin"));
```

Copy behavior can be controlled with options.

```
Files.copy(
    Path.of("src.bin"),
    Path.of("dst.bin"),
    StandardCopyOption.REPLACE_EXISTING,
    StandardCopyOption.COPY_ATTRIBUTES
);
```

> **Note**
>
> `Files.copy` throws FileAlreadyExistsException by default.
>
> Use `REPLACE_EXISTING` when overwriting is intended.

Goal	Legacy approach	Modern (Files)	Key detail
Copy file	Manual stream loop	`Files.copy(Path, Path, …)`	Options include `REPLACE_EXISTING` and `COPY_ATTRIBUTES`
Copy stream	InputStream/OutputStream	`Files.copy(InputStream, Path, …)`	Useful for uploads, downloads, and piping data
Copy directory	Manual recursion	`walkFileTree + Files.copy`	No single one-liner for full directory tree copy

35.3.5 Moving and Renaming

Renaming in legacy code typically uses `File.renameTo()`, which is notoriously unreliable and platform-dependent.

NIO.2 provides Files.move() with explicit semantics and options.

35.3.5.1 Legacy API

```
boolean ok = new File("a.txt").renameTo(new File("b.txt"));
```

`renameTo()` returns false on failure and does not explain why.

It may also fail unexpectedly across filesystems.

35.3.5.2 Modern API

```
Files.move(Path.of("a.txt"), Path.of("b.txt"));
```

Move options provide precise behavior.

```
Files.move(
    Path.of("a.txt"),
    Path.of("b.txt"),
    StandardCopyOption.REPLACE_EXISTING,
    StandardCopyOption.ATOMIC_MOVE
);
```

> **Note**
>
> ATOMIC_MOVE is only guaranteed when the move occurs within the same filesystem. Otherwise an exception is thrown.

Goal	Legacy (File)	Modern (Files)	Key detail
Rename / move	`renameTo()`	`move()`	Files.move() provides exceptions and explicit options
Atomic move	Not supported	`move(…, ATOMIC_MOVE)`	Guaranteed only within the same filesystem
Replace existing	Not explicit	`REPLACE_EXISTING`	Makes overwrite intent explicit

35.3.6 Reading and Writing Text and Bytes (Files Enhancements)

A major enhancement of NIO.2 is the `Files` utility class, which provides high-level methods for common reading and writing tasks.

These methods reduce boilerplate and improve correctness.

35.3.6.1 Legacy text reading/writing

```
try (BufferedReader r = new BufferedReader(new FileReader("file.txt"))) {
    String line = r.readLine();
}
```

```
try (BufferedWriter w = new BufferedWriter(new FileWriter("file.txt"))) {
    w.write("hello");
}
```

These legacy classes typically use the platform default charset unless you explicitly bridge with InputStreamReader/OutputStreamWriter.

35.3.6.2 Modern text reading/writing

```
List<String> lines = Files.readAllLines(Path.of("file.txt"), StandardCharsets.UTF_8);
Files.write(Path.of("file.txt"), lines, StandardCharsets.UTF_8);

Files.lines(Path.of("file.txt")).forEach(System.out::println);

String string = Files.readString(Path.of("file.txt"));
Files.writeString(Path.of("file.txt"), string);
```

35.3.6.3 Modern binary reading/writing

```
byte[] data = Files.readAllBytes(Path.of("data.bin"));
Files.write(Path.of("out.bin"), data);
```

> **Important**
>
> `readAllBytes` and `readAllLines` load the entire file into memory.
>
> Use `Files.lines()` instead which lazily process each line or, for large files, prefer streaming APIs such as newBufferedReader or newInputStream.

Task	Legacy method	NIO.2 Files method	Key detail
Read all bytes	Manual InputStream loop	`readAllBytes()`	Loads the entire file into memory
Read all lines	BufferedReader loop	`readAllLines()`	Loads the entire file into memory
Read lines lazily	BufferedReader loop	`lines()`	Lazily process each line
Write bytes	OutputStream	`write(Path, byte[])`	Simple and concise
Write lines	BufferedWriter loop	`write(Path, Iterable, …)`	Charset can be specified
Append text	FileWriter(true)	`write(…, APPEND)`	Options make intent explicit

35.3.7 newInputStream/newOutputStream and newBufferedReader/newBufferedWriter

These `factory methods` create stream/reader instances from a Path.

They are the recommended bridge between classic streaming and NIO.2 path handling.

```
try (InputStream in = Files.newInputStream(Path.of("a.bin"))) { }
try (OutputStream out = Files.newOutputStream(Path.of("b.bin"))) { }
```

```
try (BufferedReader r = Files.newBufferedReader(Path.of("t.txt"), StandardCharsets.UTF_8)) { }
try (BufferedWriter w = Files.newBufferedWriter(Path.of("t.txt"), StandardCharsets.UTF_8)) { }
```

35.3.8 Listing Directories and Traversing Trees

Legacy directory listing is based on `File.list()` and `File.listFiles()`.

These methods return arrays and provide limited error reporting.

35.3.8.1 Legacy listing

```
File dir = new File(".");
File[] children = dir.listFiles();
```

NIO.2 provides multiple approaches depending on needs.

35.3.8.2 Modern listing (DirectoryStream)

```
try (DirectoryStream<Path> ds = Files.newDirectoryStream(Path.of("."))) {
    for (Path p : ds) {
        System.out.println(p);
    }
}
```

35.3.8.3 Modern walking (Files.walk)

```
Files.walk(Path.of("."))
    .filter(Files::isRegularFile)
        .forEach(System.out::println);
```

> **Note**
>
> `Files.walk` returns a Stream that must be closed. Prefer try-with-resources when using it.

```
try (Stream<Path> s = Files.walk(Path.of("."))) {
    s.forEach(System.out::println);
}
```

35.3.8.4 Modern traversal with FileVisitor

For full control (skip subtrees, handle errors, follow links), use `walkFileTree + FileVisitor`.

```
Files.walkFileTree(Path.of("."), new SimpleFileVisitor<>() {
    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) {
        System.out.println(file);
        return FileVisitResult.CONTINUE;
    }
});
```

Goal	Legacy	Modern	Key detail
List directory	`list()` / `listFiles()`	`newDirectoryStream()`	Lazy, must be closed
Walk tree (simple)	Manual recursion	`walk()` (Stream)	Stream must be closed
Walk tree (full control)	Manual recursion	`walkFileTree()`	Fine-grained control, error handling, pruning

35.3.9 Searching and Filtering

Searching is typically implemented by traversing and filtering.

NIO.2 provides convenient building blocks: glob patterns, streams, and visitors.

```
try (DirectoryStream<Path> ds =
    Files.newDirectoryStream(Path.of("."), "*.txt")) {
        for (Path p : ds) {
            System.out.println(p);
        }
}
```

```
try (Stream<Path> s = Files.find(Path.of("."), 10,
    (p, a) -> a.isRegularFile() && p.toString().endsWith(".log"))) {
        s.forEach(System.out::println);
}
```

35.3.10 Attributes: Reading, Writing, and Views

Legacy File exposes only a few attributes (size, lastModified).

NIO.2 supports rich metadata via attribute views.

35.3.10.1 Legacy attributes

```
long size = new File("a.txt").length();
long lm = new File("a.txt").lastModified();
```

35.3.10.2 Modern attributes

```
BasicFileAttributes a =
    Files.readAttributes(Path.of("a.txt"), BasicFileAttributes.class);

long size = a.size();
FileTime modified = a.lastModifiedTime();
```

You can also access attributes using string-based names.

```
Object v = Files.getAttribute(Path.of("a.txt"), "basic:size");
Files.setAttribute(Path.of("a.txt"), "basic:lastModifiedTime", FileTime.fromMillis(0));
```

> **Note**
>
> Attribute views are filesystem-dependent.
>
> Unsupported attributes cause exceptions.

35.3.11 Symbolic Links and Link Following

NIO.2 can explicitly detect and read symbolic links.

This is critical for correct filesystem traversal and security.

```
Path link = Path.of("mylink");
boolean isLink = Files.isSymbolicLink(link);

if (isLink) {
    Path target = Files.readSymbolicLink(link);
}
```

Many methods follow links by default.

To prevent this, pass `LinkOption.NOFOLLOW_LINKS` when supported.

35.3.12 Summary: Why Files Is an Enhancement

The Files utility class improves filesystem programming by:

- reducing boilerplate (copy/move/read/write)
- providing explicit options (overwrite, atomic move, follow links)
- offering richer metadata (attributes/views)
- supporting scalable traversal and searching

Legacy APIs remain mostly for backward compatibility or when required by old libraries.

35.4 Serialization — Object Streams, Compatibility, and Traps

Serialization is the process of converting an object graph into a byte stream so it can be stored or transmitted, then reconstructed later.

In Java, classic serialization is implemented by `java.io.ObjectOutputStream` and `java.io.ObjectInputStream`.

This topic is critical because it combines:

- I/O streams and object graphs
- versioning and backward compatibility
- security concerns and safe usage patterns
- special language rules (`transient` , static, `serialVersionUID`)

35.4.1 What Serialization Does (and What It Does Not)

When an object is serialized, Java writes enough information to reconstruct it later:

- the class name
- the serialVersionUID
- the values of serializable instance fields
- references between objects (object identity)

Serialization does not automatically include:

- static fields (class-level state)

- transient fields (explicitly excluded)
- non-serializable referenced objects (unless handled specially)

35.4.2 The Two Main Marker Interfaces

Java serialization is enabled by implementing one of these interfaces.

Interface	Meaning	Control level
`Serializable`	Opt-in marker, default mechanism	Medium (custom hooks possible)
`Externalizable`	Requires manual read/write implementation	High (full control over format)

> **Note**
>
> `Serializable` has no methods. It is a marker interface.
>
> `Externalizable` extends Serializable and adds readExternal/writeExternal.

35.4.3 Basic Example: Writing and Reading an Object

This is the minimal pattern used in practice.

```
import java.io.*;

class Person implements Serializable {

    private String name;
    private int age;

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

}

public class Demo {

    public static void main(String[] args) throws Exception {

        Person p = new Person("Alice", 30);

        try (ObjectOutputStream out =
                 new ObjectOutputStream(new FileOutputStream("p.bin"))) {
            out.writeObject(p);
        }

        try (ObjectInputStream in =
                 new ObjectInputStream(new FileInputStream("p.bin"))) {
            Person copy = (Person) in.readObject();
        }
    }

}
```

> **Note**
>
> `readObject()` returns Object. A cast is required. `readObject()` can throw ClassNotFoundException.

35.4.4 Object Graphs, References, and Identity

`Serialization` preserves object identity within the same stream.

If the same object reference appears multiple times in the graph, Java writes it once and later writes back-references.

```
Person p = new Person("Bob", 40);
Object[] arr = { p, p }; // same reference twice

out.writeObject(arr);
Object[] restored = (Object[]) in.readObject();

// restored[0] and restored[1] refer to the same object instance
```

> **Note**
>
> This behavior prevents infinite recursion on cyclic graphs.

35.4.5 `serialVersionUID`: The Versioning Key

`serialVersionUID` is a long identifier used to verify compatibility between a serialized stream and a class definition.

If the UID differs, deserialization typically fails with InvalidClassException.

If you do not declare `serialVersionUID`, the JVM computes one from class details.

Small changes may change the computed UID, breaking compatibility.

```
class Person implements Serializable {

    private static final long serialVersionUID = 1L;

    private String name;
    private int age;
}
```

Change type Compatibility impact (default)
Add a field Often compatible (new field gets default)
Remove a field Often compatible (missing field ignored)
Change field type Often incompatible
Change class name/package Incompatible
Change serialVersionUID Incompatible

> **Note**
>
> Declaring a stable serialVersionUID is the standard way to control serialization compatibility.

35.4.6 `transient` and `static` Fields

`transient` fields are excluded from serialization.

On `deserialization`, transient fields are assigned default values (0, false, null) unless explicitly restored.

`static` fields belong to the class, not to instances, so they are not serialized.

```
class Session implements Serializable {

private static final long serialVersionUID = 1L;

static int counter = 0;       // not serialized
transient String token;       // not serialized
String user;                  // serialized


}
```

> **Note**
>
> If a transient field is required after deserialization, you must recompute it or restore it manually.

35.4.7 Non-Serializable Fields and NotSerializableException

If an object references a field whose type is not serializable, serialization fails with NotSerializableException.

```
class Holder implements Serializable {

    private static final long serialVersionUID = 1L;

    private Thread t; // Thread is not serializable


}
```

Typical solutions:

- mark the field transient
- replace it with a serializable representation
- use custom serialization hooks

35.4.8 Constructors and Serialization

Constructor behavior during serialization and deserialization is a frequent source of confusion.

Java serialization restores object state primarily from the byte stream, not by running constructors.

35.4.8.1 Rule: Constructors of Serializable Classes Are Not Called

During deserialization of a Serializable class, the constructors, or any static or instance blocks of that class, are NOT executed.

The instance is created without calling those constructors (or any static or instance blocks), and field values are injected from the stream.

> **Note**
>
> This is why constructors of Serializable classes must not contain essential initialization logic.
>
> That initialization would not run during deserialization.

35.4.8.2 Inheritance Rule: The First Non-Serializable Superclass Constructor Is Called

When a Serializable class has a non-Serializable superclass, deserialization must still initialize that superclass part.

Therefore, Java calls **the no-argument constructor of the first non-Serializable superclass**.

Important implications:

- the non-Serializable superclass must have an accessible no-arg constructor
- serializable subclasses skip constructors, but non-serializable superclasses do not

35.4.8.3 Summary Table: Which Constructors Run

Class type	Constructor called during deserialization
Serializable class	No
Serializable subclass	No
First non-Serializable superclass	Yes (no-arg constructor)
Externalizable class	Yes (public no-arg constructor required)

35.4.8.4 Worked Example: Which Constructors Are Called

This example prints which constructors run during normal construction and during deserialization.

```
import java.io.*;

class A {

    A() {
        System.out.println("A constructor");
    }
}

class B extends A implements Serializable {

    private static final long serialVersionUID = 1L;

    B() {
        System.out.println("B constructor");
    }
}

class C extends B { // Extending B, C is Serializable

    private static final long serialVersionUID = 1L;

    C() {
        System.out.println("C constructor");
    }
}

public class Demo {

    public static void main(String[] args) throws Exception {

        C obj = new C();

        try (ObjectOutputStream out =
                 new ObjectOutputStream(new FileOutputStream("c.bin"))) {
            out.writeObject(obj);
        }

        try (ObjectInputStream in =
                 new ObjectInputStream(new FileInputStream("c.bin"))) {
            Object restored = in.readObject();
        }
    }
}
```

Expected Output and Explanation During normal construction (new C()):

```
A constructor
B constructor
C constructor
```

During deserialization (readObject):

```
A constructor
```

Explanation:

- C is Serializable, so C() is not called during deserialization
- B is Serializable, so B() is not called during deserialization
- A is not Serializable, so A() is called (no-arg constructor)
- Fields of B and C are restored from the stream instead of constructors running

> **Note**
>
> If the first non-Serializable superclass has no accessible no-arg constructor, deserialization fails.

35.4.9 Custom Serialization Hooks: `writeObject` and `readObject`

Custom serialization hooks exist to handle cases where default Java serialization is not enough (transient state, derived fields, encryption, validation, compatibility).

They are advanced but extremely important for correct deserialization behavior.

35.4.9.1 Why Custom Serialization Exists

By default, Java serialization automatically writes and reads all non-static, non-transient instance fields of a Serializable object.

This is convenient, but it cannot express certain common needs.

Typical reasons to customize serialization:

- A field should not be stored directly (sensitive data)
- A field is derived/cached and should be recomputed after restore
- You need validation when reading (reject invalid state)
- You need backward/forward compatibility logic (support older streams)
- A referenced object is not Serializable and must be handled specially

35.4.9.2 What `writeObject` and `readObject` Really Are

To customize serialization and deserialization, a class may define two special private methods named `writeObject` and `readObject`.

These methods are not overrides of methods from any interface or superclass.

They do not belong to Serializable, and they are not part of the normal method call flow of your program.

You never call `writeObject` or `readObject` yourself.

Instead, the serialization framework (ObjectOutputStream and ObjectInputStream) checks, using reflection, whether the class defines methods with these exact names and signatures.

If such methods are found, the serialization framework calls them automatically during serialization or deserialization.

If they are not found, the framework performs default serialization instead.

> **Note**
>
> If the method signature is incorrect (wrong visibility, parameter type, return type, or declared exceptions), the serialization framework does not recognize the method and silently falls back to default serialization.
>
> This behavior often makes errors hard to diagnose.

35.4.9.3 Exact Required Signatures

```
private void writeObject(ObjectOutputStream out) throws IOException

private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException
```

Key constraints:

- must be private
- must return void
- parameter types must match exactly
- exceptions must be compatible with the required throws list

35.4.9.4 What Happens During Serialization: Step by Step

When you serialize an object, you typically call:

```
out.writeObject(obj);
```

Then the serialization mechanism does roughly this:

- Checks if the object's class implements Serializable
- Checks whether the class declares a private writeObject(ObjectOutputStream)
- If not present: default serialization runs automatically
- If present: your writeObject is called instead

A crucial point: inside writeObject, Java does not automatically write the normal fields unless you ask for it.

This is why this call exists:

```
out.defaultWriteObject();
```

`defaultWriteObject()` means: "serialize the object's normal serializable fields using the default mechanism."

After that, you may write extra data in any format you want.

35.4.9.5 Typical Pattern and the Write/Read Order Rule

The typical pattern is to keep default serialization and then extend it.

The order of reads MUST match the order of writes.

```
private void writeObject(ObjectOutputStream out) throws IOException {
    out.defaultWriteObject(); // writes normal fields
    out.writeInt(42); // writes extra custom data
}

private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {

    in.defaultReadObject();          // reads normal fields
    int x = in.readInt();            // reads extra custom data in same order

}
```

> **Note**
>
> If you write extra values (ints/strings/etc.), you must read them back in the same sequence. Otherwise deserialization will fail or restore corrupted state.

35.4.10 Example Use Case: Restoring a transient Derived Field

A classic use case is recomputing a transient cached value after deserialization.

```java
class User implements Serializable {

    private static final long serialVersionUID = 1L;

    private String firstName;
    private String lastName;

    private transient String fullName;

    User(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.fullName = firstName + " " + lastName;
    }

    private void readObject(ObjectInputStream in)
            throws IOException, ClassNotFoundException {

        in.defaultReadObject(); // restore firstName and lastName
        fullName = firstName + " " + lastName; // recompute transient field
    }
}
```

35.4.11 Externalizable: Full Control (and Full Responsibility)

Externalizable requires you to define how to write and read the object manually.

It also requires a public no-argument constructor because deserialization instantiates the object first.

```java
import java.io.*;

class Point implements Externalizable {
int x;
int y;

public Point() { } // required

public Point(int x, int y) { this.x = x; this.y = y; }

@Override
public void writeExternal(ObjectOutput out) throws IOException {
    out.writeInt(x);
    out.writeInt(y);
}

@Override
public void readExternal(ObjectInput in) throws IOException {
    x = in.readInt();
    y = in.readInt();
}


}
```

> **Note**
>
> With Externalizable, you control the format. If you change the format later, you must handle backward compatibility yourself.

35.4.12 `readObject()` Security Considerations

Deserialization of untrusted data is dangerous because it can execute code indirectly via:

- constructors and initialization logic
- readObject hooks
- gadget chains in libraries

Safe practice guidelines:

- Never deserialize untrusted bytes unless you have a strong reason
- Prefer safe data formats (JSON, protobuf) for external inputs

- If forced, apply object filters and strict validation

35.4.13 Common Traps and Practical Tips

- Serializable is marker-only; no method must be implemented
- `readObject` returns Object and may throw ClassNotFoundException
- `static fields` are never serialized
- `transient fields` reset to default values unless restored
- Missing `serialVersionUID` may break compatibility unexpectedly
- Externalizable requires public no-arg constructor
- NotSerializableException occurs when a referenced field type is not serializable

35.4.14 When to Use (or Avoid) Java Serialization

Use classic Java serialization mainly for:

- short-lived local persistence under controlled versions
- in-memory caching where both ends are trusted
- legacy systems that already depend on it

Avoid it for:

- public network protocols
- long-term storage with evolving schemas
- untrusted inputs

◀ 34. Java I/O Streams | ▲ Index | 36. Interacting with the User (Standard I/O Streams) ▶

36. Interacting with the User (Standard I/O Streams)

Table of Contents

- 36.1 The Standard I/O Streams
- 36.2 PrintStream What It Is and Why It Exists
  - 36.2.1 Key Characteristics of PrintStream
  - 36.2.2 Basic Usage of PrintStream
  - 36.2.3 Formatting Output with PrintStream
- 36.3 Reading Input as an IO Stream
  - 36.3.1 Low-Level Reading from Systemin
  - 36.3.2 Using InputStreamReader and BufferedReader
- 36.4 The Scanner Class Convenient but Subtle
  - 36.4.1 Common Scanner Pitfalls
- 36.5 Closing System Streams
- 36.6 Acquiring Input with Console
  - 36.6.1 Reading Input from Console
  - 36.6.2 Reading Passwords Securely
- 36.7 Formatting Console Output
- 36.8 Comparing Console Scanner and BufferedReader
- 36.9 Redirection and Standard Streams
- 36.10 Common Traps and Best Practices
- 36.11 Final Summary

Java programs often need to interact with the user: printing information, reading input, and formatting output.

This interaction is implemented using standard I/O streams, which are normal Java streams connected to the operating system.

This chapter explains how Java interacts with the console and standard input/output, starting from the most basic concepts and moving to higher-level APIs.

36.1 The Standard I/O Streams

Every Java program starts with three predefined streams provided by the JVM.

They are connected to the process environment (usually a terminal or console).

Stream	Field	Type	Purpose
Standard output	`System.out`	PrintStream	Normal output
Standard error	`System.err`	PrintStream	Error output
Standard input	`System.in`	InputStream	User input

> **Note**
>
> These streams are created by the JVM, not by your program.
>
> They exist for the entire lifetime of the process.

36.2 `PrintStream`: What It Is and Why It Exists

`PrintStream` is a byte-oriented output stream designed for human-readable output.

It wraps another OutputStream and adds convenient printing methods.

`System.out` and `System.err` are both instances of `PrintStream`.

36.2.1 Key Characteristics of PrintStream

- Byte-oriented stream with text-printing helpers
- Provides `print()` and `println()` methods
- Converts values to text automatically
- Does not throw `IOException` on write errors
- Optionally supports auto-flushing on newline / `println()`

> **Note**
>
> Unlike most streams, PrintStream suppresses IOExceptions.
>
> Errors must be checked using checkError().

36.2.2 Basic Usage of PrintStream

```
System.out.println("Hello");
System.out.print("Value: ");
System.out.println(42);
```

`println()` appends the platform-specific line separator automatically.

36.2.3 Formatting Output with PrintStream

PrintStream supports formatted output using `printf()` and `format()`, which are based on the same syntax as String.format().

```
System.out.printf("Name: %s, Age: %d%n", "Alice", 30);
```

Specifier	Meaning
`%s`	String
`%d`	Integer
`%f`	Floating-point
`%n`	Platform-independent newline

> **Note**
>
> `printf()` does not automatically add a newline unless you specify `%n`.

36.3 Reading Input as an I/O Stream

Standard input (System.in) is an InputStream connected to user input.

It provides raw bytes and must be adapted for practical use.

36.3.1 Low-Level Reading from System.in

At the lowest level, you can read raw bytes from System.in.

This is rarely convenient for interactive programs.

```
int b = System.in.read();
```

> **Note**
>
> `System.in.read()` blocks until input is available.

36.3.2 Using InputStreamReader and BufferedReader

To read text input, `System.in` is typically wrapped into a Reader and buffered.

```
BufferedReader reader =
new BufferedReader(new InputStreamReader(System.in));

String line = reader.readLine();
```

This converts `bytes → characters` and allows line-based input.

36.4 The Scanner Class (Convenient but Subtle)

`Scanner` is a high-level utility for parsing text input.

It is often used for console interaction, especially in small programs.

```
Scanner sc = new Scanner(System.in);
int value = sc.nextInt();
String text = sc.nextLine();
```

> **Note**
>
> `Scanner` performs tokenization and parsing, not simple reading.
>
> This makes it convenient but slower and sometimes surprising.

36.4.1 Common Scanner Pitfalls

- Mixing `nextInt()` (and other `nextXxx()`) with `nextLine()` can appear to "skip" input because the trailing newline from the numeric token is still in the buffer.
- Parsing errors throw InputMismatchException
- Scanner is relatively slow for large input

36.5 Closing System Streams

`System streams` are special and must be handled carefully.

Stream	Close explicitly?
`System.out`	No
`System.err`	No
`System.in`	Usually no

Closing `System.out` or `System.err` closes the underlying OS stream and affects the entire JVM: closing these streams affects the entire JVM process, not just the current class or method.

> **Note**
>
> In almost all applications, you should NOT close `System.out` or `System.err`.

36.6 Acquiring Input with `Console`

The `Console` class provides a higher-level, safer way to interact with the user.

It is designed specifically for interactive console programs.

```
Console console = System.console();
if (console == null) {
    throw new IllegalStateException("No console available");
}
```

> **Note**
>
> `System.console()` may return null when no console is available (e.g. IDEs, redirected input).

The presence of a console depends on the underlying platform and on how the JVM is launched.

If the JVM is started from an interactive command line and the standard input/output streams are not redirected, a console is typically available.
In this case, the console is usually connected to the keyboard and display from which the program was launched.

If the JVM is started in a non-interactive context — for example by an IDE, a background scheduler, a service manager, or with redirected standard streams — a console will usually not be available.

When a console exists, it is represented by a single unique instance of the Console class, which can be obtained by invoking the `System.console()` method. If no console device is available, this method will return `null`.

36.6.1 Reading Input from Console

```
String name = console.readLine("Name: ");
```

`readLine()` prints a prompt and reads a full line of input.

36.6.2 Reading Passwords Securely

Console allows reading passwords without echoing characters.

```
char[] password = console.readPassword("Password: ");
```

> **Note**
>
> Passwords are returned as `char[]` so they can be cleared from memory.

36.7 Formatting Console Output

Console also supports formatted output, similar to PrintStream.

```
console.printf("Welcome %s%n", name);
```

This uses the same format specifiers as `printf()`.

36.8 Comparing Console, Scanner, and BufferedReader

API	Use case	Strengths	Limitations
`BufferedReader`	Simple text input	Fast, predictable, charset explicit	Manual parsing
`Scanner`	Token-based / parsed input	Convenient, expressive	Slower, subtle token behavior
`Console`	Interactive console apps	Passwords, prompts, formatted I/O	May be unavailable (`null`)

36.9 Redirection and Standard Streams

Standard streams can be redirected by the operating system. Java code does not need to change.

```
java App < input.txt > output.txt
```

From the program's perspective, System.in and System.out still behave like normal streams.

> **Note**
>
> Redirection is handled by the operating system or shell. The Java code does not need to change to support it.

36.10 Common Traps and Best Practices

- PrintStream suppresses IOExceptions
- `System.console()` can return null
- Do not close `System.out` or `System.err`
- Scanner mixes parsing and reading
- Console is preferred for passwords
- If you use `Scanner` on `System.in`, do not close the Scanner if other parts of the program still need to read from `System.in` (closing the Scanner closes `System.in`).

36.11 Final Summary

- `System.out` and `System.err` are PrintStreams for output
- `System.in` is a byte stream that must be adapted for text
- `BufferedReader` and `Scanner` are common input strategies
- `Console` provides safe interactive input and output
- Standard streams integrate naturally with OS redirection

Java Platform Module System (JPMS)

37. Java Platform Module System (JPMS)

Table of Contents

- 37.1 Why Modules Were Introduced
  - 37.1.1 Problems with the Classpath
  - 37.1.2 Example of a Classpath Problem
- 37.2 What Is a Module
  - 37.2.1 Core Properties of Modules
  - 37.2.2 Module vs Package vs JAR
- 37.3 The module-infojava Descriptor
  - 37.3.1 Minimal Module Descriptor
- 37.4 Module Directory Structure
- 37.5 A First Modular Program
  - 37.5.1 Main Class
  - 37.5.2 Module Descriptor
- 37.6 Strong Encapsulation Explained
- 37.7 Summary of Key Ideas

The `Java Platform Module System` (**JPMS**) was introduced in Java 9.

It is a language-level and runtime-level mechanism for structuring Java applications into strongly encapsulated units called `modules`.

JPMS affects how code is:

- organized
- compiled
- linked
- packaged
- loaded at runtime

Understanding JPMS is essential for modern Java, especially for large applications, libraries, runtime images, and deployment tooling.

37.1 Why Modules Were Introduced

Before Java 9, Java applications were built using only:

- `packages`
- `JAR` files
- the `classpath`

This model had serious limitations as applications grew.

37.1.1 Problems with the Classpath

The classpath is a flat list of JARs where:

- all public classes are accessible to everyone
- there is no reliable dependency declaration
- conflicting versions are common
- encapsulation is weak or nonexistent

- duplicate classes silently overwrite each other based on classpath order

This led to well-known issues such as:

- “JAR hell”
- classpath ordering bugs
- accidental use of internal APIs
- runtime failures that were not detected at compile time

37.1.2 Example of a Classpath Problem

Suppose two libraries depend on different versions of the same third-party JAR.

Only one version can be placed on the classpath.

Which one is chosen depends on classpath order, not correctness.

> **Note**
>
> This problem cannot be reliably solved with the classpath alone.

37.2 What Is a Module?

A `module` is a named, self-describing unit of code.

In summary, a module is a collection of one or more related packages, together with a module descriptor file that explicitly defines its dependencies and the functionality it makes available.

A module therefore provides its consumers with a clearly defined and controlled set of capabilities.

Every named module has a unique name that identifies it to the compiler and module system.

It explicitly declares:

- what it depends on
- what it exposes to other modules
- what it keeps hidden

A module is stronger than a package and more structured than a JAR.

37.2.1 Core Properties of Modules

Property	Description
Strong encapsulation	Packages are hidden by default
Explicit dependencies	Dependencies must be declared
Reliable configuration	Missing dependencies cause errors early
Named identity	Each module has a unique name

37.2.2 Module vs Package vs JAR

Concept	Purpose	Encapsulation
Package	Namespace grouping	Weak (public still visible)
JAR	Packaging / deployment	None (all classes visible when on classpath)
Module	Encapsulation + dependency unit	Strong (unexported packages hidden)

37.3 The `module-info.java` Descriptor

Every `named module` is defined by a module descriptor file named:

```
module-info.java
```

This file describes the module to the compiler and the runtime.

37.3.1 Minimal Module Descriptor

A minimal module descriptor declares only the module name. The filename must be exactly `module-info.java`, and it must be located in the root of the module source tree.

```
module com.example.hello {
}
```

> **Note**
>
> **A module with no directives exports nothing and depends on nothing.**

37.4 Module Directory Structure

A modular project follows a standard directory layout. The module descriptor sits at the root of the module's source tree.

```
src/
└ com.example.hello/
    ├ module-info.java
    └ com/
        └ example/
            └ hello/
                └ Main.java
```

Key points:

- The **directory name matches the module name**
- `module-info.java` is at the top of the module source root
- packages follow standard Java naming rules

> **Note**
>
> In IDE and build-tool projects, the file structure may differ (e.g. Maven uses `src/main/java`). What always remains true: `module-info.java` sits in the root of the module source tree and package paths follow standard Java naming.

37.5 A First Modular Program

Let's create a minimal modular application.

37.5.1 Main Class

```
package com.example.hello;

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

37.5.2 Module Descriptor

```
module com.example.hello {
    exports com.example.hello;
}
```

The `exports directive` makes the `package` accessible to other modules.

Without it, the package is encapsulated and inaccessible.

37.6 Strong Encapsulation Explained

In `JPMS`, packages are NOT accessible by default.

Even public classes are hidden unless explicitly exported.

In modules, `public` means "public to other modules *only if* the containing package is exported."

Situation	Accessible from another module?
Public class in non-exported package	No
Public class in exported package	Yes
Protected member in exported package	Yes, but only via subclassing (not general access)
Package-private class/member (any package)	No
Private member	No

> **Note**
>
> This is a fundamental difference from the classpath model.

37.7 Summary of Key Ideas

- `JPMS` introduces modules as strong units of encapsulation
- Dependencies are explicit and checked
- `module-info.java` is the central descriptor
- Packages are hidden unless exported
- Classpath-based visibility no longer applies in modules
- Public visibility is no longer enough: module exports control accessibility

---

◀ 36. Interacting with the User (Standard I/O Streams) | ▲ Index | 38. Compiling, Packaging, and Running Modules ▶

38. Compiling, Packaging, and Running Modules

Table of Contents

- 38.1 The Module Path vs the Classpath
- 38.2 Module-Related Command-Line Options
    - 38.2.1 Options Available in Both java and javac
    - 38.2.2 Options Applicable Only to javac
    - 38.2.3 Options Applicable Only to java
    - 38.2.4 Important Distinctions
- 38.3 Compiling a Single Module
- 38.4 Compiling Multiple Interdependent Modules
- 38.5 Packaging a Module into a Modular JAR
- 38.6 Running a Modular Application
- 38.7 Module Directives Explained
    - 38.7.1 requires
    - 38.7.2 requires transitive
    - 38.7.3 exports
    - 38.7.4 exports-to-qualified-exports
    - 38.7.5 opens
    - 38.7.6 opens-to-qualified-opens
    - 38.7.7 Table of Core Directives
    - 38.7.8 Exports vs Opens — Compile-Time vs Runtime Access
- 38.8 Named, Automatic, and Unnamed Modules
    - 38.8.1 Named Modules
    - 38.8.2 Automatic Modules
    - 38.8.3 Unnamed Module
    - 38.8.4 Comparison Summary
- 38.9 Top-Down and Bottom-Up Approaches to Modularizing an Application
    - 38.9.1 Top-Down Approach
        - 38.9.1.1 Fundamental Rules
        - 38.9.1.2 Practical Implications
        - 38.9.1.3 Access Rules Summary
    - 38.9.2 Bottom-Up Approach
        - 38.9.2.1 Core Strategy
        - 38.9.2.2 Architectural Advantages
    - 38.9.3 Conceptual Comparison
    - 38.9.4 Migration Insight
- 38.10 Inspecting Modules and Dependencies
    - 38.10.1 Describing Modules with java
    - 38.10.2 Describing Modular JARs
    - 38.10.3 Analyzing Dependencies with jdeps
- 38.11 Creating Custom Runtime Images with jlink
- 38.12 Creating Self-Contained Applications with jpackage
- 38.13 Final Summary JPMS in Practice

Once a `module` is defined with a `module-info.java` file, it must be compiled, packaged, and executed using module-aware tools.

This section explains how the `Java toolchain` changes when modules are involved.

38.1 The Module Path vs the Classpath

`JPMS` introduces a new concept: the **module path**.

It exists alongside the traditional **classpath**, but the two behave very differently.

Aspect	Classpath	Module path
Structure	Flat list of JARs	Modules with identities
Encapsulation	None	Strong
Dependency checking	None	Strict
Split packages	Allowed	Forbidden (named modules)
Resolution order	Order-dependent	Deterministic

> **Note**
>
> - A JAR placed on the `module path` is treated as a `module`:
>   - If it contains a `module-info.class`, it becomes a `named module`.
>   - If it does not contain a module descriptor, it becomes an `automatic module`.
> - A JAR placed on the classpath is not treated as a module.
>   - Instead, it becomes part of the unnamed module, together with all other classpath entries.
> - A modular JAR (i.e., a JAR containing module-info.class) can still be used as a regular JAR.
>   - If it is placed on the classpath instead of the module path, it is treated as part of the unnamed module, allowing non-modular applications to use it without adopting the module system.
> - Split packages:
>   - Are allowed on the classpath (multiple JARs may contain classes in the same package).
>   - Are forbidden for named or automatic modules on the module path.

38.2 Module-Related Command-Line Options

When working with the Java Module System, both `java` and `javac` provide specific options for compiling and running modular applications.

Some options are shared, while others are specific to one tool.

38.2.1 Options Available in Both `java` and `javac`

These options can be used during compilation as well as execution:

- **`--module`** or **`-m`**
  Used to compile or run only the specified module.
- **`--module-path`** or **`-p`**
  Specifies the paths where `java` or `javac` will look for module definitions.

The Java Module System provides three special command-line options, usable with both `javac` and `java`, that allow you to override module access rules at runtime or compilation time without modifying the

`module-info.java` files. These options affect only the current command execution and do not permanently change the module descriptors.

The three options are:

- `--add-reads`
- `--add-exports`
- `--add-opens`

They are typically used for testing, backward compatibility, migration scenarios, or when working with third-party modules that cannot be modified.

For example, suppose `moduleA` needs to access public types from `moduleB`, but:

- `moduleA` does not declare `requires moduleB;`
- `moduleB` does not export the required package to `moduleA`

Instead of editing the `module-info.java` files, you can temporarily grant the necessary access using:

```
javac --add-reads moduleA=moduleB \
      --add-exports moduleB/com.modB.package1=moduleA \
      ...

java  --add-reads moduleA=moduleB \
      --add-exports moduleB/com.modB.package1=moduleA \
      ...
```

Here is what each option means:

- `--add-reads moduleA=moduleB`
  Temporarily declares that `moduleA` reads `moduleB`.
  This is equivalent to adding `requires moduleB;` inside `moduleA`'s descriptor.
  It allows `moduleA` to access the exported packages of `moduleB`.
- `--add-exports moduleB/com.modB.package1=moduleA`
  Temporarily exports the package `com.modB.package1` from `moduleB` to `moduleA`.
  This is equivalent to adding:
  `exports com.modB.package1 to moduleA;`
  inside `moduleB`'s descriptor.

Important distinction:

- `--add-reads` establishes module-level readability.
- `--add-exports` grants access to specific packages.
- `--add-opens` (not shown above) is similar to `--add-exports` but additionally allows deep reflection access, typically required by frameworks.

These options do not modify the compiled module metadata; they only adjust the module graph for that particular invocation of `javac` or `java`.

38.2.2 Options Applicable Only to `javac`

These options apply only at compile time:

- **`--module-source-path`**
  (no shortcut)
  Used by `javac` to locate source module definitions.
- **`-d`**
  Specifies the output directory where the `.class` files will be generated after compilation.

38.2.3 Options Applicable Only to `java`

These options apply only at runtime:

- **`--list-modules`**
  (no shortcut)
  Lists all observable modules and then exits.

- `--show-module-resolution`
  (no shortcut)
  Displays module resolution details during application startup.
- `--describe-module` or `-d`
  Describes a specified module and then exits.

38.2.4 Important Distinctions

The option `-d` has different meanings depending on the tool:

- In `javac`, `-d` defines the output directory for compiled class files.
- In `java`, `-d` is a shortcut for `--describe-module`.

Additionally, `-d` must not be confused with `-D` (uppercase D).

- `-D` is used when running a Java program to define system properties as name-value pairs on the command line.

```
java -Dconfig.file=app.properties com.example.Main
```

In this example, `-Dconfig.file=app.properties` sets a system property that can be accessed at runtime using `System.getProperty("config.file")`.

38.3 Compiling a Single Module

To compile a module, you must specify the module source path and the destination directory.

```
javac -d out \
src/com.example.hello/module-info.java \
src/com.example.hello/com/example/hello/Main.java
```

A more scalable approach uses `--module-source-path`.

```
javac --module-source-path src \
      -d out \
      $(find src -name "*.java")
```

> **Note**
>
> `--module-source-path` tells javac where to find multiple modules at once.

38.4 Compiling Multiple Interdependent Modules

When modules depend on each other, their dependencies must be resolvable at compile time.

`--module-path` **mods** (sample dir containing interdependent moules) should contain already-compiled modular JARs or compiled module directories (each with its own module-info.class).

```
javac -d out \
--module-source-path src \
--module-path mods \
$(find src -name "*.java")
```

Here:

- `--module-source-path` locates module source trees
- `--module-path` provides already-compiled modules

38.5 Packaging a Module into a Modular JAR

After compilation, modules are typically packaged as JAR files.

A modular JAR contains a `module-info.class` at its root.

If `module-info.class` is present, the JAR becomes a `named module` automatically and its `name` is taken from the descriptor (not the filename).

```
jar --create \
--file mods/com.example.hello.jar \
--main-class com.example.hello.Main \
-C out/com.example.hello .
```

> **Note**
>
> A JAR with `module-info.class` is a `named module, not an automatic module`. When a JAR contains a `module-info.class`, its module name is taken from that file and not inferred from the filename.

38.6 Running a Modular Application

To run a modular application, you use the `module path` and specify the `module name`.

```
java --module-path mods \
--module com.example.hello/com.example.hello.Main
```

You can shorten this using the `-p` and `-m` options.

```
java -p mods -m com.example.hello/com.example.hello.Main
```

> **Note**
>
> When using named modules, the classpath is ignored for resolution of module dependencies.

38.7 Module Directives Explained

The `module-info.java` file contains directives that describe dependencies, visibility, and services.

Each directive has a precise meaning.

38.7.1 `requires`

The `requires` directive declares a dependency on another module.

Without it, types from the dependency module cannot be used.

```
module com.example.app {
    requires com.example.lib;
}
```

Effects of requires:

- Dependency must be present at compile and runtime
- Exported packages of the required module become accessible

38.7.2 `requires transitive`

`requires transitive` exposes a dependency to downstream modules.

It propagates readability.

```
module com.example.lib {
    requires transitive com.example.util;
    exports com.example.lib.api;
}
```

Meaning:

- **Any module requiring com.example.lib automatically reads com.example.util**
- **Callers do not need to declare requires com.example.util explicitly**

> **Note**
>
> This is similar to “public dependencies” in other module systems.
>
> Readable ≠ exported: a transitive requirement does not export your packages automatically.

38.7.3 `exports`

`exports` makes a package accessible to other modules.

Only exported packages are visible outside the module.

```
module com.example.lib {
    exports com.example.lib.api;
}
```

Non-exported packages remain strongly encapsulated.

38.7.4 `exports ... to` (Qualified Exports)

A qualified export restricts access to specific modules.

```
module com.example.lib {
    exports com.example.internal to com.example.friend;
}
```

Only the listed modules can access the exported package.

38.7.5 `opens`

`opens` allows deep reflective access to a package.

This is primarily for frameworks using reflection.

```
module com.example.app {
    opens com.example.app.model;
}
```

> **Note**
>
> opens does NOT make a package accessible at compile time. It only affects runtime reflection.

38.7.6 `opens ... to` (Qualified Opens)

You can restrict reflective access to specific modules.

```
module com.example.app {
    opens com.example.app.model to com.fasterxml.jackson.databind;
}
```

> **Note**
>
> `opens` affects reflection; `exports` affects compilation and type visibility.

38.7.7 Table of Core Directives

Directive	Purpose
`requires`	Declare a dependency
`requires transitive`	Propagate dependency
`exports`	Expose a package
`exports ... to`	Expose to specific modules
`opens`	Allow runtime reflection
`opens ... to`	Restrict reflective access

38.7.8 Exports vs Opens — Compile-Time vs Runtime Access

Visibility	Compile-time?	Runtime reflection?
`exports`	Yes	No
`opens`	No	Yes
`exports ... to`	Yes (limited modules)	No
`opens ... to`	No	Yes (limited modules)

> **Important**
>
> `JPMS` adds a `module path`, but the `classpath` still exists. They can coexist, but named modules take precedence.

38.8 Named, Automatic, and Unnamed Modules

`JPMS` supports different kinds of modules to allow gradual migration from the classpath.

JPMS must interoperate with legacy code.

To support gradual adoption, the JVM recognizes three different module categories.

38.8.1 Named Modules

A `named module` has a `module-info.class` and a stable identity.

- Strong encapsulation
- Explicit dependencies
- Full JPMS support

38.8.2 Automatic Modules

A JAR without `module-info` placed on the `module path` becomes an `automatic module`.

Its name is derived from the JAR file name.

- Reads all other modules
- Exports all packages
- No strong encapsulation

> **Note**
>
> Automatic modules exist to ease migration. They are not suitable as a long-term design.

38.8.3 Unnamed Module

Code on the classpath belongs to the `unnamed module`.

- Reads all named modules
- All packages are open
- Cannot be required by named modules

> **Note**
>
> The `unnamed module` preserves legacy classpath behavior.

38.8.4 Comparison Summary

Module type	module-info present?	Encapsulation	Reads
`Named`	Yes	Strong	Declared only
`Automatic`	No	Weak	All modules
`Unnamed`	No	None	All modules

38.9 Top-Down and Bottom-Up Approaches to Modularizing an Application

When migrating an existing (non-modular) application to the Java Platform Module System (JPMS), two principal strategies can be adopted: **top-down** and **bottom-up**. Both approaches require a clear understanding of how **named modules**, **automatic modules**, and the **unnamed module** interact.

38.9.1 Top-Down Approach

In a `top-down approach`, you **begin by modularizing the main application** module and then progressively migrate its dependencies.

38.9.1.1 Fundamental Rules

1. **A JAR placed on the module path becomes an automatic module.**
    - Its name is determined either:
        - From the `Automatic-Module-Name` entry in its manifest, or
        - Derived from the JAR filename (hyphens are replaced with dots and version numbers are ignored).
          Example:
          `mysql-connector-java-8.0.11.jar` → `mysql.connector.java`
    - An `automatic module`:
        - Exports all its packages.
        - Reads all other modules.
2. **A JAR placed on the classpath belongs to the unnamed module.**
    - The `unnamed module`:
        - Exports all its packages.
        - Can read all other modules.
    - However, it has no name, so no other module can declare `requires` on it.
3. **Explicitly named modules (those with a `module-info.java`)**
    - Can declare dependencies using:

      ```
      requires some.module;
      ```

    - Can depend on:
        - Other named modules
        - Automatic modules
    - Cannot depend on the unnamed module (since it has no name).

Important consequence:

> A `named module` can read an `automatic module`, but it cannot read the `unnamed module`.

38.9.1.2 Practical Implications

Suppose:

- Application JAR = `A`
- `A` directly depends on `B`
- `B` depends on `C`

If you modularize `A` first:

- `A` must declare `requires B;`
- Therefore, `B` must be on the module path (named or automatic)
- If `B` becomes a named module, then:
    - `C` must also move to the module path (as named or automatic)

Thus, in a top-down migration:

- You start from the application layer.
- You progressively modularize dependencies outward.
- Automatic modules are often used temporarily during transition.

38.9.1.3 Access Rules Summary

Module Type	Exports	Can Read
`Named module`	Only declared exports	Only required modules
`Automatic module`	All packages	All modules
`Unnamed module`	All packages	All modules

> **Important**
>
> - `Automatic and unnamed modules` are **permissive**.
> - `Named modules` enforce explicit dependency and export rules.

38.9.2 Bottom-Up Approach

In a `bottom-up approach`, you begin by modularizing the `lowest-level libraries first`, and then progressively move upward toward higher-level modules `and finally the main application`.

38.9.2.1 Core Strategy

You first convert foundational libraries into proper named modules with explicit `module-info.java` descriptors.

Then:

- Modules that depend on them are modularized.
- Finally, the main application becomes a named module.

This approach emphasizes:

- Explicit `requires` relationships
- Controlled `exports`

- Strong encapsulation from the beginning

38.9.2.2 Architectural Advantages

Compared to automatic modules:

- Named modules export only what is explicitly declared.
- They do not implicitly read every other module.
- Encapsulation boundaries are clearly defined.

Bottom-up modularization generally results in:

- A cleaner dependency graph
- Better maintainability
- Stronger module boundaries

38.9.3 Conceptual Comparison

Top-Down

- Start from the main application.
- Modularize dependencies as required.
- Often rely temporarily on automatic modules.
- Faster initial migration.

Bottom-Up

- Start from core libraries.
- Define strict module descriptors early.
- Progressively move upward.
- Produces a more disciplined and robust modular architecture.

38.9.4 Migration Insight

In practice, real-world projects often combine both strategies:

- A top-down migration is used to enable modular execution quickly.
- A bottom-up refinement phase replaces automatic modules with properly defined named modules.

This hybrid approach allows incremental adoption of JPMS while gradually strengthening encapsulation and architectural clarity.

38.10 Inspecting Modules and Dependencies

38.10.1 Describing Modules with java

```
java --describe-module java.sql
```

This shows `exports`, `requires`, and `services` of a module.

38.10.2 Describing Modular JARs

```
jar --describe-module --file mylib.jar
```

38.10.3 Analyzing Dependencies with `jdeps`

`jdeps` analyzes class and module dependencies statically.

```
jdeps myapp.jar
```

```
jdeps --module-path mods --check my.module
```

To detect use of JDK internal APIs:

```
jdeps --jdk-internals myapp.jar
```

38.11 Creating Custom Runtime Images with `jlink`

`jlink` builds a minimal Java runtime containing only the modules required by an application.

```
jlink
--module-path $JAVA_HOME/jmods:mods
--add-modules com.example.app
--output runtime-image
```

Benefits:

- smaller runtime
- faster startup
- no unused JDK modules

38.12 Creating Self-Contained Applications with `jpackage`

`jpackage` builds platform-specific installers or application images.

```
jpackage
--name MyApp
--input mods
--main-module com.example.app/com.example.Main
```

`jpackage` can produce:

- .exe / .msi (Windows)
- .pkg / .dmg (macOS)
- .deb / .rpm (Linux)

38.13 Final Summary JPMS in Practice

- `JPMS` introduces `strong encapsulation` and reliable dependencies
- `Modules` replace fragile classpath conventions
- `Services` enable decoupled architectures
- `Automatic` and `unnamed modules` support migration
- `jlink` and `jpackage` enable modern deployment models

◀ 37. Java Platform Module System (JPMS) | ▲ Index | 39. Services in JPMS (The ServiceLoader Model) ▶

39. Services in JPMS (The ServiceLoader Model)

Table of Contents

- 39.1 The Problem Services Solve
  - 39.1.1 Roles in the Service Model
  - 39.1.2 Service Interface Module
  - 39.1.3 Service Provider Module
  - 39.1.4 Service Consumer Module
  - 39.1.5 Loading Services at Runtime
  - 39.1.6 Service Resolution Rules
  - 39.1.7 Service Locator Layer
  - 39.1.8 Sequential Invocation Diagram
  - 39.1.9 Component Summary Table

`JPMS` includes a built-in service mechanism that allows `modules` to discover and use implementations at runtime without hardcoding dependencies between `providers` and `consumers`.

This mechanism is based on the existing `ServiceLoader API`, but modules make it reliable, explicit, and safe.

39.1 The Problem Services Solve

Sometimes a module needs to use a capability, but should not depend on a specific implementation.

Typical examples include: - logging frameworks - database drivers - plugin systems - service providers selected at runtime

Without services, the consumer would need to depend directly on a concrete implementation.

This creates tight coupling and reduces flexibility.

39.1.1 Roles in the Service Model

The `JPMS service model` involves four distinct roles.

Role	Description
`Service interface`	Defines the contract
`Service provider`	Implements the service
`Service consumer`	Uses the service
`Service loader`	Discovers implementations at runtime

39.1.2 Service Interface Module

The `service interface` defines the API that `consumers` depend on.

It must be exported so other modules can see it.

```
package com.example.service;

public interface GreetingService {
    String greet(String name);
}
```

```
module com.example.service {
    exports com.example.service;
}
```

> **Note**
>
> The service interface module should contain no implementations.

39.1.3 Service Provider Module

A `provider module` implements the service interface and declares that it provides the service.

```
package com.example.service.impl;

import com.example.service.GreetingService;

public class EnglishGreeting implements GreetingService {
    public String greet(String name) {
        return "Hello " + name;
    }
}
```

```
module com.example.provider.english {
    requires com.example.service;
    provides com.example.service.GreetingService with com.example.service.impl.EnglishGreeting
}
```

Key points: - The `provider` depends on the `service interface` - The implementation class does not need to be exported - The `provides` directive registers the implementation

39.1.4 Service Consumer Module

The `consumer module` declares that it uses a service, but does not name any implementation.

```
module com.example.consumer {
    requires com.example.service;
    uses com.example.service.GreetingService;
}
```

> **Note**
>
> `uses` declares intent to discover implementations at runtime.
>
> A module that declares `uses` but has no matching provider on the module path compiles normally, but `ServiceLoader` returns an empty result at runtime.

39.1.5 Loading Services at Runtime

The `ServiceLoader API` performs service discovery.

It finds all providers visible to the module graph.

```
ServiceLoader<GreetingService> loader =
    ServiceLoader.load(GreetingService.class);

for (GreetingService service : loader) {
    System.out.println(service.greet("World"));
}
```

`JPMS` guarantees that only declared providers are discovered.

Classpath-based “accidental” discovery is prevented.

39.1.6 Service Resolution Rules

For a service to be discoverable by `ServiceLoader`, several conditions must be satisfied:

Rule	Meaning
Provider module must be readable	Resolved by `requires` graph
Service interface must be exported	Consumers must see it
Consumer must declare `uses`	Otherwise ServiceLoader fails
Provider must declare `provides`	Implicit discovery is forbidden

39.1.7 Service Locator Layer

It is possible to introduce an additional layer called `Service Locator`.

In this architecture:

- The `consumer` does not directly use `ServiceLoader`
- The `Service Locator` is the only component that declares `uses`
- The `consumer` depends on the `Service Locator`

Architectural structure:

```
Consumer → Service Locator → ServiceLoader → Provider
```

Service Locator module:

```
module com.example.locator {
    requires com.example.service;
    uses com.example.service.GreetingService;
}
```

Service Locator class:

```
package com.example.locator;

import java.util.ServiceLoader;
import com.example.service.GreetingService;

public class GreetingLocator {

    public static GreetingService getService() {
        return ServiceLoader
                .load(GreetingService.class)
                .findFirst()
                .orElseThrow();
    }
}
```

Consumer module:

```
module com.example.consumer {
    requires com.example.locator;
}
```

The consumer does not declare `uses` because it does not directly invoke `ServiceLoader`.

39.1.8 Sequential Invocation Diagram

Execution sequence:

1. The `Consumer` invokes `GreetingLocator.getService()`
2. The `Service Locator` invokes `ServiceLoader.load(...)`
3. The `ServiceLoader` consults the module graph
4. The system identifies modules that declare `provides`
5. The `Provider` implementation is instantiated
6. The instance is returned to the `Consumer`

Sequential diagram:

```
Consumer
   |
   | 1. getService()
   ▼
Service Locator
   |
   | 2. ServiceLoader.load()
   ▼
ServiceLoader
   |
   | 3. Provider resolution
   ▼
Provider Implementation
   |
   | 4. Instance returned
   ▼
Consumer
```

39.1.9 Component Summary Table

Component	Role	exports	requires	uses	provides
SPI	Defines contract	✅	❌	❌	❌
Provider	Implements service	❌	✅	❌	✅
Service Locator	Performs discovery	(optional)	✅	✅	❌
Consumer	Uses the service	❌	✅	❌	❌

◀ 38. Compiling, Packaging, and Running Modules | ▲ Index | ▶