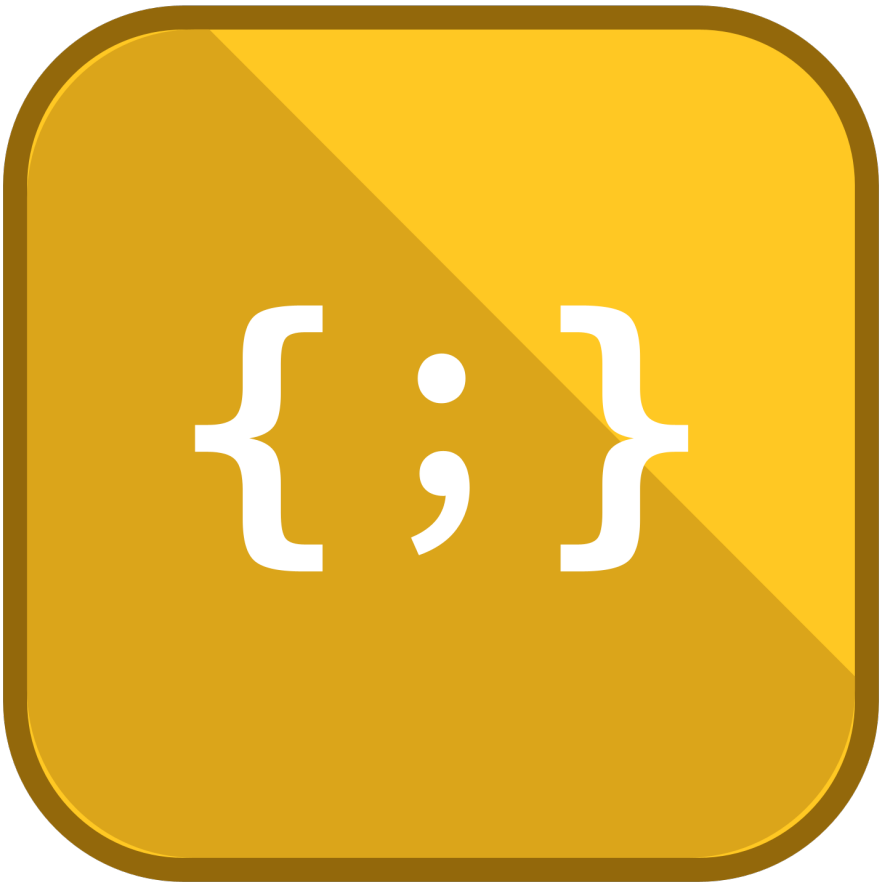
Java Syntax



readbytes

Java Syntax

From Beginner to Expert

readbytes.github.io

2025-07-11

This page is intentionally left blank.

Contents

Chapter 1.

Java Program Structure and Entry Point

1. Structure of a Java Program
2. The `main` Method Syntax
3. Class Declaration and File Structure
4. Statements, Blocks, and Semicolons
5. Command-Line Arguments Syntax
6. Comments and Formatting Best Practices

1 Java Program Structure and Entry Point

1.1 Structure of a Java Program

Understanding the structure of a Java program is the foundation of learning the language. Java follows a strict syntax and organizational style that ensures consistency and clarity, making code easier to read, maintain, and compile. Every Java program is composed of classes and methods, with a designated entry point—the `main` method—where execution begins.

Let's start with a minimal Java program:

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

This small example includes all the essential building blocks of a Java application. Let's break it down:

`public class HelloWorld`

Every Java application must have at least one class. In Java, **everything resides within a class**, even the entry point to the program. Here, `HelloWorld` is the name of the class. The keyword `public` makes it accessible from outside the class, including the Java Virtual Machine (JVM), which needs to invoke it.

- The class name should **match the filename**. So, if the class is named `HelloWorld`, the file must be named `HelloWorld.java`.
- The keyword `class` is used to define the class.

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

This is the **entry point** of any standalone Java application. The JVM looks for this exact method signature to start the program.

- `public`: So the method is accessible to the JVM.
- `static`: It allows the JVM to call the method **without creating an instance** of the class.
- `void`: Indicates that this method does **not return a value**.
- `main`: This is the predefined method name.
- `String[] args`: This is a **parameter array** that holds command-line arguments passed to the program.

Without this method, the program will compile but not run.

Curly Braces `{ }`

Java uses curly braces to define blocks of code:

- One pair for the class: `{ ... }`
- Another pair for the method: `{ ... }`

Each opening brace must be matched by a closing brace. This **block structure defines scope**, and misplacing or omitting braces results in a **compile-time error**.

```
System.out.println("Hello, Java!");
```

This is a **statement** that prints a line of text to the console. It uses the `System.out` output stream and calls its `println()` method to display the message.

- Each statement in Java **must end with a semicolon (`;`)**.
- This line shows how Java **invokes methods on objects**, even for something as simple as printing to the screen.

1.2 The `main` Method Syntax

1.2.1 The `main` Method Syntax

The `main` method is the entry point of any standalone Java application. When you run a Java program, the Java Virtual Machine (JVM) looks specifically for this method to start executing your code. Because of this, its syntax must be exact.

Here is the correct syntax for the `main` method:

```java
public static void main(String[] args) {
    // program statements go here
}
```

Let's break down each part of this declaration:

- **`public`**: This access modifier means the method is visible to all other classes, including the JVM. Since the JVM needs to call this method from outside your class, it must be `public`.
- **`static`**: This keyword means the method belongs to the class itself, not to an instance of the class. The JVM calls the `main` method **without creating an object** of the class, so `static` is mandatory.
- **`void`**: This indicates that the method does not return any value.
- **`main`**: This is the exact method name the JVM looks for when starting a program.
- **`String[] args`**: This is the method parameter—a single argument that is an array of `String` objects. It contains any command-line arguments passed to the program when it starts.

1.2.2 Common Syntax Error: Missing `static`

A very common mistake is to forget the `static` keyword:

```
public void main(String[] args) { // Incorrect!
    // ...
}
```

This compiles but will result in a runtime error: `Error: Main method not found in class ...`

Because without `static`, the JVM would need to create an instance of the class before calling `main`, but it cannot do this on its own.

1.2.3 Reflection

The `main` method is special because it signals where the Java program begins. Its exact signature is required by the JVM to correctly locate and invoke it. This design enforces consistency, allowing the JVM to start any Java program in a predictable way. As you continue learning Java, remember that while you can create many other methods with different names and signatures, the `main` method is your program's official starting point.

1.3 Class Declaration and File Structure

1.3.1 Class Declaration and File Structure

In Java, classes are the fundamental building blocks of programs. Understanding how to declare classes and how they relate to your source files is crucial for writing well-structured code.

Declaring a Java Class

A class declaration typically looks like this:

```
public class MyClass {
    // class body
}
```

- The keyword `class` is followed by the **class name**, which should follow Java naming conventions: start with a capital letter and use camel case (e.g., `MyClass`, `EmployeeData`).
- The `public` access modifier means this class can be accessed from anywhere.

Filename Must Match the Public Class

When a class is declared `public`, **the name of the file must exactly match the class name** and end with `.java`. For example, if your class is:

```
public class MyClass { }
```

then the filename **must be** `MyClass.java`. This rule is enforced by the Java compiler and is crucial for correct compilation and program organization.

Valid and Invalid Naming Examples

Class Declaration	Filename	Valid?
`public class HelloWorld`	`HelloWorld.java`	YES Valid
`class HelloWorld`	`HelloWorld.java`	YES Valid (no public)
`public class HelloWorld`	`helloWorld.java`	NO Invalid (case mismatch)
`public class HelloWorld`	`Hello.java`	NO Invalid (name mismatch)

Multiple Classes in One File

Java allows **multiple classes in a single `.java` file**, but only **one of them may be declared `public`**, and if so, the file name must match that public class.

For example, this is valid in a file named `MainApp.java`:

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

class Helper {
    void assist() {
        System.out.println("Helping...");
    }
}
```

- Here, `MainApp` is public, and the file is named `MainApp.java`.
- The `Helper` class is package-private (no access modifier) and can only be accessed within the same package.
- If no class is declared `public`, the filename can be anything, but this is uncommon for programs with an entry point.

1.4 Statements, Blocks, and Semicolons

1.4.1 Statements, Blocks, and Semicolons

In Java, a **statement** is a complete unit of execution—something the program does, such as declaring a variable or calling a method. Most statements end with a **semicolon (`;`)**, which tells the compiler where one statement finishes and the next begins.

Examples of Statements

```
int number = 10;            // variable declaration statement
number = number + 5;        // assignment statement
System.out.println(number); // method call statement
```

Each of these lines is a statement and must end with a semicolon. Missing a semicolon causes a **compile-time error** because the compiler cannot determine where the statement ends.

Blocks: Grouping Statements

A **block** is a group of statements enclosed in curly braces `{}`. Blocks are used to define the body of methods, classes, and control structures like loops and conditionals.

Example:

```
if (number > 10) {
    System.out.println("Number is greater than 10");
    number = 10;
}
```

Here, the two statements inside the `{}` form a block that executes only if the condition is true.

Blocks can be nested and help organize code logically.

Semicolon Placement

- The semicolon terminates the statement immediately before it.
- You cannot put a semicolon after control structures like `if`, `for`, or `while` before their blocks; doing so ends the statement prematurely and can cause unexpected behavior.

Incorrect example:

```
if (number > 10); {  // Incorrect semicolon here!
    System.out.println("This always prints");
}
```

1.4.2 Reflection

Java does **not enforce indentation or whitespace**, but proper formatting is crucial for human readers. Using consistent indentation and clear block organization makes code easier to understand and maintain. Blocks clearly indicate the scope of statements and control flow. While semicolons are small punctuation marks, they are fundamental to Java's syntax; missing or misplaced semicolons are a frequent cause of compilation errors for beginners. Paying attention to statements, blocks, and semicolons early will build a strong foundation for writing syntactically correct Java programs.

1.5 Command-Line Arguments Syntax

Java programs can accept input parameters when started from the command line. These inputs, called **command-line arguments**, are passed to the `main` method as an array of strings: `String[] args`.

Using `String[] args`

The parameter `args` in the `main` method holds all command-line arguments as strings. You can access them by their index, starting at `args[0]`.

Here is an example program that prints all command-line arguments:

```
public class PrintArgs {
    public static void main(String[] args) {
        System.out.println("Number of arguments: " + args.length);
        for (int i = 0; i < args.length; i++) {
            System.out.println("Argument " + i + ": " + args[i]);
        }
    }
}
```

Running the Program with Arguments

To run this program from the terminal:

1. Compile it:

```
javac PrintArgs.java
```

2. Run it with arguments, for example:

```
java PrintArgs hello world 123
```

Output:

```
Number of arguments: 3
```

```
Argument 0: hello
Argument 1: world
Argument 2: 123
```

Each word after the class name is treated as a separate argument and passed to the `args` array.

1.6 Comments and Formatting Best Practices

Comments are lines in your code that the Java compiler ignores. They help programmers explain what the code does, making it easier to read and maintain.

Types of Comments

- **Single-line comments** start with `//` and continue to the end of the line:

```java
// This is a single-line comment
int number = 10; // Inline comment after a statement
```

- **Multi-line comments** are enclosed between `/*` and `*/` and can span multiple lines:

```java
/*
 This is a multi-line comment.
 It can cover several lines.
*/
```

- **JavaDoc comments** use `/** ... */` and are specially formatted for generating documentation:

```java
/**
 * This method prints a greeting message.
 * @param name the name to greet
 */
public void greet(String name) {
    System.out.println("Hello, " + name);
}
```

JavaDoc comments help create API documentation and should be used on classes, methods, and important fields.

When and Why to Comment

Comments should **explain why** something is done, clarify complex logic, or provide context—not just restate what the code clearly shows. Over-commenting trivial code or outdated comments can confuse readers. Good comments improve collaboration and ease future maintenance.

Sample with Good Formatting and Comments

```java
public class Example {

    public static void main(String[] args) {
        // Print a welcome message
        System.out.println("Welcome to Java Syntax!");

        /*
         * Initialize a counter and increment it.
         * This simulates a simple loop without actual looping.
         */
        int counter = 0;
        counter++;
        System.out.println("Counter value: " + counter);
    }
}
```

1.6.1 Reflection

Consistent indentation and clean formatting are essential for readability, even though Java ignores whitespace. Use comments thoughtfully to guide readers without cluttering code. Clear structure and meaningful comments make your code professional and easier to understand by others — or yourself in the future.

Chapter 2.

Variables, Types, and Literals

1. Variable Declaration and Initialization
2. Primitive Data Types
3. Type Conversion and Casting
4. Constants with `final`
5. Local Variable Type Inference (`var`)
6. Numeric, Character, and Boolean Literals
7. Escape Sequences and Special Characters

2 Variables, Types, and Literals

2.1 Variable Declaration and Initialization

In Java, variables are used to store data values. Before using a variable, you must **declare** it by specifying its type and name. You can also **initialize** the variable by assigning it a value. Declaration and initialization can be done separately or combined.

Declaring Variables

A variable declaration tells the compiler about the variable's **type** and **name**:

```
int age;         // Declares an integer variable named age
double price;    // Declares a double variable named price
String name;     // Declares a String variable named name
```

At this point, the variables exist but do **not hold any meaningful value** (they have default values if they are instance or class variables, but local variables must be initialized before use).

Initializing Variables

You assign a value to a variable using the assignment operator `=`:

```
age = 25;           // Initializes age with 25
price = 19.99;      // Initializes price with 19.99
name = "Alice";     // Initializes name with the string "Alice"
```

Combined Declaration and Initialization

It is common and often clearer to declare and initialize a variable in a single statement:

```
int age = 25;
double price = 19.99;
String name = "Alice";
```

This approach reduces errors and makes your code easier to read.

Multiple Variables Declaration

You can declare multiple variables of the same type on one line, though it is best to initialize them separately for clarity:

```
int x = 10, y = 20, z = 30;  // Multiple variables declared and initialized
```

Naming Conventions and Best Practices

- Use **meaningful names** that describe the variable's purpose, such as `count`, `totalPrice`, or `userName`.
- Follow Java naming conventions: variable names start with a **lowercase letter** and

use **camelCase** for multiple words.
- Avoid using Java keywords or names that are too short or vague (`a`, `temp`), unless in very limited contexts.
- Initialize variables before using them to avoid errors.
- Prefer initializing variables at the time of declaration unless there is a reason to delay assignment.

Full runnable code:

```java
public class VariableDemo {

    public static void main(String[] args) {
        // Declaration
        int age;
        double price;
        String name;

        // Initialization
        age = 25;
        price = 19.99;
        name = "Alice";

        // Combined declaration and initialization
        int score = 100;
        double taxRate = 0.13;
        String city = "Toronto";

        // Multiple variable declarations (same type)
        int x = 10, y = 20, z = 30;

        // Output values
        System.out.println("Age: " + age);
        System.out.println("Price: $" + price);
        System.out.println("Name: " + name);
        System.out.println("Score: " + score);
        System.out.println("Tax Rate: " + taxRate);
        System.out.println("City: " + city);
        System.out.println("x: " + x + ", y: " + y + ", z: " + z);
    }
}
```

2.1.1 Reflection

Clear and consistent variable declaration and initialization make your code easier to understand and maintain. Combining declaration and initialization is a good habit that prevents uninitialized variable errors. Choosing descriptive variable names following conventions improves readability and helps others (and your future self) grasp the code's intent quickly. Mastering variables is a fundamental step toward effective Java programming.

2.2 Primitive Data Types

Java provides eight **primitive data types** that store simple values directly in memory. These types are the building blocks for storing data efficiently and performing fast operations.

The Eight Primitive Types

Type	Size (bits)	Description	Example Declaration
`byte`	8	Small integer, from -128 to 127	`byte b = 100;`
`short`	16	Larger integer, from -32,768 to 32,767	`short s = 30000;`
`int`	32	Standard integer, from -2^{31} to $2^{31}-1$	`int i = 1_000_000;`
`long`	64	Large integer, from -2^{3} to $2^{3}-1$	`long l = 10_000_000_000L;`
`float`	32	Single-precision floating point	`float f = 3.14f;`
`double`	64	Double-precision floating point	`double d = 3.14159;`
`char`	16	A single Unicode character	`char c = 'A';`
`boolean`	1 (logical)	True or false value	`boolean flag = true;`

When to Use Each Type

- Use **`int`** for most whole number calculations because it balances memory and performance well.
- Use **`long`** when values exceed the `int` range (e.g., large counters or timestamps). Remember to append `L` to literals.
- Use **`byte`** and **`short`** mainly when memory is a concern, such as large arrays, but be cautious about implicit conversions.
- Use **`float`** and **`double`** for decimal numbers. Prefer `double` for higher precision; `float` is mostly used in graphics or when memory is limited.
- Use **`char`** to store single characters, such as letters or digits. Note that it stores Unicode characters.
- Use **`boolean`** for true/false logic conditions.

Syntax Examples

```
int age = 30;
double price = 19.99;
char grade = 'A';
boolean isActive = false;
byte smallNumber = 100;
long population = 7_800_000_000L;
float piApprox = 3.14f;
short temperature = -10;
```

2.2.1 Reflection

Choosing the right primitive type affects **memory usage** and **performance**. Smaller types save memory but may require extra care when performing operations or conversions. Larger types like `long` and `double` use more space but allow a wider range or precision. Understanding these types helps you write efficient Java code tailored to your program's needs.

2.3 Type Conversion and Casting

In Java, **type conversion** happens when you assign a value from one data type to another. This can be **implicit** (automatic) or **explicit** (manual casting). Understanding these conversions is essential to avoid data loss and runtime errors.

Implicit Casting (Widening Conversion)

Java automatically converts smaller numeric types to larger ones because there is no risk of losing information. This is called **widening conversion**.

Example:

```
int i = 100;
long l = i;       // int implicitly cast to long
float f = l;      // long implicitly cast to float
```

Here, `int` converts to `long`, and then `long` converts to `float` without explicit code because the target type can hold all possible values of the source type.

Explicit Casting (Narrowing Conversion)

When converting from a larger type to a smaller one, you must explicitly cast the value. This is called **narrowing conversion** and may cause **data loss**.

Example:

```
double d = 9.78;
int i = (int) d;  // explicit cast from double to int, fractional part lost
System.out.println(i); // Output: 9
```

Casting a `double` to an `int` truncates the decimal part, which can cause loss of precision.

Another example with possible overflow:

```
int big = 130;
byte b = (byte) big;  // byte range is -128 to 127
System.out.println(b); // Output: -126 (due to overflow)
```

Because `130` exceeds the byte range, the value wraps around, leading to unexpected results.

Full runnable code:

```java
public class TypeCastingDemo {
    public static void main(String[] args) {
        // ===== Implicit Casting (Widening Conversion) =====
        int i = 100;
        long l = i;         // int -> long
        float f = l;        // long -> float

        System.out.println("=== Implicit Casting ===");
        System.out.println("int value: " + i);
        System.out.println("long value (from int): " + l);
        System.out.println("float value (from long): " + f);

        // ===== Explicit Casting (Narrowing Conversion) =====
        double d = 9.78;
        int i2 = (int) d;   // double -> int (fractional part lost)

        int big = 130;
        byte b = (byte) big;  // int -> byte (overflow expected)

        System.out.println("\n=== Explicit Casting ===");
        System.out.println("double value: " + d);
        System.out.println("int value (from double): " + i2);
        System.out.println("int value (for overflow): " + big);
        System.out.println("byte value (from int): " + b);
    }
}
```

2.3.1 Why Casting Is Needed

Casting allows flexibility in operations involving different numeric types and helps the compiler understand how to handle assignments. Without casting, certain assignments would cause compilation errors.

2.3.2 When Casting Becomes Dangerous

- **Data loss:** When converting from a larger to a smaller type, important data may be truncated or wrapped.
- **Logic errors:** Unexpected results from overflow or truncation can lead to bugs that are hard to detect.
- **Readability:** Excessive or unnecessary casting can make code harder to read and maintain.

2.3.3 Reflection

Casting is a powerful tool but must be used carefully. Always be aware of the types involved and the possible effects of narrowing conversions. When precision and data integrity are critical, avoid unnecessary casts and prefer using types that can safely hold your values. Understanding type conversion helps you write robust and predictable Java programs.

2.4 Constants with `final`

In Java, the `final` keyword is used to declare **constants**—variables whose values cannot be changed once assigned. Using constants improves code readability and helps prevent accidental modification of important values.

Declaring Constants

To declare a constant, use `final` before the variable type. By convention, constant names are written in **uppercase letters with underscores** separating words:

```
final int MAX_USERS = 100;
final double PI = 3.14159;
final String APP_NAME = "JavaSyntaxBook";
```

Reassignment Not Allowed

Once a `final` variable is initialized, attempting to change its value causes a **compile-time error**:

```
MAX_USERS = 200;  // Error: cannot assign a value to final variable MAX_USERS
```

This restriction ensures that constants remain fixed throughout the program.

Using `final` constants makes your code easier to read and maintain by clearly indicating values that are meant to stay the same. Constants serve as single sources of truth, preventing "magic numbers" or hard-coded values scattered throughout your code. This is especially useful for configuration settings, limits, or fixed parameters like mathematical constants. Declaring constants also helps prevent bugs caused by unintended variable reassignment, increasing the robustness of your programs.

2.5 Local Variable Type Inference (`var`)

Starting from Java 10, you can declare local variables using the `var` keyword, which allows the compiler to **infer** the variable's type based on the assigned value. This feature reduces verbosity while maintaining type safety.

Correct Syntax

Instead of explicitly specifying the type, you write:

```
var message = "Hello, world!";   // inferred as String
var count = 100;                 // inferred as int
var list = new ArrayList<String>(); // inferred as ArrayList<String>
```

The compiler determines the type at compile time, so `var` does not make Java dynamically typed; the variable still has a fixed type.

Limitations

- **Initialization is mandatory:** You must initialize the variable at the point of declaration so the compiler can infer its type.

```
var x;  // Error: cannot use 'var' without initialization
```

- **Cannot assign `null` without type:** Because `null` alone doesn't provide type information, you cannot do this:

```
var obj = null;  // Error: cannot infer type from null
```

- **Only for local variables:** `var` cannot be used for method parameters, fields, or return types.

When `var` Improves Clarity

- When the type is obvious or verbose, such as generic types:

```
var map = new HashMap<String, List<Integer>>();
```

- When the exact type isn't important to understand the code's intent.

When `var` Reduces Readability

- Overusing `var` with unclear or complex expressions may make it harder to understand the variable's type at a glance.

```
var result = processData();  // What is the type of result?
```

`var` helps write cleaner, less cluttered code, especially with long type names. However, use it thoughtfully—clear, explicit types can sometimes improve readability, especially for beginners or complex codebases. Balancing `var` usage with clarity ensures maintainable and understandable Java programs.

Full runnable code:

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

```java
public class VarExample {
    public static void main(String[] args) {
        // ===== Correct Syntax: Type Inference with 'var' =====
        var message = "Hello, world!";                  // inferred as String
        var count = 100;                                // inferred as int
        var list = new ArrayList<String>();             // inferred as ArrayList<String>
        var map = new HashMap<String, List<Integer>>(); // inferred as HashMap<String, List<Integer>>

        System.out.println("Message: " + message);
        System.out.println("Count: " + count);
        list.add("Java");
        System.out.println("List: " + list);
        map.put("scores", List.of(85, 90, 95));
        System.out.println("Map: " + map);

        // ===== Limitation Examples (Commented Out) =====

        // var x; // NO Error: cannot use 'var' without initializer
        // var obj = null; // NO Error: cannot infer type from null

        // ===== Readability Considerations =====
        var name = "Alice";          // clear and obvious
        var total = calculateTotal(5, 3); // not clear what type 'total' is at a glance

        System.out.println("Name: " + name);
        System.out.println("Total: " + total);
    }

    static double calculateTotal(int a, int b) {
        return a * b * 1.5;
    }
}
```

2.6 Numeric, Character, and Boolean Literals

In Java, **literals** are fixed values directly written in code. They represent data like numbers, characters, or boolean values. Understanding literal syntax helps write clear and readable code.

Integer Literals

- Decimal (base 10):

```java
int decimal = 1234;
```

- Binary (base 2), introduced in Java 7, start with `0b` or `0B`:

```java
int binary = 0b1010;  // equals decimal 10
```

- Hexadecimal (base 16), start with `0x` or `0X`:

```
int hex = 0xFF;       // equals decimal 255
```

- You can use **underscores** to improve readability in large numbers:

```
int largeNumber = 1_000_000;
```

Floating-Point Literals

- By default, decimals are `double`:

```
double pi = 3.14159;
```

- Use `f` or `F` suffix for `float` literals:

```
float gravity = 9.8f;
```

- You can also use scientific notation:

```
double avogadro = 6.022e23;
```

Character Literals

- Use single quotes `' '` to represent a single Unicode character:

```
char letter = 'A';
char digit = '7';
```

- You can also use Unicode escape sequences:

```
char smiley = '\u263A';
```

Boolean Literals

- Only two possible values: `true` and `false`:

```
boolean isJavaFun = true;
boolean isFishTasty = false;
```

Full runnable code:

```
public class LiteralsDemo {
    public static void main(String[] args) {
        // ===== Integer Literals =====
        int decimal = 1234;
        int binary = 0b1010;          // 10 in decimal
        int hex = 0xFF;               // 255 in decimal
        int largeNumber = 1_000_000;

        // ===== Floating-Point Literals =====
        double pi = 3.14159;
```

```
        float gravity = 9.8f;
        double avogadro = 6.022e23;

        // ===== Character Literals =====
        char letter = 'A';
        char digit = '7';
        char smiley = '\u263A';       //

        // ===== Boolean Literals =====
        boolean isJavaFun = true;
        boolean isFishTasty = false;

        // ===== Output =====
        System.out.println("=== Integer Literals ===");
        System.out.println("Decimal: " + decimal);
        System.out.println("Binary (0b1010): " + binary);
        System.out.println("Hexadecimal (0xFF): " + hex);
        System.out.println("Large Number (with underscores): " + largeNumber);

        System.out.println("\n=== Floating-Point Literals ===");
        System.out.println("Double (pi): " + pi);
        System.out.println("Float (gravity): " + gravity);
        System.out.println("Scientific Notation (Avogadro's number): " + avogadro);

        System.out.println("\n=== Character Literals ===");
        System.out.println("Letter: " + letter);
        System.out.println("Digit: " + digit);
        System.out.println("Smiley (Unicode): " + smiley);

        System.out.println("\n=== Boolean Literals ===");
        System.out.println("Is Java fun? " + isJavaFun);
        System.out.println("Is fish tasty? " + isFishTasty);
    }
}
```

2.6.1 Reflection

Using the appropriate literal format increases code clarity and reduces errors. Underscores help make large numbers easier to read. Choosing the right literal suffix (`f` for floats, `0x` for hex) avoids confusion and unintended data type issues. Clear, consistent literal usage is a simple but important part of writing clean Java syntax.

2.7 Escape Sequences and Special Characters

In Java, **escape sequences** are special characters used inside string and character literals to represent otherwise hard-to-type or invisible characters. They begin with a backslash \.

Common Escape Sequences

Escape Sequence	Description	Example Output
`\n`	New line	Moves to the next line
`\t`	Tab	Inserts a horizontal tab
`\\`	Backslash	Prints a single backslash \
`\"`	Double quote	Prints a double quote "
`\'`	Single quote	Prints a single quote '
`\r`	Carriage return	Returns cursor to line start
`\b`	Backspace	Deletes the previous character

Examples

```
System.out.println("Hello\nWorld!");       // Prints on two lines
System.out.println("Column1\tColumn2");    // Tab space between words
System.out.println("C:\\Program Files");   // Prints C:\Program Files
System.out.println("She said, \"Java is fun!\""); // Prints quotes inside string
```

Output:

```
Hello
World!
Column1    Column2
C:\Program Files
She said, "Java is fun!"
```

Escape sequences are essential for formatting output and including special characters inside strings. A common mistake is forgetting to escape backslashes or quotes, which leads to syntax errors or incorrect output. Using escape sequences properly allows you to produce readable and well-structured console output and handle string data that contains special characters without breaking the code. Mastery of escape sequences improves your ability to manipulate text in Java programs efficiently.

Chapter 3.

Operators and Expressions

1. Arithmetic Operators
2. Assignment and Compound Assignment
3. Relational and Equality Operators
4. Logical Operators (`&&`, `||`, `!`)
5. Bitwise Operators (`&`, `|`, `^`, `~`)
6. Bit Shift Operators (`>>`, `<<`, `>>>`)
7. Unary Operators (`++`, `--`, `+`, `-`, `!`)
8. Operator Precedence and Associativity

3 Operators and Expressions

3.1 Arithmetic Operators

Java supports the basic arithmetic operators used to perform mathematical calculations on numeric values. These operators work on both integer and floating-point types.

Basic Arithmetic Operators

Operator	Description	Example	Result
`+`	Addition	`5 + 3`	`8`
`-`	Subtraction	`10 - 4`	`6`
`*`	Multiplication	`7 * 6`	`42`
`/`	Division	`20 / 5`	`4`
`%`	Modulus (remainder)	`17 % 3`	`2`

Sample Expressions

```
int a = 15;
int b = 4;

System.out.println(a + b);  // Output: 19
System.out.println(a - b);  // Output: 11
System.out.println(a * b);  // Output: 60
System.out.println(a / b);  // Output: 3 (integer division)
System.out.println(a % b);  // Output: 3 (remainder)
```

Division Behavior

- **Integer division** truncates the decimal part:

```
int result = 7 / 2;
System.out.println(result);  // Output: 3
```

Here, `7 / 2` evaluates to `3`, not `3.5`, because both operands are integers.

- **Floating-point division** retains decimals:

```
double result = 7.0 / 2;
System.out.println(result);  // Output: 3.5
```

At least one operand must be a floating-point type (`float` or `double`) to get a precise result.

Full runnable code:

```
public class ExpressionDemo {
    public static void main(String[] args) {
        // ===== Sample Expressions =====
```

```
        int a = 15;
        int b = 4;

        System.out.println("=== Basic Arithmetic ===");
        System.out.println("a + b = " + (a + b));  // 19
        System.out.println("a - b = " + (a - b));  // 11
        System.out.println("a * b = " + (a * b));  // 60
        System.out.println("a / b = " + (a / b));  // 3 (integer division)
        System.out.println("a % b = " + (a % b));  // 3 (remainder)

        // ===== Division Behavior =====
        System.out.println("\n=== Division Behavior ===");

        int intResult = 7 / 2;
        System.out.println("7 / 2 (int division) = " + intResult);  // 3

        double floatResult = 7.0 / 2;
        System.out.println("7.0 / 2 (floating-point) = " + floatResult);  // 3.5
    }
}
```

3.1.1 Reflection

Understanding how arithmetic operators work, especially division, is crucial to avoid subtle bugs. Using integers can unintentionally truncate results, so be mindful of operand types when performing division. The modulus operator `%` is also useful for tasks like checking even/odd numbers or wrapping values within a range. Mastering arithmetic operators is a foundational step in writing effective Java expressions.

3.2 Assignment and Compound Assignment

In Java, the **assignment operator** `=` is used to store a value in a variable. Additionally, **compound assignment operators** combine arithmetic with assignment for more concise code.

Basic Assignment

```
int a = 10;  // Assigns 10 to variable a
```

Compound Assignment Operators

These operators perform an operation on the variable and assign the result back to it:

Operator	Equivalent To	Example	Result
`+=`	`a = a + value`	`a += 5;` (if `a=10`)	`a = 15`
`-=`	`a = a - value`	`a -= 3;`	`a = 7`
`*=`	`a = a * value`	`a *= 2;`	`a = 20`
`/=`	`a = a / value`	`a /= 4;`	`a = 5`
`%=`	`a = a % value`	`a %= 3;`	`a = 2`

Example:

```
int a = 10;
a += 5;     // a is now 15
a *= 2;     // a is now 30
```

Operator Chaining

You can assign multiple variables in one statement by chaining assignments right to left:

```
int a, b, c;
a = b = c = 5;
System.out.println(a + ", " + b + ", " + c);  // Output: 5, 5, 5
```

Here, `c` is assigned `5`, then `b` is assigned `c`, and finally `a` is assigned `b`.

Full runnable code:

```
public class AssignmentDemo {
    public static void main(String[] args) {
        // ===== Basic Assignment =====
        int a = 10;  // Assign 10 to a
        System.out.println("Basic Assignment: a = " + a);

        // ===== Compound Assignment Operators =====
        a += 5;  // a = a + 5
        System.out.println("After a += 5: a = " + a);  // 15

        a -= 3;  // a = a - 3
        System.out.println("After a -= 3: a = " + a);  // 12

        a *= 2;  // a = a * 2
        System.out.println("After a *= 2: a = " + a);  // 24

        a /= 4;  // a = a / 4
        System.out.println("After a /= 4: a = " + a);  // 6

        a %= 3;  // a = a % 3
        System.out.println("After a %= 3: a = " + a);  // 0

        // ===== Operator Chaining =====
        int x, y, z;
        x = y = z = 5;  // Assign 5 to z, then y = z, then x = y
        System.out.println("Chained Assignment: x = " + x + ", y = " + y + ", z = " + z);
    }
```

```
}
```

3.2.1 Reflection

Compound assignments help write shorter, cleaner code, especially for operations updating the same variable repeatedly. However, overusing chained assignments or compound operators in complex expressions can reduce clarity, making code harder to read and debug. Striking a balance between compactness and readability is key to maintainable Java code.

3.3 Relational and Equality Operators

Java provides **relational operators** to compare values and return a boolean result (`true` or `false`). These are essential for decision-making in programs.

Relational Operators

Operator	Meaning	Example	Result
`<`	Less than	`5 < 10`	`true`
`>`	Greater than	`10 > 3`	`true`
`<=`	Less than or equal	`7 <= 7`	`true`
`>=`	Greater than or equal	`9 >= 12`	`false`

These operators are commonly used with numeric primitives (`int`, `double`, etc.).

Equality Operators

Operator	Meaning	Example	Result
`==`	Equal to	`5 == 5`	`true`
`!=`	Not equal to	`5 != 10`	`true`

For **primitive types**, `==` compares values directly. For example:

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

Comparing Objects (Strings)

For **objects** like `String`, `==` compares **references** (memory addresses), not the content. To compare the actual text, use `.equals()`:

```java
String s1 = new String("Java");
String s2 = new String("Java");

System.out.println(s1 == s2);          // false, different objects
System.out.println(s1.equals(s2));     // true, same content
```

Full runnable code:

```java
public class RelationalAndEqualityDemo {
    public static void main(String[] args) {
        // ===== Relational Operators =====
        int a = 5, b = 10, c = 7, d = 9;

        System.out.println("=== Relational Operators ===");
        System.out.println("a < b: " + (a < b));        // true
        System.out.println("b > a: " + (b > a));        // true
        System.out.println("c <= 7: " + (c <= 7));      // true
        System.out.println("d >= 12: " + (d >= 12));    // false

        // ===== Equality Operators (Primitive Types) =====
        int x = 5, y = 5, z = 10;

        System.out.println("\n=== Equality Operators (Primitives) ===");
        System.out.println("x == y: " + (x == y));      // true
        System.out.println("x != z: " + (x != z));      // true

        // ===== Comparing Objects (Strings) =====
        String s1 = new String("Java");
        String s2 = new String("Java");
        String s3 = "Java";
        String s4 = "Java";

        System.out.println("\n=== String Comparison ===");
        System.out.println("s1 == s2: " + (s1 == s2));               // false
        System.out.println("s1.equals(s2): " + s1.equals(s2));       // true

        System.out.println("s3 == s4: " + (s3 == s4));               // true (same string pool object)
        System.out.println("s3.equals(s4): " + s3.equals(s4));       // true
    }
}
```

3.3.1 Reflection

A common mistake is using `==` to compare objects, leading to unexpected false results even when contents match. Always use `.equals()` for object content comparison. For primitives, `==` works as expected. Understanding this distinction helps avoid subtle bugs in Java programs.

3.4 Logical Operators (`&&`, `||`, `!`)

3.4.1 Logical Operators (`&&`, `||`, `!`)

Logical operators are used in Java to combine or invert boolean expressions, which is essential for controlling program flow in conditional statements.

Logical AND (`&&`)

- Returns `true` if **both** operands are `true`.
- If the first operand is `false`, the second operand **is not evaluated** (short-circuiting).

Example:

```
boolean a = true;
boolean b = false;

System.out.println(a && b);  // Output: false
System.out.println(b && a);  // Output: false (second operand not evaluated)
```

Logical OR (`||`)

- Returns `true` if **at least one** operand is `true`.
- If the first operand is `true`, the second operand **is not evaluated** (short-circuiting).

Example:

```
boolean a = true;
boolean b = false;

System.out.println(a || b);  // Output: true (second operand not evaluated)
System.out.println(b || a);  // Output: true
```

Logical NOT (`!`)

- Inverts the boolean value: `true` becomes `false`, and `false` becomes `true`.

Example:

```
boolean a = false;
System.out.println(!a);  // Output: true
```

Short-Circuiting and Method Calls

Short-circuiting means that Java stops evaluating as soon as the result is known. This can affect method calls in conditions:

```
boolean checkA() {
    System.out.println("checkA called");
    return false;
}
```

```
boolean checkB() {
    System.out.println("checkB called");
    return true;
}

if (checkA() && checkB()) {
    System.out.println("Both true");
}
// Output:
// checkA called
// (checkB not called because checkA is false)
```

Because `checkA()` returns `false`, `checkB()` is never called.

Full runnable code:

```
public class LogicalOperatorsDemo {

    public static void main(String[] args) {
        // === Logical AND (&&) ===
        boolean a = true;
        boolean b = false;

        System.out.println("=== Logical AND (&&) ===");
        System.out.println("a && b: " + (a && b));  // false
        System.out.println("b && a: " + (b && a));  // false (a not evaluated)

        // === Logical OR (||) ===
        System.out.println("\n=== Logical OR (||) ===");
        System.out.println("a || b: " + (a || b));  // true (b not evaluated)
        System.out.println("b || a: " + (b || a));  // true

        // === Logical NOT (!) ===
        boolean c = false;
        System.out.println("\n=== Logical NOT (!) ===");
        System.out.println("!c: " + (!c));  // true

        // === Short-circuiting and method calls ===
        System.out.println("\n=== Short-Circuiting with && ===");
        if (checkA() && checkB()) {
            System.out.println("Both returned true");
        }

        System.out.println("\n=== Short-Circuiting with || ===");
        if (checkA() || checkB()) {
            System.out.println("At least one returned true");
        }
    }

    static boolean checkA() {
        System.out.println("checkA called");
        return false;
    }

    static boolean checkB() {
        System.out.println("checkB called");
        return true;
```

```
    }
}
```

3.4.2 Reflection

Logical operators combined with short-circuiting allow efficient condition checks and prevent unnecessary method executions. Understanding this behavior helps avoid side effects and improves program performance. Use `&&`, `||`, and `!` thoughtfully in conditions to create clear, concise, and correct logic.

3.5 Bitwise Operators (`&`, `|`, `^`, `~`)

Bitwise operators work at the **binary level**, manipulating individual bits of integer values. They are powerful tools for low-level programming tasks such as flags, permissions, and performance optimizations.

Common Bitwise Operators

Operator	Description	Example
`&`	Bitwise AND	`0b1100 & 0b1010`
|	Bitwise OR	0b1100 | 0b1010
`^`	Bitwise XOR (exclusive)	`0b1100 ^ 0b1010`
`~`	Bitwise NOT (complement)	`~0b1100`

Bit-Level Examples

Let's consider two 4-bit numbers:

```
int a = 0b1100;  // binary 1100 (decimal 12)
int b = 0b1010;  // binary 1010 (decimal 10)
```

Operation	Binary Result	Decimal Result
`a & b` (AND)	`1000`	8
a | b (OR)	`1110`	14
`a ^ b` (XOR)	`0110`	6
`~a` (NOT)	`...0011`	(depends on bit-width, e.g., -13 for 32-bit)

Truth Table for `&`, `|`, and `^`

A	B	A & B	A | B	A ^ B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Full runnable code:

```
public class BitwiseOperatorsDemo {
    public static void main(String[] args) {
        int a = 0b1100;  // 12 in decimal
        int b = 0b1010;  // 10 in decimal

        System.out.println("=== Bit-Level Values ===");
        System.out.println("a (0b1100): " + a + " => " + toBinary(a));
        System.out.println("b (0b1010): " + b + " => " + toBinary(b));

        // Bitwise AND
        int andResult = a & b;
        System.out.println("\nBitwise AND (a & b): " + andResult + " => " + toBinary(andResult));

        // Bitwise OR
        int orResult = a | b;
        System.out.println("Bitwise OR (a | b): " + orResult + " => " + toBinary(orResult));

        // Bitwise XOR
        int xorResult = a ^ b;
        System.out.println("Bitwise XOR (a ^ b): " + xorResult + " => " + toBinary(xorResult));

        // Bitwise NOT
        int notResult = ~a;
        System.out.println("Bitwise NOT (~a): " + notResult + " => " + toBinary(notResult));
    }

    // Helper method to display binary with 32 bits
    private static String toBinary(int value) {
        return String.format("%32s", Integer.toBinaryString(value)).replace(' ', '0');
    }
}
```

3.5.1 Reflection

Bitwise operators are niche but powerful, enabling efficient manipulation of individual bits. Common real-world uses include:

- Managing **flags** and **permissions** compactly (e.g., file permissions).
- Performing **masking** to extract or modify specific bits.
- Optimizing performance-critical code where direct bit control is beneficial.

While often unnecessary in everyday Java programming, mastering bitwise operators opens up advanced capabilities for working with binary data and system-level tasks.

3.6 Bit Shift Operators (>>, <<, >>>)

Bit shift operators move the bits of an integer value to the left or right, effectively multiplying or dividing the number by powers of two at the binary level. They are useful for optimization and low-level data manipulation.

Left Shift (<<)

- Shifts bits to the **left** by a specified number of positions.
- Zeros are shifted into the least significant bits.
- Equivalent to multiplying by 2 for each shift position.

Example:

```
int a = 0b0001_0101;  // decimal 21
int leftShift = a << 2; // shifts bits 2 places left
System.out.println(Integer.toBinaryString(leftShift));  // Output: 1010100 (decimal 84)
```

Right Shift (>>) Arithmetic Shift

- Shifts bits to the **right** by a specified number of positions.
- Preserves the **sign bit** (leftmost bit), maintaining the number's sign.
- Used for signed integers.

Example:

```
int b = 0b1111_1000;  // decimal -8 (two's complement)
int rightShift = b >> 2;
System.out.println(Integer.toBinaryString(rightShift));  // Output: 11111110 (decimal -2)
```

Unsigned Right Shift (>>>) Logical Shift

- Shifts bits to the **right**, filling zeros in the leftmost bits.
- Does **not** preserve the sign bit.
- Used to treat numbers as unsigned.

Example:

```
int c = 0b1111_1000;  // decimal -8
int unsignedShift = c >>> 2;
System.out.println(Integer.toBinaryString(unsignedShift)); // Output: 00111111 (decimal 1073741821)
```

Full runnable code:

```java
public class ShiftOperatorsDemo {
    public static void main(String[] args) {
        // === Left Shift (<<) ===
        int a = 0b0001_0101;  // 21 in decimal
        int leftShift = a << 2;

        System.out.println("=== Left Shift (<<) ===");
        System.out.println("Original a        : " + toBinary(a) + " = " + a);
        System.out.println("a << 2 (mult by 4): " + toBinary(leftShift) + " = " + leftShift);

        // === Arithmetic Right Shift (>>) ===
        int b = (byte) 0b1111_1000;  // -8 in decimal (signed byte extended to int)
        int rightShift = b >> 2;

        System.out.println("\n=== Right Shift (>>) ===");
        System.out.println("Original b        : " + toBinary(b) + " = " + b);
        System.out.println("b >> 2            : " + toBinary(rightShift) + " = " + rightShift);

        // === Unsigned Right Shift (>>>) ===
        int c = (byte) 0b1111_1000;  // -8 again
        int unsignedShift = c >>> 2;

        System.out.println("\n=== Unsigned Right Shift (>>>) ===");
        System.out.println("Original c        : " + toBinary(c) + " = " + c);
        System.out.println("c >>> 2           : " + toBinary(unsignedShift) + " = " + unsignedShift);
    }

    // Helper: returns 32-bit binary string
    private static String toBinary(int value) {
        return String.format("%32s", Integer.toBinaryString(value)).replace(' ', '0');
    }
}
```

3.6.1 Reflection

Bit shift operators provide efficient ways to multiply or divide integers by powers of two, which is faster than arithmetic operations in some contexts. They are critical in low-level programming, such as encoding/decoding protocols, graphics, and cryptography. Understanding arithmetic vs. logical shifts is key when dealing with signed vs. unsigned data to avoid unexpected behavior. Mastery of these operators enables more control over binary data and optimized performance in Java programs.

3.7 Unary Operators (++, --, +, -, !)

Unary operators operate on a single operand and are commonly used to modify or evaluate values quickly.

Unary Operators in Java

- `++` : Increment operator — increases an integer value by 1.
- `--` : Decrement operator — decreases an integer value by 1.
- `+` : Unary plus — indicates a positive value (usually redundant).
- `-` : Unary minus — negates a numeric value.
- `!` : Logical NOT — inverts a boolean value (`true` to `false`, and vice versa).

Prefix vs. Postfix ++ and --

- **Prefix (`++a` or `--a`)**: The variable is incremented or decremented *before* the expression is evaluated.
- **Postfix (`a++` or `a--`)**: The current value is used in the expression *before* the increment or decrement happens.

Example:

```
int a = 5;
int x = ++a;  // a is incremented to 6, then x is assigned 6
int y = a++;  // y is assigned 6, then a is incremented to 7
```

Subtle Behavior in Expressions

Expressions combining prefix and postfix operators can be confusing:

```
int a = 3;
int result = ++a + a++;
// Step 1: ++a increments a to 4, then uses 4
// Step 2: a++ uses current a (4), then increments to 5
// result = 4 + 4 = 8
System.out.println(result);  // Output: 8
```

Full runnable code:

```
public class UnaryOperatorsDemo {
    public static void main(String[] args) {
        // Basic unary operators
        int a = 5;
        System.out.println("Initial a: " + a);

        int x = ++a;  // prefix increment: a=6, x=6
        System.out.println("After ++a: a = " + a + ", x = " + x);

        int y = a++;  // postfix increment: y=6, a=7
        System.out.println("After a++: a = " + a + ", y = " + y);

        // Unary plus and minus
        int positive = +a;
        int negative = -a;
        System.out.println("Unary plus (+a): " + positive);
        System.out.println("Unary minus (-a): " + negative);

        // Logical NOT
        boolean flag = true;
```

```
        System.out.println("Original flag: " + flag);
        System.out.println("Logical NOT (!flag): " + !flag);

        // Subtle behavior in expressions combining prefix and postfix
        int b = 3;
        int result = ++b + b++;
        // ++b increments b to 4, returns 4
        // b++ returns 4, then increments to 5
        // result = 4 + 4 = 8
        System.out.println("b after expression (++b + b++): " + b);
        System.out.println("Result of ++b + b++: " + result);
    }
}
```

3.7.1 Reflection

Understanding the difference between prefix and postfix is crucial for writing clear, bug-free code. Misusing them in complex expressions can lead to unexpected results and maintenance challenges. It's often best to keep increments and decrements on separate lines for clarity. Unary operators remain essential tools for concise value changes and logical negation in Java.

3.8 Operator Precedence and Associativity

Understanding **operator precedence** and **associativity** is essential for predicting how Java evaluates complex expressions without parentheses.

Operator Precedence

Precedence determines the order in which operators are evaluated in an expression. Operators with higher precedence are evaluated before operators with lower precedence.

Example without parentheses:

```
int result = 5 + 3 * 2;  // Multiplication (*) has higher precedence than addition (+)
System.out.println(result);  // Output: 11
```

Here, `3 * 2` is evaluated first, then added to `5`.

Using Parentheses

Parentheses can override the default precedence to specify the intended order explicitly:

```
int result = (5 + 3) * 2;  // Parentheses force addition first
System.out.println(result);  // Output: 16
```

Operator Associativity

Associativity determines the order in which operators of the **same precedence** are evaluated:

- Most operators in Java are **left-associative**, meaning evaluation proceeds left to right.
- Assignment operators and the unary operators are **right-associative**.

Example:

```
int a = 10;
int b = 5;
int c = 2;
int result = a - b - c;  // Evaluated as (a - b) - c
System.out.println(result);  // Output: 3
```

Full runnable code:

```
public class OperatorPrecedenceDemo {
    public static void main(String[] args) {
        // Operator precedence: multiplication before addition
        int result1 = 5 + 3 * 2;  // 3 * 2 = 6, then 5 + 6 = 11
        System.out.println("5 + 3 * 2 = " + result1);

        // Using parentheses to override precedence
        int result2 = (5 + 3) * 2;  // (5 + 3) = 8, then 8 * 2 = 16
        System.out.println("(5 + 3) * 2 = " + result2);

        // Operator associativity: left to right for subtraction
        int a = 10, b = 5, c = 2;
        int result3 = a - b - c;  // (10 - 5) - 2 = 3
        System.out.println("10 - 5 - 2 = " + result3);

        // Right-associative example: assignment operators
        int x, y, z;
        x = y = z = 100;  // Right to left: z=100, y=100, x=100
        System.out.println("x = " + x + ", y = " + y + ", z = " + z);
    }
}
```

Operator Precedence Table

Java's operator precedence table is available in official documentation and reference sites, showing exact precedence and associativity for all operators. This table is vital for writing and understanding complex expressions correctly.

3.8.1 Reflection

While Java follows clear rules for precedence and associativity, relying solely on these can make code hard to read and error-prone. Using parentheses enhances readability and prevents subtle bugs by making the order of evaluation explicit. Always prioritize clarity in your expressions to produce maintainable and reliable Java code.

Chapter 4.

Control Flow Syntax

1. `if`, `else if`, and `else`
2. `switch` Statements and Fallthrough
3. Pattern Matching with `instanceof` (Java 16)
4. `while`, `do-while`, and `for` Loops
5. Enhanced `for` Loop (for-each)
6. Control Statements: `break`, `continue`, and Labels

4 Control Flow Syntax

4.1 `if`, `else if`, and `else`

Conditional branching is a fundamental control flow structure in Java that allows the program to execute different code blocks based on conditions.

Basic Syntax

The `if` statement checks a boolean condition. If `true`, its block executes. Optionally, you can add an `else if` to check additional conditions, and an `else` block as a fallback.

```
int score = 75;

if (score >= 90) {
    System.out.println("Grade: A");
} else if (score >= 80) {
    System.out.println("Grade: B");
} else if (score >= 70) {
    System.out.println("Grade: C");
} else {
    System.out.println("Grade: F");
}
```

How Conditions Are Evaluated

- The program evaluates conditions **in order**, top to bottom.
- The first `true` condition's block executes.
- The rest are skipped.
- If none match, the `else` block executes if present.

Nested `if` Statements

You can nest `if` inside another `if` to check complex conditions:

```
int age = 20;
boolean hasLicense = true;

if (age >= 18) {
    if (hasLicense) {
        System.out.println("Allowed to drive.");
    } else {
        System.out.println("Needs a license.");
    }
} else {
    System.out.println("Too young to drive.");
}
```

Full runnable code:

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

```
        // Example 1: Grading using if-else if-else
        int score = 75;

        if (score >= 90) {
            System.out.println("Grade: A");
        } else if (score >= 80) {
            System.out.println("Grade: B");
        } else if (score >= 70) {
            System.out.println("Grade: C");
        } else {
            System.out.println("Grade: F");
        }

        // Example 2: Nested if statements
        int age = 20;
        boolean hasLicense = true;

        if (age >= 18) {
            if (hasLicense) {
                System.out.println("Allowed to drive.");
            } else {
                System.out.println("Needs a license.");
            }
        } else {
            System.out.println("Too young to drive.");
        }
    }
}
```

Reflection

Proper indentation and clear structure make `if-else` logic easy to follow and maintain. Omitting an `else` may cause missed cases, leading to unexpected behavior. Be cautious with overlapping conditions and always test edge cases. Using `if-else if-else` correctly ensures your program handles all possibilities gracefully.

4.2 `switch` Statements and Fallthrough

The `switch` statement provides a clean way to execute different code blocks based on the value of a variable, often used as an alternative to long `if-else if` chains when testing one variable against multiple possible values.

Basic Syntax

```
switch (variable) {
    case value1:
        // code block
        break;
    case value2:
        // code block
```

```
        break;
    default:
        // code block executed if no case matches
}
```

- The **`switch` expression** evaluates once.
- Execution jumps to the matching `case`.
- The **`break`** statement prevents the program from continuing ("falling through") to the next case.
- The **`default`** case runs if no matching case is found; it's optional but recommended.

Example with `break`

```
int day = 3;
String dayName;

switch (day) {
    case 1:
        dayName = "Monday";
        break;
    case 2:
        dayName = "Tuesday";
        break;
    case 3:
        dayName = "Wednesday";
        break;
    default:
        dayName = "Invalid day";
}
System.out.println(dayName);  // Output: Wednesday
```

Intentional Fallthrough

If you omit `break`, execution continues into subsequent cases:

```
int month = 12;

switch (month) {
    case 12:
    case 1:
    case 2:
        System.out.println("Winter");
        break;
    case 3:
    case 4:
    case 5:
        System.out.println("Spring");
        break;
    default:
        System.out.println("Other season");
}
```

Here, months 12, 1, and 2 all print "Winter" because of fallthrough without intervening `break`s.

Full runnable code:

```
public class SwitchDemo {
    public static void main(String[] args) {
        // Example 1: switch with break
        int day = 3;
        String dayName;

        switch (day) {
            case 1:
                dayName = "Monday";
                break;
            case 2:
                dayName = "Tuesday";
                break;
            case 3:
                dayName = "Wednesday";
                break;
            default:
                dayName = "Invalid day";
        }
        System.out.println("Day " + day + " is " + dayName);  // Output: Wednesday

        // Example 2: Intentional fallthrough
        int month = 12;
        System.out.print("Month " + month + " is in ");

        switch (month) {
            case 12:
            case 1:
            case 2:
                System.out.println("Winter");
                break;
            case 3:
            case 4:
            case 5:
                System.out.println("Spring");
                break;
            default:
                System.out.println("Other season");
        }
    }
}
```

`switch` is often cleaner and more readable than multiple `if-else if` statements when testing a single variable against many discrete values. It can improve maintainability and reduce errors related to complex nested conditions. However, `switch` works only with discrete values (like primitives, enums, and strings), while `if` statements handle a broader range of conditions, including ranges and complex expressions. Being mindful of `break` usage is crucial to prevent unexpected fallthrough bugs.

4.2.1 Switch Expressions (Java 14)

Starting with Java 14, the traditional `switch` statement was enhanced with **switch expressions**, introducing a more concise and expressive way to handle multiple cases and return values directly.

What Are Switch Expressions?

Unlike the classic `switch` statement, which is used primarily for control flow, **switch expressions** can produce a value. This means you can assign the result of a `switch` expression directly to a variable.

Basic Syntax

```
int day = 3;

String dayName = switch (day) {
    case 1 -> "Monday";
    case 2 -> "Tuesday";
    case 3 -> "Wednesday";
    default -> "Invalid day";
};

System.out.println(dayName);  // Output: Wednesday
```

- The `->` arrow replaces the colon (`:`) and removes the need for explicit `break` statements.
- Each case returns a value.
- The entire `switch` is an expression that evaluates to a result.
- The `default` case is required if not all possible cases are covered.

Multiple Labels in One Case

Switch expressions support grouping multiple labels to share the same result:

```
String season = switch (month) {
    case 12, 1, 2 -> "Winter";
    case 3, 4, 5 -> "Spring";
    default -> "Other";
};
```

Block Bodies in Switch Expressions

For more complex logic, use a block with `yield` to return a value:

```
String category = switch (score) {
    case 90, 100 -> "Excellent";
    case 80, 89 -> "Good";
    default -> {
        System.out.println("Score below 80");
        yield "Needs Improvement";
    }
};
```

Full runnable code:

```
public class SwitchExpressionDemo {
    public static void main(String[] args) {
        int day = 3;
        String dayName = switch (day) {
            case 1 -> "Monday";
            case 2 -> "Tuesday";
            case 3 -> "Wednesday";
            default -> "Invalid day";
        };
        System.out.println("Day " + day + " is " + dayName);  // Output: Wednesday

        int month = 12;
        String season = switch (month) {
            case 12, 1, 2 -> "Winter";
            case 3, 4, 5 -> "Spring";
            default -> "Other";
        };
        System.out.println("Month " + month + " is in " + season);

        int score = 75;
        String category = switch (score) {
            case 90, 100 -> "Excellent";
            case 80, 89 -> "Good";
            default -> {
                System.out.println("Score below 80");
                yield "Needs Improvement";
            }
        };
        System.out.println("Score " + score + " category: " + category);
    }
}
```

4.2.2 Reflection

Switch expressions simplify code by combining control flow and value production, reducing verbosity and errors like missing `breaks`. They make your logic clearer and more concise. However, since switch expressions require Java 14 or later, ensure your environment supports them before use. When used appropriately, switch expressions improve readability and maintainability in decision-making code.

4.3 Pattern Matching with `instanceof` (Java 16)

Traditionally, the `instanceof` operator in Java is used to check if an object is an instance of a specific class or interface. However, this often requires an explicit type cast after the check, making the code verbose and repetitive.

Traditional `instanceof` Usage

```
Object obj = "Hello, World!";

if (obj instanceof String) {
    String str = (String) obj;  // explicit cast required
    System.out.println(str.toUpperCase());
}
```

Here, after confirming `obj` is a `String`, you must cast it before using `String` methods like `toUpperCase()`.

Pattern Matching with `instanceof`

Java 16 introduced **pattern matching for `instanceof`**, which combines the check and cast in one step:

```
Object obj = "Hello, World!";

if (obj instanceof String str) {
    // str is automatically cast to String within this block
    System.out.println(str.toUpperCase());
}
```

The variable `str` is declared and initialized if the `instanceof` check succeeds, eliminating the need for a separate cast.

Full runnable code:

```
public class InstanceofDemo {
    public static void main(String[] args) {
        Object obj1 = "Hello, World!";
        Object obj2 = 123;

        // Traditional instanceof usage with explicit cast
        if (obj1 instanceof String) {
            String str = (String) obj1;  // explicit cast required
            System.out.println("Traditional: " + str.toUpperCase());
        }

        // Pattern matching with instanceof (Java 16+)
        if (obj2 instanceof String str) {
            // This block won't execute because obj2 is not a String
            System.out.println("Pattern matching: " + str.toUpperCase());
        } else {
            System.out.println("Pattern matching: obj2 is not a String");
        }
    }
}
```

Benefits of Pattern Matching

- **Simplifies code** by removing redundant casts.
- **Improves readability** by keeping the condition and the cast close together.

- **Enhances safety** by limiting the scope of the pattern variable to the `if` block, reducing accidental misuse elsewhere.
- **Supports cleaner control flow** when combined with `else` or nested conditions.

4.3.1 Reflection

Pattern matching with `instanceof` streamlines common type-checking scenarios, making Java code more concise and expressive. It encourages safer coding practices by tying type checks directly to variable declarations, avoiding the boilerplate and risks associated with manual casting. As a result, developers write cleaner, easier-to-understand code when working with polymorphic types.

4.4 `while`, `do-while`, and `for` Loops

Loops are essential in Java for executing code repeatedly based on a condition. Java provides three common loop types: `while`, `do-while`, and `for`. Each serves specific use cases depending on when the condition should be checked and how the loop is structured.

`while` Loop

The `while` loop checks its condition *before* each iteration. If the condition is `true`, the loop body runs; otherwise, it exits immediately.

```
int count = 1;
while (count <= 5) {
    System.out.println("Count: " + count);
    count++;
}
```

Use case: When the number of iterations is unknown and the loop may not run at all if the condition is initially `false`.

`do-while` Loop

The `do-while` loop executes the loop body *at least once* before checking the condition at the end of each iteration.

```
int count = 1;
do {
    System.out.println("Count: " + count);
    count++;
} while (count <= 5);
```

Use case: When you need the loop body to run at least once regardless of the condition.

`for` Loop

The `for` loop is compact and ideal when the number of iterations is known or controlled by a counter.

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

It has three parts:

- **Initialization:** `int i = 1` — runs once before looping starts.
- **Condition:** `i <= 5` — checked before each iteration.
- **Update:** `i++` — runs after each iteration.

Full runnable code:

```
public class LoopExamples {
    public static void main(String[] args) {
        System.out.println("While loop:");
        int count1 = 1;
        while (count1 <= 5) {
            System.out.println("Count: " + count1);
            count1++;
        }

        System.out.println("\nDo-while loop:");
        int count2 = 1;
        do {
            System.out.println("Count: " + count2);
            count2++;
        } while (count2 <= 5);

        System.out.println("\nFor loop:");
        for (int i = 1; i <= 5; i++) {
            System.out.println("Count: " + i);
        }
    }
}
```

4.4.1 Reflection

- Use `while` when the loop should run only if a condition is true upfront.
- Use `do-while` if the loop must run at least once before checking.
- Use `for` when you have a clear iteration count or need a concise loop control structure.

Choosing the appropriate loop improves readability and logic clarity. Misusing loops—like using `while` when the body must run once—can cause bugs or unnecessarily complex code. Understanding these differences empowers you to write efficient, predictable loops in your Java programs.

4.5 Enhanced `for` Loop (for-each)

The enhanced `for` loop, also known as the **for-each** loop, provides a simplified syntax for iterating over arrays and collections without the need to manage an index variable explicitly.

Syntax

```
int[] numbers = {1, 2, 3, 4, 5};

for (int num : numbers) {
    System.out.println(num);
}
```

Here, `num` takes on each value in the `numbers` array, one by one, until the loop completes.

Comparison with Regular `for` Loop

The equivalent traditional loop looks like this:

```
for (int i = 0; i < numbers.length; i++) {
    System.out.println(numbers[i]);
}
```

The enhanced for loop is **more concise** and eliminates boilerplate code such as index initialization, condition checking, and incrementing.

Usage with Collections

The for-each loop works similarly with any class implementing the `Iterable` interface, such as `ArrayList`:

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

for (String name : names) {
    System.out.println(name);
}
```

Full runnable code:

```
import java.util.List;

public class ForEachDemo {
    public static void main(String[] args) {
        // Using enhanced for-each with an array
        int[] numbers = {1, 2, 3, 4, 5};
        System.out.println("Enhanced for-each with array:");
        for (int num : numbers) {
            System.out.println(num);
        }

        // Equivalent traditional for loop
        System.out.println("\nTraditional for loop with array:");
        for (int i = 0; i < numbers.length; i++) {
```

```
            System.out.println(numbers[i]);
        }

        // Using enhanced for-each with a List
        List<String> names = List.of("Alice", "Bob", "Charlie");
        System.out.println("\nEnhanced for-each with List:");
        for (String name : names) {
            System.out.println(name);
        }
    }
}
```

Reflection

The enhanced for loop improves **readability** by focusing on elements rather than indices, reducing common mistakes like off-by-one errors. However, it does have limitations:

- You **cannot modify the index** during iteration.
- You **cannot remove elements** from the collection safely.
- It's **not suitable** if you need to iterate in a non-sequential or reverse order.

Despite these, the enhanced for loop is an excellent choice for simple, safe, and clear iteration over arrays and collections in Java.

4.6 Control Statements: `break`, `continue`, and Labels

Java provides control statements like `break` and `continue` to manage loop execution more precisely. These statements help alter the flow inside loops, and labeled statements extend this control especially in nested loops.

`break` Statement

The `break` statement immediately exits the nearest enclosing loop or `switch` block:

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

Here, the loop terminates early when `i` equals 3.

`continue` Statement

The `continue` statement skips the current iteration and proceeds to the next one:

```java
for (int i = 1; i <= 5; i++) {
    if (i == 3) {
        continue;  // Skip printing when i is 3
    }
    System.out.println(i);
}
// Output: 1 2 4 5
```

The iteration where `i` is 3 is skipped.

Labeled Statements

In nested loops, `break` and `continue` only affect the innermost loop by default. Labels allow you to specify which loop to control.

```java
outerLoop:
for (int i = 1; i <= 3; i++) {
    innerLoop:
    for (int j = 1; j <= 3; j++) {
        if (i == 2 && j == 2) {
            break outerLoop;  // Exit the outer loop entirely
        }
        System.out.println("i=" + i + ", j=" + j);
    }
}
```

This code exits both loops when `i` is 2 and `j` is 2.

Full runnable code:

```java
public class BreakContinueExample {
    public static void main(String[] args) {
        System.out.println("Break example:");
        for (int i = 1; i <= 5; i++) {
            if (i == 3) {
                break;  // Exit loop when i is 3
            }
            System.out.print(i + " ");
        }
        System.out.println();

        System.out.println("Continue example:");
        for (int i = 1; i <= 5; i++) {
            if (i == 3) {
                continue;  // Skip printing when i is 3
            }
            System.out.print(i + " ");
        }
        System.out.println();

        System.out.println("Labeled break example:");
        outerLoop:
        for (int i = 1; i <= 3; i++) {
            innerLoop:
            for (int j = 1; j <= 3; j++) {
                if (i == 2 && j == 2) {
```

```
                    break outerLoop;  // Exit the outer loop entirely
                }
                System.out.println("i=" + i + ", j=" + j);
            }
        }
    }
}
```

4.6.1 Reflection

While `break` and `continue` can simplify complex looping logic by allowing early exits or skipping iterations, overusing them—especially with labeled loops—can make code harder to read and maintain. Use these statements judiciously and document their intent clearly to keep your code understandable and bug-free.

Chapter 5.

Methods and Scope

1. Method Declaration Syntax
2. Parameters, Return Types, and `return`
3. Method Overloading
4. Recursion Syntax
5. Scope Rules and Variable Lifetime

5 Methods and Scope

5.1 Method Declaration Syntax

In Java, a method is a reusable block of code that performs a specific task. Methods are declared inside classes and can accept parameters, return values, or perform actions without returning anything.

Basic Syntax

```
accessModifier returnType methodName(parameterList) {
    // method body
}
```

- **accessModifier**: Determines visibility (`public`, `private`, etc.).
- **returnType**: Type of value returned (e.g., `int`, `void`, `String`).
- **methodName**: Follows camelCase convention and describes what the method does.
- **parameterList**: A comma-separated list of input variables with type annotations.

Example

```
public class Calculator {

    public int add(int a, int b) {
        int result = a + b;
        return result;
    }

    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println(calc.add(5, 3));  // Output: 8
    }
}
```

In this example:

- The method `add` is declared as `public` (accessible from other classes),
- It returns an `int`,
- It takes two `int` parameters, and
- It returns the sum of those two values.

Method Signature

The *method signature* consists of the method's name and parameter types. It's how the Java compiler differentiates methods. For example:

```
public void log(String message)
public void log(String message, int level)
```

These are two different methods due to different parameter lists.

Reflection

Good method naming is crucial—names should be descriptive but concise. Avoid overly generic names like `doStuff()` or `handle()` unless within a very narrow, self-evident context.

Access modifiers affect code organization. Use `private` for helper methods and `public` for functionality exposed to other classes.

A well-declared method improves code clarity, modularity, and testability. Understanding method declaration syntax lays the groundwork for building scalable, maintainable Java applications.

5.2 Parameters, Return Types, and `return`

In Java, methods can accept *parameters* (inputs) and optionally *return* a value. These features enable methods to process dynamic data and produce results, supporting modular, reusable code.

Defining Parameters

Parameters are defined inside parentheses in the method declaration. Each parameter has a type and a name:

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

The `greet` method takes one parameter, `name`, of type `String`.

You can define multiple parameters by separating them with commas:

```
public int multiply(int a, int b) {
    return a * b;
}
```

Return Types and `return`

Every method must declare a return type:

- Use `void` if the method does not return a value.
- Specify a type (`int`, `double`, `String`, etc.) if it returns a value.

Example with return value:

```
public double areaOfCircle(double radius) {
    return Math.PI * radius * radius;
}
```

The `return` statement:

- Ends the method's execution.
- Sends a value back to the caller.

```
int result = multiply(4, 5);  // result = 20
```

Important: If a method has a return type other than `void`, it *must* include a `return` statement that returns a matching type. Missing a return in such cases causes a compilation error.

Example of a void method:

```
public void printLine() {
    System.out.println("----------");
}
```

Full runnable code:

```
public class MethodExample {

    // Method with one parameter, no return (void)
    public void greet(String name) {
        System.out.println("Hello, " + name + "!");
    }

    // Method with multiple parameters and a return value
    public int multiply(int a, int b) {
        return a * b;
    }

    // Method returning a double value
    public double areaOfCircle(double radius) {
        return Math.PI * radius * radius;
    }

    // Void method with no parameters
    public void printLine() {
        System.out.println("----------");
    }

    public static void main(String[] args) {
        MethodExample example = new MethodExample();

        example.greet("Alice");

        int product = example.multiply(4, 5);
        System.out.println("4 * 5 = " + product);

        double area = example.areaOfCircle(3);
        System.out.println("Area of circle with radius 3 = " + area);

        example.printLine();
    }
}
```

Reflection

Choosing meaningful parameter names improves code clarity. Avoid vague names like `x` or `val` unless in mathematical or short-lived contexts.

A well-designed method should:

- Do one thing only (single-responsibility principle),
- Accept only necessary parameters, and
- Clearly indicate its result through the return value.

Understanding how parameters and `return` work is fundamental to writing functional and expressive Java methods.

5.3 Method Overloading

Method overloading allows you to define multiple methods with the same name but different parameter lists in the same class. This enables flexibility while keeping method names intuitive and consistent.

Basic Syntax and Examples

Java distinguishes overloaded methods by the **number**, **type**, or **order** of parameters.

```
public class Printer {

    // Overloaded methods
    public void print(String message) {
        System.out.println("String: " + message);
    }

    public void print(int number) {
        System.out.println("Integer: " + number);
    }

    public void print(String message, int number) {
        System.out.println("String and Integer: " + message + ", " + number);
    }

    public void print(int number, String message) {
        System.out.println("Integer and String: " + number + ", " + message);
    }

    public static void main(String[] args) {
        Printer p = new Printer();
        p.print("Hello");
        p.print(42);
        p.print("Score", 100);
        p.print(100, "Score");
    }
}
```

Each method has a unique signature (name + parameter types), even though they all share the same name.

How Java Chooses the Right Method

At compile time, Java selects the method that best matches the provided argument types. If multiple methods match equally well, a compilation error may occur due to ambiguity.

For example:

```
p.print(10);  // Calls print(int number)
```

If only `print(String)` existed, Java would convert `10` to a `String` and call that one. But with both versions, it chooses the exact type match.

Reflection

Method overloading can improve **code readability** and **flexibility**, especially for utility methods or constructors. However, excessive or confusing overloads can make code harder to maintain. Be cautious when overloading methods with subtle differences in parameter types, like `float` vs. `double`, or argument order, as these can lead to unintended calls and hard-to-spot bugs.

Use overloading when it genuinely simplifies usage—not just to show off multiple method forms.

5.4 Recursion Syntax

Recursion is the process in which a method calls itself to solve a smaller part of the problem. It is a powerful concept, but must be used carefully to avoid stack overflow errors due to infinite recursion.

Example: Factorial Calculation

Here's a classic example using recursion to calculate the factorial of a number:

```
public class RecursionExample {

    public static int factorial(int n) {
        // Base case
        if (n == 0) {
            return 1;
        }
        // Recursive call
        return n * factorial(n - 1);
    }

    public static void main(String[] args) {
        System.out.println(factorial(5)); // Output: 120
```

```
    }
}
```

In this example:

- The **base case** (`n == 0`) prevents infinite recursion.
- The **recursive step** (`n * factorial(n - 1)`) gradually reduces the problem size.

Without a base case, the method would keep calling itself indefinitely, causing a runtime error (`StackOverflowError`).

Reflection on When to Use Recursion

Use recursion when:

- The problem is naturally recursive (e.g., tree traversal, backtracking, combinatorics).
- The recursive solution is clearer and more elegant than loops.

Avoid recursion when:

- The depth of recursion is large and could lead to stack overflow.
- An iterative approach is simpler and more efficient in terms of memory.

For instance, computing the nth Fibonacci number recursively is readable but inefficient due to redundant calculations:

```
public static int fibonacci(int n) {
    if (n <= 1) return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Recursion showcases elegant syntax and conceptual power, but understanding the role of **termination** and **problem reduction** is crucial. Always make sure a recursive method moves toward the base case and eventually stops.

5.5 Scope Rules and Variable Lifetime

In Java, **scope** refers to the region of the program where a variable is accessible. Java enforces strict scoping rules to promote clarity and reduce errors. Understanding variable scope is crucial for writing safe and predictable code.

5.5.1 Types of Scope

1. **Block Scope** A variable declared inside a pair of `{}` braces is visible only within that block.

```
if (true) {
    int x = 5;
    System.out.println(x); // Valid
}
// System.out.println(x); // Error: x is out of scope
```

2. **Method Scope** A variable declared within a method exists only for the duration of that method call.

```
public void show() {
    int y = 10;
    System.out.println(y);
}
// y cannot be accessed outside the method
```

3. **Class Scope (Instance and Static Fields)** Variables declared directly in a class are accessible by all methods within that class, depending on access modifiers.

```
public class Counter {
    private int count = 0; // Instance variable (class scope)

    public void increment() {
        count++;
    }

    public void display() {
        System.out.println(count);
    }
}
```

5.5.2 Variable Lifetime

- **Local variables** (declared in methods/blocks) exist temporarily—created when the block/method is entered and destroyed when exited.
- **Instance variables** persist as long as the object exists.
- **Static variables** exist for the life of the class in memory.

5.5.3 Shadowing and Naming Conflicts

Shadowing happens when a variable declared in a nested block or method has the same name as one in an outer scope. The inner variable "shadows" the outer one, potentially causing confusion:

```
public class ShadowExample {
    int value = 10;

    public void printValue() {
```

```
        int value = 20; // Shadows the instance variable
        System.out.println(value); // Prints 20
    }
}
```

Use the `this` keyword to refer to the instance variable when it's shadowed:

```
System.out.println(this.value); // Refers to instance variable
```

Full runnable code:

```
public class ScopeExample {

    // Class scope (instance variable)
    private int count = 0;

    public void demonstrateBlockScope() {
        if (true) {
            int x = 5;  // Block scope variable
            System.out.println("Block scope x = " + x); // Valid inside block
        }
        // Uncommenting the next line will cause a compile error:
        // System.out.println(x); // Error: x is out of scope
    }

    public void demonstrateMethodScope() {
        int y = 10; // Method scope variable
        System.out.println("Method scope y = " + y);
        // y cannot be accessed outside this method
    }

    public void incrementCount() {
        count++; // Access instance variable
    }

    public void displayCount() {
        System.out.println("Instance variable count = " + count);
    }

    public void demonstrateShadowing() {
        int count = 100;  // Shadows instance variable
        System.out.println("Shadowed count = " + count);
        System.out.println("Instance variable count via this = " + this.count);
    }

    public static void main(String[] args) {
        ScopeExample example = new ScopeExample();

        example.demonstrateBlockScope();
        example.demonstrateMethodScope();

        example.incrementCount();
        example.incrementCount();
        example.displayCount();

        example.demonstrateShadowing();
```

```
    }
}
```

5.5.4 Reflection

Being mindful of scope helps avoid **naming collisions**, **unexpected behavior**, and **memory leaks**. Always declare variables in the narrowest scope needed, use meaningful names, and avoid shadowing unless there's a good reason. Proper scoping leads to more readable, maintainable, and bug-resistant code.

Chapter 6.

Classes, Objects, and Members

1. Defining Classes
2. Creating Objects
3. Instance and Static Fields
4. Instance and Static Methods
5. Access Modifiers: `public`, `private`, `protected`
6. Static Blocks and Initialization Blocks
7. Using `this` Keyword

6 Classes, Objects, and Members

6.1 Defining Classes

In Java, a **class** is the fundamental building block of object-oriented programming. A class defines a **blueprint** for objects—each object created from a class can have its own data (fields) and behavior (methods).

6.1.1 Basic Class Structure

Here's a simple class definition:

```java
public class Person {
    // Fields (also called instance variables)
    String name;
    int age;

    // Method
    void greet() {
        System.out.println("Hello, my name is " + name);
    }
}
```

- `public` is an **access modifier** that makes the class accessible from other packages.
- `class` is the keyword to declare a class.
- `Person` is the **class name**, following Java naming conventions (capitalize each word—PascalCase).
- Inside the class, we define:
  - **Fields** like `name` and `age`, which store data for each object.
  - **Methods** like `greet()`, which define behavior the object can perform.

6.1.2 Naming Conventions and File Structure

- The **filename must match the public class name**. For the class above, the file should be named `Person.java`.
- Class names should start with an uppercase letter and follow **PascalCase** (e.g., `BankAccount`, `StudentRecord`).
- Field and method names usually use **camelCase** (e.g., `accountBalance`, `calculateInterest`).

6.1.3 Multiple Classes per File

Only **one public class** is allowed per `.java` file, and it must match the filename. However, you can define other non-public (package-private) classes in the same file.

```
class Helper {
    void help() {
        System.out.println("Helping...");
    }
}
```

But in practice, each class is usually placed in its own file for clarity and maintainability.

6.1.4 Reflection

A class serves as a **template**—you don't use the class itself directly but create **objects** (instances) based on it. Defining a class with clear structure, good naming, and organized methods makes your code reusable, readable, and scalable as your program grows. Understanding how to properly define a class is a vital first step in mastering Java's object-oriented features.

6.2 Creating Objects

In Java, **objects** are instances of classes. Once you've defined a class, you can create objects using the `new` keyword, which calls the class's constructor and allocates memory for the new instance.

6.2.1 Basic Object Instantiation

Here's how to create an object from a class and access its members:

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

    void greet() {
        System.out.println("Hi, I'm " + name + " and I'm " + age + " years old.");
    }
}

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

```
        Person p1 = new Person(); // Object creation using 'new'
        p1.name = "Alice";
        p1.age = 30;
        p1.greet(); // Output: Hi, I'm Alice and I'm 30 years old.
    }
}
```

- `new Person()` creates a new instance of the `Person` class.
- `p1` is a **reference variable** that points to the object.
- Fields like `name` and `age` are accessed using **dot notation** (`p1.name`).
- Methods are called the same way: `p1.greet()`.

6.2.2 Constructor Role

Constructors are special methods invoked during object creation. If you don't define one, Java provides a **default constructor**.

You can also define custom constructors to initialize values:

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

Then call it like:

```
Person p2 = new Person("Bob", 25);
```

6.2.3 Forgetting `new`

If you forget to use `new`, like:

```
Person p;
p.name = "Error!";
```

You'll get a **NullPointerException** because `p` has not been initialized. Always instantiate with `new` before using the object.

Full runnable code:

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

    // Custom constructor
```

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

    // Default constructor (optional to declare, Java provides it if no other constructor exists)
    Person() {
        this.name = "Unknown";
        this.age = 0;
    }

    void printInfo() {
        System.out.println("Name: " + name + ", Age: " + age);
    }

    public static void main(String[] args) {
        // Using custom constructor
        Person p1 = new Person("Alice", 30);
        p1.printInfo();

        // Using default constructor
        Person p2 = new Person();
        p2.printInfo();

        // Uncommenting the following will cause NullPointerException at runtime
        /*
        Person p3; // Declared but not initialized
        p3.name = "Error!"; // NullPointerException
        */
    }
}
```

6.2.4 Reflection

Creating and using objects is the heart of Java programming. Constructors simplify initialization, and using object references promotes clean and reusable code. Mastering object creation helps you structure your programs around real-world entities.

6.3 Instance and Static Fields

In Java, **fields** are variables declared within a class. These can be either **instance fields** (each object has its own copy) or **static fields** (shared across all objects of the class). Understanding the difference is essential for writing efficient and clear code.

6.3.1 Instance Fields

Instance fields are tied to individual objects. Each object created from a class gets its own unique copy of these fields.

```
public class Car {
    String model; // instance field
    int year;
}

public class Main {
    public static void main(String[] args) {
        Car car1 = new Car();
        car1.model = "Toyota";
        car1.year = 2020;

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

        System.out.println(car1.model); // Toyota
        System.out.println(car2.model); // Honda
    }
}
```

Here, `car1` and `car2` have their own separate `model` and `year` fields.

6.3.2 Static Fields

Static fields belong to the class itself and are **shared among all instances**.

```
public class Car {
    String model;
    int year;
    static int totalCars = 0; // static field

    Car() {
        totalCars++;
    }
}

public class Main {
    public static void main(String[] args) {
        new Car();
        new Car();
        System.out.println(Car.totalCars); // 2
    }
}
```

Notice that `totalCars` is accessed via the class name `Car.totalCars`, and not through an instance. Every time a new `Car` is created, the static field is updated globally.

6.3.3 Use Cases and Reflection

Use **instance fields** for data that varies between objects (like `name`, `age`, or `model`). Use **static fields** for:

- Shared configuration settings
- Constants (e.g., `public static final double PI = 3.14159;`)
- Utility counters (like `totalCars`)

Using static fields carefully helps conserve memory and avoids redundancy—but overusing them can reduce flexibility and break encapsulation. Always consider whether a field should represent object-specific state or shared class-level data.

6.4 Instance and Static Methods

In Java, methods are blocks of code that define behavior. Like fields, methods can be either **instance** (belonging to an object) or **static** (belonging to the class). Understanding how to use each appropriately is key to writing well-structured Java code.

6.4.1 Instance Methods

An instance method is associated with an object. You must create an instance of the class to call the method.

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

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

Here, `add()` is called on the `calc` object, and each object can have its own internal state that instance methods can work with.

6.4.2 Static Methods

A static method belongs to the class, not any instance. You can call it without creating an object.

```
public class MathUtils {
    public static int square(int x) {
        return x * x;
    }
}

public class Main {
    public static void main(String[] args) {
        int result = MathUtils.square(4); // static method call
        System.out.println(result);       // 16
    }
}
```

Static methods are often used for utility or helper functions.

6.4.3 Access Rules

- A **static method cannot** directly access instance methods or fields—because it does not have a reference to an object.
- An **instance method can** access both instance and static members—because it's always tied to an object and thus the class.

```
public class Example {
    static void staticMethod() {
        // Cannot access instanceMethod() directly
    }

    void instanceMethod() {
        staticMethod(); // Valid
    }
}
```

6.4.4 When to Use What

- Use **instance methods** when behavior depends on the state of a particular object.
- Use **static methods** when behavior is general-purpose and does not require object state (e.g., math utilities, formatting helpers).

Choosing the right method type improves encapsulation, performance, and clarity.

6.5 Access Modifiers: `public`, `private`, `protected`

In Java, **access modifiers** control the visibility of classes, methods, and variables. They define **how accessible a member is** from other classes and packages. Java provides four main levels of access control:

6.5.1 `public`

A `public` member is **accessible from anywhere** in the program.

```
public class Example {
    public int number = 42;
}
```

If another class imports or refers to `Example`, it can access `number` directly.

6.5.2 `private`

A `private` member is **only accessible within the same class**.

```
public class Example {
    private int secret = 123;

    public int getSecret() {
        return secret;
    }
}
```

Other classes **cannot** directly access `secret`. Instead, controlled access is typically provided via public methods like `getSecret()`. This is a core part of **encapsulation**.

6.5.3 `protected`

A `protected` member is accessible:

- Within the **same package**
- By **subclasses** (even in different packages)

```
package animals;

public class Animal {
    protected void speak() {
```

```
        System.out.println("Animal speaks");
    }
}
```

```
package zoo;

import animals.Animal;

public class Dog extends Animal {
    public void bark() {
        speak(); // Legal: subclass can access protected method
    }
}
```

6.5.4 Default (Package-Private)

If no modifier is used, the member is **package-private**, meaning it is accessible **only within the same package**.

```
class Example {
    int data = 100; // default/package-private
}
```

6.5.5 Access Summary Table

Modifier	Same Class	Same Package	Subclass (Other Package)	Other Packages
`public`	YES	YES	YES	YES
`protected`	YES	YES	YES	NO
(default)	YES	YES	NO	NO
`private`	YES	NO	NO	NO

6.5.6 Reflection

Choosing the correct access modifier is critical for **encapsulation**—a principle where internal details are hidden to protect and simplify object usage. It's best to start with `private` and **only increase visibility as needed**, following the **principle of least privilege**. This keeps your classes well-encapsulated, secure, and easier to maintain.

6.6 Static Blocks and Initialization Blocks

In Java, you can perform complex setup tasks using **initializer blocks**, which are special code blocks that run automatically when a class or object is loaded or created. These blocks come in two types:

6.6.1 Static Initializer Block

A **static block** runs **once**, when the class is **first loaded** into memory. It's typically used to initialize **static variables**.

Syntax:

```
public class Example {
    static int staticValue;

    static {
        staticValue = 100;
        System.out.println("Static block executed");
    }
}
```

Execution Order:

- Static blocks run **once**, **before** any static method is called or any object is created.
- If a class has multiple static blocks, they execute **in order of appearance**.

6.6.2 Instance Initializer Block

An **instance block** runs **every time** an object is created, **before the constructor**.

Syntax:

```
public class Example {
    int instanceValue;

    {
        instanceValue = 42;
        System.out.println("Instance initializer executed");
    }

    public Example() {
        System.out.println("Constructor executed");
    }
}
```

Output:

```
Instance initializer executed
Constructor executed
```

Execution Order:

- When an object is created:
    1. Instance fields are initialized.
    2. Instance initializer blocks run.
    3. Constructor runs.

Full runnable code:

```java
public class Example {
    static int staticValue;

    // Static block runs once when the class is loaded
    static {
        staticValue = 100;
        System.out.println("Static block executed");
    }

    int instanceValue;

    // Instance initializer block runs every time an object is created, before constructor
    {
        instanceValue = 42;
        System.out.println("Instance initializer executed");
    }

    public Example() {
        System.out.println("Constructor executed");
    }

    public static void main(String[] args) {
        System.out.println("Main started");

        // Creating first object
        Example ex1 = new Example();

        // Creating second object
        Example ex2 = new Example();
    }
}
```

6.6.3 Reflection

Use initializer blocks for logic that is **shared across constructors** or for complex static initialization that cannot be expressed with a simple assignment. However, **use them sparingly**—for most cases, static variables should be initialized where declared, and constructor logic should remain in constructors for clarity.

Initializer blocks can help reduce code duplication and organize setup logic, but overusing them can make your class harder to read. When in doubt, prefer clear constructors and static factory methods.

6.7 Using `this` Keyword

In Java, the `this` keyword is a special reference to the **current object**—the instance whose method or constructor is being executed. It helps differentiate between **instance variables** and **parameters or local variables**, and supports constructor chaining and passing object references.

6.7.1 Disambiguating Fields with `this`

When a constructor or method parameter shares a name with an instance variable, `this` is used to clarify intent:

```
public class Book {
    private String title;

    public Book(String title) {
        this.title = title; // Assigns parameter to instance variable
    }
}
```

Without `this`, `title = title;` would refer to the parameter on both sides, leaving the field unchanged.

6.7.2 Constructor Chaining with `this()`

Java allows one constructor to call another in the same class using `this()`. This avoids repeating initialization logic:

```
public class Rectangle {
    int width, height;

    public Rectangle() {
        this(10, 20); // Calls the parameterized constructor
    }

    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
```

```
    }
}
```

Note: `this()` must be the **first statement** in a constructor.

6.7.3 Passing the Current Object

You can pass `this` as an argument to methods that expect an instance of the current class:

```
public class Printer {
    void printInfo(Book b) {
        System.out.println("Book title: " + b.title);
    }

    void start() {
        printInfo(this); // Passes current object
    }
}
```

Full runnable code:

```
public class Main {

    public static void main(String[] args) {
        // Testing Book with this
        Book myBook = new Book("Effective Java");
        System.out.println("Book title: " + myBook.getTitle());

        // Testing Rectangle constructor chaining
        Rectangle rect1 = new Rectangle();
        System.out.println("Rectangle default: " + rect1.width + "x" + rect1.height);

        Rectangle rect2 = new Rectangle(5, 8);
        System.out.println("Rectangle custom: " + rect2.width + "x" + rect2.height);

        // Testing passing this
        Printer printer = new Printer(myBook);
        printer.start();
    }
}

class Book {
    private String title;

    public Book(String title) {
        this.title = title; // disambiguate instance variable and parameter
    }

    public String getTitle() {
        return title;
    }
}
```

```java
class Rectangle {
    int width, height;

    public Rectangle() {
        this(10, 20); // constructor chaining: calls the parameterized constructor
    }

    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }
}

class Printer {
    private Book book;

    public Printer(Book book) {
        this.book = book;
    }

    void printInfo(Book b) {
        System.out.println("Printing book info: " + b.getTitle());
    }

    void start() {
        printInfo(this.book); // pass current object state (the book)
    }
}
```

6.7.4 Reflection

Using `this` enhances **clarity** and reduces **ambiguity**, especially when naming conventions overlap. It's essential in constructor chaining and useful when an object needs to refer to itself. However, overusing it can clutter code unnecessarily. Use it **where it improves readability or is required by syntax**, and let naming clarity handle the rest.

Chapter 7.

Constructors and Initialization

1. Constructor Declaration
2. Overloaded Constructors
3. Default and No-Arg Constructors
4. Constructor Chaining
5. Instance Initializer Blocks

7 Constructors and Initialization

7.1 Constructor Declaration

In Java, a **constructor** is a special block of code used to **initialize objects** when they are created. Unlike regular methods, constructors:

- Have **no return type**, not even `void`
- Must have the **same name as the class**
- Are called automatically when you use the `new` keyword

7.1.1 Basic Syntax and Example

Here's a simple constructor that sets the initial values of an object's fields:

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

    // Constructor
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

In the example above:

- The constructor is named `Person`, matching the class name
- It takes two parameters (`name` and `age`) and uses `this` to assign them to instance variables

To create an object using this constructor:

```
Person p = new Person("Alice", 30);
```

This creates a `Person` object with the name "Alice" and age 30.

Full runnable code:

```
public class Main {
    public static void main(String[] args) {
        Person p = new Person("Alice", 30);
        System.out.println("Name: " + p.name);
        System.out.println("Age: " + p.age);
    }
}

class Person {
```

```
    String name;
    int age;

    // Constructor
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

7.1.2 How Constructors Differ from Methods

Constructors are often confused with methods, but they are fundamentally different:

Feature	Constructor	Method
Name	Same as class name	Any valid identifier
Return type	None (not even `void`)	Must have return type
Invocation	Automatically during `new`	Manually called

7.1.3 Reflection: Why Constructors Matter

Constructors are **essential** to creating well-formed objects. They enforce **initialization logic**, helping avoid partially constructed or invalid objects. For instance, if a `BankAccount` requires a non-zero starting balance, a constructor can enforce that rule.

By controlling how objects are created, constructors help maintain **object integrity**, support **encapsulation**, and make your code more **robust and predictable**.

7.2 Overloaded Constructors

In Java, **constructor overloading** allows a class to have **multiple constructors** with different parameter lists. This gives you the flexibility to create objects with varying levels of detail or different initialization contexts.

7.2.1 Syntax and Example

Consider the following class with overloaded constructors:

```java
public class Rectangle {
    int width;
    int height;

    // Constructor with two parameters
    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }

    // Constructor with one parameter (square)
    public Rectangle(int side) {
        this.width = side;
        this.height = side;
    }

    // No-arg constructor (default values)
    public Rectangle() {
        this.width = 1;
        this.height = 1;
    }
}
```

You can now create `Rectangle` objects in different ways:

```java
Rectangle r1 = new Rectangle(10, 5);   // Custom width and height
Rectangle r2 = new Rectangle(7);       // Square with side = 7
Rectangle r3 = new Rectangle();        // Default 1x1 rectangle
```

Full runnable code:

```java
public class Main {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(10, 5);   // Custom width and height
        Rectangle r2 = new Rectangle(7);       // Square with side = 7
        Rectangle r3 = new Rectangle();        // Default 1x1 rectangle

        System.out.println("r1: " + r1.width + " x " + r1.height);
        System.out.println("r2: " + r2.width + " x " + r2.height);
        System.out.println("r3: " + r3.width + " x " + r3.height);
    }
}

class Rectangle {
    int width;
    int height;

    // Constructor with two parameters
    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }

    // Constructor with one parameter (square)
    public Rectangle(int side) {
        this.width = side;
```

```
        this.height = side;
    }

    // No-arg constructor (default values)
    public Rectangle() {
        this.width = 1;
        this.height = 1;
    }
}
```

7.2.2 How Java Chooses the Right Constructor

When you instantiate an object with `new`, Java checks the constructor signatures and chooses the one that matches the number and type of arguments provided. This resolution is done at **compile time**, not runtime, and is based on **method signature matching**.

7.2.3 Reflection: Why Use Overloaded Constructors?

Overloaded constructors make your classes **more flexible** and **user-friendly**. They allow developers to initialize objects in different ways without needing to remember default values or create multiple setup methods.

For example, if you're building a `User` object, you might want one constructor for name and email, another for just the username, and a default constructor for anonymous users.

By offering multiple initialization paths, overloading improves **code clarity**, **reusability**, and **maintainability** while keeping object creation concise and consistent.

7.3 Default and No-Arg Constructors

In Java, constructors are essential for initializing objects. There are two important types of constructors to understand early on: the **default constructor** and the **no-argument (no-arg) constructor**.

7.3.1 Default Constructor

A **default constructor** is a constructor **automatically provided by the compiler only if you don't define any constructor in your class**. It has no parameters and simply

calls the superclass constructor.

Example:

```
public class Animal {
    // No constructors defined
    // Compiler adds: Animal() { super(); }
}
```

You can then create an object like:

```
Animal a = new Animal();
```

7.3.2 No-Arg Constructor

A **no-arg constructor** is a constructor that **you write explicitly** that takes no parameters. It may include custom initialization logic.

Example:

```
public class Animal {
    public Animal() {
        System.out.println("An animal is created.");
    }
}
```

Even though this constructor takes no arguments, it's not the compiler-generated default—it's user-defined and won't be added automatically.

7.3.3 Important Distinction

If you define **any constructor** (with or without parameters), **Java will no longer generate a default constructor**. This means if you define a parameterized constructor and still want to allow object creation without arguments, **you must explicitly write your own no-arg constructor**.

Example:

```
public class Animal {
    public Animal(String name) {} // Custom constructor

    // No default or no-arg constructor now!
}
```

Full runnable code:

```java
public class Main {
    public static void main(String[] args) {
        // Using class with no constructor defined - compiler provides default constructor
        AnimalDefault animalDefault = new AnimalDefault();
        System.out.println("Created AnimalDefault");

        // Using class with explicit no-arg constructor
        AnimalNoArg animalNoArg = new AnimalNoArg();

        // Using class with parameterized constructor only
        AnimalParam animalParam = new AnimalParam("Buddy");
        // The following line would cause a compile error if uncommented because no no-arg constructor
        // AnimalParam animalParamNoArg = new AnimalParam();
    }
}

// No constructor defined - compiler adds default constructor
class AnimalDefault {
    // no constructors explicitly defined
}

// Explicit no-arg constructor with custom logic
class AnimalNoArg {
    public AnimalNoArg() {
        System.out.println("An animal is created.");
    }
}

// Class with only parameterized constructor, no default or no-arg constructor
class AnimalParam {
    public AnimalParam(String name) {
        System.out.println("Animal created with name: " + name);
    }
}
```

7.3.4 Reflection

Define your own no-arg constructor when you want objects to be easily created without needing parameters, such as in frameworks, serialization, or testing. It ensures better control over object state and prevents unintended behavior due to missing constructors.

7.4 Constructor Chaining

Constructor chaining allows one constructor to call another within the same class using the special keyword `this(...)`. This helps **eliminate code duplication** by centralizing common initialization logic, which improves readability and maintainability.

7.4.1 Syntax of Constructor Chaining

To call another constructor from the current one, use `this(...)` as the **first statement** in the constructor.

```
public class Book {
    String title;
    int pages;

    // Constructor 1
    public Book() {
        this("Untitled", 0); // Calls Constructor 2
    }

    // Constructor 2
    public Book(String title) {
        this(title, 100); // Calls Constructor 3
    }

    // Constructor 3
    public Book(String title, int pages) {
        this.title = title;
        this.pages = pages;
    }
}
```

In this example:

- `Book()` chains to `Book(String, int)`
- `Book(String)` also chains to `Book(String, int)`
- Only the third constructor actually contains the initialization logic

7.4.2 Why Use Constructor Chaining?

Without chaining, each constructor would repeat field initialization:

```
public Book(String title) {
    this.title = title;
    this.pages = 100;
}
```

If you have three constructors initializing the same fields with different defaults, you'd have to repeat code multiple times. This repetition can introduce bugs if one constructor is updated and others are not.

By chaining constructors, **only one place** contains the actual logic for setting values, making the code easier to maintain and modify.

Full runnable code:

```java
public class Main {
    public static void main(String[] args) {
        Book b1 = new Book();
        System.out.println("b1: " + b1.title + ", pages: " + b1.pages);

        Book b2 = new Book("Java Programming");
        System.out.println("b2: " + b2.title + ", pages: " + b2.pages);

        Book b3 = new Book("Effective Java", 350);
        System.out.println("b3: " + b3.title + ", pages: " + b3.pages);
    }
}

class Book {
    String title;
    int pages;

    // Constructor 1
    public Book() {
        this("Untitled", 0); // Calls Constructor 3
    }

    // Constructor 2
    public Book(String title) {
        this(title, 100); // Calls Constructor 3
    }

    // Constructor 3
    public Book(String title, int pages) {
        this.title = title;
        this.pages = pages;
    }
}
```

7.4.3 Reflection

Constructor chaining encourages the **DRY principle** (Don't Repeat Yourself). It also gives flexibility by allowing multiple ways to create an object while ensuring consistent initialization. However, always use `this(...)` carefully—make sure it forms a clear and non-circular chain, or the compiler will reject it.

7.5 Instance Initializer Blocks

In Java, **instance initializer blocks** are blocks of code enclosed in `{}` that are **not part of any method or constructor**, but are executed every time an object is created. These blocks run **after field initializations and before any constructor code**.

7.5.1 Syntax and Execution Order

Here's how you declare an instance initializer block:

```
public class Sample {
    int x;

    {
        // Instance initializer block
        System.out.println("Initializer block running...");
        x = 10;
    }

    public Sample() {
        System.out.println("Constructor running...");
    }
}
```

Output when creating a new object:

```
Initializer block running...
Constructor running...
```

This shows that the initializer block executes **before the constructor**, but **after field initializers**, in the order they appear in the code.

7.5.2 Field Initializer vs. Initializer Block vs. Constructor

```
public class OrderExample {
    int a = initializeA();

    {
        System.out.println("Instance initializer block");
    }

    public OrderExample() {
        System.out.println("Constructor");
    }

    int initializeA() {
        System.out.println("Field initializer");
        return 42;
    }
}
```

Output:

```
Field initializer
Instance initializer block
```

`Constructor`

Full runnable code:

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Creating Sample object:");
        Sample sample = new Sample();

        System.out.println("\nCreating OrderExample object:");
        OrderExample order = new OrderExample();
    }
}

class Sample {
    int x;

    {
        // Instance initializer block
        System.out.println("Initializer block running...");
        x = 10;
    }

    public Sample() {
        System.out.println("Constructor running...");
    }
}

class OrderExample {
    int a = initializeA();

    {
        System.out.println("Instance initializer block");
    }

    public OrderExample() {
        System.out.println("Constructor");
    }

    int initializeA() {
        System.out.println("Field initializer");
        return 42;
    }
}
```

7.5.3 When to Use Initializer Blocks

Instance initializer blocks can be helpful when:

- You have **shared initialization logic** for **all constructors**
- You want to run code **once per object**, regardless of which constructor is called
- You need logic that doesn't naturally fit in a field initializer

However, **they can also be confusing** if overused or mixed with complex constructor logic. Most initialization should happen in constructors or field initializers. Use initializer blocks **only when necessary**, and keep them simple for readability.

Summary: Instance initializer blocks offer fine-grained control over object initialization order, but should be used judiciously to avoid obscuring object creation flow.

Chapter 8.

Inheritance and Polymorphism Syntax

1. Using `extends`
2. Method Overriding
3. `super` Keyword Usage
4. Object Slicing (syntax implications)
5. Final Classes and Methods (`final`)
6. `instanceof` Operator

8 Inheritance and Polymorphism Syntax

8.1 Using `extends`

In Java, the `extends` keyword is used to create a subclass (also called a derived class) that inherits fields and methods from a superclass (also called a base class). This is the foundation of **inheritance**, one of the core principles of object-oriented programming.

8.1.1 Basic Syntax

```
class Animal {
    void speak() {
        System.out.println("The animal makes a sound.");
    }
}

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

In this example, `Dog` is a subclass of `Animal`. It **inherits** the `speak()` method from `Animal`, and adds a new method `bark()`.

8.1.2 What Gets Inherited

When a class extends another:

- **Public and protected methods and fields** are inherited.
- **Private members** are **not inherited** directly, but may still be accessed via public/protected methods (getters/setters) if defined.
- Constructors are **not inherited**, but the subclass can call the superclass constructor using `super()`.

```
public class Main {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.speak(); // Inherited from Animal
        d.bark();  // Defined in Dog
    }
}
```

Full runnable code:

```java
class Animal {
    void speak() {
        System.out.println("The animal makes a sound.");
    }
}

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

public class Main {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.speak(); // Inherited from Animal
        d.bark();  // Defined in Dog
    }
}
```

8.1.3 Whats Not Inherited

- Constructors
- Private fields and methods
- Static blocks and initialization blocks

8.1.4 Reflection on Code Reuse

Using `extends` helps **avoid code duplication** by centralizing shared behavior in a base class. Subclasses can:

- Use inherited functionality as-is
- **Override** methods to provide specialized behavior
- **Extend** the class with new methods or fields

This approach supports the **"is-a" relationship**, where `Dog` **is an** `Animal`, making the code more organized, modular, and reusable.

Use inheritance thoughtfully, ensuring it models a logical hierarchy. When used well, `extends` promotes clean, maintainable code through shared behavior and abstraction.

8.2 Method Overriding

Method overriding allows a subclass to provide a specific implementation of a method that is already defined in its superclass. This is a key aspect of polymorphism in Java, enabling objects to exhibit different behaviors based on their actual type at runtime.

8.2.1 Basic Syntax and Example

```
class Animal {
    void speak() {
        System.out.println("The animal makes a sound.");
    }
}

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

In this example, `Dog` overrides the `speak()` method inherited from `Animal`. The `@Override` annotation tells the compiler that this method is intended to override a superclass method, helping catch errors such as mismatched method signatures.

8.2.2 Runtime Behavior: Dynamic Dispatch

When you call an overridden method on a superclass reference that points to a subclass object, **Java decides at runtime** which version of the method to execute—this is called **dynamic dispatch**:

```
public class Main {
    public static void main(String[] args) {
        Animal myPet = new Dog();  // Upcasting
        myPet.speak();             // Outputs: "The dog barks."
    }
}
```

Although `myPet` is declared as type `Animal`, it actually refers to a `Dog` object. The overridden `speak()` method in `Dog` is executed.

Full runnable code:

```
class Animal {
    void speak() {
```

```
        System.out.println("The animal makes a sound.");
    }
}

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

public class Main {
    public static void main(String[] args) {
        Animal myPet = new Dog();  // Upcasting
        myPet.speak();             // Outputs: The dog barks.
    }
}
```

8.2.3 Rules of Overriding

- Method signature must match (name, parameters, return type).
- Access level cannot be more restrictive than the original.
- Can only override methods that are not `private`, `static`, or `final`.

8.2.4 Reflection: Why Override?

Method overriding supports **polymorphism**, allowing one interface (e.g., `Animal`) to represent multiple behaviors (e.g., `Dog.speak()`, `Cat.speak()`).

This makes your code:

- More **flexible** and **extensible**
- Easier to **maintain**
- Aligned with object-oriented design principles like **open/closed** and **Liskov substitution**

In essence, overriding allows subclasses to customize or extend behaviors, making your codebase better organized and more powerful.

8.3 `super` Keyword Usage

The `super` keyword in Java is used to refer to the **immediate superclass** of a class. It is particularly useful in two main situations:

- Calling a superclass **constructor**
- Accessing a superclass **method** or **field** that has been overridden or hidden in the subclass

8.3.1 Calling the Superclass Constructor with `super()`

If a subclass needs to initialize part of the parent class, it can call the superclass constructor using `super()`.

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

class Dog extends Animal {
    Dog() {
        super("Buddy"); // Calls Animal(String) constructor
        System.out.println("Dog constructor");
    }
}
```

When `new Dog()` is called, it first invokes the `Animal` constructor through `super("Buddy")`, then proceeds with the rest of the `Dog` constructor. This call must be the **first statement** in the subclass constructor.

8.3.2 Accessing Overridden Methods with `super.methodName()`

A subclass can override a method but still invoke the parent version using `super.methodName()`:

```
class Animal {
    void speak() {
        System.out.println("Animal speaks");
    }
}

class Dog extends Animal {
    @Override
    void speak() {
        super.speak(); // Calls Animal's speak method
        System.out.println("Dog barks");
    }
}
```

This is helpful when the subclass wants to **extend** rather than completely replace the behavior of the superclass method.

Full runnable code:

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

    void speak() {
        System.out.println("Animal speaks");
    }
}

class Dog extends Animal {
    Dog() {
        super("Buddy"); // Calls Animal(String) constructor
        System.out.println("Dog constructor");
    }

    @Override
    void speak() {
        super.speak(); // Calls Animal's speak method
        System.out.println("Dog barks");
    }
}

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

8.3.3 Reflection: When and Why to Use `super`

Using `super` is important when you:

- Need to **reuse logic** from the superclass
- Ensure proper **object construction** in inheritance chains
- Preserve or extend inherited behavior rather than replacing it

However, overusing `super` can indicate tight coupling. It's best used deliberately, where superclass behavior is truly required in the subclass.

8.4 Object Slicing (syntax implications)

Object slicing is a concept that arises when a subclass object is referenced by a superclass variable. In Java, all objects are accessed through references, so the term "object slicing" is often used more conceptually than literally—as Java does not physically slice objects like

some other languages (e.g., C++). However, the effect is similar: when you use a superclass reference to hold a subclass object, you lose direct access to subclass-specific members.

8.4.1 Example:

```
class Animal {
    void speak() {
        System.out.println("Animal speaks");
    }
}

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

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        Animal myAnimal = myDog;  // Polymorphic assignment

        myAnimal.speak();  // Works fine, Animal method accessible
        // myAnimal.bark(); // Compile-time error: method not found in Animal
    }
}
```

Here, `myAnimal` holds a reference to a `Dog` object, but only the methods declared in `Animal` are accessible. The subclass-specific `bark()` method cannot be called directly from the `myAnimal` reference.

8.4.2 Reflection:

This limitation is due to **static typing**: the compiler checks methods based on the reference type, not the actual object type. Although **dynamic dispatch** ensures overridden methods are correctly called at runtime, subclass-specific members require casting to access.

This behavior promotes type safety but restricts access to subclass features unless explicitly cast, reflecting a trade-off between flexibility and safety in Java's polymorphism model. Understanding this helps avoid confusion when working with inheritance and references.

8.5 Final Classes and Methods (`final`)

In Java, the `final` keyword can be used to **prevent inheritance and method overriding**, helping enforce design constraints and improve code safety.

8.5.1 Declaring a `final` class

A `final` class **cannot be subclassed**. This means no class can extend it:

```
public final class ImmutableClass {
    // Class content
}

// The following will cause a compile-time error:
// public class ExtendedClass extends ImmutableClass { }
```

Making a class `final` is useful when you want to create immutable or security-sensitive classes that shouldn't be altered by inheritance.

8.5.2 Declaring a `final` method

A `final` method **cannot be overridden** by subclasses, even if the class itself is not `final`:

```
class Base {
    public final void display() {
        System.out.println("Base display method");
    }
}

class Derived extends Base {
    // The following will cause a compile-time error:
    // public void display() { System.out.println("Derived display"); }
}
```

Using `final` on methods ensures critical behavior remains unchanged in subclasses.

Full runnable code:

```
final class ImmutableClass {
    public void show() {
        System.out.println("ImmutableClass method");
    }
}

// Uncommenting this will cause a compile-time error:
// class ExtendedClass extends ImmutableClass {}
```

```java
class Base {
    public final void display() {
        System.out.println("Base display method");
    }
}

class Derived extends Base {
    // This override would cause a compile-time error:
    // public void display() {
    //     System.out.println("Derived display");
    // }
}

public class Main {
    public static void main(String[] args) {
        ImmutableClass imm = new ImmutableClass();
        imm.show();

        Derived d = new Derived();
        d.display();
    }
}
```

8.5.3 Reflection

Restricting inheritance or overriding with `final` helps maintain **design stability**, preventing unintended modifications or misuse. For example, security-sensitive classes like `java.lang.String` are `final` to guarantee consistent, reliable behavior.

At the same time, overusing `final` can reduce flexibility, so it's best applied thoughtfully, balancing safety and extensibility.

8.6 `instanceof` Operator

The `instanceof` operator in Java is used to **test whether an object is an instance of a specific class or interface** before performing operations like casting. This helps avoid `ClassCastException` at runtime by ensuring the cast is safe.

8.6.1 Traditional Usage

Here is a typical example where `instanceof` is used to check the type of an object before casting it:

```java
Object obj = "Hello, Java!";

if (obj instanceof String) {
    String str = (String) obj;  // Safe cast after check
    System.out.println("String length: " + str.length());
} else {
    System.out.println("Not a String object");
}
```

In this example, `obj` is declared as `Object` but actually references a `String`. The `instanceof` check confirms this before casting, preventing runtime errors.

8.6.2 How `instanceof` Enables Safe Polymorphism

In polymorphic code, you often have variables of a superclass or interface type pointing to subclass instances. The `instanceof` operator allows you to:

- Identify the actual object type at runtime.
- Safely downcast to access subclass-specific members.
- Write conditional logic depending on the object's real type.

This improves flexibility by letting a single reference hold various subclass types, while still enabling safe and controlled behavior specific to those types.

8.6.3 Enhanced Pattern Matching with `instanceof` (Java 16)

Starting with Java 16, `instanceof` was enhanced with **pattern matching**, which simplifies type checks and casts into one step:

```java
if (obj instanceof String str) {
    System.out.println("String length: " + str.length());
}
```

Here, if `obj` is a `String`, it's automatically cast and assigned to `str`, removing the need for a separate cast line. This feature improves readability and reduces boilerplate.

Full runnable code:

```java
public class InstanceofExample {
    public static void main(String[] args) {
        Object obj = "Hello, Java!";

        // Traditional usage
        if (obj instanceof String) {
            String str = (String) obj;  // Safe cast after check
            System.out.println("Traditional: String length = " + str.length());
```

```
        } else {
            System.out.println("Traditional: Not a String object");
        }

        // Enhanced pattern matching (Java 16+)
        if (obj instanceof String str) {
            System.out.println("Pattern matching: String length = " + str.length());
        }
    }
}
```

8.6.4 Reflection

Using `instanceof` judiciously enhances type safety and clarity in polymorphic code. While it helps prevent errors, excessive use may indicate design issues—sometimes better addressed with polymorphic methods or interfaces. However, combined with Java's newer pattern matching, `instanceof` remains a powerful tool for safe and readable type-dependent logic.

Chapter 9.

Interfaces and Abstract Types

1. Declaring and Implementing Interfaces
2. Functional Interfaces and `@FunctionalInterface`
3. Default and Static Methods in Interfaces
4. Abstract Classes and Methods

9 Interfaces and Abstract Types

9.1 Declaring and Implementing Interfaces

In Java, **interfaces** provide a powerful way to define contracts that classes can implement. An interface declares method signatures (and, since Java 8, default and static methods) without providing full implementations, allowing different classes to share a common set of behaviors without forcing a specific class hierarchy.

Defining an Interface

To declare an interface, use the `interface` keyword followed by the interface name and a body containing method signatures:

```
public interface Vehicle {
    void start();
    void stop();
}
```

In this example, `Vehicle` is an interface with two methods: `start()` and `stop()`. These methods are **implicitly public and abstract**, so you don't specify these modifiers explicitly.

Implementing an Interface

A class **implements** an interface by using the `implements` keyword and providing concrete implementations for all of the interface's methods:

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

    @Override
    public void stop() {
        System.out.println("Car is stopping.");
    }
}
```

Here, `Car` commits to the `Vehicle` contract by implementing `start()` and `stop()`. The `@Override` annotation helps catch errors if method signatures don't match.

Implementing Multiple Interfaces

Java supports **multiple interface inheritance**, meaning a class can implement more than one interface:

```
public interface Electric {
    void charge();
}
```

```java
public class ElectricCar implements Vehicle, Electric {
    @Override
    public void start() {
        System.out.println("ElectricCar is starting silently.");
    }

    @Override
    public void stop() {
        System.out.println("ElectricCar is stopping.");
    }

    @Override
    public void charge() {
        System.out.println("ElectricCar is charging.");
    }
}
```

`ElectricCar` implements both `Vehicle` and `Electric`, so it must implement all methods from both interfaces. This ability lets you **mix behaviors** flexibly without tying your classes to a rigid class inheritance structure.

Interface Inheritance vs. Class Inheritance

- **Class Inheritance** (`extends`) creates a parent-child relationship where the subclass inherits state (fields) and behavior (methods) from its superclass. This relationship is **concrete** and can only extend one class.
- **Interface Inheritance** (`implements`) defines a **contract** that any implementing class must fulfill, but without dictating the class hierarchy or shared implementation. A class can implement **multiple** interfaces, enabling polymorphism without the complexity of multiple inheritance of classes.

Why Are Interfaces Useful for Abstraction?

1. **Decoupling:** Interfaces separate *what* a class should do from *how* it does it. This encourages coding to an interface rather than to an implementation, promoting loose coupling and easier maintenance.
2. **Multiple Behavior Inheritance:** Since Java does not support multiple inheritance of classes, interfaces allow a class to have multiple capabilities by implementing multiple interfaces.
3. **Polymorphism:** You can write code that works on the interface type, allowing different implementations to be plugged in without changing client code.

```java
public void testVehicle(Vehicle v) {
    v.start();
    v.stop();
}
```

This method can accept any object that implements `Vehicle`, whether it's a `Car`, `ElectricCar`, or another class entirely.

4. **API Design:** Interfaces define clear, minimal contracts for APIs and libraries, improving readability and enforceability.

Summary Example

```
public interface Flyable {
    void fly();
}

public class Bird implements Flyable {
    @Override
    public void fly() {
        System.out.println("Bird is flying.");
    }
}

public class Airplane implements Flyable {
    @Override
    public void fly() {
        System.out.println("Airplane is flying.");
    }
}
```

Both `Bird` and `Airplane` share the `Flyable` interface, but their implementations of `fly()` differ, illustrating polymorphism with interfaces.

Full runnable code:

```
class Animal {
    void speak() {
        System.out.println("The animal makes a sound.");
    }
}

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

public class Main {
    public static void main(String[] args) {
        Animal myPet = new Dog();  // Upcasting
        myPet.speak();             // Outputs: The dog barks.
    }
}
```

9.1.1 Reflection

Interfaces provide a way to enforce consistent behavior across unrelated classes, enabling flexibility and extensibility in Java programs. Unlike class inheritance, which shares implementation and can lead to complex hierarchies, interfaces define a clear contract without imposing a fixed inheritance structure. This abstraction fosters modular, maintainable, and testable code — essential qualities in modern software development.

By mastering interfaces and their implementation, you unlock a core aspect of Java's design philosophy, empowering you to write clean, scalable, and robust applications.

9.2 Functional Interfaces and `@FunctionalInterface`

In Java, a **functional interface** is a special kind of interface that has exactly one abstract method. This single abstract method defines the contract of the interface and enables instances of the interface to be created with lambda expressions, method references, or anonymous classes — a core foundation of functional programming in Java.

Defining a Functional Interface

A functional interface contains only one abstract method, but it can have any number of **default** or **static** methods. Here's an example of a functional interface named `Calculator`:

```
@FunctionalInterface
public interface Calculator {
    int calculate(int a, int b);
}
```

- The interface has a single abstract method, `calculate`.
- The `@FunctionalInterface` annotation is optional but highly recommended. It explicitly marks the interface as functional and allows the compiler to enforce this rule.

Purpose of `@FunctionalInterface`

The `@FunctionalInterface` annotation serves several important roles:

1. **Compile-Time Enforcement:** It ensures the interface has exactly one abstract method. If you accidentally add another abstract method, the compiler will produce an error. This prevents accidental violations of the functional interface contract.
2. **Documentation:** It clearly signals to readers that the interface is intended for functional programming use, making the code easier to understand and maintain.
3. **Compatibility:** Some APIs and frameworks may specifically require or optimize for functional interfaces marked with this annotation.

Without this annotation, an interface with one abstract method can still be used as a

functional interface, but you lose the safety net provided by the compiler.

Using Functional Interfaces with Lambda Expressions

Functional interfaces are critical for using **lambda expressions** in Java. Lambdas provide a concise syntax to implement the single abstract method without writing verbose anonymous classes.

For example, using the `Calculator` interface:

```
Calculator add = (a, b) -> a + b;
Calculator multiply = (a, b) -> a * b;

System.out.println(add.calculate(5, 3));       // Output: 8
System.out.println(multiply.calculate(5, 3));  // Output: 15
```

Here, `add` and `multiply` are lambda expressions implementing the `calculate` method. The lambda `(a, b) -> a + b` means: given `a` and `b`, return their sum.

This style dramatically reduces boilerplate and improves code readability, especially when used with standard Java libraries that use functional interfaces, such as the `java.util.function` package.

Common Built-In Functional Interfaces

Java provides many ready-to-use functional interfaces in the `java.util.function` package, including:

- `Predicate<T>` — represents a boolean-valued function of one argument (`test(T t)`).
- `Function<T, R>` — represents a function that accepts one argument and produces a result (`apply(T t)`).
- `Consumer<T>` — represents an operation that accepts a single input argument and returns no result (`accept(T t)`).
- `Supplier<T>` — represents a supplier of results, no input and returns a value (`get()`).
- `BinaryOperator<T>` — represents an operation upon two operands of the same type, producing a result of the same type.

Each of these has a single abstract method, making them functional interfaces suitable for lambdas.

Functional Interfaces and Functional Programming

The introduction of functional interfaces was a major step toward embracing **functional programming paradigms** in Java:

- **Immutability:** Functional programming encourages avoiding side effects and mutating shared state.
- **Higher-order functions:** Functions can be passed as parameters, returned from methods, or stored in variables, enabling more flexible and reusable code.
- **Declarative style:** Code focuses on *what* to do rather than *how* to do it, improving clarity.

By defining and using functional interfaces, Java enables these patterns while maintaining backward compatibility with the object-oriented core of the language.

Example: Custom Functional Interface with Default Methods

Functional interfaces can also contain default methods without breaking their contract:

```
@FunctionalInterface
public interface Printer {
    void print(String message);

    default void printTwice(String message) {
        print(message);
        print(message);
    }
}
```

This interface has one abstract method, `print`, and a default method, `printTwice`. A lambda can implement `print`, and the default method can be used as-is:

```
Printer printer = msg -> System.out.println(msg);
printer.print("Hello");        // Prints "Hello"
printer.printTwice("Hello");   // Prints "Hello" twice
```

Full runnable code:

```
@FunctionalInterface
interface Calculator {
    int calculate(int a, int b);
}

@FunctionalInterface
interface Printer {
    void print(String message);

    default void printTwice(String message) {
        print(message);
        print(message);
    }
}

public class Main {
    public static void main(String[] args) {
        // Using Calculator interface with lambdas
        Calculator add = (a, b) -> a + b;
        Calculator multiply = (a, b) -> a * b;

        System.out.println("Addition: " + add.calculate(5, 3));       // 8
        System.out.println("Multiplication: " + multiply.calculate(5, 3)); // 15

        // Using Printer interface with a lambda
        Printer printer = msg -> System.out.println(msg);
        printer.print("Hello");
        printer.printTwice("World");
    }
```

```
}
```

9.2.1 Reflection

Functional interfaces are a cornerstone of modern Java programming, bridging the gap between traditional object-oriented approaches and functional programming techniques. The `@FunctionalInterface` annotation provides clarity and safety, ensuring that your interfaces remain suitable for lambda expressions and method references.

By leveraging functional interfaces, Java developers can write more concise, expressive, and modular code. They unlock the power of functional programming patterns—such as higher-order functions and immutability—while retaining the strengths and familiarity of Java's object-oriented roots.

Understanding and using functional interfaces effectively will prepare you for working with streams, asynchronous programming, and modern APIs that rely heavily on these constructs.

9.3 Default and Static Methods in Interfaces

Before Java 8, interfaces could only declare abstract methods — methods without a body — meaning any class implementing the interface had to provide its own implementation. This limitation made evolving interfaces difficult because adding a new method to an interface would break all existing implementations.

Default Methods

Java 8 introduced **default methods** to address this problem. A default method provides a method body inside the interface itself, allowing the interface to supply a *default implementation*. Classes that implement the interface can either use this default or override it.

Here's an example of an interface with a default method:

```
public interface Vehicle {
    void start();

    default void stop() {
        System.out.println("Stopping the vehicle.");
    }
}
```

In this example:

- The `start()` method is abstract and must be implemented.

- The `stop()` method has a default implementation that can be inherited as-is.

A class implementing `Vehicle` can choose to override `stop()` or use the provided default:

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

    // stop() method inherited by default
}
```

Static Methods

Java 8 also added **static methods** to interfaces. These are methods that belong to the interface itself rather than any instance, similar to static methods in classes.

Example:

```
public interface MathUtils {
    static int square(int x) {
        return x * x;
    }
}
```

Static methods in interfaces are called using the interface name:

```
int result = MathUtils.square(5);  // Returns 25
```

Full runnable code:

```
interface Vehicle {
    void start();

    default void stop() {
        System.out.println("Stopping the vehicle.");
    }
}

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

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

public class Main {
```

```
    public static void main(String[] args) {
        Vehicle myCar = new Car();
        myCar.start();  // Output: Car started.
        myCar.stop();   // Output: Stopping the vehicle.

        int result = MathUtils.square(5);
        System.out.println("Square of 5: " + result);  // Output: Square of 5: 25
    }
}
```

Why Were Default and Static Methods Added?

The main reasons were:

- **Backward Compatibility:** They allow library designers to add new methods to interfaces without breaking existing implementations. If a new method can have a reasonable default, it can be added safely.
- **Shared Logic:** Interfaces can now provide reusable behavior, reducing code duplication in implementing classes.
- **Better Abstractions:** Interfaces can evolve into richer APIs that combine abstract declarations with concrete behavior.

Reflection

Default and static methods in interfaces represent a significant evolution in Java's type system. They enable *interface evolution* without sacrificing backward compatibility — a critical concern for large-scale, widely used APIs.

By allowing interfaces to hold shared logic, Java promotes cleaner design and easier maintenance. However, care should be taken to avoid overly complex default methods that can obscure the primary role of interfaces as contracts.

In summary, default and static methods enhance flexibility and expressiveness in Java interfaces, bridging the gap between pure abstraction and practical implementation.

9.4 Abstract Classes and Methods

In Java, **abstract classes** provide a way to define classes that cannot be instantiated on their own but serve as a foundation for other classes. Abstract classes can contain both abstract methods (methods without implementation) and concrete methods (methods with implementation). This allows for **partial implementation and code reuse** across subclasses.

Declaring an Abstract Class and Abstract Method

To declare an abstract class, use the `abstract` keyword in the class declaration. Similarly, abstract methods inside the class are declared with `abstract` and do not have a body.

```java
public abstract class Animal {
    // Abstract method: no body
    public abstract void makeSound();

    // Concrete method: has implementation
    public void eat() {
        System.out.println("This animal is eating.");
    }
}
```

Here, `Animal` is an abstract class with an abstract method `makeSound()` and a concrete method `eat()`.

Concrete Subclass Implementing Abstract Method

A subclass that extends an abstract class must implement all abstract methods, or it too becomes abstract.

```java
public class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Woof!");
    }
}
```

Now, `Dog` is a concrete class with an implementation of `makeSound()`. We can create objects of `Dog` but not of `Animal`.

Abstract Classes vs. Interfaces

Aspect	Abstract Class	Interface
Can contain	Abstract methods, concrete methods, fields	Only abstract methods (pre-Java 8), now can have default and static methods
Multiple inheritance	No (Java doesn't support multiple inheritance)	Yes (a class can implement multiple interfaces)
Use case	Partial implementation, shared state (fields)	Pure abstraction, defining contracts
Constructors	Can have constructors	Cannot have constructors

When to use which?

- Use **abstract classes** when you want to share code and state (fields) among related classes, or provide default behavior.
- Use **interfaces** to define a contract or capability that can be added to unrelated classes, allowing multiple inheritance of type.

Full runnable code:

```java
// Abstract class with an abstract and a concrete method
abstract class Animal {
    public abstract void makeSound();  // Abstract method

    public void eat() {                // Concrete method
        System.out.println("This animal is eating.");
    }
}

// Concrete subclass that implements the abstract method
class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Woof!");
    }
}

public class Main {
    public static void main(String[] args) {
        // Animal a = new Animal(); // NO Not allowed: cannot instantiate abstract class

        Dog d = new Dog();             // YES Concrete class
        d.makeSound();                 // Woof!
        d.eat();                       // This animal is eating.
    }
}
```

Reflection

Abstract classes play a vital role in Java's type system by allowing **partial implementations**, promoting code reuse while still enforcing a contract for subclasses. Unlike interfaces, abstract classes can hold state and concrete methods, enabling shared functionality and reducing code duplication.

However, since Java supports only single inheritance of classes, abstract classes should be used when there is a clear "is-a" relationship and shared implementation is needed. Interfaces, enhanced since Java 8 with default methods, are better suited for defining capabilities that can cross-cut unrelated classes.

By balancing the use of abstract classes and interfaces, Java programmers can create flexible, maintainable designs that leverage both abstraction and reuse effectively.

Chapter 10.

Nested, Inner, and Anonymous Classes

1. Static Nested Classes
2. Inner Classes (Non-static)
3. Local Classes
4. Anonymous Classes

10 Nested, Inner, and Anonymous Classes

10.1 Static Nested Classes

In Java, **nested classes** are classes defined within other classes. They provide a way to logically group classes that are only used in one place, increasing encapsulation and readability. Among nested classes, there are two main types: **static nested classes** and **inner classes** (non-static). This section focuses on **static nested classes** — what they are, how they differ from inner classes, and when to use them.

What Are Static Nested Classes?

A **static nested class** is a nested class that is declared with the `static` keyword inside an outer class. Unlike inner classes, static nested classes do **not** have access to instance variables or methods of the outer class. They behave much like any other top-level class but are scoped inside the outer class for organizational purposes.

Key differences from inner classes:

- **Static nested classes do not require an instance of the outer class** to be created.
- They cannot directly access non-static (instance) members of the outer class.
- They can access **static members** of the outer class.

This makes static nested classes ideal for grouping helper or utility classes closely related to the outer class but not needing access to instance-specific data.

Syntax Example of a Static Nested Class

```
public class OuterClass {
    private static int staticValue = 10;
    private int instanceValue = 20;

    // Static nested class
    public static class NestedStaticClass {
        public void display() {
            // Can access static members of outer class
            System.out.println("Static value: " + staticValue);

            // Cannot access instance members directly
            // System.out.println("Instance value: " + instanceValue); // Compile error!
        }
    }
}
```

In this example, `NestedStaticClass` is declared inside `OuterClass` with the `static` keyword. Inside its `display()` method, it can access `staticValue` (a static member of `OuterClass`) but **cannot** access `instanceValue` because it is an instance member.

Instantiating Static Nested Classes

Because static nested classes do not depend on an instance of the outer class, you instantiate them **using the outer class name**:

```
public class Test {
    public static void main(String[] args) {
        // Instantiate the static nested class
        OuterClass.NestedStaticClass nestedObject = new OuterClass.NestedStaticClass();
        nestedObject.display();
    }
}
```

Notice how we use `OuterClass.NestedStaticClass` to create an instance of the nested class. This contrasts with inner classes, which require an instance of the outer class to instantiate.

Full runnable code:

```
public class Main {
    static class OuterClass {
        private static int staticValue = 10;
        private int instanceValue = 20;

        // Static nested class
        public static class NestedStaticClass {
            public void display() {
                // Accessing static member of OuterClass
                System.out.println("Static value: " + staticValue);

                // Accessing instanceValue here would cause a compile error
                // System.out.println("Instance value: " + instanceValue);
            }
        }
    }

    public static void main(String[] args) {
        OuterClass.NestedStaticClass nestedObject = new OuterClass.NestedStaticClass();
        nestedObject.display();
    }
}
```

When and Why Use Static Nested Classes?

1. **Logical Grouping of Related Classes** Static nested classes help organize code by grouping classes that are closely related to the outer class but do not need access to its instance members. This improves **encapsulation** and keeps the outer class namespace cleaner.
2. **Avoid Unnecessary References to Outer Instances** Since static nested classes don't have implicit references to outer class instances, they **avoid memory overhead** and potential leaks caused by such references. This can be important for performance-sensitive applications.
3. **Implementing Helper or Utility Classes** Static nested classes are often used to

implement **helper classes** or **builders** that support the functionality of the outer class without being part of its core state.

4. **Enhanced Readability and Maintainability** Grouping related classes together reduces the need for separate files and makes the code easier to navigate and maintain.

Summary Reflection

Static nested classes provide a way to nest classes inside another for better organization while maintaining independence from the outer class's instance state. This independence reduces coupling and memory footprint, making static nested classes well-suited for utility or helper classes that logically belong inside an outer class but don't require access to its instance data.

By contrast, **inner classes** (non-static nested classes) are tightly coupled with an instance of the outer class, with direct access to its members, which is beneficial when inner behavior must directly modify or depend on the outer instance's state.

Using static nested classes wisely leads to cleaner, more modular code and helps prevent unintended memory retention that can occur with inner classes holding references to outer instances. It also improves maintainability by grouping logically connected code in one place without exposing unnecessary implementation details externally.

10.2 Inner Classes (Non-static)

Java allows you to declare classes inside other classes, known as **nested classes**. When these nested classes are **non-static**, they are called **inner classes**. Unlike static nested classes, inner classes maintain a strong relationship with an instance of the enclosing (outer) class, allowing them to access the outer class's members—including private fields and methods—directly. This capability makes inner classes powerful tools for logically grouping behavior closely tied to a particular instance of the outer class.

What Are Inner Classes?

An **inner class** is a non-static class defined within another class. Each instance of an inner class is implicitly linked to an instance of the outer class. This linkage allows the inner class to:

- Access all members of the outer class, including private fields and methods.
- Reference the outer class instance via the special syntax `OuterClassName.this`.

This makes inner classes ideal for scenarios where a helper or utility class needs to tightly interact with the state of a particular outer class object.

Syntax and Example

Here's an example demonstrating how to declare and use a non-static inner class that modifies a private field of its enclosing class:

```java
public class OuterClass {
    private int count = 0;

    // Inner class declaration
    public class InnerClass {
        public void incrementCount() {
            // Directly accessing and modifying private member of OuterClass
            count++;
            System.out.println("Count incremented to: " + count);
        }

        public void displayOuterValue() {
            System.out.println("Accessing from inner class: count = " + count);
        }
    }

    public void showCount() {
        System.out.println("Current count: " + count);
    }
}
```

In this example, the inner class `InnerClass` directly accesses and modifies the private `count` field of the outer class. This access demonstrates one of the key benefits of inner classes: encapsulated yet privileged access to the outer instance's data.

Instantiating an Inner Class

Because an inner class instance is tied to an outer class instance, you **cannot instantiate an inner class without first having an instance of the outer class**.

Here's how you create an instance of the inner class from outside the outer class:

```java
public class TestInnerClass {
    public static void main(String[] args) {
        OuterClass outer = new OuterClass();           // Create an instance of the outer class
        OuterClass.InnerClass inner = outer.new InnerClass();  // Create inner class instance tied to '

        inner.incrementCount();  // Modify outer class's private field through inner class
        inner.displayOuterValue();

        outer.showCount();       // Verify the count was updated
    }
}
```

Note the syntax `outer.new InnerClass()`: this explicitly ties the inner class instance to the specific `outer` instance.

Full runnable code:

```java
public class Main {

    static class OuterClass {
        private int count = 0;

        // Non-static inner class
```

```
        public class InnerClass {
            public void incrementCount() {
                count++;
                System.out.println("Count incremented to: " + count);
            }

            public void displayOuterValue() {
                System.out.println("Accessing from inner class: count = " + count);
            }
        }

        public void showCount() {
            System.out.println("Current count: " + count);
        }
    }

    public static void main(String[] args) {
        OuterClass outer = new OuterClass();
        OuterClass.InnerClass inner = outer.new InnerClass();

        inner.incrementCount();      // Count incremented to: 1
        inner.displayOuterValue();  // Accessing from inner class: count = 1

        outer.showCount();           // Current count: 1
    }
}
```

Use Cases and Advantages of Inner Classes

Inner classes are particularly useful in the following scenarios:

1. **Encapsulation of Logic Tied to Outer Instance** Inner classes let you group helper classes inside the outer class without exposing them externally. Since they can access private fields, they serve as a natural way to implement closely coupled functionality.
2. **Event Listeners and Callbacks** Inner classes are widely used in GUI programming (e.g., Swing, Android) to implement event listeners that need access to the state of the enclosing object.
3. **Improved Code Organization** By nesting related classes, you keep your codebase cleaner and make it easier to maintain relationships between classes.
4. **Stateful Helpers** When a helper class needs to maintain or modify the state of the enclosing instance, inner classes provide a straightforward approach without exposing internals through public APIs.

Reflection: Benefits and Cautions

- **Benefits:** Inner classes improve encapsulation by keeping helper classes tightly bound to their outer instances. They can simplify access to private members without requiring getters/setters. They also help reduce clutter in your package by limiting the visibility scope of supporting classes.
- **Cautions:** Because inner classes hold an implicit reference to their enclosing instance,

they can unintentionally cause memory leaks if instances outlive their outer class objects (common in long-lived listener patterns). Additionally, the syntax for creating inner class instances is more verbose and can confuse beginners.

- **Design tip:** Use inner classes when the logic they encapsulate is genuinely tied to the lifecycle and state of the outer instance. For more loosely coupled functionality, prefer static nested classes or top-level classes.

10.2.1 Summary

Non-static inner classes in Java are a powerful feature that lets you define classes closely associated with an outer instance, granting them full access to that instance's members—even private ones. Their unique instantiation syntax reflects this close binding. Inner classes improve encapsulation and code clarity by grouping tightly coupled logic, especially when helper classes need direct access to the outer class's internals. However, they should be used judiciously to avoid unintended complexity or memory retention issues.

10.3 Local Classes

In Java, **local classes** are a special type of nested class declared **within a method, constructor, or initializer block**. Unlike static nested classes or inner classes, local classes exist only within the scope of the block in which they are declared. This limited scope means local classes are **not visible outside the enclosing method or block**, making them useful for encapsulating helper functionality tightly bound to a specific method's logic.

Declaring a Local Class

A local class is declared just like any other class but inside a method body. It can have fields, methods, and constructors just like a normal class. However, its visibility and lifetime are confined to the method where it's declared.

Here's the general form:

```
public void someMethod() {
    class LocalHelper {
        void greet() {
            System.out.println("Hello from the local class!");
        }
    }

    LocalHelper helper = new LocalHelper();
    helper.greet();
}
```

In this example, `LocalHelper` is a class local to the `someMethod()` method. You cannot create an instance of `LocalHelper` outside this method.

Scope and Lifecycle

- **Scope:** The local class exists only within the block (usually a method) where it is defined.
- **Visibility:** It is not accessible outside the method; hence, it cannot be referenced elsewhere in the class.
- **Lifecycle:** Instances of the local class can only be created during the execution of the enclosing method. Once the method completes, the local class and any of its instances are subject to garbage collection if no references remain.

This limited scope makes local classes perfect for utility or helper classes used only inside a single method, avoiding cluttering the outer class or package namespace.

Access to Enclosing Methods Variables

Local classes can access:

- **Instance members** of the enclosing class (fields and methods).
- **Local variables and parameters** of the enclosing method **only if they are effectively final** (meaning they are not modified after initialization).

For example:

```
public void printMessage(String message) {
    class Printer {
        void print() {
            System.out.println(message); // Accesses effectively final local variable
        }
    }
    Printer p = new Printer();
    p.print();
}
```

Note: Before Java 8, the local variables had to be explicitly declared `final`. Now, they just need to be effectively final (unchanged after initialization).

Practical Example: Utility Class Within a Method

Suppose you need a helper class to process data only within a specific method:

```
public void processData(int[] data) {
    class DataAnalyzer {
        int sum() {
            int total = 0;
            for (int num : data) {
                total += num;
            }
            return total;
        }
```

```
        double average() {
            return (double) sum() / data.length;
        }
    }

    DataAnalyzer analyzer = new DataAnalyzer();
    System.out.println("Sum: " + analyzer.sum());
    System.out.println("Average: " + analyzer.average());
}
```

In this example, the `DataAnalyzer` class helps organize code related to analyzing `data`. It doesn't need to be visible outside `processData()`, so defining it locally improves encapsulation and readability.

Full runnable code:

```
public class Main {

    public void someMethod() {
        class LocalHelper {
            void greet() {
                System.out.println("Hello from the local class!");
            }
        }

        LocalHelper helper = new LocalHelper();
        helper.greet();
    }

    public void printMessage(String message) {
        class Printer {
            void print() {
                System.out.println("Printing message: " + message);
            }
        }

        Printer p = new Printer();
        p.print();
    }

    public void processData(int[] data) {
        class DataAnalyzer {
            int sum() {
                int total = 0;
                for (int num : data) {
                    total += num;
                }
                return total;
            }

            double average() {
                return data.length == 0 ? 0 : (double) sum() / data.length;
            }
        }

        DataAnalyzer analyzer = new DataAnalyzer();
        System.out.println("Sum: " + analyzer.sum());
```

```
        System.out.println("Average: " + analyzer.average());
    }

    public static void main(String[] args) {
        Main demo = new Main();
        demo.someMethod();
        demo.printMessage("Java local classes rock!");
        demo.processData(new int[] {5, 10, 15});
    }
}
```

Reflection: When to Use Local Classes

Advantages:

- **Encapsulation:** Local classes keep helper logic confined to the method that uses it, preventing pollution of the class or package namespace.
- **Readability:** By grouping related functionality inside the method, it can make the method's intent clearer, especially when the helper logic is not complex enough to justify a separate top-level class.
- **Access:** Local classes have privileged access to instance members and effectively final local variables, making them very flexible within their scope.

Considerations:

- **Readability risk:** Overusing local classes, especially for large or complex logic, can make methods harder to read and maintain.
- **Limited reuse:** Since local classes are invisible outside their method, they cannot be reused elsewhere without duplication.
- **Access restrictions:** The need for variables to be effectively final can sometimes require code restructuring.

10.4 Anonymous Classes

Anonymous classes in Java provide a concise way to declare and instantiate a class **at the same time**—without giving the class an explicit name. They are commonly used to create quick implementations of interfaces or abstract classes, especially when the implementation is short-lived or used only once.

Syntax of Anonymous Classes

The syntax of an anonymous class is usually combined with the creation of an instance of an interface or abstract class. It looks like this:

```
InterfaceName instance = new InterfaceName() {
    // Implement methods here
};
```

or

```
AbstractClass instance = new AbstractClass() {
    // Override abstract methods here
};
```

The braces `{}` after the interface or class name define the body of the anonymous class. You can override methods or add new ones inline.

Example: Anonymous Class Implementing an Interface

A classic use of anonymous classes is in event handling or callbacks. For example, with the `Runnable` interface used for threads:

```
Runnable task = new Runnable() {
    @Override
    public void run() {
        System.out.println("Task is running in a separate thread!");
    }
};

Thread thread = new Thread(task);
thread.start();
```

Here, the anonymous class implements the `Runnable` interface and overrides the `run()` method inline, without creating a separate named class.

Example: Overriding Methods Inline

Another example is using an anonymous class to handle a button click event in GUI programming:

```
button.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button clicked!");
    }
});
```

This way, the logic for handling the click is placed directly where it is needed, keeping related code together and avoiding cluttering the codebase with one-off classes.

Full runnable code:

```
public class Main {
    public static void main(String[] args) {
        // Anonymous class implementing the Runnable interface
        Runnable task = new Runnable() {
            @Override
            public void run() {
                System.out.println("Task is running in a separate thread!");
            }
        };
```

```
        Thread thread = new Thread(task);
        thread.start();
    }
}
```

Benefits of Anonymous Classes

- **Conciseness:** They reduce boilerplate code by eliminating the need to write a full class declaration.
- **Encapsulation:** The implementation is kept close to where it is used, making it easy to understand the behavior in context.
- **Convenience:** Perfect for short, simple implementations like callbacks, event listeners, or small thread tasks.

Drawbacks and Considerations

- **Readability:** Overusing anonymous classes or writing complex logic inside them can make the code harder to read and maintain.
- **Debugging:** Since anonymous classes don't have explicit names, stack traces and debugging sessions can be less clear.
- **Reusability:** Anonymous classes cannot be reused outside the point of declaration, so if the logic grows or needs to be shared, refactoring into a named class is better.
- **Verbosity:** Compared to lambda expressions (introduced in Java 8), anonymous classes are more verbose for simple single-method interfaces.

Summary

Anonymous classes offer a powerful way to provide inline implementations of interfaces or abstract classes without cluttering your code with extra class files or declarations. They're ideal for brief, context-specific behavior such as event handling and threading. However, they should be used judiciously—complex or widely used logic is better served by named classes to preserve clarity and maintainability.

With the introduction of lambda expressions in Java 8, many uses of anonymous classes—especially for functional interfaces—can be simplified, but anonymous classes still play a key role when more than one method needs overriding or when dealing with abstract classes.

Chapter 11.

Enums Syntax

1. Declaring `enum` Types
2. Enum Constants
3. Enums with Fields and Methods
4. Using Enums in `switch` Statements

11 Enums Syntax

11.1 Declaring `enum` Types

In Java, an `enum` (short for *enumeration*) is a special reference type that represents a fixed set of constant values. Unlike traditional constants defined with `static final` fields, enums provide a **type-safe** and expressive way to represent a group of related constants. This makes your code clearer, more maintainable, and less error-prone.

Basic Syntax of an `enum`

To declare an enum, use the `enum` keyword followed by the name of the type and a list of constant values separated by commas:

```
public enum Day {
    SUNDAY,
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY
}
```

Here, `Day` is an enum type with seven predefined constants representing days of the week.

Enums Are Reference Types

Though enums look like constants, they are actually **full-fledged reference types**—similar to classes. This means:

- You can use enums like any other object.
- They have methods, can implement interfaces, and have fields (covered in later sections).
- Each enum constant is a singleton instance of the enum class.

Using the Enum

You can declare variables of the enum type and assign one of its constants:

```
Day today = Day.WEDNESDAY;

if (today == Day.WEDNESDAY) {
    System.out.println("It's midweek.");
}
```

The comparison uses `==` because enum constants are unique instances, guaranteeing safe reference equality checks. This is a significant advantage over using plain constants (like `int` or `String`), where equality comparisons can be error-prone.

Minimal Example in a Class

```
public class EnumExample {
    public enum TrafficLight {
        RED,
        YELLOW,
        GREEN
    }

    public static void main(String[] args) {
        TrafficLight signal = TrafficLight.RED;

        switch (signal) {
            case RED:
                System.out.println("Stop!");
                break;
            case YELLOW:
                System.out.println("Get ready.");
                break;
            case GREEN:
                System.out.println("Go!");
                break;
        }
    }
}
```

This example demonstrates how enums make control flow with related constants straightforward and readable.

Why Use Enums Over Plain Constants?

Before enums, developers often used `public static final` constants or integer `#define`-style constants for sets of related values. For example:

```
public static final int RED = 0;
public static final int YELLOW = 1;
public static final int GREEN = 2;
```

While functional, this approach suffers from several issues:

- **Lack of type safety:** Any integer can be assigned, even if it doesn't represent a valid color.
- **Poor readability:** The meaning of values like `0`, `1`, or `2` is not self-evident.
- **No namespace grouping:** Constants are often spread across the class or interface, making maintenance harder.

Enums solve these problems elegantly:

- You can only assign valid enum constants to variables of that enum type.
- Enum constants have meaningful names improving code clarity.
- Enums group related constants into one logical unit.

Summary Reflection

Declaring `enum` types elevates your code from primitive constants to meaningful, type-safe abstractions. This clarity reduces bugs, improves readability, and aligns your design closer to the problem domain. Enums also serve as a foundation for more advanced features such as associating data and behavior with constants, which you'll explore in the next sections.

Using enums is a best practice for representing fixed sets of values in Java — a modern and robust alternative to traditional constant patterns.

11.2 Enum Constants

In Java, **enum constants** are the predefined fixed instances declared inside an enum type. Each constant represents a unique, immutable object of that enum class. Understanding how to define, access, and iterate over these constants is fundamental to leveraging enums effectively.

Defining Enum Constants

Enum constants are declared as comma-separated identifiers within the enum body, usually written in **all uppercase letters** to follow Java naming conventions for constants. For example:

```
public enum Season {
    WINTER,
    SPRING,
    SUMMER,
    FALL
}
```

Here, `WINTER`, `SPRING`, `SUMMER`, and `FALL` are the enum constants of type `Season`. They are implicitly `public static final` instances created by the Java compiler.

Accessing Enum Constants

You can access enum constants using the dot notation directly through the enum type:

```
Season currentSeason = Season.SUMMER;
System.out.println("The current season is " + currentSeason);
```

This assigns the constant `SUMMER` to the variable `currentSeason` and prints it. Enum constants are singleton instances, so comparison using `==` is safe and recommended:

```
if (currentSeason == Season.SUMMER) {
    System.out.println("Time for the beach!");
}
```

Iterating Over Enum Constants

Java provides a built-in static method called `values()` for every enum type, which returns an array of all declared constants in the order they are defined. This is useful for iteration:

```
for (Season s : Season.values()) {
    System.out.println(s);
}
```

Output:

```
WINTER
SPRING
SUMMER
FALL
```

Using `values()` enables you to loop through all possible enum constants, which is particularly helpful for displaying options or validating input.

Enum Constant Characteristics

- **Immutability:** Enum constants are immutable, meaning their state cannot change after creation. This ensures consistency and thread safety.
- **Uniqueness:** Each constant is a single, unique instance of the enum type. This guarantees reference equality with `==`.
- **Implicit `toString()` Implementation:** By default, calling `toString()` on an enum constant returns its name, e.g., `"SUMMER"`.

Naming Conventions

The standard convention for enum constants is to use uppercase letters with words separated by underscores (_), similar to static final constants. This convention improves readability and instantly signals their constant nature:

```
public enum TrafficSignal {
    RED,
    YELLOW,
    GREEN
}
```

Following naming conventions makes your code more consistent and easier to understand by other developers.

Full runnable code:

```
public class Main {
    // Define the enum
    public enum Season {
        WINTER,
        SPRING,
        SUMMER,
```

```
        FALL
    }

    public static void main(String[] args) {
        // Accessing an enum constant
        Season currentSeason = Season.SUMMER;
        System.out.println("The current season is " + currentSeason);

        // Comparison using ==
        if (currentSeason == Season.SUMMER) {
            System.out.println("Time for the beach!");
        }

        // Iterating over enum constants
        System.out.println("All seasons:");
        for (Season s : Season.values()) {
            System.out.println(s);
        }
    }
}
```

Summary Reflection

Enum constants provide a clean, type-safe, and immutable way to represent fixed sets of values. Their well-defined lifecycle—created once and never modified—makes them reliable throughout your application. The ability to iterate over all constants using `values()` simplifies many common programming tasks, like menus or state handling.

Moreover, the naming conventions for enums contribute to clear communication of their purpose in code, aligning with Java's broader emphasis on readability and maintainability.

By using enum constants properly, you ensure your code is both robust and expressive, avoiding the pitfalls of traditional integer or string constants.

11.3 Enums with Fields and Methods

Java enums are more than just simple collections of constants—they can have **fields**, **constructors**, and **methods**, allowing you to attach data and behavior directly to each enum constant. This makes enums powerful alternatives to small classes when you need to group related values with associated logic.

Adding Fields and Constructors

You can define fields inside an enum and set them via a private constructor. For example, consider a `Day` enum that stores a display name for each day:

```
public enum Day {
    MONDAY("Mon"),
    TUESDAY("Tue"),
```

```
    WEDNESDAY("Wed"),
    THURSDAY("Thu"),
    FRIDAY("Fri"),
    SATURDAY("Sat"),
    SUNDAY("Sun");

    private final String displayName;

    // Private constructor to set the displayName
    Day(String displayName) {
        this.displayName = displayName;
    }

    // Getter method for displayName
    public String getDisplayName() {
        return displayName;
    }
}
```

Each enum constant calls the constructor with a specific `displayName`. The constructor is implicitly private to prevent external instantiation.

Using Methods in Enums

Enums can have methods that operate on their fields or implement logic unique to the enum type. For instance:

```
public class Main {
    public static void main(String[] args) {
        Day today = Day.MONDAY;
        System.out.println("Today is " + today.getDisplayName());  // Output: Today is Mon
    }
}
```

This shows how you can retrieve a friendly name for each day through `getDisplayName()`. You can add more methods to your enum to encapsulate behavior related to its constants.

Full runnable code:

```
public class Main {

    public enum Day {
        MONDAY("Mon"),
        TUESDAY("Tue"),
        WEDNESDAY("Wed"),
        THURSDAY("Thu"),
        FRIDAY("Fri"),
        SATURDAY("Sat"),
        SUNDAY("Sun");

        private final String displayName;

        Day(String displayName) {
            this.displayName = displayName;
        }
```

```
        public String getDisplayName() {
            return displayName;
        }
    }

    public static void main(String[] args) {
        Day today = Day.MONDAY;
        System.out.println("Today is " + today.getDisplayName());

        System.out.println("All days:");
        for (Day day : Day.values()) {
            System.out.println(day.name() + " -> " + day.getDisplayName());
        }
    }
}
```

Reflection: Enums as Powerful Alternatives to Classes

By combining data and behavior, enums become self-contained, strongly typed objects, often eliminating the need for separate classes to represent fixed sets of related values. This:

- **Reduces boilerplate code:** No need for separate classes with fields and getters.
- **Improves maintainability:** Grouping related logic with constants helps keep code organized.
- **Enhances safety:** Enums enforce a fixed set of instances, preventing invalid values.

Overall, enums with fields and methods offer a concise yet expressive way to represent complex concepts that naturally fit into a fixed set of named instances.

11.4 Using Enums in `switch` Statements

Enums integrate seamlessly with Java's `switch` statement, providing a clear, type-safe way to branch logic based on a fixed set of predefined constants. Using enums in a `switch` block is both expressive and less error-prone compared to using primitive types like `int` or `String`.

Syntax and Basic Example

Here's a straightforward example that prints messages based on the day of the week using an enum:

```
public enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}

public class Main {
    public static void printMessage(Day day) {
        switch (day) {
            case MONDAY:
                System.out.println("Start of the workweek!");
```

```
                break;
            case FRIDAY:
                System.out.println("Almost weekend!");
                break;
            case SATURDAY:
            case SUNDAY:
                System.out.println("It's the weekend!");
                break;
            default:
                System.out.println("Midweek days are productive.");
                break;
        }
    }

    public static void main(String[] args) {
        printMessage(Day.MONDAY);    // Output: Start of the workweek!
        printMessage(Day.WEDNESDAY); // Output: Midweek days are productive.
        printMessage(Day.SUNDAY);    // Output: It's the weekend!
    }
}
```

How the Compiler Ensures Type Safety

When you use an enum in a `switch` statement, the compiler ensures:

- **Only valid enum constants are used** in `case` labels. If you try to switch on a value not part of the enum, the code won't compile.
- **The `switch` variable must be of the enum type** or compatible with it, preventing accidental mixing of unrelated types.
- **Exhaustiveness checking (optional in newer Java versions):** While the compiler does not force you to handle all enum constants, some IDEs and static analyzers can warn if certain constants are not covered in your `switch`, reducing the chance of missing cases.

This level of type safety prevents common bugs caused by mistyping string literals or incorrect integer constants.

Readability and Reduced Logic Errors

Using enums in `switch` statements makes your code:

- **More readable:** Named constants like `MONDAY` and `SUNDAY` are clearer than arbitrary numbers or strings.
- **Less error-prone:** No risk of typos that occur when using strings or incorrect numeric values.
- **Easier to maintain:** Adding or removing enum constants is straightforward, and IDE support helps update `switch` statements accordingly.
- **Supports grouping:** As shown in the example, multiple cases can share code (e.g., `SATURDAY` and `SUNDAY` both print "It's the weekend!"), improving conciseness.

Additional Notes on `switch` and Enums

- You can use `enum` values directly in `case` labels without prefixing them with the enum type (e.g., `case MONDAY:` instead of `case Day.MONDAY:`).
- The `default` case handles any unmatched enum values, although it may be unnecessary if you cover all constants explicitly.
- In Java 12 and later, the enhanced `switch` expression can return values directly, further improving clarity when working with enums.

Reflection

Switching on enums leverages Java's type system to create expressive and safe branching logic. Compared to switching on strings or integers, enums reduce the risk of runtime errors caused by invalid values, making your code robust and easier to understand. They serve as a natural fit for controlling program flow where a variable can only have a limited set of valid states.

Chapter 12.

Records Syntax (Java 14)

1. Declaring Records
2. Compact Constructors
3. Records vs Classes
4. Using Records for Data Aggregation

12 Records Syntax (Java 14)

12.1 Declaring Records

Introduced in Java 14 as a preview feature and standardized in Java 16, **records** provide a compact syntax for declaring classes whose main purpose is to **carry immutable data**. Records automatically generate much of the boilerplate code that traditional Java classes require, such as constructors, getters, `equals()`, `hashCode()`, and `toString()`. This makes records ideal for modeling simple data aggregates with minimal fuss.

Basic Syntax

Declaring a record is straightforward using the `record` keyword followed by the record name and a **header** listing its components (fields):

```
public record Point(int x, int y) { }
```

This single line defines a record named `Point` with two components, `x` and `y`, both of type `int`.

What the Record Provides Automatically

When you declare a record like the above, Java implicitly provides the following:

- **A canonical constructor** that takes parameters matching the components (`Point(int x, int y)`), initializing the fields.
- **Accessor methods** for each component, named exactly as the components (`x()` and `y()`), which return the respective values.
- **`equals(Object o)` and `hashCode()`** methods that consider all components, allowing meaningful equality checks and usage in hash-based collections.
- **A `toString()` method** that returns a string representation of the record including component names and values, e.g., `Point[x=3, y=5]`.

These generated methods ensure your record is a fully functional, immutable data carrier with minimal manual coding.

Using the Record

Here's an example of how to use the `Point` record:

```
public class Main {
    public static void main(String[] args) {
        Point p1 = new Point(3, 5);

        System.out.println(p1.x()); // Prints: 3
        System.out.println(p1.y()); // Prints: 5
        System.out.println(p1);     // Prints: Point[x=3, y=5]

        Point p2 = new Point(3, 5);
        System.out.println(p1.equals(p2)); // Prints: true
```

```
    }
}
```

Notice that to access the values, you call `x()` and `y()`, not `getX()` or `getY()`. This naming convention differs from traditional JavaBeans getters but is consistent across all records.

Full runnable code:

```
public class Main {

    // Record declaration
    public record Point(int x, int y) {}

    public static void main(String[] args) {
        Point p1 = new Point(3, 5);

        System.out.println("x: " + p1.x());      // Accessor method
        System.out.println("y: " + p1.y());
        System.out.println("Point: " + p1);      // toString()

        Point p2 = new Point(3, 5);
        System.out.println("p1 equals p2? " + p1.equals(p2));  // equals()
        System.out.println("HashCode of p1: " + p1.hashCode()); // hashCode()
    }
}
```

Immutability by Design

Records are **implicitly final**, which means you cannot extend a record class. Each component is also **final** and private under the hood, so once a record instance is created, its data cannot be changed.

This design enforces **immutability**, making records ideal for representing values or data transfer objects (DTOs) where thread safety and predictability are important.

Reflection on the Intent of Records

Records represent a significant evolution in Java's approach to data modeling:

- **Reducing boilerplate:** Records eliminate the need to manually write constructors, accessors, `equals()`, `hashCode()`, and `toString()`. This streamlines development and reduces potential errors in repetitive code.
- **Encouraging immutability:** Since records are immutable by default, they promote safer, more predictable designs.
- **Expressing intent:** Using a record clearly communicates that the type is a simple, transparent data carrier, not a complex entity with mutable state or intricate behavior.
- **Supporting pattern matching:** Records integrate well with newer Java features like pattern matching, further enhancing clarity and conciseness.

Limitations and Considerations

While records simplify many cases, they are not a replacement for all classes:

- You cannot define instance fields outside of the record components.
- Records cannot extend other classes (but can implement interfaces).
- Behavior should be minimal; records are meant for data, not heavy business logic.

In summary, records offer a powerful, concise way to define immutable data structures with built-in functionality, making your Java code cleaner, more maintainable, and expressive. This modern feature aligns Java with other languages that have had similar constructs for years, such as Kotlin's data classes or C#'s records.

12.2 Compact Constructors

Records in Java automatically generate a **canonical constructor** matching the record components, which assigns parameter values to fields. However, sometimes you need to add **validation** or **transformation logic** when creating instances. For this purpose, Java allows you to define a **compact constructor** inside a record.

What Is a Compact Constructor?

A compact constructor is a concise way to add code to the canonical constructor **without repeating the parameter list or field assignments**. It uses the record's component names directly and automatically assigns parameters to fields unless you explicitly override the assignments.

Example: Compact Constructor with Validation

```
public record Person(String name, int age) {
    public Person {
        if (name == null || name.isBlank()) {
            throw new IllegalArgumentException("Name cannot be null or blank");
        }
        if (age < 0) {
            throw new IllegalArgumentException("Age cannot be negative");
        }
        // Fields 'name' and 'age' are automatically assigned after this block
    }
}
```

In this example:

- We define a compact constructor with the syntax `public Person { ... }`.
- Inside, we validate the inputs.
- We **do not need to write** `this.name = name; this.age = age;` because this happens implicitly after the constructor body unless overridden.

Comparison with Canonical Constructors

A canonical constructor requires full parameter declaration and explicit field assignment:

```java
public record Person(String name, int age) {
    public Person(String name, int age) {
        if (name == null || name.isBlank()) {
            throw new IllegalArgumentException("Name cannot be null or blank");
        }
        if (age < 0) {
            throw new IllegalArgumentException("Age cannot be negative");
        }
        this.name = name;
        this.age = age;
    }
}
```

While this works, it is more verbose and repetitive, especially for records with many components.

Full runnable code:

```java
public class Main {

    public record Person(String name, int age) {
        public Person {
            if (name == null || name.isBlank()) {
                throw new IllegalArgumentException("Name cannot be null or blank");
            }
            if (age < 0) {
                throw new IllegalArgumentException("Age cannot be negative");
            }
            // Fields are assigned automatically
        }
    }

    public static void main(String[] args) {
        try {
            Person p1 = new Person("Alice", 30);
            System.out.println("Created person: " + p1);

            Person p2 = new Person("", 25); // Triggers exception
            System.out.println("Created person: " + p2);
        } catch (IllegalArgumentException e) {
            System.out.println("Validation failed: " + e.getMessage());
        }

        try {
            Person p3 = new Person("Bob", -5); // Triggers exception
            System.out.println("Created person: " + p3);
        } catch (IllegalArgumentException e) {
            System.out.println("Validation failed: " + e.getMessage());
        }
    }
}
```

Reflection

Compact constructors keep records **concise yet flexible** by allowing you to insert validation, normalization, or other logic directly at the construction phase without sacrificing the brevity

that makes records attractive. They promote clean and maintainable code, keeping the focus on data immutability and integrity.

12.3 Records vs Classes

With the introduction of **records** in Java 14, a new concise syntax for data-carrying classes has emerged. Records provide a lightweight way to model **immutable data aggregates** while reducing boilerplate code common in plain old Java objects (POJOs). This section compares **records** with traditional **classes**, highlighting their syntax differences, capabilities, and limitations, and reflects on when to prefer records over classes.

Syntax Comparison

Traditional Class:

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

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

    public String name() {
        return name;
    }

    public int age() {
        return age;
    }

    @Override
    public String toString() {
        return "Person[name=" + name + ", age=" + age + "]";
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Person)) return false;
        Person other = (Person) o;
        return age == other.age && name.equals(other.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, age);
    }
}
```

Record:

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

Notice how a record declaration is dramatically more concise, automatically generating:

- A canonical constructor
- Final private fields
- Public accessor methods (named after fields)
- `equals()`, `hashCode()`, and `toString()` implementations

What Records Can Do

- **Immutable data storage:** All fields are implicitly `private` and `final`.
- **Automatic implementations:** Constructors, getters, `equals()`, `hashCode()`, and `toString()` are generated.
- **Compact constructors:** Allow custom validation/logic with concise syntax.
- **Implement interfaces:** Records can implement interfaces just like classes.
- **Serialization support:** Records can be serialized like classes.

What Records Cannot Do

- **No mutable fields:** Records enforce immutability; you cannot declare non-final instance fields.
- **No explicit setters:** Since fields are final, you cannot mutate record fields after construction.
- **Cannot extend other classes:** Records implicitly extend `java.lang.Record` and cannot extend any other class.
- **Cannot be extended:** Records are implicitly `final`; you cannot create subclasses of a record.
- **Limited customization of state:** You cannot add instance fields beyond the declared components, though you can add static fields.
- **No support for inheritance hierarchy:** Records don't support traditional polymorphic class hierarchies.

When to Use Records

- **Data-centric modeling:** Records shine when your class is primarily a **simple data carrier** — for example, modeling points, configurations, or DTOs.
- **Reduce boilerplate:** For classes that require lots of boilerplate code for constructors, accessors, and common methods, records provide a **clean, declarative alternative**.
- **Immutability:** When you want to guarantee the object's state won't change after construction, records enforce this automatically.
- **Value-based equality:** If you need two objects to be considered equal based on their contents rather than their identity, records implement this by default.
- **Improved readability:** The concise syntax clarifies the intent, focusing on the data structure itself.

When to Prefer Classes

- **Mutable objects:** If your design requires mutable state or setters, records are not suitable.
- **Complex behavior or inheritance:** When you need to extend other classes, override behavior extensively, or use inheritance hierarchies, classes remain the right choice.
- **Custom state management:** If your class has non-canonical fields or needs to manage internal state beyond constructor parameters, classes provide more flexibility.
- **Legacy code or frameworks:** Some libraries or frameworks may have compatibility assumptions around traditional classes and might not fully support records.

Reflection

Records introduce a powerful tool for Java developers to write **clearer, safer, and more maintainable** code by focusing on immutable data and automatically handling common tasks like equality and string representation. They promote **value-based design** and reduce boilerplate, which leads to fewer errors and more concise code.

However, they do not replace traditional classes for all scenarios. The limitations around immutability, inheritance, and customization mean classes are still essential for more complex or mutable domain models.

In summary, use records when your class primarily models **data with fixed state**, and prefer classes when your design requires richer behavior or mutability. This thoughtful use of both enables you to write idiomatic, clean, and robust Java code.

12.4 Using Records for Data Aggregation

Records are an elegant and concise way to group related data into a single, immutable object. This feature makes them ideal for **data aggregation**—combining multiple values into one logical unit that can be passed around your application safely and clearly.

Grouping Multiple Values: Example with Coordinates

Imagine you are working on a program that processes points on a 2D plane. Traditionally, you might create a class like this:

```
public class Point {
    private final double x;
    private final double y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double getX() { return x; }
    public double getY() { return y; }
```

```
    @Override
    public String toString() {
        return "(" + x + ", " + y + ")";
    }
}
```

With records, this can be simplified drastically:

```
public record Point(double x, double y) {}
```

This single line creates a fully immutable type with two components, `x` and `y`, and automatically generates constructor, accessor methods, `equals()`, `hashCode()`, and `toString()`.

Simplifying Method Returns

Records can make method signatures and implementations clearer by returning multiple values without needing to create separate classes manually:

```
public class Geometry {
    public static Point midpoint(Point a, Point b) {
        return new Point(
            (a.x() + b.x()) / 2,
            (a.y() + b.y()) / 2
        );
    }
}
```

Here, `midpoint` returns a `Point` record that clearly groups the two coordinates. Before records, you might have returned an array or a custom class, both less clear or more verbose.

Grouping Complex Data: User Information

Records also work well for aggregating more complex data, such as user profiles:

```
public record User(String username, String email, int age) {}
```

In this example, a `User` record can be used throughout your application to represent user data consistently, safely, and without risk of accidental modification.

Using Records in Collections

Records are especially powerful when used in collections like lists or maps:

```
List<User> users = List.of(
    new User("alice", "alice@example.com", 30),
    new User("bob", "bob@example.com", 25)
);

Map<String, Point> cityCoordinates = Map.of(
    "New York", new Point(40.7128, -74.0060),
    "Los Angeles", new Point(34.0522, -118.2437)
);
```

Because records provide reliable implementations of `equals()` and `hashCode()`, they work seamlessly as keys or values in collections without requiring extra code.

Records and Functional-Style Programming

Records promote **functional programming principles** in Java by emphasizing **immutability** and **value-based equality**. Since record components are `final`, you cannot change the state after construction, eliminating side effects and making your programs easier to reason about.

When paired with streams and lambda expressions, records enable concise, declarative code for data transformation and aggregation:

```
List<User> adults = users.stream()
    .filter(user -> user.age() >= 18)
    .collect(Collectors.toList());
```

This code filters a list of immutable `User` records to those who are adults, leveraging the safety and clarity records provide.

Full runnable code:

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

public class Main {

    public record Point(double x, double y) {}

    public record User(String username, String email, int age) {}

    public static class Geometry {
        public static Point midpoint(Point a, Point b) {
            return new Point(
                (a.x() + b.x()) / 2,
                (a.y() + b.y()) / 2
            );
        }
    }

    public static void main(String[] args) {
        // Example 1: Returning a record from a method
        Point p1 = new Point(2, 4);
        Point p2 = new Point(6, 8);
        Point mid = Geometry.midpoint(p1, p2);
        System.out.println("Midpoint: " + mid);

        // Example 2: Grouping data with a record
        List<User> users = List.of(
            new User("alice", "alice@example.com", 30),
            new User("bob", "bob@example.com", 17),
            new User("carol", "carol@example.com", 22)
        );

        System.out.println("\nAll users:");
```

```
        users.forEach(System.out::println);

        // Example 3: Using records in collections and functional style
        List<User> adults = users.stream()
            .filter(user -> user.age() >= 18)
            .collect(Collectors.toList());

        System.out.println("\nAdults:");
        adults.forEach(System.out::println);

        Map<String, Point> cityCoordinates = Map.of(
            "New York", new Point(40.7128, -74.0060),
            "Los Angeles", new Point(34.0522, -118.2437)
        );

        System.out.println("\nCity coordinates:");
        cityCoordinates.forEach((city, point) ->
            System.out.println(city + ": " + point));
    }
}
```

Reflection: The Role of Records in Modern Java

Records fill an important gap in Java by providing a **native, language-level construct** optimized for grouping data. They reduce boilerplate, improve readability, and support immutability, all of which are hallmarks of modern software design.

Using records for data aggregation helps developers:

- **Avoid error-prone mutable state:** Since records are immutable, they prevent bugs from unintended modifications.
- **Express intent clearly:** The concise syntax signals the purpose of the class as a pure data carrier.
- **Write safer APIs:** Returning and accepting records in method signatures makes contracts explicit and reliable.
- **Improve maintainability:** Less boilerplate and clearer semantics reduce the cognitive load on maintainers.

In summary, records provide a **streamlined way to represent aggregates of related data** in Java, perfectly suited to both simple and complex use cases. Their design aligns with contemporary programming styles that prioritize immutability, clarity, and expressive code, making them an essential tool in any modern Java programmer's toolkit.

Chapter 13.

Exception Handling Syntax

1. `try`, `catch`, and `finally`
2. Multiple Catch Blocks
3. `throw` and `throws`
4. Creating Custom Exceptions

13 Exception Handling Syntax

13.1 `try`, `catch`, and `finally`

Exception handling is a core feature in Java that helps manage runtime errors gracefully, preventing program crashes and enabling more robust code. The basic structure involves three key blocks: `try`, `catch`, and `finally`.

The Basic Structure

- **`try` block:** Contains code that might throw an exception. You "try" to execute this code safely.
- **`catch` block:** Handles specific exceptions thrown within the `try`. You can have multiple `catch` blocks for different exception types.
- **`finally` block:** Contains code that always runs after the `try` and `catch`, regardless of whether an exception occurred. It's used for cleanup tasks.

Syntax Example

```
try {
    // Code that may throw an exception
} catch (ExceptionType1 e1) {
    // Handle ExceptionType1
} catch (ExceptionType2 e2) {
    // Handle ExceptionType2
} finally {
    // Cleanup code, always executed
}
```

Practical Example: Division by Zero

Consider a program that divides two integers entered by the user. Dividing by zero throws an `ArithmeticException`, which we can catch and handle:

```
public class DivisionExample {
    public static void main(String[] args) {
        int numerator = 10;
        int denominator = 0;
        try {
            int result = numerator / denominator;  // This throws ArithmeticException
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Error: Cannot divide by zero.");
        } finally {
            System.out.println("Execution of try-catch-finally block completed.");
        }
        System.out.println("Program continues...");
    }
}
```

Output:

```
Error: Cannot divide by zero.
Execution of try-catch-finally block completed.
Program continues...
```

Flow Explanation

- The `try` block attempts to execute the division.
- When division by zero occurs, an `ArithmeticException` is thrown.
- The `catch` block matching `ArithmeticException` catches it, preventing a program crash, and prints an error message.
- The `finally` block executes afterward, whether an exception was caught or not.
- After the whole block, the program continues normally.

Another Example: Array Access

Exception handling is also useful for handling invalid array indices:

```java
try {
    int[] numbers = {1, 2, 3};
    System.out.println(numbers[5]);  // ArrayIndexOutOfBoundsException
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Invalid array index accessed.");
} finally {
    System.out.println("Cleanup or final operations go here.");
}
```

This prevents the program from terminating unexpectedly and allows graceful recovery.

Why Use Exception Handling?

- **Improves program stability:** Instead of abrupt crashes, exceptions can be caught and managed, enabling the program to handle unexpected conditions smoothly.
- **Centralizes error handling:** Using `catch` blocks allows error processing in a controlled, organized manner.
- **Ensures cleanup:** The `finally` block guarantees that important finalization code (like closing files or releasing resources) runs no matter what.
- **Separates error logic from business logic:** This separation improves code readability and maintainability.

Key Points About `finally`

- The `finally` block always executes after `try` and `catch`, even if the `try` block returns early or an exception is rethrown.
- It is ideal for closing files, database connections, or other resources that must be released.
- If `System.exit()` is called in the `try` or `catch`, the JVM will terminate immediately, and `finally` will not execute.

Reflection

Exception handling is fundamental in building reliable Java applications. By anticipating errors and preparing responses, you can maintain control flow integrity and avoid unexpected failures. The combination of `try`, `catch`, and `finally` blocks not only safeguards program execution but also fosters cleaner, more maintainable code. This structure allows developers to separate normal logic from error recovery and cleanup, which ultimately leads to more robust and professional-grade applications.

13.2 Multiple Catch Blocks

Java's exception handling mechanism allows you to catch and handle different types of exceptions that might be thrown in your program. Using multiple `catch` blocks enables fine-grained control over how various exceptions are handled, improving program robustness and clarity.

Basic Multiple Catch Blocks

You can write several `catch` blocks after a single `try` block, each tailored to handle a specific type of exception. The Java runtime will execute the first `catch` block with an exception type matching the thrown exception.

```
try {
    // Code that may throw exceptions
} catch (FileNotFoundException e) {
    System.out.println("File not found: " + e.getMessage());
} catch (IOException e) {
    System.out.println("I/O error: " + e.getMessage());
} catch (Exception e) {
    System.out.println("General error: " + e.getMessage());
}
```

In this example:

- If a `FileNotFoundException` occurs, the first `catch` block executes.
- If any other `IOException` occurs, the second block executes.
- If any other exception (subclass of `Exception`) occurs, the last block executes.

Ordering of Catch Blocks: Most Specific First

It is important to order your `catch` blocks from **most specific** to **most general**. This ensures that exceptions are caught by the most appropriate handler.

Why?

If a general exception type (like `Exception`) is caught before a more specific one (like `FileNotFoundException`), the more specific block becomes unreachable, leading to a compile-time error.

Example of incorrect order causing compile error:

```
try {
    // ...
} catch (Exception e) {  // Catches all exceptions first
    // ...
} catch (IOException e) {  // Unreachable: compile error
    // ...
}
```

The compiler prevents this because the second block can never be reached.

Multi-Catch Syntax (`catch (IOException | SQLException e)`)

Introduced in Java 7, multi-catch lets you handle multiple exception types in a single `catch` block using the pipe (`|`) operator:

```
try {
    // Code that may throw IOException or SQLException
} catch (IOException | SQLException e) {
    System.out.println("Error occurred: " + e.getMessage());
}
```

Benefits:

- Reduces code duplication when handling different exceptions similarly.
- Keeps code concise and easier to maintain.

Restrictions:

- The exception types in a multi-catch must not have a subclass-superclass relationship.
- The caught exception variable `e` is implicitly final—you cannot assign a new value to it inside the catch block.

Practical Example

```
import java.io.*;
import java.sql.*;

public class MultiCatchExample {
    public static void main(String[] args) {
        try {
            readFile("data.txt");
            queryDatabase("SELECT * FROM users");
        } catch (FileNotFoundException e) {
            System.out.println("File missing: " + e.getMessage());
        } catch (IOException e) {
            System.out.println("I/O error: " + e.getMessage());
        } catch (SQLException e) {
            System.out.println("Database error: " + e.getMessage());
        } catch (Exception e) {
            System.out.println("Unexpected error: " + e.getMessage());
        }
    }
```

```
    static void readFile(String filename) throws IOException {
        // Simulate file operation
        throw new FileNotFoundException("File " + filename + " not found.");
    }

    static void queryDatabase(String query) throws SQLException {
        // Simulate database operation
        throw new SQLException("Invalid SQL query.");
    }
}
```

Here, exceptions are handled with granularity, allowing different messages based on the exact error type.

Using multi-catch, the above could be simplified if `FileNotFoundException` and `IOException` were handled the same way:

```
try {
    readFile("data.txt");
    queryDatabase("SELECT * FROM users");
} catch (FileNotFoundException | SQLException e) {
    System.out.println("Error: " + e.getMessage());
} catch (IOException e) {
    System.out.println("I/O error: " + e.getMessage());
} catch (Exception e) {
    System.out.println("Unexpected error: " + e.getMessage());
}
```

Reflection: Why Use Multiple Catch Blocks?

- **Clarity:** Different `catch` blocks clearly communicate how various errors are handled, making the code more understandable.
- **Granularity:** Allows tailored recovery or logging for specific errors.
- **Maintainability:** You can modify the handling logic for one exception type without affecting others.
- **Safety:** Ensures that the program responds correctly depending on the exception type, improving robustness.

Summary

Multiple `catch` blocks and multi-catch syntax enhance Java's exception handling by offering:

- The ability to **distinguish between exception types** and respond accordingly.
- **Cleaner, more readable code** by avoiding nested `try-catch` or excessive general `catch` blocks.
- **Control over exception handling flow**, respecting exception hierarchies.

Properly ordering catch blocks and using multi-catch where appropriate are best practices that help produce reliable, maintainable, and clear Java applications.

13.3 `throw` and `throws`

Exception handling in Java not only involves catching exceptions but also explicitly raising (throwing) them and declaring which exceptions a method might propagate. The keywords `throw` and `throws` serve these purposes.

The `throw` Statement: Manually Raising Exceptions

The `throw` keyword is used inside a method or block to **manually raise an exception**. When a `throw` statement executes, it immediately stops the current flow and passes control to the nearest matching `catch` block or propagates the exception up the call stack.

Syntax:

```
throw new ExceptionType("Error message");
```

Example:

```
public void setAge(int age) {
    if (age < 0) {
        throw new IllegalArgumentException("Age cannot be negative");
    }
    this.age = age;
}
```

In this example, if an invalid age is passed, the method explicitly throws an `IllegalArgumentException`, signaling a problem to the caller.

The `throws` Clause: Declaring Exceptions in Method Signatures

The `throws` keyword is used in a method declaration to **indicate that the method might throw one or more exceptions**. This informs callers that they need to handle or further declare these exceptions.

Syntax:

```
public void methodName() throws IOException, SQLException {
    // method code that might throw IOException or SQLException
}
```

Example with checked exceptions:

```
public void readFile(String filename) throws IOException {
    FileReader file = new FileReader(filename); // may throw FileNotFoundException (subclass of IOExcep
    // Reading logic...
}
```

Here, `readFile` declares that it throws `IOException`. Any code that calls `readFile` must either handle the exception with a try-catch block or declare it further up.

Checked vs. Unchecked Exceptions

Java distinguishes between **checked** and **unchecked** exceptions, which affects how `throw` and `throws` are used.

- **Checked Exceptions:** Subclasses of `Exception` but *not* `RuntimeException`. The compiler forces you to declare or handle these exceptions. Examples include `IOException`, `SQLException`, and `ClassNotFoundException`.
- **Unchecked Exceptions:** Subclasses of `RuntimeException` or `Error`. These exceptions do not require declaration or mandatory handling. Examples are `NullPointerException`, `IllegalArgumentException`, `ArithmeticException`.

Implications:

- For **checked exceptions**, you **must** use `throws` in the method signature if the method can throw them, and callers must handle or declare them.
- For **unchecked exceptions**, you can use `throw` to raise them but do **not** need to declare them with `throws`.

Why Use `throw` and `throws`?

- **`throw` improves clarity and control:** It lets you signal error conditions deliberately. For instance, when validating input or detecting illegal states, throwing an exception stops incorrect processing immediately.
- **`throws` enforces safety:** Declaring checked exceptions forces the caller to be aware of potential failure points. This helps prevent unchecked runtime crashes and encourages robust error handling.

Practical Example Combining `throw` and `throws`

```
import java.io.*;

public class FileProcessor {

    // Declare that this method throws IOException (checked)
    public void processFile(String filename) throws IOException {
        if (filename == null || filename.isEmpty()) {
            // Throw unchecked exception: IllegalArgumentException
            throw new IllegalArgumentException("Filename cannot be null or empty");
        }
        FileReader reader = new FileReader(filename); // May throw FileNotFoundException
        // Read and process file...
    }

    public static void main(String[] args) {
        FileProcessor processor = new FileProcessor();

        try {
            processor.processFile("");
        } catch (IllegalArgumentException e) {
```

```
            System.out.println("Invalid argument: " + e.getMessage());
        } catch (IOException e) {
            System.out.println("I/O failure: " + e.getMessage());
        }
    }
}
```

Explanation:

- `processFile` throws an unchecked `IllegalArgumentException` if the input is invalid — no need to declare this in `throws`.
- It declares `throws IOException` because opening a file might throw a checked exception.
- The caller handles both exceptions appropriately.

Reflection on Exception Handling Design

The separation between `throw` and `throws` along with checked and unchecked exceptions is a key design aspect of Java's error handling:

- **Checked exceptions** improve program safety by making error handling explicit.
- **Unchecked exceptions** represent programming errors (e.g., null dereference) that usually cannot be recovered from, so they don't clutter method signatures.
- Proper use of `throw` and `throws` helps create clear contracts between methods and callers, documenting what can go wrong and ensuring that exceptional conditions are not silently ignored.

13.3.1 Summary

- Use **`throw`** to raise exceptions manually within methods.
- Use **`throws`** in method declarations to specify checked exceptions that must be handled or propagated.
- Understand the difference between **checked** and **unchecked** exceptions and their implications on declaring exceptions.
- Combining these mechanisms leads to more robust, maintainable, and predictable Java programs by making error conditions explicit and manageable.

13.4 Creating Custom Exceptions

In Java, while the standard exception classes cover many common error conditions, sometimes your application requires **custom exceptions** to represent specific problems unique to your domain or business logic. Custom exceptions improve clarity, enable more precise error handling, and make your code more expressive.

Defining a Custom Exception Class

To create a custom exception, you define a new class that **extends** either:

- `Exception` — to create a **checked exception** that must be declared or handled.
- `RuntimeException` — to create an **unchecked exception** that does not require explicit handling.

Basic syntax:

```
// Checked exception
public class InvalidUserInputException extends Exception {
    public InvalidUserInputException(String message) {
        super(message);
    }
}

// Unchecked exception
public class InsufficientBalanceException extends RuntimeException {
    public InsufficientBalanceException(String message) {
        super(message);
    }
}
```

These classes typically provide constructors that pass a message or cause to the superclass, making them easy to instantiate and throw with informative error details.

Example: Throwing and Catching a Custom Exception

Let's create a simple banking example where withdrawing more money than available balance raises a custom unchecked exception:

```
// Custom unchecked exception
public class InsufficientBalanceException extends RuntimeException {
    public InsufficientBalanceException(String message) {
        super(message);
    }
}

public class BankAccount {
    private double balance;

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

    public void withdraw(double amount) {
        if (amount > balance) {
            throw new InsufficientBalanceException(
                "Withdrawal amount exceeds available balance"
            );
        }
        balance -= amount;
    }

    public double getBalance() {
```

```
        return balance;
    }
}

public class Main {
    public static void main(String[] args) {
        BankAccount account = new BankAccount(1000);

        try {
            account.withdraw(1500);
        } catch (InsufficientBalanceException e) {
            System.out.println("Transaction failed: " + e.getMessage());
        }
    }
}
```

What happens here?

- When attempting to withdraw 1500, the `withdraw` method throws the `InsufficientBalanceExceptio`
- The exception is caught in the `try-catch` block in `main`.
- A meaningful message is printed to inform the user about the failure.

Checked vs. Unchecked Custom Exceptions

You can create either **checked** or **unchecked** custom exceptions depending on your error-handling needs:

- **Checked exceptions** (extending `Exception`) enforce explicit handling or declaration. Use these when the caller can reasonably recover or take alternative action.
- **Unchecked exceptions** (extending `RuntimeException`) are for programming errors or conditions that generally cannot be anticipated or recovered from. They do not require explicit handling.

For example, if invalid user input is recoverable, you might create:

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

And force callers to handle or propagate it.

Why Create Custom Exceptions?

Custom exceptions are not just a formalism—they make your business logic **more meaningful and expressive**:

- They **document intent** clearly. Instead of a generic `Exception` or `RuntimeException`, a custom exception tells readers exactly what went wrong.
- They **enable granular error handling**. Catch specific exceptions to respond differently depending on the failure type.

- They **improve debugging**. Custom exceptions carry semantic names and can include additional fields or methods to convey extra context.
- They **help separate concerns** by encapsulating error conditions related to your domain logic.

Best Practices for Custom Exceptions

- **Provide multiple constructors:** Include ones with just a message, with a message and cause (`Throwable`), and a no-argument constructor for flexibility.
- **Avoid excessive custom exceptions:** Use them judiciously to avoid cluttering your codebase with too many types.
- **Document exceptions:** Clearly document when and why exceptions are thrown to aid maintainers and users of your API.
- **Consider checked vs. unchecked carefully:** Base your choice on how recoverable the exception is.

13.4.1 Summary

- Custom exceptions are created by extending `Exception` (checked) or `RuntimeException` (unchecked).
- They allow you to represent application-specific error conditions with meaningful types.
- Throwing and catching custom exceptions enhances program clarity, expressiveness, and error management.
- Thoughtful use of custom exceptions leads to cleaner, more maintainable, and more robust code aligned with your business logic.

Chapter 14.

Generics Syntax

1. Generic Classes
2. Generic Methods
3. Bounded Type Parameters (`T extends Number`)
4. Wildcards: `?`, `? extends`, `? super`

14 Generics Syntax

14.1 Generic Classes

Generics are a powerful feature in Java that enable you to write **type-safe**, reusable code by allowing classes, interfaces, and methods to operate on **parameterized types**. Instead of working with raw `Object` types and casting manually, generics allow you to specify a placeholder for the type that the class will handle, making your code cleaner and less error-prone.

Defining a Generic Class

A **generic class** is declared by adding a **type parameter** in angle brackets `<T>` after the class name. The type parameter `T` is a placeholder that gets replaced by a concrete type when an object of that class is instantiated.

Syntax:

```
public class Box<T> {
    private T content;

    public Box(T content) {
        this.content = content;
    }

    public T getContent() {
        return content;
    }

    public void setContent(T content) {
        this.content = content;
    }
}
```

Here, `Box<T>` is a generic class with a type parameter `T`. The field `content` can hold any type that replaces `T`. The constructor, getter, and setter also work with this generic type.

Using a Generic Class with Different Types

When creating instances of a generic class, you specify the actual type to use in place of `T`:

```
public class Main {
    public static void main(String[] args) {
        Box<String> stringBox = new Box<>("Hello");
        System.out.println(stringBox.getContent());  // Output: Hello

        Box<Integer> intBox = new Box<>(123);
        System.out.println(intBox.getContent());     // Output: 123

        Box<Double> doubleBox = new Box<>(45.67);
        System.out.println(doubleBox.getContent());  // Output: 45.67
    }
}
```

Here we created three `Box` objects with different types: `String`, `Integer`, and `Double`. The compiler enforces type safety, ensuring that only the declared type is used. For example, you cannot accidentally put an `Integer` into a `Box<String>`.

Generic Classes with Multiple Type Parameters

You can also declare generic classes with **multiple type parameters**, allowing more flexibility:

```
public class Pair<K, V> {
    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey() {
        return key;
    }

    public V getValue() {
        return value;
    }
}
```

Usage example:

```
Pair<String, Integer> entry = new Pair<>("Age", 30);
System.out.println(entry.getKey() + ": " + entry.getValue());  // Output: Age: 30

Pair<Integer, String> reversedEntry = new Pair<>(100, "Score");
System.out.println(reversedEntry.getKey() + ": " + reversedEntry.getValue());  // Output: 100: Score
```

This `Pair` class can hold any two types, named `K` and `V` here for key and value, but you can use any valid identifier for type parameters.

Full runnable code:

```
public class GenericDemo {

    // Generic Box class
    public static class Box<T> {
        private T content;

        public Box(T content) {
            this.content = content;
        }

        public T getContent() {
            return content;
        }

        public void setContent(T content) {
```

```
            this.content = content;
        }
    }

    // Generic Pair class with two type parameters
    public static class Pair<K, V> {
        private K key;
        private V value;

        public Pair(K key, V value) {
            this.key = key;
            this.value = value;
        }

        public K getKey() {
            return key;
        }

        public V getValue() {
            return value;
        }
    }

    public static void main(String[] args) {
        // Using Box with different types
        Box<String> stringBox = new Box<>("Hello");
        Box<Integer> intBox = new Box<>(123);
        Box<Double> doubleBox = new Box<>(45.67);

        System.out.println("Box Contents:");
        System.out.println("String: " + stringBox.getContent());
        System.out.println("Integer: " + intBox.getContent());
        System.out.println("Double: " + doubleBox.getContent());

        // Using Pair with different type combinations
        Pair<String, Integer> entry = new Pair<>("Age", 30);
        Pair<Integer, String> reversedEntry = new Pair<>(100, "Score");

        System.out.println("\nPair Entries:");
        System.out.println(entry.getKey() + ": " + entry.getValue());
        System.out.println(reversedEntry.getKey() + ": " + reversedEntry.getValue());
    }
}
```

Benefits of Using Generic Classes

1. **Type Safety** Generics eliminate the need for explicit casting and reduce the risk of `ClassCastException` at runtime. The compiler checks that only the correct types are used, catching errors early.
2. **Code Reuse** Instead of writing separate classes for each data type (e.g., `StringBox`, `IntegerBox`), you write one generic class that works for all types, improving maintainability and reducing boilerplate code.
3. **Expressiveness** Generics communicate your intent clearly. When you see `Box<String>`, you immediately know what kind of data that box holds, improving readability.

4. **Interoperability with Collections** Java Collections API heavily uses generics, so understanding generic classes is essential for working effectively with lists, sets, maps, and more.

Important Notes

- Generics use **type erasure**: at runtime, the generic type parameters are erased and replaced with their bounds or `Object`. This means you cannot use generics for certain operations (like creating arrays of generic types).
- Generic type parameters are **invariant**: `Box<Number>` is not a superclass or subclass of `Box<Integer>`, even though `Integer` is a subclass of `Number`. This is important to understand when working with generic collections.

14.1.1 Summary

- A **generic class** defines one or more **type parameters** in angle brackets `<T>`, which are placeholders for types specified upon instantiation.
- You can instantiate generic classes with different concrete types, ensuring **type safety** without casting.
- Generic classes support **multiple type parameters**, enabling flexible data structures like pairs or maps.
- Using generics improves **code reuse**, **readability**, and helps catch type errors at compile time.
- Understanding generics is crucial for leveraging Java's collections and writing robust, maintainable code.

Generics transform Java from a language with raw types to a much more type-safe and expressive one, paving the way for modern Java programming.

14.2 Generic Methods

Generic methods extend the power of generics by allowing you to define methods with their own type parameters—**independent** of whether the enclosing class is generic or not. This means you can write a **single method** that works with various types, improving flexibility and code reuse without having to make the entire class generic.

Syntax of a Generic Method

A generic method declares its type parameter(s) before the return type, enclosed in angle brackets `<>`. This signals to the compiler that the method can operate on one or more type parameters.

Basic syntax:

```
public <T> void printArray(T[] array) {
    for (T element : array) {
        System.out.println(element);
    }
}
```

Here, `<T>` declares a generic type parameter `T` just for the method `printArray`. The method takes an array of `T` elements and prints each one. Note that `T` is a **method-level type parameter**, separate from any generic types the class might have.

Generic Methods Inside Non-Generic Classes

You can define generic methods inside **regular (non-generic) classes**. This is common when the generic behavior is limited to a few methods rather than the whole class.

Example:

```
public class Utility {

    public static <T> void printArray(T[] array) {
        for (T element : array) {
            System.out.print(element + " ");
        }
        System.out.println();
    }

    public static <T> T getFirstElement(T[] array) {
        if (array == null || array.length == 0) {
            return null;
        }
        return array[0];
    }
}
```

Usage:

```
public class Main {
    public static void main(String[] args) {
        Integer[] intArray = {1, 2, 3, 4};
        String[] strArray = {"apple", "banana", "cherry"};

        Utility.printArray(intArray);  // Output: 1 2 3 4
        Utility.printArray(strArray);  // Output: apple banana cherry

        System.out.println("First int: " + Utility.getFirstElement(intArray));  // Output: First int: 1
        System.out.println("First string: " + Utility.getFirstElement(strArray));  // Output: First str
    }
}
```

Notice that the `Utility` class itself is **not generic**, but the methods are. This pattern is useful for stateless utility methods that operate on different types.

Full runnable code:

```java
public class GenericMethodDemo {

    // Utility class with generic methods
    public static class Utility {

        public static <T> void printArray(T[] array) {
            for (T element : array) {
                System.out.print(element + " ");
            }
            System.out.println();
        }

        public static <T> T getFirstElement(T[] array) {
            if (array == null || array.length == 0) {
                return null;
            }
            return array[0];
        }
    }

    // Main method to demonstrate generic utility methods
    public static void main(String[] args) {
        Integer[] intArray = {1, 2, 3, 4};
        String[] strArray = {"apple", "banana", "cherry"};

        System.out.println("Integer Array:");
        Utility.printArray(intArray);  // Output: 1 2 3 4

        System.out.println("\nString Array:");
        Utility.printArray(strArray);  // Output: apple banana cherry

        System.out.println("\nFirst int: " + Utility.getFirstElement(intArray));     // Output: First i
        System.out.println("First string: " + Utility.getFirstElement(strArray));    // Output: First s
    }
}
```

Generic Methods with Multiple Type Parameters

Methods can have multiple type parameters, separated by commas:

```java
public class PairUtil {

    public static <K, V> void printPair(K key, V value) {
        System.out.println("Key: " + key + ", Value: " + value);
    }
}
```

Usage:

```java
PairUtil.printPair("Name", "Alice");
PairUtil.printPair(1001, 99.5);
```

This method works with any combination of key and value types.

Generic Return Types

Generic methods can also return values of generic types. The compiler enforces type safety when the method is called, so you get the correct type inferred automatically:

```
public static <T> T getMiddleElement(T[] array) {
    if (array == null || array.length == 0) {
        return null;
    }
    return array[array.length / 2];
}
```

Usage:

```
String[] fruits = {"apple", "banana", "cherry"};
String middle = Utility.getMiddleElement(fruits);
System.out.println(middle);  // Output: banana
```

The method returns a value of the generic type `T`, inferred by the compiler based on the argument type.

Full runnable code:

```
public class GenericReturnDemo {

    // Utility class with a generic method that returns the middle element
    public static class Utility {

        public static <T> T getMiddleElement(T[] array) {
            if (array == null || array.length == 0) {
                return null;
            }
            return array[array.length / 2];
        }
    }

    public static void main(String[] args) {
        String[] fruits = {"apple", "banana", "cherry"};
        Integer[] numbers = {10, 20, 30, 40, 50};

        String middleFruit = Utility.getMiddleElement(fruits);
        Integer middleNumber = Utility.getMiddleElement(numbers);

        System.out.println("Middle fruit: " + middleFruit);   // Output: banana
        System.out.println("Middle number: " + middleNumber); // Output: 30
    }
}
```

Benefits of Generic Methods

1. **Flexibility Without Class-Level Generics** You can keep your classes simple and non-generic but still write reusable, type-safe methods that work with any type.
2. **Code Reuse and Reduced Duplication** Instead of writing overloaded methods for different types, a generic method handles them all in one place.

3. **Improved Type Safety** The compiler ensures you don't accidentally mix incompatible types, eliminating many potential runtime errors.
4. **Clearer APIs** Using generic methods makes your APIs expressive about the types they handle, which aids in maintenance and readability.

Important Notes

- The type parameter(s) of a generic method are **declared before the return type**.
- Generic methods can be **static or instance** methods.
- When calling generic methods, you usually don't need to specify the type explicitly; Java infers it based on arguments.
- If needed, you can specify type arguments explicitly:

```
Utility.<String>printArray(new String[]{"a", "b", "c"});
```

but this is rarely required.

14.2.1 Summary

- A **generic method** declares its own type parameter(s)** inside a regular or generic class.
- Syntax involves putting the type parameter before the return type, e.g., `<T> void methodName(...)`.
- Generic methods enable flexible, reusable, and type-safe code without making the whole class generic.
- They support multiple type parameters and can return generic types.
- This approach leads to cleaner, safer code and is especially common in utility classes and libraries.

Generic methods expand your programming toolkit, allowing you to write more generalized code while maintaining strong typing guarantees, making Java development both easier and safer.

14.3 Bounded Type Parameters (`T extends Number`)

In Java generics, **bounded type parameters** allow you to restrict the types that can be used as arguments for a generic type. This gives you more control over what types are valid, enabling you to write methods or classes that operate only on a subset of types—while maintaining type safety and flexibility.

What Are Bounded Type Parameters?

A bounded type parameter uses the `extends` keyword to limit the types you can use. Despite the keyword `extends`, the bound works for both classes and interfaces (including abstract classes), and it means **"is a subtype of"**.

For example, if you write:

```
<T extends Number>
```

it means that `T` can be any class that **is a subclass of** `Number` or `Number` itself. This includes types like `Integer`, `Double`, `Float`, etc., but **not** types like `String` or custom classes unrelated to `Number`.

Why Use Bounded Type Parameters?

Suppose you want a method that works only with numeric types, so you can perform numeric operations safely. Without bounds, your generic method would accept any type, including types that cannot be used numerically. Bounded type parameters solve this problem by restricting acceptable types, enabling you to use numeric methods like `doubleValue()` or `intValue()` on the generic parameter.

Example: Generic Method with Numeric Bound

Here is a simple generic method that calculates the sum of an array of numbers:

```
public class MathUtils {

    // T must extend Number, so only numeric types are allowed
    public static <T extends Number> double sumArray(T[] numbers) {
        double sum = 0.0;
        for (T number : numbers) {
            sum += number.doubleValue();  // safe because T extends Number
        }
        return sum;
    }
}
```

Usage:

```
public class Main {
    public static void main(String[] args) {
        Integer[] intArray = {1, 2, 3, 4};
        Double[] doubleArray = {1.5, 2.5, 3.5};

        System.out.println(MathUtils.sumArray(intArray));    // Output: 10.0
        System.out.println(MathUtils.sumArray(doubleArray)); // Output: 7.5

        // The following line would cause a compile-time error:
        // String[] strArray = {"a", "b", "c"};
        // MathUtils.sumArray(strArray); // Not allowed because String doesn't extend Number
    }
}
```

Full runnable code:

```
public class NumericSumDemo {

    // Math utility class with a generic method to sum numeric arrays
    public static class MathUtils {

        // T must extend Number to ensure only numeric types are accepted
        public static <T extends Number> double sumArray(T[] numbers) {
            double sum = 0.0;
            for (T number : numbers) {
                sum += number.doubleValue();  // safe conversion for all Number subclasses
            }
            return sum;
        }
    }

    public static void main(String[] args) {
        Integer[] intArray = {1, 2, 3, 4};
        Double[] doubleArray = {1.5, 2.5, 3.5};

        System.out.println("Sum of intArray: " + MathUtils.sumArray(intArray));      // Output: 10.0
        System.out.println("Sum of doubleArray: " + MathUtils.sumArray(doubleArray)); // Output: 7.5

        // The following would fail to compile because String is not a subclass of Number
        // String[] strArray = {"a", "b", "c"};
        // System.out.println(MathUtils.sumArray(strArray));
    }
}
```

Multiple Bounds

Java also allows **multiple bounds** by using `&` to separate interfaces/classes. For example:

```
<T extends Number & Comparable<T>>
```

means `T` must be a subtype of `Number` **and** implement `Comparable<T>`. This is useful when you need both numeric behavior and the ability to compare objects.

Bounded Type Parameters in Classes

Bounds can also be applied at the class level:

```
public class NumericBox<T extends Number> {
    private T value;

    public NumericBox(T value) {
        this.value = value;
    }

    public double doubleValue() {
        return value.doubleValue();
    }
}
```

Here, `NumericBox` can only be instantiated with subclasses of `Number`, ensuring safe numeric operations.

Reflection on When to Use Bounds

1. **Balance Flexibility and Safety** Bounded types let you accept a wide range of compatible types (like all numeric types) while preventing invalid types (like `String`). This preserves the flexibility generics provide without sacrificing compile-time type safety.
2. **Enable Use of Specific Methods** Without bounds, you cannot safely call methods specific to certain classes. For example, without bounding `T extends Number`, calling `doubleValue()` on `T` would cause a compile error because not all types have this method.
3. **Enhance Code Readability and Intent** Bounds make your code's intent clearer. When you declare `<T extends Number>`, other developers immediately understand the method or class is meant for numeric types.
4. **Avoid Casting and Runtime Errors** By restricting types, you reduce the need for unsafe casting and prevent runtime errors that would occur if a method assumed numeric behavior on non-numeric types.

Summary

- **Bounded type parameters** restrict generic types using the `extends` keyword, e.g., `<T extends Number>`.
- This ensures only compatible types (subclasses of `Number`) can be used, enabling type-safe numeric operations.
- You can use bounded type parameters in methods and classes.
- Multiple bounds are supported to further refine type constraints.
- Bounds improve **type safety**, **clarity**, and **flexibility** by balancing the generic method/class's usability and correctness.
- They are essential in situations where you need to guarantee that type parameters possess certain capabilities or methods.

Using bounded type parameters effectively helps you harness the power of generics while avoiding pitfalls of overly general or unsafe code. It's a key technique to write reusable, robust, and expressive Java code.

14.4 Wildcards: `?`, `? extends`, `? super`

Generics in Java provide great flexibility and type safety, but sometimes you want to write methods that can work with a range of related types without specifying the exact type parameter. This is where **wildcards** come into play. Wildcards allow you to express uncertainty or flexibility in generic type arguments, enabling powerful and reusable APIs.

What Is a Wildcard?

The wildcard `?` represents an **unknown type** in generics. It means "some type, but I don't know which." For example:

```
List<?> list = new ArrayList<String>();
```

Here, `list` can hold any kind of `List`, but you cannot add elements to it (except `null`) because the exact type is unknown.

Why Use Wildcards?

Wildcards are useful when you want a method to accept generic types of unknown or varying types, especially when the method doesn't need to modify the collection or wants to maintain flexibility.

14.4.1 Wildcard Variants and Their Meanings

There are three main wildcard forms:

1. **Unbounded wildcard:** `?`
2. **Upper bounded wildcard:** `? extends Type`
3. **Lower bounded wildcard:** `? super Type`

Unbounded Wildcard: `?`

This represents any type. It's often used when you only need to **read** data but don't care about the specific type.

Example:

```
public void printList(List<?> list) {
    for (Object elem : list) {
        System.out.println(elem);
    }
}
```

You can call `printList` with a `List<String>`, `List<Integer>`, or any other `List<T>`. But since the type is unknown, you cannot add elements except `null` because it might violate type safety.

Upper Bounded Wildcard: `? extends Type`

This means the unknown type **is a subtype of** `Type` (or `Type` itself). It is useful when you want to **read** from a collection and ensure that everything in it is at least of type `Type`.

Example:

```java
public double sumNumbers(List<? extends Number> list) {
    double sum = 0.0;
    for (Number num : list) {
        sum += num.doubleValue();
    }
    return sum;
}
```

You can pass `List<Integer>`, `List<Double>`, or any `List` of a subclass of `Number`. This guarantees safe reading since every element is at least a `Number`.

Important: You **cannot add** anything to `list` except `null`, because the exact subtype is unknown. For example, if `list` is actually a `List<Double>`, adding an `Integer` would be unsafe.

Lower Bounded Wildcard: `? super Type`

This means the unknown type is a **supertype of** `Type` (or `Type` itself). It is useful when you want to **write** to a collection safely, ensuring you can add elements of a specific type or its subclasses.

Example:

```java
public void addIntegers(List<? super Integer> list) {
    list.add(10);
    list.add(20);
}
```

Here, you can pass `List<Integer>`, `List<Number>`, or `List<Object>`. Because the list is guaranteed to accept `Integer` objects (or their subclasses), it's safe to add `Integer` values.

Note: You cannot safely read specific types out of such a list other than `Object`, because the exact type could be any supertype.

14.4.2 Practical Summary: PECS Rule

To remember when to use `extends` or `super`, the Java community uses the **PECS** acronym:

- **Producer Extends:** If your code only **produces** (reads) values from a generic collection, use `? extends T`.
- **Consumer Super:** If your code only **consumes** (writes to) a generic collection, use `? super T`.

14.4.3 Example: Reading vs Writing

```
public void processNumbers(List<? extends Number> producer) {
    Number n = producer.get(0);  // safe to read as Number
    // producer.add(3); // Compile error! Can't add to ? extends Number
}

public void fillNumbers(List<? super Integer> consumer) {
    consumer.add(5);   // safe to add Integer
    // Integer n = consumer.get(0); // Compile error! get() returns Object
}
```

Full runnable code:

```
import java.util.*;

public class WildcardDemo {

    // Unbounded wildcard: can read but not add
    public static void printList(List<?> list) {
        System.out.println("Printing list:");
        for (Object elem : list) {
            System.out.println("  " + elem);
        }
    }

    // Upper bounded wildcard: reads from a list of Numbers or subclasses
    public static double sumNumbers(List<? extends Number> list) {
        double sum = 0.0;
        for (Number num : list) {
            sum += num.doubleValue();
        }
        return sum;
    }

    // Lower bounded wildcard: writes Integers to a list of Integer or supertypes
    public static void addIntegers(List<? super Integer> list) {
        list.add(10);
        list.add(20);
    }

    public static void main(String[] args) {
        List<String> stringList = Arrays.asList("apple", "banana", "cherry");
        List<Integer> intList = new ArrayList<>(Arrays.asList(1, 2, 3));
        List<Double> doubleList = Arrays.asList(1.1, 2.2, 3.3);
        List<Number> numberList = new ArrayList<>();

        // Unbounded wildcard demo
        printList(stringList);
        printList(intList);

        // Upper bounded wildcard demo
        System.out.println("\nSum of intList: " + sumNumbers(intList));      // Output: 6.0
        System.out.println("Sum of doubleList: " + sumNumbers(doubleList)); // Output: 6.6

        // Lower bounded wildcard demo
```

```
        addIntegers(numberList);  // Safe to add Integers
        printList(numberList);

        // Demonstrating PECS in action
        processNumbers(intList);
        fillNumbers(numberList);
    }

    // Supporting PECS examples
    public static void processNumbers(List<? extends Number> producer) {
        System.out.println("\nFirst number (producer): " + producer.get(0));
        // producer.add(100); // Compile-time error
    }

    public static void fillNumbers(List<? super Integer> consumer) {
        consumer.add(42); // Safe to add Integer
        System.out.println("Added 42 to consumer");
    }
}
```

14.4.4 Why Wildcards Matter: Avoiding Unsafe Operations

Without wildcards, you might write overly restrictive code or use unsafe casts. Wildcards allow you to express flexible contracts safely.

For example, a method that accepts `List<Number>` cannot accept `List<Integer>`, because generics are **invariant** in Java (`List<Integer>` is NOT a subtype of `List<Number>`). Using `List<? extends Number>` relaxes this restriction, allowing covariance.

14.4.5 Reflection on Wildcards

- Wildcards greatly enhance the **flexibility** of APIs that work with generics.
- Using `? extends` supports **covariance** (safe reading).
- Using `? super` supports **contravariance** (safe writing).
- The PECS rule helps avoid confusion and mistakes when designing methods.
- Overusing wildcards can sometimes reduce readability, so use them judiciously.
- Avoid adding to collections declared with `? extends` to maintain type safety.
- Always prefer exact generic types if flexibility is not needed.

14.4.6 Summary Table

Wildcard	Meaning	Use case	Can you add elements?	Can you read elements?
`?` (unbounded)	Unknown type	Read-only or general APIs	No (except null)	Yes, as `Object`
`? extends T`	Subtype of `T` (covariant)	Reading from collection	No	Yes, as type `T` or supertype
`? super T`	Supertype of `T` (contravariant)	Writing to collection	Yes, type `T` or subtype	Yes, only as `Object`

14.4.7 Conclusion

Wildcards are a powerful feature of Java generics that let you write flexible, reusable code while preserving type safety. Understanding how to use `?`, `? extends`, and `? super` effectively—and applying the PECS principle—enables you to design APIs that work cleanly across a variety of types, making your code more robust and maintainable.

Chapter 15.

Lambda Expressions and Method References

1. Lambda Syntax
2. Using Lambdas with Functional Interfaces
3. Method References (`Class::method`)
4. Constructor References

15 Lambda Expressions and Method References

15.1 Lambda Syntax

Lambda expressions are one of the most powerful features introduced in Java 8, enabling developers to write concise, functional-style code. At their core, lambdas provide a simple syntax for defining anonymous functions—functions without a name—that can be passed around as values. This dramatically reduces boilerplate code, especially when working with collections or APIs designed for functional programming.

15.1.1 Basic Syntax of a Lambda Expression

A lambda expression in Java has the general form:

```
(parameters) -> expression
```

- **Parameters:** Zero or more input parameters, enclosed in parentheses. The compiler can often infer the parameter types.
- **Arrow token (->):** Separates the parameter list from the body.
- **Expression or block:** The code executed when the lambda runs. This can be a single expression or a block of statements.

15.1.2 Examples of Lambda Expressions

No Parameters, Single Expression

```
Runnable r = () -> System.out.println("Hello, Lambda!");
r.run();
```

Here, the lambda takes no parameters (`()`), and its body is a single expression: printing a message. It implements the `Runnable` interface, which has a single abstract method `run()` with no parameters.

Single Parameter, Single Expression

```
Consumer<String> printer = message -> System.out.println(message);
printer.accept("Hello, World!");
```

Since there is only one parameter, the parentheses around `message` can be omitted. The lambda simply calls `System.out.println` with the given `message`. The `Consumer<T>` interface represents an operation that accepts a single input argument and returns no result.

Multiple Parameters, Single Expression

```
BinaryOperator<Integer> adder = (a, b) -> a + b;
System.out.println(adder.apply(5, 3));  // Output: 8
```

When there are multiple parameters, parentheses are required around the parameter list. This lambda takes two integers and returns their sum.

Multiple Parameters, Block Body

```
BiPredicate<String, String> startsWith = (str, prefix) -> {
    if (str == null || prefix == null) return false;
    return str.startsWith(prefix);
};

System.out.println(startsWith.test("Lambda", "Lam"));  // Output: true
```

A block body, enclosed in curly braces `{}`, allows multiple statements. In this case, we perform a null check before returning whether `str` starts with `prefix`.

15.1.3 Parameter Types and Type Inference

While parameter types can be explicitly declared, they are often inferred from the context:

```
// Explicit types
(BiFunction<Integer, Integer, Integer>) (Integer a, Integer b) -> a * b;

// Inferred types (more common)
(a, b) -> a * b;
```

In most cases, explicit types are unnecessary and would clutter code.

15.1.4 Returning Values

For single-expression lambdas, the expression's value is implicitly returned. For block bodies, you must use the `return` keyword:

```
// Single-expression (implicit return)
Function<String, Integer> length = s -> s.length();

// Block body (explicit return)
Function<String, Integer> lengthBlock = s -> {
    return s.length();
};
```

15.1.5 Why Use Lambdas?

Before lambdas, implementing behavior often meant creating anonymous classes:

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

This is verbose and harder to read. Lambdas express the same logic concisely.

15.1.6 Enabling Functional-Style Programming

Java collections and streams embrace lambdas to support functional programming:

```
List<String> names = Arrays.asList("Anna", "Bob", "Clara");

// Print all names using lambda
names.forEach(name -> System.out.println(name));

// Filter names starting with 'B'
List<String> filtered = names.stream()
    .filter(name -> name.startsWith("B"))
    .collect(Collectors.toList());

System.out.println(filtered);  // Output: [Bob]
```

Lambdas let you describe *what* to do with data, not *how* to iterate over it.

Full runnable code:

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

public class LambdaExamples {

    public static void main(String[] args) {
        // No Parameters, Single Expression
        Runnable r = () -> System.out.println("Hello, Lambda!");
        r.run();

        // Single Parameter, Single Expression
        Consumer<String> printer = message -> System.out.println(message);
        printer.accept("Hello, World!");

        // Multiple Parameters, Single Expression
        BinaryOperator<Integer> adder = (a, b) -> a + b;
        System.out.println("5 + 3 = " + adder.apply(5, 3));
```

```
        // Multiple Parameters, Block Body
        BiPredicate<String, String> startsWith = (str, prefix) -> {
            if (str == null || prefix == null) return false;
            return str.startsWith(prefix);
        };
        System.out.println("Does 'Lambda' start with 'Lam'? " + startsWith.test("Lambda", "Lam"));

        // Parameter Types and Type Inference
        BiFunction<Integer, Integer, Integer> multiplier = (Integer a, Integer b) -> a * b;
        System.out.println("4 * 6 = " + multiplier.apply(4, 6));

        BiFunction<Integer, Integer, Integer> inferredMultiplier = (a, b) -> a * b;
        System.out.println("7 * 2 = " + inferredMultiplier.apply(7, 2));

        // Returning Values
        Function<String, Integer> length = s -> s.length();
        System.out.println("Length of 'Lambda': " + length.apply("Lambda"));

        Function<String, Integer> lengthBlock = s -> {
            return s.length();
        };
        System.out.println("Length of 'Expression': " + lengthBlock.apply("Expression"));

        // Functional-Style Programming with Lambdas
        List<String> names = Arrays.asList("Anna", "Bob", "Clara");

        System.out.println("\nNames:");
        names.forEach(name -> System.out.println("  " + name));

        List<String> filtered = names.stream()
            .filter(name -> name.startsWith("B"))
            .collect(Collectors.toList());

        System.out.println("Names starting with 'B': " + filtered);
    }
}
```

15.1.7 Reflection on Lambdas

- **Reduce Boilerplate:** Lambdas drastically reduce the need for anonymous inner classes and verbose code, especially in event handling, callbacks, and collection operations.
- **Enhance Readability:** Compact syntax makes intent clearer. Instead of focusing on the class structure, the focus shifts to behavior.
- **Enable Higher-Order Functions:** Methods can accept or return functions, enabling composition, reuse, and functional programming idioms.
- **Encourage Immutability and Statelessness:** Lambdas promote writing stateless functions, which leads to more predictable, thread-safe code.
- **Fit Into Java's Type System:** Lambdas are tightly integrated with **functional**

interfaces—interfaces with a single abstract method—allowing seamless use with existing APIs.

15.1.8 Summary

- Lambda expressions are anonymous functions with a concise syntax: `(parameters) -> expression` or `(parameters) -> { statements }`.
- Parameters and return types are usually inferred by the compiler.
- Lambdas replace verbose anonymous classes and enable functional-style programming.
- They shine when used with Java's functional interfaces like `Runnable`, `Consumer`, `Function`, and more.
- Using lambdas promotes clearer, more maintainable, and more expressive Java code.

15.1.9 Final Code Example

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class LambdaDemo {
    public static void main(String[] args) {
        List<String> fruits = Arrays.asList("Apple", "Banana", "Cherry", "Date");

        // Print all fruits
        fruits.forEach(fruit -> System.out.println(fruit));

        // Filter fruits with length > 5
        List<String> longFruits = fruits.stream()
            .filter(fruit -> fruit.length() > 5)
            .collect(Collectors.toList());

        System.out.println("Fruits with more than 5 letters: " + longFruits);
    }
}
```

Here, lambdas make the code concise, readable, and focused on *what* is done with the data, not *how*.

15.2 Using Lambdas with Functional Interfaces

Lambda expressions in Java are tightly connected with **functional interfaces**—interfaces that declare exactly one abstract method. This single abstract method is the "target" for the

lambda, meaning the lambda provides the implementation for that method.

15.2.1 What Is a Functional Interface?

A **functional interface** is an interface with only one abstract method (aside from any default or static methods). This design enables lambdas to be assigned directly to instances of that interface, dramatically simplifying code.

The Java standard library includes many common functional interfaces, such as:

- **`Runnable`** — no arguments, no return value.
- **`Comparator<T>`** — compares two objects.
- **`Consumer<T>`** — accepts one argument, returns nothing.
- **`Function<T,R>`** — takes one argument, returns a result.

15.2.2 Assigning Lambdas to Functional Interfaces

Let's start with some classic examples:

`Runnable`

```
Runnable task = () -> System.out.println("Task running...");
new Thread(task).start();
```

Here, the lambda `() -> System.out.println("Task running...")` implements the single abstract method `run()` from the `Runnable` interface. This replaces the older verbose anonymous class approach.

`ComparatorString`

```
Comparator<String> lengthComparator = (s1, s2) -> s1.length() - s2.length();

List<String> names = Arrays.asList("Anna", "Bob", "Clara");
Collections.sort(names, lengthComparator);

System.out.println(names);  // Output: [Bob, Anna, Clara]
```

This lambda implements the `compare` method, which accepts two strings and returns their length difference. It enables flexible, inline sorting logic without boilerplate.

15.2.3 Defining a Custom Functional Interface

You can define your own functional interfaces and assign lambdas to them. For clarity, you should annotate your interface with `@FunctionalInterface` (optional but recommended), which instructs the compiler to enforce the single-abstract-method rule.

```
@FunctionalInterface
interface StringProcessor {
    String process(String input);
}
```

Now, a lambda can implement this interface succinctly:

```
StringProcessor toUpperCase = s -> s.toUpperCase();
StringProcessor addExclamation = s -> s + "!";

System.out.println(toUpperCase.process("hello"));       // Output: HELLO
System.out.println(addExclamation.process("hello"));    // Output: hello!
```

This pattern allows you to create flexible, reusable behaviors without writing full class implementations.

15.2.4 Why Must There Be a Single Abstract Method?

The single abstract method ensures that the compiler knows exactly which method the lambda implements. Without this restriction, it would be ambiguous what the lambda corresponds to. This rule is central to lambda compatibility and type inference.

Interfaces with multiple abstract methods cannot be implemented with lambdas but can still be implemented with classes or anonymous classes.

15.2.5 How Lambdas Bring Flexibility to Interface-Based Design

Before lambdas, to provide custom behavior you had to write concrete classes or anonymous inner classes implementing interfaces:

```
Comparator<String> comp = new Comparator<String>() {
    @Override
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
};
```

Lambdas reduce this to a single expression, improving readability and maintainability.

Because lambdas work seamlessly with functional interfaces, Java's API design often exposes behavior points as functional interfaces, empowering developers to pass behavior easily.

15.2.6 Example: Higher-Order Function Using Custom Functional Interface

```
public class Processor {
    public static String applyOperation(String input, StringProcessor processor) {
        return processor.process(input);
    }

    public static void main(String[] args) {
        String result = applyOperation("java", s -> s + " is awesome");
        System.out.println(result);  // Output: java is awesome
    }
}
```

Here, the method `applyOperation` takes a string and a `StringProcessor`. Passing a lambda directly simplifies client code and enables flexible behavior injection.

Full runnable code:

```
import java.util.*;

// Custom functional interface
@FunctionalInterface
interface StringProcessor {
    String process(String input);
}

public class LambdaInterfaceDemo {

    // Higher-order function using the custom interface
    public static String applyOperation(String input, StringProcessor processor) {
        return processor.process(input);
    }

    public static void main(String[] args) {

        // Runnable lambda
        Runnable task = () -> System.out.println("Task running...");
        new Thread(task).start();

        // Comparator lambda for sorting strings by length
        Comparator<String> lengthComparator = (s1, s2) -> s1.length() - s2.length();
        List<String> names = Arrays.asList("Anna", "Bob", "Clara");
        Collections.sort(names, lengthComparator);
        System.out.println("Sorted by length: " + names); // [Bob, Anna, Clara]

        // Custom StringProcessor implementations
        StringProcessor toUpperCase = s -> s.toUpperCase();
        StringProcessor addExclamation = s -> s + "!";
        System.out.println(toUpperCase.process("hello"));     // HELLO
```

```
        System.out.println(addExclamation.process("hello"));  // hello!

        // Using a lambda as a higher-order function argument
        String result = applyOperation("java", s -> s + " is awesome");
        System.out.println(result);  // java is awesome
    }
}
```

15.2.7 Functional Interfaces in the Standard Library

The Java API provides a rich set of functional interfaces in the `java.util.function` package, designed for common use cases:

Interface	Description	Abstract Method
`Predicate<T>`	Tests a condition on `T`	`boolean test(T t)`
`Function<T,R>`	Transforms `T` into `R`	`R apply(T t)`
`Consumer<T>`	Accepts `T` but returns nothing	`void accept(T t)`
`Supplier<T>`	Provides a `T` without input	`T get()`
`UnaryOperator<T>`	Takes and returns the same type `T`	`T apply(T t)`
`BinaryOperator<T>`	Takes two `T`s and returns `T`	`T apply(T t1, T t2)`

You can easily assign lambdas to any of these, which makes them invaluable building blocks.

15.2.8 Reflection

- **Functional interfaces** are the bridge that connects Java's traditional object-oriented model with functional programming concepts.
- Lambdas provide a concise way to implement these interfaces without verbosity.
- The requirement of a single abstract method ensures clarity and compiler support.
- Custom functional interfaces allow you to model domain-specific behaviors flexibly.
- Leveraging lambdas with functional interfaces improves expressiveness, reduces boilerplate, and leads to cleaner, more maintainable code.
- The Java standard library's rich set of functional interfaces ensures broad applicability, encouraging functional-style programming patterns.

15.3 Method References (`Class::method`)

Method references are a concise and readable way to refer to existing methods or constructors in Java, introduced alongside lambdas in Java 8. They act as shorthand for certain lambda expressions by directly pointing to methods, allowing cleaner and more expressive code.

15.3.1 Types of Method References

There are **three main types** of method references in Java:

1. **Static Method References:** `ClassName::staticMethod`
2. **Instance Method References of a Particular Object:** `instance::instanceMethod`
3. **Constructor References:** `ClassName::new`

15.3.2 Static Method References

Static method references point to a static method in a class. They are useful when the lambda just calls a static method.

Example: Sorting strings by length using a static helper method.

```
import java.util.Arrays;
import java.util.Comparator;

public class StaticMethodExample {

    // Static method to compare strings by length
    public static int compareByLength(String s1, String s2) {
        return s1.length() - s2.length();
    }

    public static void main(String[] args) {
        String[] names = {"Anna", "Bob", "Clara"};

        // Using lambda
        Arrays.sort(names, (a, b) -> StaticMethodExample.compareByLength(a, b));

        // Using method reference to static method (simpler!)
        Arrays.sort(names, StaticMethodExample::compareByLength);

        System.out.println(Arrays.toString(names));  // Output: [Bob, Anna, Clara]
    }
}
```

Here, `StaticMethodExample::compareByLength` replaces the lambda `(a, b) -> StaticMethodExample.compareByLength(a, b)` without changing behavior.

15.3.3 Instance Method References of a Particular Object

You can reference an instance method of a particular object, often useful when passing existing object behavior.

Example: Using a `PrintStream` object's `println` method.

```
import java.util.Arrays;

public class InstanceMethodExample {
    public static void main(String[] args) {
        String[] messages = {"Hello", "World", "!"};

        // Using lambda to print each message
        Arrays.stream(messages).forEach(s -> System.out.println(s));

        // Using method reference to instance method of System.out
        Arrays.stream(messages).forEach(System.out::println);
    }
}
```

Here, `System.out::println` is a method reference to the instance method `println` on the specific `PrintStream` object `System.out`.

15.3.4 Constructor References

Constructor references refer to a constructor and can be used to instantiate new objects, replacing lambdas that call constructors.

Example: Creating `Person` objects via constructor references.

```
import java.util.function.Function;

class Person {
    String name;

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

    @Override
    public String toString() {
        return "Person: " + name;
    }
}

public class ConstructorReferenceExample {
    public static void main(String[] args) {
        Function<String, Person> personFactory = Person::new; // Constructor reference

        Person p = personFactory.apply("Alice");
        System.out.println(p);  // Output: Person: Alice
```

```
    }
}
```

Here, `Person::new` is equivalent to the lambda `name -> new Person(name)` but is cleaner and more intuitive.

15.3.5 How Method References Simplify Lambda Expressions

Method references improve code clarity by removing redundancy. Instead of writing:

```
x -> someObject.someMethod(x)
```

You write:

```
someObject::someMethod
```

Or instead of:

```
(a, b) -> SomeClass.staticMethod(a, b)
```

You write:

```
SomeClass::staticMethod
```

And for constructors:

```
arg -> new ClassName(arg)
```

becomes:

```
ClassName::new
```

This eliminates boilerplate, making your code easier to read and maintain.

15.3.6 When Should You Prefer Method References?

- **Readability:** If a lambda simply calls a method without additional logic, prefer method references.
- **Conciseness:** Method references reduce verbosity.
- **Expressiveness:** They convey intent directly, showing you're passing an existing method.
- **Code Style Consistency:** Using method references promotes uniform, functional-style code.

15.3.7 When Not to Use Method References

- When the lambda contains additional logic beyond just invoking a method.
- When method references reduce clarity because they separate the call from context.
- When method reference syntax does not easily express complex parameter transformations.

15.3.8 Summary

Method Reference Type	Syntax	Example	Replaces Lambda
Static method	`ClassName::method`	`Math::max`	`(a, b) -> Math.max(a, b)`
Instance method (object)	`instance::method`	`System.out::println`	`x -> System.out.println(x)`
Constructor	`ClassName::new`	`Person::new`	`name -> new Person(name)`

Method references in Java provide a powerful, succinct way to refer to methods and constructors, enabling cleaner and more expressive functional programming patterns. They improve readability and maintainability, especially when working with APIs designed around functional interfaces.

15.4 Constructor References

Constructor references are a special kind of method reference introduced in Java 8 that allow you to refer directly to a class constructor. Instead of writing a lambda expression that explicitly creates a new object, you can use a concise constructor reference to make the code cleaner and easier to read.

15.4.1 Basic Syntax of Constructor References

The syntax for a constructor reference is:

```
ClassName::new
```

This works similarly to other method references but points to a constructor instead of a

method.

15.4.2 Using Constructor References with Functional Interfaces

Constructor references are typically used with functional interfaces from `java.util.function` like:

- `Supplier<T>` — no-argument constructors
- `Function<T,R>` — constructors with one argument
- `BiFunction<T,U,R>` — constructors with two arguments

15.4.3 Example 1: Using `Supplier` with a No-Arg Constructor

Suppose you have a simple class `Car`:

```
class Car {
    public Car() {
        System.out.println("Car created!");
    }
}
```

You can use a `Supplier<Car>` to create new instances of `Car` with a constructor reference:

```
import java.util.function.Supplier;

public class ConstructorReferenceDemo {
    public static void main(String[] args) {
        Supplier<Car> carSupplier = Car::new;  // Constructor reference

        Car myCar = carSupplier.get();  // Calls Car()
    }
}
```

Here, `Car::new` is shorthand for the lambda `() -> new Car()`. When you call `carSupplier.get()`, it creates a new `Car` instance.

Full runnable code:

```
import java.util.function.Supplier;

public class SupplierConstructorDemo {

    // Simple Car class with a no-arg constructor
    static class Car {
        public Car() {
            System.out.println("Car created!");
        }
```

```
    }

    public static void main(String[] args) {
        // Using constructor reference with Supplier
        Supplier<Car> carSupplier = Car::new;

        // Create a new Car using the supplier
        Car myCar = carSupplier.get();  // Output: Car created!
    }
}
```

15.4.4 Example 2: Using `Function` with a Parameterized Constructor

If the constructor takes parameters, you can use `Function` or other functional interfaces with arguments.

```
class Person {
    private String name;

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

    @Override
    public String toString() {
        return "Person: " + name;
    }
}
```

Now, use a `Function<String, Person>` to create new `Person` objects:

```
import java.util.function.Function;

public class ConstructorReferenceDemo {
    public static void main(String[] args) {
        Function<String, Person> personFactory = Person::new;

        Person alice = personFactory.apply("Alice");
        Person bob = personFactory.apply("Bob");

        System.out.println(alice);  // Output: Person: Alice
        System.out.println(bob);    // Output: Person: Bob
    }
}
```

The `Person::new` constructor reference replaces the lambda `name -> new Person(name)`.

Full runnable code:

```
import java.util.function.Function;
```

```java
public class ConstructorReferenceDemo {

    // Simple Person class
    static class Person {
        private String name;

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

        @Override
        public String toString() {
            return "Person: " + name;
        }
    }

    public static void main(String[] args) {
        // Using constructor reference with Function
        Function<String, Person> personFactory = Person::new;

        Person alice = personFactory.apply("Alice");
        Person bob = personFactory.apply("Bob");

        System.out.println(alice);  // Output: Person: Alice
        System.out.println(bob);    // Output: Person: Bob
    }
}
```

15.4.5 Example 3: Using `BiFunction` with Two Parameters

For constructors with two parameters, `BiFunction` fits naturally.

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

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public String toString() {
        return "(" + x + ", " + y + ")";
    }
}
```

Using a constructor reference with `BiFunction`:

```java
import java.util.function.BiFunction;

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

```
        BiFunction<Integer, Integer, Point> pointFactory = Point::new;

        Point p1 = pointFactory.apply(3, 4);
        Point p2 = pointFactory.apply(7, 9);

        System.out.println(p1);  // Output: (3, 4)
        System.out.println(p2);  // Output: (7, 9)
    }
}
```

Full runnable code:

```
import java.util.function.BiFunction;

public class ConstructorReferenceDemo {

    // Point class with a two-argument constructor
    static class Point {
        private int x, y;

        public Point(int x, int y) {
            this.x = x;
            this.y = y;
        }

        @Override
        public String toString() {
            return "(" + x + ", " + y + ")";
        }
    }

    public static void main(String[] args) {
        BiFunction<Integer, Integer, Point> pointFactory = Point::new;

        Point p1 = pointFactory.apply(3, 4);
        Point p2 = pointFactory.apply(7, 9);

        System.out.println(p1);  // Output: (3, 4)
        System.out.println(p2);  // Output: (7, 9)
    }
}
```

15.4.6 Why Use Constructor References?

- **Less Boilerplate:** Constructor references eliminate the need to write explicit lambda expressions when all you're doing is creating a new object.
- **Improved Readability:** The syntax directly states your intention: "Use this constructor."
- **Clean Factories and Callbacks:** Constructor references are ideal for factories, streams, and callback code where new instances need to be created on demand without

clutter.

15.4.7 Real-World Use Case: Creating Objects from Streams

Imagine processing a list of strings and converting them to `Person` objects:

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class ConstructorReferenceDemo {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Anna", "Brian", "Catherine");

        List<Person> people = names.stream()
                                   .map(Person::new)  // Constructor reference
                                   .collect(Collectors.toList());

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

Without constructor references, the `map` would look like `.map(name -> new Person(name))`. Using `Person::new` simplifies the code and emphasizes clarity.

15.4.8 Reflection: Constructor References in Modern Java

Constructor references bring flexibility to interface-driven designs by reducing verbosity and focusing on what matters—the creation of new objects. They fit naturally into functional programming paradigms, such as streams and callbacks, where object instantiation needs to be concise and expressive.

By combining functional interfaces with constructor references, Java lets you build elegant factory-like mechanisms without cluttering your code with anonymous classes or explicit lambdas. This makes your programs easier to maintain and understand.

15.4.9 Summary

Functional Interface	Constructor Reference Example	Lambda Equivalent
`Supplier<T>`	`Car::new`	`() -> new Car()`
`Function<T, R>`	`Person::new`	`name -> new Person(name)`

Functional Interface	Constructor Reference Example	Lambda Equivalent
`BiFunction<T,U,R>`	`Point::new`	`(x, y) -> new Point(x, y)`

Constructor references provide a powerful and succinct way to instantiate objects, boosting code clarity and promoting a functional style in Java applications.

Next, we will explore how method references and constructor references can be combined with advanced features like streams and collectors for even more expressive and compact Java code.

Chapter 16.

Annotations Syntax

1. Built-in Annotations (`@Override`, `@Deprecated`, etc.)
2. Defining Custom Annotations
3. Applying Annotations
4. Retention and Target Policies

16 Annotations Syntax

16.1 Built-in Annotations (`@Override`, `@Deprecated`, etc.)

Java annotations are metadata that provide information to the compiler or runtime environment. Built-in annotations serve as flags to communicate intent, affect compilation behavior, or help IDEs provide warnings or suggestions. This section focuses on three of the most widely used built-in annotations: `@Override`, `@Deprecated`, and `@SuppressWarnings`.

16.1.1 `@Override`

The `@Override` annotation indicates that a method is intended to override a method declared in a superclass or implement an interface method.

Example:

```
class Animal {
    void speak() {
        System.out.println("Animal speaks");
    }
}

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

If you accidentally misspell the method name or signature, the compiler will flag an error:

```
class Cat extends Animal {
    @Override
    void speek() { // Error: Method does not override a superclass method
        System.out.println("Cat meows");
    }
}
```

Why use `@Override`?

- **Compiler Safety:** Catches accidental mismatches.
- **Improved Clarity:** Makes the intent of method overriding explicit.
- **Better Maintenance:** Helps developers quickly understand inheritance structure.

16.1.2 `@Deprecated`

The `@Deprecated` annotation marks a method, field, or class as outdated and signals that it should no longer be used.

Example:

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

    int sum(int... numbers) {
        int result = 0;
        for (int num : numbers) {
            result += num;
        }
        return result;
    }
}
```

When `add()` is used, the compiler will issue a warning:

```
Calculator calc = new Calculator();
int result = calc.add(3, 4); // Warning: add(int, int) is deprecated
```

Enhancing with `@Deprecated` Javadoc:

```
/**
 * @deprecated Use {@link #sum(int...)} instead.
 */
@Deprecated
int add(int a, int b) {
    return a + b;
}
```

Why use `@Deprecated`?

- **Backward Compatibility:** Allows existing code to continue compiling.
- **Guidance:** Directs developers toward better alternatives.
- **API Evolution:** Encourages gradual deprecation instead of breaking changes.

16.1.3 `@SuppressWarnings`

The `@SuppressWarnings` annotation tells the compiler to ignore specific warnings for a code block, method, or class. It's useful for avoiding unnecessary or known warnings in special cases.

Common Values:

- `"unchecked"` — to suppress unchecked type operations (e.g., raw generics)
- `"deprecation"` — to suppress warnings about deprecated APIs
- `"rawtypes"` — to suppress warnings about using raw types

Example:

```
@SuppressWarnings("unchecked")
void rawListUsage() {
    List list = new ArrayList(); // Raw type warning suppressed
    list.add("Hello");
}
```

You can apply it to methods, classes, or local variables:

```
@SuppressWarnings({"unchecked", "deprecation"})
void mixedUsage() {
    List list = new ArrayList();
    list.add("Hello");

    Calculator calc = new Calculator();
    int result = calc.add(2, 3); // Deprecated method, warning suppressed
}
```

Why use `@SuppressWarnings`?

- **Cleaner Compilation:** Reduces noise from known issues.
- **Targeted Ignoring:** Encourages controlling rather than disabling all warnings.
- **Documentation:** Signals to others that the warning is understood and intentionally ignored.

16.1.4 Reflection: How Built-in Annotations Improve Java Code

Built-in annotations serve three primary purposes:

1. **Correctness:** `@Override` helps prevent bugs by ensuring methods are overridden as intended.
2. **Maintainability and API Evolution:** `@Deprecated` signals safe migration paths and supports progressive changes without breaking legacy code.
3. **Control over Compiler Feedback:** `@SuppressWarnings` allows developers to quiet compiler warnings selectively, enabling focus on real issues.

These annotations are not merely decorative—they guide the compiler, enhance documentation, and clarify design choices. IDEs like IntelliJ IDEA and Eclipse leverage annotations to provide better code suggestions, autocomplete, and error detection, making development faster and more robust.

Full runnable code:

```java
import java.util.ArrayList;
import java.util.List;

class Animal {
    void speak() {
        System.out.println("Animal speaks");
    }
}

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

class Calculator {
    /**
     * @deprecated Use {@link #sum(int...)} instead.
     */
    @Deprecated
    int add(int a, int b) {
        return a + b;
    }

    int sum(int... numbers) {
        int result = 0;
        for (int n : numbers) {
            result += n;
        }
        return result;
    }
}

public class Main {
    @SuppressWarnings({"unchecked", "deprecation"})
    public static void main(String[] args) {
        // @Override demonstration
        Animal a = new Dog();
        a.speak(); // Dog barks

        // @Deprecated usage
        Calculator calc = new Calculator();
        int result = calc.add(3, 4); // Deprecated method
        System.out.println("Deprecated add() result: " + result);

        // @SuppressWarnings usage
        List list = new ArrayList(); // Raw type
        list.add("Unchecked warning suppressed");

        System.out.println("Raw list first element: " + list.get(0));
    }
}
```

16.1.5 Summary

Annotation	Purpose	Typical Usage
`@Override`	Ensure method correctly overrides parent	Method overriding in subclasses
`@Deprecated`	Warn about outdated or unsafe features	Mark old methods or classes
`@SuppressWarnings`	Suppress known compiler warnings	Raw types, deprecated use, etc.

Using these annotations is not just good practice—it's essential for writing clear, maintainable, and forward-compatible Java code. As you develop more complex systems, understanding and applying annotations effectively will help you build more reliable software.

16.2 Defining Custom Annotations

In Java, annotations are not limited to built-in ones like `@Override` or `@Deprecated`. Developers can define their **own annotations** to describe metadata for classes, methods, fields, parameters, and more. These custom annotations enable **meta-programming**, allowing tools, libraries, and frameworks to behave dynamically based on annotated elements.

16.2.1 Declaring a Custom Annotation

A custom annotation is defined using the `@interface` keyword. The syntax resembles that of an interface, but it does not contain methods in the traditional sense—rather, it defines **elements** that act like configuration parameters.

Basic Syntax:

```
public @interface MyAnnotation {
    String value(); // an element with no default value
}
```

This annotation can be applied like so:

```
@MyAnnotation("Hello")
public class Greeting {
    // ...
}
```

In this case, the `value` element is specified directly. If the annotation has only one element

named `value`, Java allows shorthand usage by omitting the name.

16.2.2 Multiple Elements and Default Values

Annotations can include multiple elements, and elements can have default values.

Example:

```
public @interface Author {
    String name();
    String date();
    int version() default 1; // optional element with default
}
```

Usage:

```
@Author(name = "Jane Doe", date = "2025-06-20")
public class Document {
    // ...
}
```

Here, `version` is omitted because it uses the default value of `1`. You can include it explicitly if needed:

```
@Author(name = "Jane Doe", date = "2025-06-20", version = 2)
```

16.2.3 Supported Element Types

The types allowed in annotation elements are limited:

- Primitive types (`int`, `double`, etc.)
- `String`
- `Class` or `Class<?>`
- Enumerations
- Other annotations
- One-dimensional arrays of the above types

Invalid types like `List<String>` or `Map<String, String>` are not permitted.

16.2.4 Custom Annotation Example: Validation Metadata

Suppose you're building a validation tool and want to annotate fields that must not be null.

```java
public @interface NotNull {
    String message() default "This field cannot be null.";
}
```

You can use this annotation on fields:

```java
public class User {
    @NotNull
    private String name;

    @NotNull(message = "Email is required.")
    private String email;
}
```

Although Java itself won't enforce this constraint, you can write a tool or framework to **read the annotation using reflection** and perform validation.

16.2.5 Reading Custom Annotations with Reflection

You can retrieve annotation metadata at runtime using the `java.lang.reflect` API.

Example:

```java
import java.lang.reflect.Field;

public class Validator {
    public static void validate(Object obj) throws Exception {
        Class<?> clazz = obj.getClass();
        for (Field field : clazz.getDeclaredFields()) {
            if (field.isAnnotationPresent(NotNull.class)) {
                field.setAccessible(true);
                Object value = field.get(obj);
                if (value == null) {
                    NotNull annotation = field.getAnnotation(NotNull.class);
                    throw new Exception(annotation.message());
                }
            }
        }
    }
}
```

Full runnable code:

```java
import java.lang.annotation.*;
import java.lang.reflect.Field;

// Define the custom annotation
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
@interface NotNull {
    String message() default "This field cannot be null.";
```

```
}

// Class using the annotation
class User {
    @NotNull
    private String name;

    @NotNull(message = "Email is required.")
    private String email;

    public User(String name, String email) {
        this.name = name;
        this.email = email;
    }
}

// Validator that uses reflection to check @NotNull fields
class Validator {
    public static void validate(Object obj) throws Exception {
        Class<?> clazz = obj.getClass();
        for (Field field : clazz.getDeclaredFields()) {
            if (field.isAnnotationPresent(NotNull.class)) {
                field.setAccessible(true);
                Object value = field.get(obj);
                if (value == null) {
                    NotNull annotation = field.getAnnotation(NotNull.class);
                    throw new Exception(annotation.message());
                }
            }
        }
    }
}

// Main class to test it all
public class CustomAnnotationDemo {
    public static void main(String[] args) {
        try {
            User user1 = new User("Alice", "alice@example.com");
            Validator.validate(user1);  // Should pass

            User user2 = new User("Bob", null);
            Validator.validate(user2);  // Should throw an exception
        } catch (Exception e) {
            System.out.println("Validation error: " + e.getMessage());
        }
    }
}
```

16.2.6 Why Use Custom Annotations?

Custom annotations are a key enabler of **declarative programming**—where developers express what should happen, not how.

Benefits:

- **Clean and reusable configuration:** Instead of repeating logic across methods or fields, annotations centralize intent.
- **Framework integration:** Libraries like Spring, JUnit, and Hibernate rely heavily on custom annotations to configure behavior.
- **Tool-friendly metadata:** IDEs and compilers can use annotations to provide smarter checks and assistance.
- **Non-intrusive metadata:** Annotations do not affect class inheritance or implementation and remain easy to remove or change.

16.2.7 Real-World Use Cases

- **Dependency Injection:** `@Inject`, `@Autowired`
- **Testing:** `@Test`, `@BeforeEach`
- **Serialization:** `@JsonProperty`, `@XmlElement`
- **Security:** `@RolesAllowed`, `@Secured`
- **Validation:** `@NotNull`, `@Min`, `@Email`

All of these are implemented as custom annotations that are processed at compile-time or runtime.

16.2.8 Summary Table

Feature	Description
`@interface`	Declares a new annotation type
Elements	Configuration values (must return specific supported types)
Default values	Optional elements can have default values
Usage	Apply to class, method, field, etc. using `@YourAnnotation`
Accessing annotations	Use reflection (`Class.getDeclaredFields()`, etc.)

16.2.9 Final Reflection

Custom annotations are a **powerful design tool** in Java. While they don't enforce behavior on their own, they enable you to **express intent declaratively** and build tooling around it. This is especially powerful in large systems or frameworks, where consistent behavior can be encoded through simple annotations rather than boilerplate logic.

By learning how to define and apply your own annotations, you open the door to building cleaner, more expressive APIs and participating in the modern Java ecosystem of annotation-driven development.

16.3 Applying Annotations

In Java, annotations provide metadata that describe how a program should behave, rather than implementing the behavior directly. Applying annotations to various elements—classes, methods, fields, parameters, local variables—makes code more expressive, declarative, and tool-friendly. This section explores how annotations are applied and interpreted in different contexts.

16.3.1 Applying Annotations: Basic Syntax

Annotations are placed directly above the element they apply to, using the `@AnnotationName` syntax.

Example 1: Annotating a Class

```
@Deprecated
public class OldService {
    // ...
}
```

Here, `@Deprecated` marks the entire class as outdated. IDEs and compilers will usually show a warning when this class is used.

16.3.2 Annotating Methods

Annotations are commonly used with methods, especially for overriding, testing, and lifecycle control.

Example 2: Using `@Override` on a method

```
class Animal {
    void makeSound() {
        System.out.println("Generic animal sound");
    }
}
```

```java
class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Woof!");
    }
}
```

The `@Override` annotation instructs the compiler to ensure that `makeSound()` actually overrides a method from the superclass. If it doesn't, a compile-time error occurs.

16.3.3 Annotating Fields

Field annotations are often used in frameworks like Spring, Hibernate, or validation libraries.

Example 3: Applying an annotation to a field

```java
public class User {
    @NotNull
    private String username;

    @Size(min = 8)
    private String password;
}
```

These annotations act as **metadata**. Tools can use them to automatically validate the field values at runtime.

16.3.4 Annotating Parameters

Method parameters can also be annotated, often for validation or documentation purposes.

Example 4: Annotating a parameter

```java
public void registerUser(@NotNull String email) {
    // ...
}
```

Here, a tool or library could detect the `@NotNull` annotation and reject null values automatically.

16.3.5 Annotating Local Variables

Since Java 8, even local variables can be annotated.

Example 5: Local variable annotation

```
public void process() {
    @SuppressWarnings("unused")
    int temp = 42;
}
```

This usage is less common but can be helpful for controlling compiler behavior in specific scopes.

16.3.6 Multiple Annotations on a Single Element

You can apply more than one annotation to a single element by stacking them.

Example 6: Multiple annotations

```
@Deprecated
@SuppressWarnings("unchecked")
public void legacyMethod() {
    // ...
}
```

The order of annotations generally doesn't matter, unless one annotation depends on another (rare). Some annotations, like `@Repeatable` ones, can be applied multiple times in certain ways (covered later).

16.3.7 Annotation Placement: Quick Reference

Target	Example
Class	`@Deprecated class Foo {}`
Method	`@Override void bar() {}`
Field	`@NotNull String name;`
Parameter	`void setName(@NotNull String name)`
Local Variable	`@SuppressWarnings String s = "";`

16.3.8 Annotation with Parameters

Annotations can include values to customize their behavior.

Example 7: Passing values

```
@Test(timeout = 1000)
public void testPerformance() {
    // ...
}
```

Here, the `@Test` annotation from JUnit includes a `timeout` parameter.

Full runnable code:

```
import java.util.ArrayList;
import java.util.List;

// Custom stubs to simulate behavior
@interface NotNull {}
@interface Size {
    int min();
}

public class AnnotationDemo {

    @Deprecated
    static class OldService {
        void doSomething() {
            System.out.println("OldService logic");
        }
    }

    static class Animal {
        void makeSound() {
            System.out.println("Generic animal sound");
        }
    }

    static class Dog extends Animal {
        @Override
        void makeSound() {
            System.out.println("Woof!");
        }
    }

    static class User {
        @NotNull
        String username;

        @Size(min = 8)
        String password;
    }

    public void registerUser(@NotNull String email) {
        System.out.println("Registering: " + email);
```

```
    }

    public void process() {
        @SuppressWarnings("unused")
        int temp = 42;
    }

    @Deprecated
    @SuppressWarnings("unchecked")
    public void legacyMethod() {
        List list = new ArrayList(); // raw type
        list.add("legacy");
        System.out.println("Legacy method ran: " + list.get(0));
    }

    public static void main(String[] args) {
        OldService old = new OldService(); // Will show deprecation warning
        old.doSomething();

        Animal a = new Dog();
        a.makeSound();

        AnnotationDemo demo = new AnnotationDemo();
        demo.registerUser("user@example.com");
        demo.legacyMethod();
    }
}
```

16.3.9 Annotations and Declarative Programming

Annotations shift the focus from *how* to do something to *what* should be done. Instead of manually wiring services or validating input, annotations allow developers to declare their intent.

Benefits:

- **Clarity:** Annotations express behavior concisely (e.g., `@Transactional`, `@Test`).
- **Consistency:** Tools can enforce uniform rules across the codebase.
- **Modularity:** Annotations decouple logic from implementation, making code easier to refactor.
- **Extensibility:** Frameworks can evolve by introducing new annotations without changing existing logic.

16.3.10 Final Reflection

Applying annotations correctly is a foundational skill in modern Java development. From basic compiler hints (`@Override`) to complex framework configurations (`@Autowired`, `@Entity`),

annotations make your code more **readable**, **robust**, and **framework-friendly**.

When used thoughtfully, annotations cleanly encode metadata that improves program structure and behavior—without cluttering logic with extra code. This is especially powerful in large projects, where declarative style can lead to cleaner APIs and reduced boilerplate.

16.4 Retention and Target Policies

Annotations in Java are powerful tools that enhance code with metadata. However, for them to be truly useful, Java provides two important meta-annotations from the `java.lang.annotation` package: `@Retention` and `@Target`. These meta-annotations define how annotations behave—specifically, **where** they can be applied and **how long** they are retained in the compiled code.

16.4.1 `@Retention`: Controlling Annotation Lifetime

The `@Retention` annotation specifies how long an annotation is retained in the program. It takes a value from the `RetentionPolicy` enum:

16.4.2 RetentionPolicy Options:

1. **SOURCE**
    - The annotation exists only in the source code.
    - It is discarded by the compiler and not included in the `.class` file.
    - Common use case: documentation tools like `@SuppressWarnings`.
2. **CLASS**
    - The annotation is stored in the compiled `.class` file but is **not** available at runtime via reflection.
    - This is the default retention policy if none is specified.
3. **RUNTIME**
    - The annotation is retained in the `.class` file **and** available at runtime through reflection.
    - Required for annotations used by frameworks or libraries that analyze classes dynamically (e.g., Spring, JUnit).

16.4.3 Example: Defining a Runtime-Retained Annotation

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.RUNTIME)
public @interface LogExecution {
}
```

This `@LogExecution` annotation will be available during runtime, which is necessary if we want to scan classes with reflection to detect its presence.

16.4.4 `@Target`: Controlling Where Annotations Can Be Applied

The `@Target` meta-annotation limits where your custom annotation can legally appear. It uses constants from the `ElementType` enum.

16.4.5 Common `ElementType` Values:

- `TYPE`: class, interface, enum, or annotation declaration
- `FIELD`: fields (including enum constants)
- `METHOD`: methods
- `PARAMETER`: method parameters
- `CONSTRUCTOR`: constructors
- `LOCAL_VARIABLE`: local variables
- `ANNOTATION_TYPE`: another annotation type
- `PACKAGE`: package declaration
- `TYPE_USE`: anywhere a type is used (advanced, e.g., `List<@NotNull String>`)

16.4.6 Example: Annotation for Methods Only

```
import java.lang.annotation.Target;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Audit {
}
```

In this example, `@Audit` can **only** be applied to methods. Trying to use it on a class or a field would cause a compilation error.

16.4.7 Combining `@Retention` and `@Target`

Typically, you'll specify both `@Retention` and `@Target` when defining a custom annotation, especially if you expect it to be used by frameworks or tooling.

16.4.8 Example: Logging Annotation for Methods

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Log {
    String value() default "INFO";
}
```

This annotation can now be used like this:

```
public class UserService {

    @Log("DEBUG")
    public void createUser() {
        // Business logic
    }
}
```

And later, a framework can reflectively inspect this annotation:

```
Method m = UserService.class.getMethod("createUser");
if (m.isAnnotationPresent(Log.class)) {
    Log log = m.getAnnotation(Log.class);
    System.out.println("Log level: " + log.value());
}
```

16.4.9 Multiple Targets: Allowing More Flexibility

You can allow annotations on multiple types by passing an array to `@Target`.

```
@Target({ElementType.METHOD, ElementType.CONSTRUCTOR})
public @interface Timed {
```

```
}
```

This allows `@Timed` to be applied to **both** methods and constructors.

Full runnable code:

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

// Define the annotation
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface Log {
    String value() default "INFO";
}

// Use the annotation
class UserService {

    @Log("DEBUG")
    public void createUser() {
        System.out.println("User created.");
    }

    @Log
    public void deleteUser() {
        System.out.println("User deleted.");
    }

    public void updateUser() {
        System.out.println("User updated.");
    }
}

// Main class to inspect and use the annotation
public class AnnotationDemo {
    public static void main(String[] args) throws Exception {
        UserService service = new UserService();
        Method[] methods = UserService.class.getDeclaredMethods();

        for (Method m : methods) {
            if (m.isAnnotationPresent(Log.class)) {
                Log log = m.getAnnotation(Log.class);
                System.out.println("Method: " + m.getName() + ", Log Level: " + log.value());
                m.invoke(service); // Call the method
            }
        }
    }
}
```

16.4.10 Reflection and `RetentionPolicy.RUNTIME`

If you plan to use reflection to detect annotations at runtime—like in dependency injection frameworks, testing tools, or ORM systems—you **must** declare the annotation with `RetentionPolicy.RUNTIME`. Otherwise, the JVM will not retain the annotation, and your reflection code won't find it.

16.4.11 Final Reflection

The `@Retention` and `@Target` meta-annotations are not just technical details—they are **essential** to the proper use and design of custom annotations. They give you:

- **Control over when annotations are visible** (`@Retention`)
- **Control over where annotations can be applied** (`@Target`)

Used wisely, these meta-annotations enforce good design patterns and ensure that your annotations are applied in meaningful, predictable ways. Whether you're building a lightweight library, a testing tool, or a full-fledged framework, these policies help manage annotation behavior efficiently—keeping your metadata robust, safe, and appropriately scoped.

Chapter 17.

Arrays Syntax

1. Declaring and Initializing Arrays
2. Multidimensional Arrays
3. Accessing Elements and Array Length
4. Enhanced For Loop with Arrays

17 Arrays Syntax

17.1 Declaring and Initializing Arrays

In Java, **arrays** are objects that store multiple values of the same type. Arrays are indexed collections with a fixed length determined at the time of creation. They are commonly used to store sequences of numbers, strings, objects, or any other data type in contiguous memory.

17.1.1 Declaring Arrays

There are two standard ways to declare an array in Java:

```
int[] numbers;     // Preferred style
int numbers[];     // Also valid, common in C/C++-influenced code
```

This declaration doesn't allocate memory—it only defines a reference to an array.

17.1.2 Initializing Arrays

With a Fixed Length

After declaring an array, you can initialize it with a specified size:

```
int[] numbers = new int[5];  // creates an array of 5 integers
```

This creates an array with indexes `0` to `4`. Each element is initialized to a **default value** depending on the array's type:

Type	Default Value
`int`, `short`, `byte`, `long`	`0`
`float`, `double`	`0.0`
`char`	`'\u0000'` (null character)
`boolean`	`false`
Reference types (`String`, custom classes, etc.)	`null`

Example:

```
String[] names = new String[3];  // names[0], names[1], names[2] all default to null
```

With Specified Values (Array Literals)

You can also initialize an array using **array literals**:

```
int[] primes = {2, 3, 5, 7, 11};
String[] fruits = {"Apple", "Banana", "Cherry"};
```

This automatically sets the size and initializes elements in one line. The compiler counts the number of items and creates the array accordingly.

17.1.3 Separate Declaration and Initialization

It is legal to declare and later initialize:

```
double[] weights;
weights = new double[] {65.5, 70.2, 55.0};
```

Note: If you're using `new type[]`, you must use the `new` keyword explicitly when separating declaration and initialization.

17.1.4 Array Initialization Patterns

Java arrays can be initialized using **loops**—useful when values follow a pattern:

```
int[] squares = new int[10];
for (int i = 0; i < squares.length; i++) {
    squares[i] = i * i;
}
```

This sets each index of the `squares` array to the square of its index (`0, 1, 4, 9, ..., 81`).

Full runnable code:

```
public class ArrayDemo {
    public static void main(String[] args) {
        // Declaration
        int[] numbers1;         // Preferred style
        int numbers2[];         // Also valid

        // Initialization with fixed length
        numbers1 = new int[5];  // Default values: 0

        // Assigning values manually
        for (int i = 0; i < numbers1.length; i++) {
            numbers1[i] = i + 1;
        }
```

```
        // Literal initialization
        int[] primes = {2, 3, 5, 7, 11};

        // Separate declaration and initialization
        double[] weights;
        weights = new double[] {65.5, 70.2, 55.0};

        // Reference type array
        String[] fruits = new String[3];
        fruits[0] = "Apple";
        fruits[1] = "Banana";
        fruits[2] = "Cherry";

        // Initialization with loop (squares of index)
        int[] squares = new int[10];
        for (int i = 0; i < squares.length; i++) {
            squares[i] = i * i;
        }

        // Output results
        System.out.println("Numbers1: ");
        for (int n : numbers1) System.out.print(n + " ");
        System.out.println();

        System.out.println("Primes: ");
        for (int p : primes) System.out.print(p + " ");
        System.out.println();

        System.out.println("Weights: ");
        for (double w : weights) System.out.print(w + " ");
        System.out.println();

        System.out.println("Fruits: ");
        for (String f : fruits) System.out.print(f + " ");
        System.out.println();

        System.out.println("Squares: ");
        for (int s : squares) System.out.print(s + " ");
        System.out.println();
    }
}
```

17.1.5 Reflection: Array Limitations and Use Considerations

While arrays in Java are useful for storing fixed-size sequences, they come with **limitations**:

- **Fixed Size**: Once created, an array's length cannot change. To work with dynamic collections, prefer `ArrayList` or other classes from the `java.util` package.
- **Type Uniformity**: Arrays store elements of a single type only.
- **Zero-Based Indexing**: The first element is at index `0`, which can lead to off-by-one errors if not carefully managed.

Despite their limitations, arrays are efficient and straightforward, making them a good choice

for fixed-size data and performance-sensitive applications.

17.2 Multidimensional Arrays

In Java, **multidimensional arrays** are arrays of arrays. The most common use is the **two-dimensional array** (2D array), often visualized as a table or matrix with rows and columns. Java also supports **jagged arrays**, where each row can have a different length, providing greater flexibility.

17.2.1 Declaring and Initializing 2D Arrays

Rectangular 2D Arrays

A **rectangular** 2D array is one where each row has the same number of columns.

Declaration and initialization:

```
int[][] matrix = new int[3][4]; // 3 rows, 4 columns
```

This creates a grid like:

```
[ [0, 0, 0, 0],
  [0, 0, 0, 0],
  [0, 0, 0, 0] ]
```

Each `int` element is initialized to `0` by default.

Alternative declaration with values:

```
int[][] matrix = {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 0, 1, 2}
};
```

Here, the size is inferred from the initializer: `3` rows and `4` columns.

Jagged Arrays (Arrays of Arrays)

A **jagged array** allows each row to have a different length:

```
int[][] triangle = new int[3][];
triangle[0] = new int[1];    // Row 0 has 1 column
triangle[1] = new int[2];    // Row 1 has 2 columns
triangle[2] = new int[3];    // Row 2 has 3 columns
```

This creates a triangular structure often used in algorithms like Pascal's Triangle or in graphics buffers.

With values:

```
int[][] triangle = {
    {1},
    {2, 3},
    {4, 5, 6}
};
```

17.2.2 Accessing and Assigning Elements

To work with individual elements of a 2D array, use two indices:

```
int value = matrix[1][2];     // Access value in 2nd row, 3rd column
matrix[0][0] = 99;            // Assign value to 1st row, 1st column
```

Note that indexing starts at `0`, so `matrix[1][2]` accesses the second row and third column.

17.2.3 Looping Through 2D Arrays:

You can use nested loops to iterate over a 2D array:

```
for (int i = 0; i < matrix.length; i++) {           // Rows
    for (int j = 0; j < matrix[i].length; j++) {    // Columns
        System.out.print(matrix[i][j] + " ");
    }
    System.out.println();
}
```

Using `matrix.length` gives the number of rows, and `matrix[i].length` gives the number of columns in row `i`—which is especially important for jagged arrays.

Full runnable code:

```
public class TwoDArrayDemo {
    public static void main(String[] args) {
        // Rectangular 2D array declaration and initialization
        int[][] matrix = {
            {1, 2, 3, 4},
            {5, 6, 7, 8},
            {9, 0, 1, 2}
        };

        // Access and modify elements
        matrix[0][0] = 99;
```

```
        int value = matrix[1][2];

        System.out.println("Rectangular matrix:");
        for (int i = 0; i < matrix.length; i++) {
            for (int j = 0; j < matrix[i].length; j++) {
                System.out.print(matrix[i][j] + " ");
            }
            System.out.println();
        }

        System.out.println("\nValue at matrix[1][2]: " + value);

        // Jagged array declaration
        int[][] triangle = new int[3][];
        triangle[0] = new int[] {1};
        triangle[1] = new int[] {2, 3};
        triangle[2] = new int[] {4, 5, 6};

        System.out.println("\nJagged triangle array:");
        for (int i = 0; i < triangle.length; i++) {
            for (int j = 0; j < triangle[i].length; j++) {
                System.out.print(triangle[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

17.2.4 Reflection: Use Cases and Considerations

Multidimensional arrays are well-suited to **tabular data**, such as:

- Grids or game boards (e.g., chess, Sudoku)
- Matrices for numerical computations
- Image processing (e.g., pixels stored as `[row][column]`)
- Graph algorithms (adjacency matrices)

Readability can become an issue with deep nesting, especially in jagged arrays. It's important to use clear variable names and formatting to make code easier to understand.

17.2.5 Performance Considerations:

- **2D arrays** in Java are implemented as arrays of arrays, so memory is not contiguous like in some languages (e.g., C). This can have a slight performance impact on large arrays.
- **Jagged arrays** save memory when rows differ in size but require careful index checking to avoid `NullPointerException`.

Multidimensional arrays are powerful tools in Java for modeling structured data. Choosing between rectangular and jagged arrays depends on the problem domain and how uniform the data is. In the next section, we'll look at how to access array elements and determine their length dynamically.

17.3 Accessing Elements and Array Length

Once an array is declared and initialized, you can **access** and **modify** its elements using index-based notation. Arrays in Java use **zero-based indexing**, meaning the first element is at index `0`, the second at index `1`, and so on.

17.3.1 Accessing and Modifying Array Elements

To **read** an element from an array, use square brackets:

```
int[] numbers = {10, 20, 30, 40};
int first = numbers[0];  // Gets 10
int last = numbers[3];   // Gets 40
```

To **write** or update an element:

```
numbers[1] = 25;         // Now numbers = {10, 25, 30, 40}
```

Accessing an index outside the valid range will throw an exception:

```
numbers[4] = 50;         // Throws ArrayIndexOutOfBoundsException
```

Always ensure that the index is **greater than or equal to 0** and **less than `array.length`**.

17.3.2 Determining Array Length

Java arrays have a **public `length` field**, which holds the number of elements in the array.

```
int size = numbers.length;  // 4
```

> Note: This is **not a method**—there are **no parentheses**.

To avoid `ArrayIndexOutOfBoundsException`, use `.length` when iterating:

```java
for (int i = 0; i < numbers.length; i++) {
    System.out.println(numbers[i]);
}
```

17.3.3 Safe Iteration Patterns

Using `.length` in loops ensures that your code stays **flexible** and **resilient to size changes**:

```java
String[] fruits = {"Apple", "Banana", "Cherry"};
for (int i = 0; i < fruits.length; i++) {
    System.out.println("Fruit " + i + ": " + fruits[i]);
}
```

This pattern avoids hardcoding array sizes and improves maintainability.

17.3.4 `.length` vs. `.length()` vs. `.size()`

It's important to distinguish between how different types expose their size:

Type	Syntax	Notes
Array	`array.length`	Public field, no parentheses
String	`str.length()`	Method call
Collection/List	`list.size()`	Method call

Example:

```java
String text = "Hello";
System.out.println(text.length());   // 5

int[] values = {1, 2, 3};
System.out.println(values.length);  // 3
```

Confusing these can lead to compilation errors.

Full runnable code:

```java
public class ArrayAccessDemo {
    public static void main(String[] args) {
        // Accessing array elements
        int[] numbers = {10, 20, 30, 40};
        int first = numbers[0];  // 10
        int last = numbers[3];   // 40

        System.out.println("First element: " + first);
```

```
        System.out.println("Last element: " + last);

        // Modifying an element
        numbers[1] = 25;          // Now numbers = {10, 25, 30, 40}
        System.out.println("Modified second element: " + numbers[1]);

        // Getting array length
        System.out.println("Array length: " + numbers.length);

        // Safe iteration
        System.out.println("All elements:");
        for (int i = 0; i < numbers.length; i++) {
            System.out.println("numbers[" + i + "] = " + numbers[i]);
        }

        // Demonstrating ArrayIndexOutOfBoundsException
        try {
            numbers[4] = 50; // Invalid index
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Caught exception: " + e);
        }

        // Comparing .length, .length(), and .size()
        String text = "Hello";
        System.out.println("String length: " + text.length());

        int[] values = {1, 2, 3};
        System.out.println("Array length: " + values.length);
    }
}
```

17.3.5 Reflection

Accessing array elements and knowing their length is foundational to array manipulation. By using `.length` properly, you can avoid common pitfalls like index-out-of-bounds errors. Understanding the differences between `.length`, `.length()`, and `.size()` helps prevent subtle bugs, especially when transitioning between arrays, strings, and collections.

17.4 Enhanced For Loop with Arrays

Java provides an **enhanced `for` loop** (also known as the **for-each loop**) to simplify iteration over arrays. It offers a concise and readable way to loop through each element, especially when you don't need the index or aren't modifying the array.

17.4.1 Syntax and Example

The enhanced `for` loop follows this structure:

```
for (type variable : array) {
    // Use variable
}
```

Here's a basic example:

```
int[] numbers = {10, 20, 30, 40};

for (int num : numbers) {
    System.out.println("Number: " + num);
}
```

In this case, `num` takes on the value of each element in the `numbers` array, one at a time. You **cannot** use `num` to modify the original array elements—changes to `num` do not affect the array.

17.4.2 Works with Reference Types Too

The enhanced loop also works well with arrays of objects:

```
String[] fruits = {"Apple", "Banana", "Cherry"};

for (String fruit : fruits) {
    System.out.println("Fruit: " + fruit);
}
```

This makes iterating through strings, custom objects, and other types much simpler and less error-prone.

17.4.3 When Not to Use It

While convenient, there are times when the enhanced `for` loop is **not suitable**:

1. **Modifying elements in place**: You cannot change values in the original array directly.
   ```
   for (int num : numbers) {
       num = num * 2;  // This does NOT change the actual array
   }
   ```
2. **Accessing by index**: If you need to know the position of the element or compare elements by index, use a traditional `for` loop:

```java
for (int i = 0; i < numbers.length; i++) {
    numbers[i] = numbers[i] * 2;  // This correctly updates the array
}
```

Full runnable code:

```java
public class EnhancedForLoopDemo {
    public static void main(String[] args) {
        int[] numbers = {10, 20, 30, 40};

        System.out.println("Using enhanced for loop:");
        for (int num : numbers) {
            System.out.println("Number: " + num);
        }

        System.out.println("\nAttempting to modify array (fails silently):");
        for (int num : numbers) {
            num = num * 2; // This does not affect the original array
        }

        // Show that original array is unchanged
        for (int num : numbers) {
            System.out.println("Still original: " + num);
        }

        System.out.println("\nModifying array using traditional for loop:");
        for (int i = 0; i < numbers.length; i++) {
            numbers[i] = numbers[i] * 2;
        }

        for (int num : numbers) {
            System.out.println("Modified: " + num);
        }

        System.out.println("\nEnhanced for loop with reference types:");
        String[] fruits = {"Apple", "Banana", "Cherry"};
        for (String fruit : fruits) {
            System.out.println("Fruit: " + fruit);
        }
    }
}
```

17.4.4 Reflection

The enhanced `for` loop improves **readability and reduces boilerplate** when you simply want to process each element without modifying them or tracking indexes. It shines in scenarios like printing values, summing numbers, or checking conditions.

However, when you need **control over the index**—such as replacing elements, accessing adjacent values, or skipping certain items—a classic index-based loop remains the better choice.

In general, prefer enhanced `for` loops for **cleaner and more declarative code**, and fall back to index-based loops when precision and mutation are required.