OCP: Java SE 21 Developer Study Guide

Christian Barahona

Contents

Introduction

Java, released by Sun Microsystems in 1995, has become one of the most popular programming languages. How does a programming language stay relevant in the fast-paced world of technology for almost 30 years?

Several factors contribute to Java's longevity, including its cross-platform compatibility, large and active developer community, and strong emphasis on backward compatibility.

One factor that stands out is Java's continuous updates and improvements to keep up with the latest technology trends. Oracle, the company that now owns Java, releases a new version of the language every six months, with each release bringing new features, performance improvements, and security enhancements. This ensures Java remains competitive with other programming languages and frameworks, maintaining its popularity for modern application development.

In this evolving programming landscape, it's essential for you to stay current with the latest technology. Earning a Java 21 certification not only validates your expertise in Java but also signals to employers that you are committed to ongoing professional development and staying ahead of the curve.

However, a Java certification is not just about passing an exam. It's about building a strong foundation in Java. By studying for and passing the certification exam, you will gain a deeper understanding of the language and its core principles.

This is my intention with this book. Here you will find clear and concise explanations of the fundamental concepts that you need to grasp in order to pass the Java SE 21 Developer Exam (1Z0-830).

Here are some details about the exam: - It consists of 50 multiple-choice questions. - The time allotted for the exam is 120 minutes. - The passing score is 68%. - It is available online through the Oracle University platform.

You can find more information here: https://education.oracle.com/product/pexam_1Z0-830.

Who Should Read This Book

This book is designed for programmers who are already familiar with Java programming, its core concepts, and perhaps even have some practical experience. It is ideal for:

- **Java developers** looking to upgrade their skills and knowledge to the Java 21 version.
- **Intermediate Java programmers** who are comfortable with earlier versions of Java and want to deepen their understanding of the features and improvements introduced in Java 21.
- **Certification aspirants** aiming to pass the Java 21 certification exam and requiring a comprehensive resource that covers all necessary topics and provides insights into the exam's structure and expectations.

However, this book may not be the best starting point for complete beginners to programming or those with no prior experience in Java. While I will explain everything required to understand the objectives covered by the exam, the book assumes a basic understanding of Java concepts and programming principles. If you're new to Java, I recommend starting with introductory materials before this certification guide.

How This Book Is Organized

The book is divided into 16 chapters and one appendix as follows:

- **Chapter 1. Utilizing Java Object-Oriented Approach - Part 1**. This chapter introduces fundamental concepts of object-oriented programming in Java, including classes, objects, and their lifecycle. It covers key language features such as keywords, comments, packages, access modifiers, fields, methods, constructors, initializers, and nested classes.
- **Chapter 2. Utilizing Java Object-Oriented Approach - Part 2**. This chapter goes deeper into Java's object-oriented features, exploring variable scopes, inheritance, polymorphism, and advanced

concepts like abstract classes, interfaces, and sealed classes. It covers topics such as method overriding, the `this` and `super` keywords, type casting, and the `instanceof` operator.

- **Chapter 3. Working with Records and Enums**. This chapter introduces two specialized Java types: records, which provide a concise way to create immutable data carriers with built-in methods, and enums, which define sets of predefined constants. It explores the features, limitations, and best practices for both, including custom constructors, methods, and fields.
- **Chapter 4. Working with Data**. This chapter provides an overview of Java's data handling capabilities, covering primitive and reference types, wrapper classes, operators, and string manipulation. It explores advanced topics such as autoboxing, operator precedence, bitwise operations, the immutability of strings, the efficiency of `StringBuilder`, text blocks, and mathematical operations using the `Math` class.
- **Chapter 5. Controlling Program Flow**. This chapter explores Java's control flow structures, including conditional statements (`if`, `else if`, `else`), `switch` statements and expressions, and various loop constructs (`while`, `do-while`, `for`, enhanced `for`). It covers advanced topics such as pattern matching in `if` statements, labeled loops, and the use of `break` and `continue` statements to manage program execution flow efficiently.
- **Chapter 6. Arrays, Generics, and Collections**. This chapter covers three fundamental concepts in Java: arrays for fixed-size data storage, generics for type-safe programming with different data types, and the Collections Framework for flexible data management. It explores array manipulation, generic classes and methods, wildcard types, and key collection interfaces and utilities, providing a comprehensive understanding of Java's data structure capabilities.
- **Chapter 7. Error Handling and Exceptions**. This chapter explores Java's exception handling mechanism, covering the hierarchy of exception classes, the difference between checked and unchecked exceptions, and techniques for throwing and catching exceptions.
- **Chapter 8. Functional Interfaces and Lambda Expressions**. This chapter introduces functional programming concepts, focusing on functional interfaces, lambda expressions, and method references. It covers the definition and use of functional interfaces, the syntax and applications of lambda expressions, built-in functional interfaces from the `java.util.function` package, and the various types of method references.
- **Chapter 9. Streams**. This chapter explores the Stream API. It covers the creation and manipulation of streams, including intermediate and terminal operations, primitive streams, short-circuiting, and advanced concepts like reduction and collection, while also introducing the `Optional` class for safer `null` handling.
- **Chapter 10. Concurrency and Multithreading**. This chapter talks about Java's concurrency and multithreading capabilities, covering thread creation, lifecycle, and synchronization mechanisms. It explores advanced topics such as the Concurrency API, thread pools, concurrent collections, parallel streams, and strategies for avoiding common pitfalls like deadlocks and race conditions.
- **Chapter 11. The Date/Time API**. This chapter explores the Date/Time API, focusing on key classes such as `LocalDate`, `LocalTime`, `LocalDateTime`, `Instant`, `Period`, and `Duration`. It covers date and time manipulation, formatting, parsing, and working with time zones, daylight savings, and offsets.
- **Chapter 12. File I/O**. This chapter is about Java's file input/output capabilities, focusing on the NIO.2 API and stream-based operations for both byte and character data. It covers essential file operations, including reading, writing, copying, moving, and deleting files, as well as working with file attributes, directory traversal, and object serialization.
- **Chapter 13. The Java Platform Module System**. This chapter explores the Java Platform Module System (JPMS), covering module creation, dependencies, and encapsulation. It covers module types, service providers, migration strategies, and tools like `jdeps`, `jmod`, and `jlink`.

- **Chapter 14. Localization**. This chapter reviews Java's localization capabilities, covering `Locale` handling, resource bundles, and internationalization of messages, numbers, dates, and times. It covers key classes like `ResourceBundle`, `MessageFormat`, `NumberFormat`, `DateFormat`, and `DateTimeFormatter`.

The following table shows the chapter where each exam objective and sub-objective is covered:

Exam Objectives	Chapter
Handling Date, Time, Text, Numeric and Boolean Values	4
Use primitives and wrapper classes. Evaluate arithmetic and boolean expressions, using the Math API and by applying precedence rules, type conversions, and casting.	4
Manipulate text, including text blocks, using String and StringBuilder classes.	4
Manipulate date, time, duration, period, instant and time-zone objects including daylight saving time using Date-Time API.	11
Controlling Program Flow	5
Create program flow control constructs including if/else, switch statements and expressions, loops, and break and continue statements.	5
Using Object-Oriented Concepts in Java	1, 2, 3, 5
Declare and instantiate Java objects including nested class objects, and explain the object life-cycle including creation, reassigning references, and garbage collection.	1
Create classes and records, and define and use instance and static fields and methods, constructors, and instance and static initializers.	1, 3
Implement overloading, including var-arg methods.	1
Understand variable scopes, apply encapsulation, and create immutable objects. Use local variable type inference.	2
Implement inheritance, including abstract and sealed types as well as record classes. Override methods, including that of the Object class. Implement polymorphism and differentiate between object type and reference type. Perform reference type casting, identify object types using the instanceof operator, and pattern matching with the instanceof operator and the switch construct.	2, 5
Create and use interfaces, identify functional interfaces, and utilize private, static, and default interface methods.	2
Create and use enum types with fields, methods, and constructors.	3
Handling Exceptions	7
Handle exceptions using try/catch/finally, try-with-resources, and multi-catch blocks, including custom exceptions.	7
Working with Arrays and Collections	6
Create arrays, List, Set, Map and Deque collections, and add, remove, update, retrieve and sort their elements.	6
Working with Streams and Lambda expressions	8, 9
Use Java object and primitive Streams, including lambda expressions implementing functional interfaces, to create, filter, transform, process, and sort data.	8, 9
Perform decomposition, concatenation, and reduction, and grouping and partitioning on sequential and parallel streams.	9
Packaging and Deploying Java Code	13
Define modules and expose module content, including that by reflection, and declare module dependencies, define services, providers, and consumers.	13
Compile Java code, create modular and non-modular jars, runtime images, and implement migration to modules using unnamed and automatic modules.	13
Managing Concurrent Code Execution	10
Create both platform and virtual threads. Use both Runnable and Callable objects, manage the thread lifecycle, and use different Executor services and concurrent API to run tasks.	10
Develop thread-safe code, using locking mechanisms and concurrent API.	10
Process Java collections concurrently and utilize parallel streams.	10

Exam Objectives	Chapter
Using Java I/O API	12
Read and write console and file data using I/O streams.	12
Serialize and de-serialize Java objects.	12
Construct, traverse, create, read, and write Path objects and their properties using the java.nio.file API.	12
Implementing Localization	14
Implement localization using locales and resource bundles. Parse and format messages, dates, times, and numbers, including currency and percentage values.	14

At the end of each chapter, you will find a set of practice questions to measure your knowledge of the topics covered in the chapter.

Here are a few strategies to maximize the benefits of these practice questions:

1. **Attempt all questions**: Even if you feel confident about a topic, attempting every question ensures comprehensive coverage of the material.
2. **Review explanations**: For each question, detailed explanations are provided, highlighting why each answer is correct or incorrect. Carefully review these explanations to understand the rationale behind each question, which is important for mastering the material.
3. **Revisit difficult questions**: If you find certain questions challenging, make a note of them and revisit these topics in the chapters. This iterative process of testing and reviewing will solidify your understanding.
4. **Track your progress**: Use the practice questions to gauge your understanding and track your progress over time. This can help identify areas where further review is needed.

As you work through the practice and even the real exam questions, consider these tips to improve your success rate:

- **Read Carefully:** Take the time read each question and all possible answers, paying attention to keywords and qualifiers like "all," "none," "only," "best," and "most" to understand what the question is really asking.
- **Identify the Core Question:** Focus on on the question's true intent. Questions often aim to test specific facets of a concept, pinpointing this can guide your thought process.
- **Rephrase the Question:** If the question is complex or confusing, try rephrasing it in your own words. This can help clarify what is being asked and make it easier to identify the correct answer.
- **Eliminate Clearly Wrong Answers:** Start by eliminating any answer choices that are clearly incorrect. Even if you're unsure about the correct answer, narrowing down the options increases your chances of choosing the right one.
- **Look for Distinct Patterns:** Sometimes, incorrect answer choices have patterns in common (such as syntactical errors or implausible values) that you can identify and eliminate.
- **Use Partial Knowledge:** Even if you're not 100% sure about an answer, use your partial knowledge of the topic to eliminate choices that don't fit what you know.
- **Manage Your Time:** If you find yourself stuck on a question, it's often better to skip it and move on. This prevents you from spending too much time on a single question and running out of time for others.
- **Mark for Review:** If available, use the exam's feature to mark questions for later review. This allows you to revisit challenging questions if time permits.
- **First Instincts:** If you must guess, go with your first instinct unless you find clear evidence to change your answer upon review. Often, your initial choice is influenced by your subconscious knowledge of the subject.
- **Context Clues:** Use any given context or code snippets to guide your answer. The context can often eliminate answers that are correct in general but not suitable for the specific scenario presented.

Integrating these tactics with a comprehensive study plan is essential for a thorough preparation. Now,

let's explore some tips for creating an effective study strategy that goes beyond merely answering practice questions.

Tips for Studying

1. Understand the Exam Objectives

First of all, visit the official Oracle Certification website to get detailed information on the objectives, the structure, and the topics covered for the 1Z0-830 exam.

However, understanding the exam objectives is not just about knowing what topics will be on the exam; it's about comprehensively integrating this knowledge into a study plan, ensuring you're well-prepared for the breadth and depth of questions you'll encounter.

Study guides like this one offer a structured way of learning and often include practice questions, study tips, and detailed explanations of topics. However, there are more resources you can use to prepare for the exam: - **Oracle Documentation**: Oracle's official documentation for Java is another important resource. It provides comprehensive details on the Java language and APIs. Familiarity with Oracle's documentation can also help you in your professional work, beyond passing the exam. - **Official or Recognized Training Courses**: Oracle offers an official training course for the Java programmer certification. Courses taught by Oracle-certified instructors or recognized professionals can offer deep insights into Java programming and the certification objectives. They can also provide answers to complex questions and clarify difficult concepts. - **Forums and Discussion Groups**: Online forums and social media groups dedicated to Java certification are excellent places to ask questions, share study tips, and connect with others programmers interested on the Java certification exams. I can recommend Coderanch.

2. Create a Study Plan

You need to approach your exam preparation strategically. A good plan addresses not only what you need to learn but also how you learn best, ensuring that, when exam day arrives, you're confident in your knowledge and ready to succeed. Here's how to create an effective study plan:

1. **Define Your Study Timeline.** Evaluate how familiar you are with the exam topics. This assessment will help you estimate how much time you'll need to prepare for each section and set a target date for taking the exam. Based on your current knowledge and the exam date, allocate a specific number of weeks or months for preparation. Ensure you include extra time for revision and practice exams.
2. **Break Down Exam Objectives into Study Sessions.** Divide the exam objectives into manageable sections or topics, which could be based on the official breakdown provided by Oracle or chapters in a study guide.
3. **Schedule Regular Study Times.** Establish a daily or weekly routine that dedicates specific times to studying. Consistency is important for long-term retention and staying on track with your study plan. However, remember to incorporate short breaks into your study sessions to prevent burnout and enhance productivity. Techniques like the Pomodoro Technique can be beneficial.
4. **Set Milestones and Review Points.** Set specific goals for what you want to achieve each week or month, such as mastering a particular topic or completing a set number of practice questions. Also, schedule regular review sessions to go over previously studied material. This repetition is vital for memory retention.
5. **Adjust the Plan as Needed.** Regularly assess your progress against the study plan. Be prepared to adjust your schedule if you're moving faster or slower than anticipated. Life events may require modifications to your study plan. The key is to stay flexible and adapt while keeping your goal in sight.

3. Practice Coding by Hand

While programmers heavily rely on Integrated Development Environments (IDEs) for coding, the ability to write code by hand (without the assistance of auto-completion or syntax highlighting) is important, especially in the context of certification exams. Coding by hand compels you to recall syntax and programming constructs from memory, reinforcing your knowledge and understanding of Java fundamentals.

Begin practicing with simple programs that cover basic concepts, such as loops, conditionals, data types, and array manipulations. Gradually increase the complexity of these programs as you become more comfortable. This practice will not only improve your coding skills but also will deepen your understanding of these concepts.

Before starting to code, consider outlining your program in pseudocode. This step helps structure your thoughts and approach to problem-solving, allowing you to focus on the logic of your solution without getting bogged down by syntax. Pseudocode is a valuable skill in both exam scenarios and real-world problem-solving.

After writing your code, review it line by line to check for syntax errors, logical mistakes, and other potential issues. Take the time to understand any errors you encounter and why they occurred. This reflective practice is important for learning and improvement. If possible, have someone else review your handwritten code. A fresh perspective can offer new insights and identify errors that you may have overlooked.

4. Include Practice Exams in Your Plan

In addition to the sample questions provided by this book, practice exams help you become familiar with the exam's format, including the wording of questions and the time constraints. This approach enables you to identify areas of weakness, allowing for more targeted and efficient study on topics needing improvement.

Don't postpone taking practice exams until the last minute. Instead, integrate them early and consistently into your study plan to assess your understanding and monitor your progress. Here are some tips:

- **Timed Sessions:** Simulate exam conditions by taking practice exams within set time limits to improve your time management skills. This practice is important for completing all questions within the given time frame during the actual exam.
- **Study in Blocks:** If tackling a full-length exam is too daunting, consider dividing practice exams into smaller segments focused on specific topics for more concentrated study sessions.
- **Simulate the Exam Environment:** Create an exam-like environment by finding a quiet, distraction-free space where you can concentrate on the practice exam without interruptions.
- **Review Incorrect Answers:** Make it a priority to review and understand the rationale behind each incorrect answer and the logic of the correct ones. This process is key to learning from your mistakes and avoiding them in the future.
- **Take Notes on Mistakes:** Maintain a record of errors and challenging topics in a notebook or digital file. Refer to these notes when revising your study plan, emphasizing these weaker areas.
- **Retake Exams:** Revisiting practice exams can be valuable, particularly after some time has elapsed since your initial attempt. However, avoid over-reliance on rote memorization of questions and answers, as it might lead to a misleading sense of readiness.
- **Diverse Set of Questions:** Engage with a wide array of practice exams to encounter various questions and scenarios. This diversity helps prevent the pitfall of memorization and fosters a genuine comprehension of the underlying concepts.
- **Refine Your Study Plan:** Leverage the insights gained from practice exams to fine-tune your study plan. Dedicate additional time to areas of lower performance and continue practice until you observe consistent score improvements.

5. Stay Healthy and Motivated

Studying for the Java certification exam can be a time-consuming and stressful process. Remember, it's important to take breaks, get enough sleep, exercise regularly, and eat a healthy diet to stay focused and energized.

What you eat significantly affects your brain function and energy levels. Consuming a balanced diet with plenty of fruits, vegetables, lean proteins, and whole grains can give you the steady energy necessary for extended study periods. Try to limit your intake of caffeine and sugar to avoid the inevitable energy crashes they can cause.

Regular exercise enhances blood flow to the brain, helping in memory retention and stress alleviation. Even brief intervals of physical activity, such as walking or stretching, can offer substantial benefits. Strive for at least 30 minutes of moderate exercise on most days.

To avoid burnout, incorporate regular breaks into your study plan. Use this time for enjoyable activities, whether it's reading, listening to music, or socializing with friends and family.

Never underestimate the importance of quality sleep, particularly in the days leading up to the exam. Sleep plays a vital role in memory consolidation and overall cognitive functionality. Try to establish a consistent sleep schedule that allows for 7-9 hours of rest each night.

Finally, keep a positive and confident outlook as you navigate your exam preparation. Trust in your capabilities and remind yourself of the reasons behind your pursuit of Java certification. Whether motivated by professional growth, personal achievement, or specific career aspirations, focusing on your initial motivations can help keep your spirits high and your motivation intact throughout challenging study periods.

All right, let's get stated!

Chapter ONE

Utilizing Java Object-Oriented Approach - Part 1

Chapter Content

- Introduction to Object-Oriented Programming
  - Objects and Classes
  - Higher-Level OOP Principles
- Object Life-Cycle in Java
  - Reference Reassignment
  - Garbage Collection
- Keywords
- Comments
- Organizing Classes into Packages
  - Creating a Package
  - Using Import Statements
  - Special Cases and Best Practices
  - Redundant Imports
  - Access Control
- Access Modifiers
- Declaring Classes
- Static and Instance Members
- Declaring Fields
  - Accessing and Modifying Fields
- Declaring Methods
  - Method Signatures
  - Calling a method
  - Using Access Modifiers with Methods
  - Passing Arguments Among Methods
  - Method Overloading
  - Varargs
  - The `main` Method

- Constructors and Initializers
  - Constructors
  - Instance Initializers
  - Static Initializers
  - Initialization Order
- Extending from `java.lang.Object`
- Nested Classes
  - Static Nested Classes
  - Non-static Nested Classes
  - Local Classes
  - Anonymous Classes
- Classes and Source Files
- Key Points
- Practice Questions

Introduction to Object-Oriented Programming

As the name implies, object-oriented programming (OOP) is a programming paradigm centered around the concept of objects. Rather than structure programs around procedures and functions (like procedural programming), OOP organizes code into objects, which represent real-world entities containing data (attributes) and behaviors (methods). This approach offers several advantages:

- Improved organization and modularity
- Code reuse through inheritance
- Real-world modeling

Java is an OOP language, so its basic building blocks are objects and classes.

Objects and Classes

Objects are distinct instances in code that contain data and behaviors. Classes, on the other hand, are blueprints or templates that define the data and behaviors common to all objects of that class.

To better understand these concepts, think of cookies made from a cookie cutter. The cookie cutter defines the shape and size of the cookies, just as classes define what attributes and methods the object instances will have. Each cookie can be unique, with different chocolate chip placements, just as objects contain distinct data values.

For example, we can define a `Cookie` class that specifies the attributes of cookies, such as flavor, shape, topping, etc. You can also define methods, which are functions that operate on the data. Methods allow objects to perform actions. Our `Cookie` objects could have an `eat()` method:

```
public class Cookie {
    // Attributes
    String flavor;
    int size;

    // Behavior (Method)
    public void eat() {
        System.out.println("That was yummy!");
    }
}
```

- `public class Cookie` defines a new `Cookie` class.
- `public` makes this class accessible from other classes.
- `String flavor;` declares a new String attribute called `flavor`.

- `int size;` declares an integer `size` attribute.
- `public void eat()` defines a public `eat` method that does not return a value (`void`).
- The class and the method bodies are wrapped in `{ }` brackets.
- `System.out.println();` prints text to the standard output (usually the console or terminal window).

And we can instantiate cookie objects from the `Cookie` class:

```
Cookie chocoChip = new Cookie();
chocoChip.flavor = "Chocolate Chip";
chocoChip.size = 2;

Cookie oatmealRaisin = new Cookie();
oatmealRaisin.flavor = "Oatmeal Raisin";
oatmealRaisin.size = 1;
```

- `Cookie chocoChip = new Cookie();` instantiates a new `Cookie` object called `chocoChip`.
- We use the class name `Cookie` and the default constructor `new Cookie()`.
- `chocoChip.flavor = "Chocolate Chip";` sets the flavor attribute of `chocoChip`.
- `chocoChip.size = 2;` sets the size attribute to `2`.
- We repeat the process for `oatmealRaisin`, creating another unique cookie object.

The objects `chocoChip` and `oatmealRaisin` are both cookies with the same methods defined by the `Cookie` class. However, they contain different data values for attributes like flavor and size.

A common misconception is that objects and classes are the same. However, while objects and classes are related, they serve distinct purposes:

- Classes define object structure.
- Objects represent unique instances.

The class acts as the mold, while objects are the cookies produced.

Higher-Level OOP Principles

Once you understand objects and classes, grasping the higher-level principles of OOP, like inheritance, encapsulation, and polymorphism, becomes easier:

- **Inheritance** enables code reuse and the creation of class hierarchies. It's like having a basic cookie recipe that serves as a template for many types of cookies. This basic recipe (the parent class) includes common ingredients and methods (attributes and behaviors) that all cookies share. Specialized recipes (subclasses) for different types of cookies, like chocolate chip or oatmeal raisin, inherit common elements but also introduce unique ingredients or steps.
- **Encapsulation** involves bundling data attributes and behaviors into class definitions. It's like wrapping up your cookie dough and recipe instructions into a neat package. Each type of cookie, like chocolate chip or oatmeal raisin, has its own box containing everything needed to make it: ingredients (data) and steps (methods). This package ensures that all the secrets to baking the perfect cookie are held tightly together, accessible only through a specific opening in the box.
- **Polymorphism** enables customizing inherited parent behaviors in subclasses, like overriding the parent `eat()` method inside `ChocolateChip` to print `"Mmm chocolate chip!"`.

Bringing this full circle, we can model real-world cookie hierarchies through:

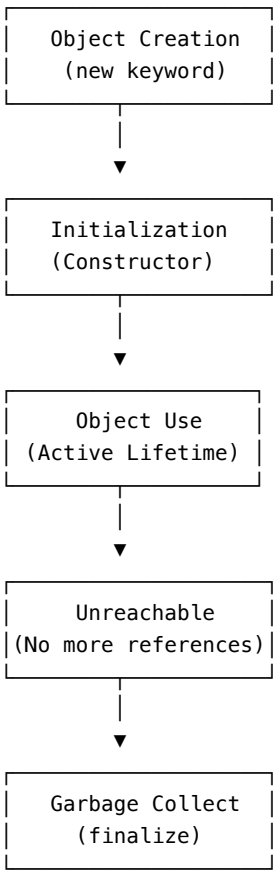
- **Inheritance** – Leveraging parent cookie traits and expanding on them.
- **Encapsulation** - Bundling cookie ingredients and recipes.
- **Polymorphism** - Customizing behaviors like `eat()` per subclass.

Together, these core OOP concepts enable flexible, modular cookie class design. We'll review these concepts in more detail in the next chapter. First, let's talk about the life-cycle of an object.

Object Life-Cycle in Java

Understanding the different stages of an object's life-cycle is essential in Java's object-oriented programming. This includes the creation of objects, how reference variables access them, and how unused objects are managed by Java's garbage collector.

Here's a diagram that illustrates the typical life-cycle of a Java object, from creation to garbage collection:



But to illustrate the life stages of a Java object, let's use the analogy of a library book. When a new book arrives at the library, it is similar to constructing a new object using the `new` keyword. For example:

```
Book javaBook = new Book("The Java Book");
```

Let's break down what happens in that single line step-by-step:

1. **Declaring the Reference Variable:** `java  Book javaBook;` This declares a variable called `javaBook` of type `Book`. At this point, no `Book` object exists yet; we have just created a reference variable that can point to a `Book` object.
2. **Instantiating the Object:** `java  = new Book("The Java Book");` The `new` keyword instantiates or constructs a new `Book` object. This allocates memory on the heap for the object, passes the string argument to the `Book` constructor to initialize its state, and returns a reference to the newly created object.
3. **Assigning the Reference:** The `=` operator assigns the reference of the new `Book` object to the `javaBook` variable.

So, `javaBook` now contains a reference pointing to the new `Book` instance in memory:

```
javaBook --> [New Book object]
```

Here, `javaBook` is the reference variable pointing to the newly created `Book` instance on the Java heap.

Reference Reassignment

Like library books being checked out by different people, object references in Java can be reassigned. For example:

```
Book refBook = javaBook; // Assign second reference
javaBook = null; // Remove original reference
```

Let's review this step by step:

1. **Creating a Second Reference:** `java  Book refBook = javaBook;` This creates a new reference variable `refBook` and assigns it the value of `javaBook`. Both `javaBook` and `refBook` now point to the same `Book` object.

   ```
   javaBook --> [Book object]
   refBook --> [Book object]
   ```

2. **Nullifying the Original Reference:** `java  javaBook = null;` This sets `javaBook` to `null`, meaning it no longer refers to any object.

   ```
   javaBook --> null
   refBook --> [Book object]
   ```

Only `refBook` now points to the `Book` object. The object does not qualify for garbage collection because `refBook` still references it.

Garbage Collection

Books no longer borrowed are eventually removed from a library's catalog. Similarly, in Java, objects with no references are cleaned up by the garbage collector:

```
refBook = null; // Unreferenced object eligible for garbage collection
```

When all references to an object are gone, it becomes eligible for garbage collection.

The garbage collection process can be summarized as follows:

1. **Identifying Unused Objects:** The garbage collector (GC) periodically scans the heap to find objects no longer referenced by any part of the application.
2. **Reclaiming Memory:** Unreferenced objects, which cannot be accessed anymore, are considered *garbage*. The GC frees the memory occupied by these objects, returning it to the pool of available memory on the heap.
3. **Automatic Management:** Garbage collection happens automatically in the background, without explicit program triggering, ensuring that memory management is handled efficiently.

In languages like C, memory must be managed manually by allocating and freeing memory. Java automates this process with garbage collection, increasing programmer productivity and reducing the risk of memory leaks and other related issues.

Now, let's discuss some concepts we'll use to declare a class and other elements.

Keywords

In Java, a keyword is a reserved word that has a predefined meaning in the language. Keywords define the structure and syntax of Java programs. They cannot be used as identifiers (names for variables, methods, classes, etc.) because they are reserved for specific purposes.

Java includes a set of keywords fundamental to the language. Some commonly used keywords include:

- `class`: Used to declare a class.
- `public`, `private`, `protected`: Access modifiers that determine the visibility and accessibility of classes, methods, and variables.
- `static`: Indicates that a member belongs to the class itself rather than instances of the class.
- `void`: Specifies that a method does not return a value.
- `if`, `else`, `switch`, `case`: Used for conditional statements.
- `for`, `while`, `do`: Used for looping and iteration.
- `return`: Used to return a value from a method.
- `new`: Used to create new instances of a class.
- `try`, `catch`, `finally`: Used for exception handling.
- `import`: Used to import classes or packages.

Always keep in mind that each keyword has a specific purpose and is used to define the structure and behavior of Java programs.

Also, it's important to note that keywords are case-sensitive in Java. For example, `class` is a keyword, but `Class` is not. Additionally, you cannot use keywords as identifiers, such as variable or method names, because they are reserved by the language.

Here's an example demonstrating the usage of some keywords:

```
public class MyClass {
    private static int myVariable;

    public static void myMethod() {
        if (myVariable > 0) {
            System.out.println("Positive");
        } else {
            System.out.println("Negative");
        }
    }
}
```

In this example, `public`, `class`, `private`, `static`, `int`, `void`, `if`, and `else` are all keywords used to define the structure and behavior of the `MyClass` class.

We'll review these and other keywords in the upcoming sections and chapters.

Comments

Comments are annotations in the code that are ignored by the compiler. They can be used to: - Describe or explain what the code does. - Document the purpose of specific blocks of code. - Explain the logic behind complex algorithms. - Mark sections of the code.

Java supports three types of comments:

1. Single-line comments
2. Multi-line comments
3. Documentation (javadoc) comments

Single-line comments start with two forward slashes (`//`). Anything following `//` on the same line is ignored by the Java compiler:

```
// This is a single-line comment
int variable = 1; // This is another single-line comment
```

Multi-line comments, also known as block comments, start with `/*` and end with `*/`. Everything between `/*` and `*/` is considered a comment, regardless of how many lines it spans:

```java
/* This is a multi-line comment
   and it can span multiple lines. */
int variable = 1;
```

Documentation comments, or javadoc comments, are designed to document the Java code. They start with `/**` and end with `*/`. These comments can be extracted to a HTML document using the Javadoc tool. Documentation comments are mostly used before definitions of classes, interfaces, methods, and fields:

```java
/**
 * This is a documentation comment.
 * It can be used to describe classes, interfaces, methods, and fields.
 */
public class MyClass {
    /**
     * This method adds up two int values.
     *
     * @param a First value
     * @param b Second value
     * @return The sum of a and b
     */
    public int add(int a, int b) {
        return a + b;
    }
}
```

Organizing Classes into Packages

A package organizes related classes, interfaces, and sub-packages into a single unit.

For example, imagine you own a grocery store that sells many types of products. To keep things organized and easy to find, you decide to group similar products together in different sections or aisles of the store.

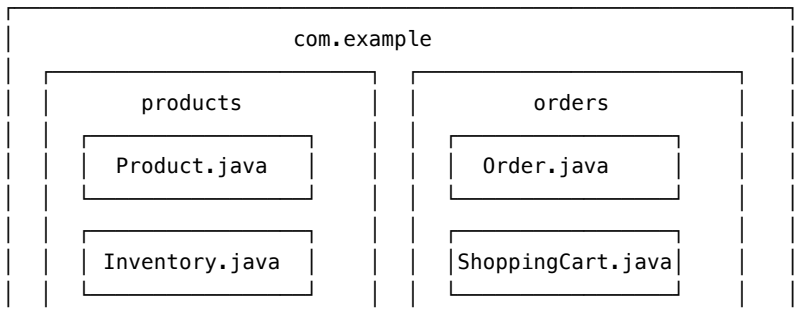
In this analogy: - The grocery store represents your Java project. - The sections or aisles in the store represent packages in Java. - The products on the shelves represent classes and interfaces in Java.

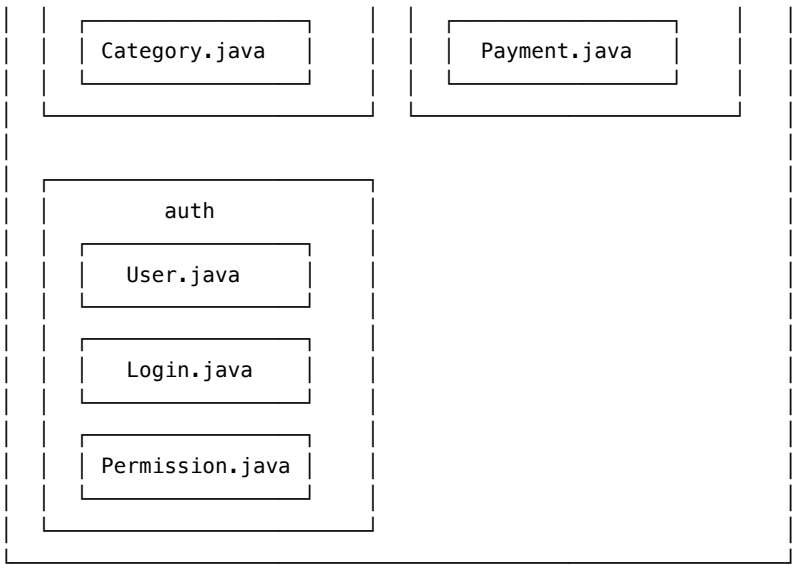
Just like how you group related products together in the same section of the store, you group related classes and interfaces together in the same package in Java.

For example, in your grocery store, you might have: - A *Fruits* section where you place all the different types of fruits like apples, bananas, and oranges. - A *Dairy* section for milk, cheese, yogurt, and other dairy products. - A *Beverages* section for various drinks like water, juice, and soda.

Similarly, in your Java project, you can have: - A `com.example.products` package for classes related to product management, such as `Product`, `Inventory`, and `Category`. - A `com.example.orders` package for classes related to order processing, such as `Order`, `ShoppingCart`, and `Payment`. - A `com.example.auth` package for classes related to user authentication, such as `User`, `Login`, and `Permission`.

Here's a visual representation of these packages and classes:





By organizing your classes into packages, you create a logical structure that makes it easier to locate and manage related code elements, just like how organizing products into sections makes it easier for customers to find what they need in the grocery store.

Creating a Package

To create a package, use the `package` keyword followed by the package name at the top of your Java source file. For example:

```
package com.example.mypackage;
```

The package name should be in lowercase and follow the reverse domain name convention to ensure uniqueness.

The package declaration must be the first statement in the source file, before any import statements or class declarations. The following will not compile:

```
import java.util.ArrayList; // Import statement before the package declaration

package mypackage; // Package declaration not at the beginning

public class MyClass {
    public static void main(String[] args) {
        System.out.println("This will not compile.");
    }
}
```

Using Import Statements

`import` statements are used to bring classes or interfaces from other packages into the current namespace. Instead of using the fully qualified name each time you refer to a class from another package, you can use an `import` statement to refer to the class by its name. For example:

```
import java.util.ArrayList;
// ...
ArrayList list = new ArrayList();
```

If you choose not to use an `import` statement for a class from another package, you would have to use the class's fully qualified name every time you reference it in your code. Remember, the fully qualified name

includes both the package name and the class name.

For example, if you do not import the `ArrayList` class from the `java.util` package, you would have to use `java.util.ArrayList` every time you want to create or use an `ArrayList` object in your code:

```
// No import statement for java.util.ArrayList
// ...
java.util.ArrayList list = new java.util.ArrayList();
```

Special Cases and Best Practices

There are a couple of exceptions or special cases to the rule regarding the use of fully qualified names and import statements:

1. **Classes in the `java.lang` Package**: Classes and interfaces in the `java.lang` package do not need to be imported explicitly, as they are automatically available. For example, you don't need to import classes like `String`, `Math`, `System`, or wrapper classes like `Integer`, `Double`, etc.
2. **Same Package**: Classes and interfaces that are in the same package as the class you're writing do not require an import statement. Java automatically looks in the current package for other classes and interfaces if it doesn't find the referenced class or interface in the imported packages.
3. **Fully Qualified Name Collision**: When two classes have the same name but are in different packages, and you need to use both in the same file, you cannot import both directly because of the name collision. In such cases, at least one (and possibly both) must be referred to by their fully qualified names to avoid ambiguity.

Here's an example to illustrate this last point:

```
import java.sql.Date;

public class Example {
    public static void main(String[] args) {
        Date sqlDate = new Date(System.currentTimeMillis());
        java.util.Date utilDate = new java.util.Date();
    }
}
```

In this example, `Date` from `java.sql` is imported, so it can be referred to by its simple name. However, since we also want to use `Date` from `java.util`, we must refer to it by its fully qualified name to distinguish it from `java.sql.Date`.

You can also use a wildcard ('*') to import all the classes from a package. For example:

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

However, it's generally recommended to import specific classes rather than using wildcards because they can make the code less readable, lead to naming conflicts if multiple packages have classes with the same name, and add redundancy, such as including a class twice.

Redundant Imports

Although the compiler allows redundant imports, they can clutter your code and reduce readability.

For example, assuming we have two classes, `MyClass` and `HelperClass`, in the same package, `mypackage`:

```
// File: HelperClass.java
package mypackage;

public class HelperClass {
    public static void doSomething() {
```

```
        System.out.println("Doing something...");
    }
}
```

The following class illustrates redundant imports:

```
package mypackage;

import mypackage.HelperClass; // Redundant import because HelperClass is in the same package
import java.util.List; // Redundant import because it's not used in the class

public class MyClass {
    public static void main(String[] args) {
        HelperClass.doSomething();
    }
}
```

In this example: - The import statement `import mypackage.HelperClass;` is redundant because `HelperClass` is already in the same package as `MyClass`. Remember, classes in the same package are automatically available to each other without the need for import statements. - The import statement `import java.util.List;` is also redundant because the `List` interface is not used anywhere in `MyClass`.

Removing these redundant imports would make the code cleaner without affecting its functionality.

Access Control

Packages provide a level of access control, similar to how certain sections of the store might be restricted to authorized personnel only. You can use access modifiers (`public`, `protected`, default, `private`) to control the visibility and accessibility of classes and members within and across packages.

For example, let's say you have a package named `com.example.internals` that contains classes and methods intended for internal use only within that package:

```
package com.example.internals;
class InternalClass {
    void internalMethod() {
        // Internal implementation
    }
}
```

Now, consider another package `com.example.api`:

```
package com.example.api;
import com.example.internals.InternalClass;
public class APIClass {
    public void someMethod() {
        InternalClass obj = new InternalClass(); // Not accessible
        obj.internalMethod(); // Not accessible
    }
}
```

In this example, the `InternalClass` and its methods have default (package-private) access. They are accessible within the `com.example.internals` package but not from other packages. The `APIClass` in the `com.example.api` package cannot access the `InternalClass` or its methods directly.

Let's review in more detail the available access modifiers.

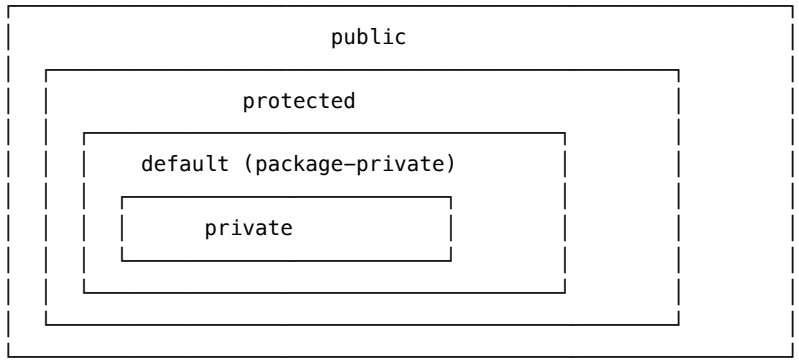
Access Modifiers

Access modifiers are keywords used in classes, methods, or variable declarations to control the visibility of that member from other parts of the program. There are four main types of access modifiers in Java:

1. **`public`**: The `public` access modifier specifies that the member is accessible from any other class in the Java application, regardless of the package it belongs to. Using the `public` modifier means there are no restrictions on accessing the member.
2. **`protected`**: The `protected` access modifier allows the member to be accessed within its own package and also by subclasses of its class in other packages. This is less restrictive than package-private but more restrictive than `public`.
3. **`default`** (also known as **package-private**): If no access modifier is specified, the member has package-private access by default. This means the member is accessible only within its own package and is not visible to classes outside the package. It's important to note that there is no explicit `default` keyword in Java; you simply omit the access modifier.
4. **`private`**: The `private` access modifier specifies that the member is accessible only within the class it is declared in. It is the most restrictive access level and is used to ensure that the member cannot be accessed from outside its own class, not even by subclasses.

Each of these access modifiers serves a specific purpose in the context of object-oriented design and encapsulation. They allow you to structure your code in a way that protects sensitive data and implementation details while exposing necessary functionality to other parts of your application.

Here's a diagram to understand the scope of each access modifier more easily:




```
Access Levels (from most restrictive to least restrictive):
private   : Same class only
default   : Same package
protected : Same package + subclasses in other packages
public    : Accessible from anywhere
```

In the next sections, we'll explain access modifiers in the context of classes, fields, and methods. But first, let's review how to properly declare a class.

Declaring Classes

A class in Java acts as a blueprint for objects, encapsulating both data and behavior.

The syntax to declare a class follows this format:

```
[accessModifier] class ClassName [extends Superclass] [implements Interface1, Interface2, ...] {
    // class body
}
```

For example, a class declaration might look like this:

```
public class MyClass extends MySuperClass implements MyInterface {
    private int myField;

    public MyClass() {
        // Constructor body
    }

    public void myMethod() {
        // Method body
    }
}
```

First, you can optionally specify an access modifier to determine the visibility and accessibility of the class to other parts of a Java application:

- **`public`**: The class is accessible from any other class across different packages.
- **Default (Package-Private)**: If no access modifier is specified, the class is only accessible by other classes within the same package. This is useful for grouping related classes together without exposing them to the entire application.

Following the optional access modifier, you have to use the `class` keyword, followed by the name of the class.

A class name or class identifier should follow the following rules:

1. **Unicode characters**: Java allows the use of Unicode characters in identifiers, which means you can use letters from non-Latin alphabets as well. However, this is not commonly used and can lead to code that is difficult to read and maintain.
2. **Alphabetic characters, digits, underscores (_), and dollar signs ($)**: **These are the most common characters used in identifiers. Any combination of these characters is allowed, but class names** must not** begin with a digit.
3. **No special characters**: Other than underscores and dollar signs, special characters such as `@`, `%`, `!`, `?`, `#`, `&`, `*`, `^`, `~`, `_`, `-`, `+`, `=`, `{`, `}`, `[`, `]`, `|`, `,`, `;`, `<`, `>`, `/`, `\`, or `'`, are not allowed in class identifiers.
4. **Class names should not contain spaces**. This would make the code invalid and lead to compilation errors.
5. **Cannot be a Java reserved word**: Identifiers cannot use any of Java's reserved words (like `int`, `if`, `for`, etc.). Reserved words have specific meanings in Java and cannot be used for class names, variable names, or any other identifiers.
6. **Case Sensitivity**: Java is case-sensitive, meaning identifiers like `MyClass`, `myclass`, and `MYCLASS` will be considered different.
7. **Length**: There is no length limit for class names in Java.

These rules ensure that class name are syntactically correct and avoid conflicts with Java's built-in language features. It's also good practice to follow Java naming conventions on top of these rules, like starting class names with a capital letter and using camel case for multi-word names (like using `MyClass` instead of `myclass` or `MY_CLASS`). But again, this is just a convention, not a rule.

After the class name, you can optionally extend a superclass using the `extends` keyword, followed by the name of the superclass. Java supports single inheritance, meaning a class can only extend one superclass.

However, you can implement one or more interfaces using the `implements` keyword, followed by a comma-separated list of interface names:

```
public class MyClass implements MyInterface1, MyInterface2, MyInterface3 {
    // ...
}
```

Finally, you define the class body within a pair of curly braces `{}`. The class body contains the members of the class, including fields, methods, constructors, and nested classes.

This way, in the next example:

```
public class MyClass extends MySuperClass implements MyInterface {
    /* Class body begins */
    // Fields
    private int myField;

    // Constructor
    public MyClass() {
        // Constructor body
    }

    // Methods
    public void myMethod() {
        // Method body
    }
    /* Class body ends */
}
```

- `public` is the access modifier, indicating that the class is accessible from anywhere.
- `class` is the keyword used to declare a class.
- `MyClass` is the name of the class.
- `extends MySuperClass` specifies that `MyClass` inherits from the `MySuperClass` superclass.
- `implements MyInterface` indicates that `MyClass` implements the `MyInterface` interface.
- The class body contains a `private` field `myField`, a `public` constructor `MyClass()`, and a `public` method `myMethod()`.

Now, before reviewing how to declare fields and methods in more detail, let's talk about static and instance members.

Static and Instance Members

Classes can have two types of members: static members and instance members. Let's use the analogy of a TV model to better understand these types of members.

Imagine different TV sets of the same model in different homes. Each TV set represents an instance (object) of the `Television` class. The TV model itself represents the class.

Instance members, such as instance variables and instance methods, belong to each individual TV set (object): - Each TV set has its own set of instance variables, like its current channel, volume, and whether it's turned on or off. - Instance methods, such as `changeChannel()` or `adjustVolume()`, are actions that each TV set can perform independently. - Instance members are accessed using the instance (object) of the class.

Static members, such as static variables and static methods, belong to the TV model (class) itself: - The TV model has static variables that are shared among all the TV sets, like the manufacturer's logo or model number. - Static methods, such as `getManufacturerInfo()` or `getModelNumber()`, are actions that belong to the TV model and can be accessed without creating an instance of the `Television` class. - Static members are accessed using the class name itself, without the need for creating an instance.

Here's the `Television` class:

```java
public class Television {
    // Instance fields
    private int currentChannel;
    private int volume;
    private boolean isOn;

    // Static field
    private static String manufacturerLogo = "MyBrand";

    // Instance method
    public void changeChannel(int channel) {
        this.currentChannel = channel;
        System.out.println("Channel changed to: " + channel);
    }

    // Static method
    public static void getManufacturerInfo() {
        System.out.println("All TVs by: " + manufacturerLogo);
    }
}
```

In this example: - The `currentChannel`, `volume`, and `isOn` fields are instance variables. Each TV set (object) has its own set of these variables. - The `manufacturerLogo` field is a `static` variable. It belongs to the class itself and is shared among all TV sets. - The `changeChannel()` method is an instance method. Each TV set can invoke this method independently. - The `getManufacturerInfo()` method is a static method. It belongs to the class and can be invoked without creating an instance of the `Television` class.

To access instance members, you need to create an instance of the class:

```java
Television tv1 = new Television();
tv1.changeChannel(5); // Changes channel of tv1
```

But to access static members, you can use the class name directly:

```java
Television.getManufacturerInfo();
```

Static members are useful for representing class-level data and behavior that is shared among all instances of the class. They can be accessed without creating an instance of the class, making them memory-efficient. However, static members cannot access instance members directly, as they are not associated with any specific instance.

It is important to note that Java allows static members (fields and methods) to be accessed through instances of a class. For example, the static method `getManufacturerInfo()` can be used this way too:

```java
tv1.getManufacturerInfo();
```

However, this is not recommended practice, as it does not clearly convey that the member is static and belongs to the class rather than the instance.

Instance members, on the other hand, are associated with each individual instance of the class. They hold data specific to each object and can access both static and instance members.

Now, you might be thinking: Why can static members be accessed without creating an instance of the class? Does this not go against the idea of object-oriented programming??

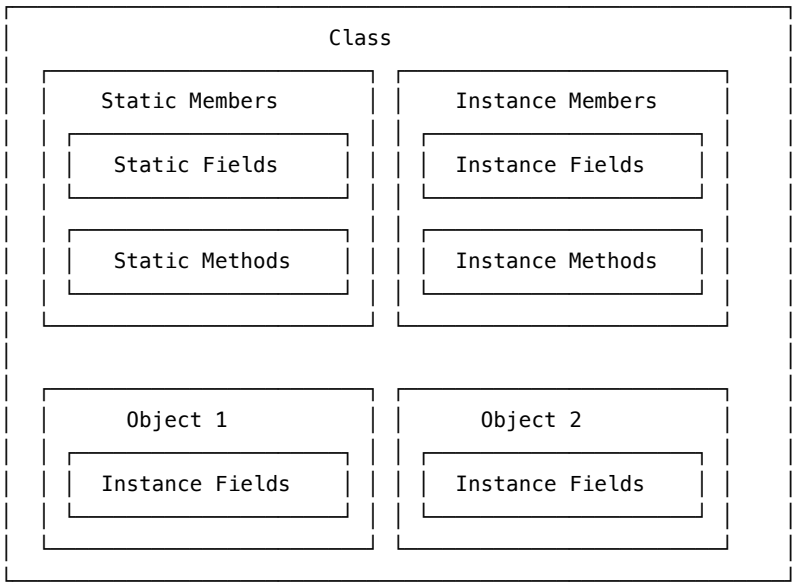
Well, this doesn't necessarily go against the principles of object-oriented programming (OOP), but rather complements them by providing a mechanism for defining class-level behavior and state.

Static methods can be used to implement utility or helper functions that do not depend on the state of an object instance. This is common in utility classes, such as the `Math` class, where all methods are static

because they do not require access to instance-level data.

Also, static members allow for global access. Granted, there's some controversy about this due to the potential for increased coupling and harder-to-test code, however, it can be appropriate for global constants that need to be accessed from various points in an application.

This diagram illustrates several key points about static and instance members in Java:



- The class contains both static and instance members.
- Static members (fields and methods) are associated with the class itself.
- Instance members (fields and methods) are associated with objects of the class.
- Multiple objects of the class each have their own instance members.
- All objects share the same static members.

Now, let's review in more detail how to declare fields.

Declaring Fields

A field is a variable that is declared at the class level. Fields, which are also referred to as attributes or instance variables, are used to hold the state of an object.

To declare a field, you use the following syntax:

```
[accessModifier] [specifiers] type fieldName [= initialValue];
```

Here are some examples:

```java
public class MyClass {
    public static final int MAX_VALUE = 100;
    private String name;
    protected double salary;
    boolean active = true;

    // ...
}
```

The access modifier is optional and can be `public`, `private`, `protected` or default (package-private) access if none is specified. Notice that unlike classes, fields can use all four types of access modifiers. Depending on

the access modifiers used, fields can be accessed from inside the class, subclasses, classes in the same package or any other class. More on this later.

The specifiers part is also optional and can include keywords like `static`, `final`, `transient`, and `volatile`. You can specify zero or more specifiers (like in the first field declaration), but the `final` keyword can only be applied once:

- A field declared as `static` belongs to the class itself, not to a particular instance. There will only be one copy of a `static` field shared by all instances of the class. A common use of `static` fields is to define constants.
- A field declared as `final` cannot be reassigned to refer to a different object or value. If it's a primitive type, the value cannot be modified. If it's a reference type, the reference cannot be changed to point to another object, but the internal state of the object can be altered if it's mutable. Final fields can be used for constants or to make fields read-only after initialization. However, while the field itself becomes read-only, objects referenced by final fields can still have their internal state changed if they are mutable.
- The `transient` and `volatile` keywords are more advanced and relate to serialization and multi-threading. We'll cover them in later chapters.

The type of the field follows the specifiers. It can be a primitive type like `int`, `boolean`, etc. or a reference type like `String`, `LocalDate`, `ArrayList`, etc.

The field name follows standard Java identifier naming rules. Here are the main rules you need to remember for field identifiers:

1. **Unicode Characters**: Java allows Unicode characters in identifiers, which means you can use characters from non-Latin character sets. However, this is not commonly used for field names, as it can make the code harder to read and maintain.
2. **Characters Allowed**: Field identifiers can only include alphanumeric characters (`A–Z`, `a–z`, `0–9`), underscore (_), and dollar sign (`$`). The identifier must begin with a letter (`A–Z` or `a–z`), underscore (_), or dollar sign (`$`). It cannot start with a digit.
3. **No Reserved Words**: Identifiers cannot be Java reserved words. Reserved words include keywords like `int`, `if`, `class`, and so on. These are part of the Java language syntax and have specific meanings to the compiler.
4. **Case Sensitivity**: Java is case-sensitive, meaning identifiers like `myField`, `MyField`, and `MYFIELD` would be considered distinct.
5. **Unlimited Length**: Technically, there is no limit to the length of an identifier, but it's essential to keep it reasonable for readability and maintainability.

It's important to differentiate between rules and conventions. Rules must be followed for the Java code to compile, while conventions, such as starting field names with a lowercase letter or using `camelCase` for multiple words, are best practices designed to make the code more readable and maintainable but are not enforced by the compiler.

Finally, providing an initial value is optional. If none is provided, fields will be initialized with their default values (`0`, `false` or `null` depending on the type). However, the initial value must be a compile-time constant for static final fields.

Once a field is declared, you can access it to read its value or modify it by assigning a new value. The way you access a field depends on whether it's an instance field or a static field and what access modifier it uses.

Accessing and Modifying Fields

To access an instance field, you first need an instance of the class. Then you can read the field's value using the dot (`.`) operator like this:

```
instanceVariable.fieldName
```

For example:

```
String name = person.firstName;
int age = employee.age;
```

To modify an instance field, you use the assignment operator (=) like this:

```
person.firstName = "John";
employee.age = 45;
```

Accessing `static` fields is a bit different. Since they belong to the class itself, you don't need an instance. You can access a static field using the class name and the dot operator:

```
ClassName.fieldName
```

For example:

```
double pi = Math.PI;
int max = Integer.MAX_VALUE;
```

Inside the same class that declares a field, you can access it directly by its name, without any prefix, regardless of the access modifier used. The only exception is accessing a static field, it's recommended to use the class name even within the same class for readability.

The access modifiers `public`, `private`, `protected` and default(package) control the visibility of a field and determine whether it can be accessed directly from outside the class.

Let's look at some examples to illustrate the different access levels.

```
public class Person {
    public String name;
    private int age;
    protected String email;
    double height;
}
```

The `name` field is `public`, so it can be accessed from any other class:

```
Person p = new Person();
p.name = "Alice";
```

The `age` field is `private`. It can only be accessed within the `Person` class. Trying to access it directly from outside the class will result in a compile error:

```
// This will not compile
p.age = 30;
```

The `email` field is `protected`. It can be accessed within the same class, any subclass, and other classes in the same package:

```
// This is okay
String email = p.email;

// This is also valid in a subclass, even in a different package
class Employee extends Person {
    public void setEmail(String e) {
         email = e;
    }
}
```

The `height` field has default (package) access since no modifier is specified. It can be accessed by other classes within the same package:

```java
// This is okay if Person and Student are in same package
class Student {
    public void printHeight(Person p) {
        System.out.println(p.height);
    }
}
```

It's common to declare fields as `private` and access them through getter and setter methods. `public` and `protected` fields are used less frequently. Default (package-private) access is useful for related classes within the same package.

Declaring Methods

A method is a block of code that performs a specific task and optionally returns a value. Methods are used to define the behavior of an object. They provide a way to encapsulate complex logic, break down a program into manageable parts, and enable code reuse.

To declare a method, use the following syntax:

```java
[accessModifier] [specifiers] returnType methodName([parameters]) [throws ExceptionType1, ExceptionType2, ...] {
    // method body
}
```

For example:

```java
public static String addParenthesis(String s) {
    return "(" + s + ")";
}

private int sum(int a, int b) {
    return a + b;
}

protected void setName(String name) throws IllegalArgumentException {
    if (name == null || name.isEmpty()) {
        throw new IllegalArgumentException("Name cannot be null or empty");
    }
    this.name = name;
}
```

The access modifier is optional and controls the visibility of the method. It can be `public`, `private`, `protected`, or default (package) access if none is specified. The same rules apply as for fields, which we discussed earlier.

The specifiers are also optional and can include keywords like `static`, `final`, `abstract`, and `synchronized`. These keywords modify the behavior of the method:

- `static` methods belong to the class itself and can be called without an instance of the class.
- `final` methods cannot be overridden by subclasses.
- `abstract` methods have no implementation in the current class and must be overridden by non-abstract subclasses.
- `synchronized` methods can only be executed by one thread at a time.

The return type specifies the type of value the method returns. It can be a primitive type, a reference type, or void if the method doesn't return anything. Every method declaration must have a return type.

The method name follows the same naming conventions as classes and fields, typically using `camelCase`. Choose meaningful names that describe the purpose of the method.

The parameters are specified within parentheses after the method name. There can be zero or more parameters. Multiple parameters are separated by commas. Parameters are variables that receive the values passed

to the method when it is called. Each parameter consists of two, optionally three, parts:

`[parameterModifier] parameterType parameterName`

The parameter modifier is optional and can be `final`. If a parameter is declared as `final`, it means that the value of the parameter cannot be changed inside the method body. Here's an example:

```
public void printMessage(final String message) {
    // message = "Hello"; // This would cause a compile error
    System.out.println(message);
}
```

The parameter type is required and specifies the data type of the parameter. It can be a primitive type (like `int`, `double`, `boolean`) or a reference type (like `String`, `ArrayList`, or custom classes).

The parameter name is also mandatory and follows the same naming conventions as class, fields and method identifiers, typically using `camelCase`. The parameter name is used to refer to the passed value within the method body.

Here are a few examples of parameter definitions:

```
// A single parameter of type int
public void printNumber(int number) {
    System.out.println("The number is: " + number);
}

// Multiple parameters of different types
public void printPersonDetails(String name, int age, boolean isStudent) {
    System.out.println("Name: " + name);
    System.out.println("Age: " + age);
    System.out.println("Is a student? " + isStudent);
}

// A parameter with a modifier
public void calculateDiscount(final double price, double discountPercentage) {
    double discountAmount = price * (discountPercentage / 100);
    double finalPrice = price - discountAmount;
    System.out.println("Discounted price: " + finalPrice);
}
```

Back to the parts of a method declaration, the `throws` clause is optional and specifies any checked exceptions that the method might throw. Multiple exceptions are separated by commas.

The method body is enclosed in curly braces `{}` and contains the code that implements the method's functionality. It can include variable declarations, loops, conditionals, method calls, and other statements.

If the method has a return type other than void, it must include a `return` statement that specifies the value to be returned. The return value must be compatible with the declared return type:

```
// A simple method that returns a string
public String getName() {
    return "Mark";
}
```

Method Signatures

A method signature uniquely identifies a method within a class. It consists of the method's name and the ordered list of parameter types. The access modifiers (sucha as `public` or `private`), return types (such as `void` or `int`), and parameter names are not part of the method signature:

```
methodName(parameterType1, parameterType2, ...)
```

For example, consider the following method declarations:

```
public void printMessage(String message) {
    System.out.println(message);
}

public int calculateSum(int a, int b) {
    return a + b;
}

private void updateUser(String username, int age, boolean isActive) {
    // method body
}
```

The method signatures for these methods are:

- `printMessage(String)`
- `calculateSum(int, int)`
- `updateUser(String, int, boolean)`

Calling a Method

When calling a method, you pass arguments that match the types and order of the parameters declared in the method signature. The arguments are the actual values that are passed to the method.

This way, to call a method, you need to use the method name followed by parentheses and provide any required arguments. The syntax is:

```
[ObjectReference.]methodName([arguments]);
```

If the method is an instance method (non-static), you need to have an object of the class that contains the method. You can then call the method using the object reference followed by the dot operator and the method name.

If the method is a `static` method, you can call it directly using the class name followed by the dot operator and the method name. You don't need an object instance to call a static method.

Here are a few examples of calling methods:

```
// Calling an instance method
Person person = new Person();
person.setName("John");
String name = person.getName();

// Calling a static method
int max = Math.max(10, 20);
double random = Math.random();

// Calling a method with arguments
Calculator calculator = new Calculator();
int sum = calculator.add(5, 3);
double result = calculator.multiply(2.5, 4.0);
```

Make sure to provide the correct number and type of arguments as defined in the method signature. If there is a mismatch, the compiler will throw an error.

Using Access Modifiers with Methods

Just like with fields, access modifiers control the visibility and accessibility of methods. The same four access modifiers can be used: `public`, `private`, `protected`, and default (package-private).

Consider this class:

```
package com.my.package;

public class MathUtils {
    public static int add(int a, int b) {
        return a + b;
    }

    private static int subtract(int a, int b) {
        return a - b;
    }

    protected static int multiply(int a, int b) {
        return a * b;
    }

    static int divide(int a, int b) {
        return a / b;
    }
}
```

The `add` method is declared as `public`, so it can be called from any other class:

```
int sum = MathUtils.add(1, 2);
```

The `subtract` method is declared as `private`. It can only be called from within the `MathUtils` class itself. Trying to call it from another class will result in a compile error:

```
// This will not compile
int difference = MathUtils.subtract(10, 7);
```

The `multiply` method is declared as `protected`. It can be called from within the same class, any subclass (even in a different package), and other classes in the same package:

```
package com.my.other.package;

// Calling from a subclass in a different package
public class AdvancedMathUtils extends MathUtils {
    public static int square(int a) {
        return multiply(a, a);
    }
}
```

The `divide` method has default (package-private) access since no explicit modifier is specified. Remember, this means that the method is accessible only to classes within the same package:

```
package com.my.package;

// Calling from another class in the same package
public class ArithmeticOperations {
    public static int performDivision(int a, int b) {
        return MathUtils.divide(a, b);
```

```
    }
}
```

Passing Arguments Among Methods

In Java, when you pass arguments to a method, they are always passed by value. This means that a copy of the value is passed to the method, rather than a reference to the original variable. However, the behavior of pass-by-value differs depending on whether you are passing a primitive type (like an `int`) or a reference type (like an object such as `String`).

When you pass a primitive type to a method, the method receives a copy of the value. Any changes made to the parameter inside the method do not affect the original variable outside the method.

Here's an example:

```
public void testPrimitive() {
    int num = 10;
    modifyPrimitive(num);
    System.out.println(num); // Output: 10
}

public void modifyPrimitive(int value) {
    value = 20;
}
```

In this example, the `modifyPrimitive` method receives a copy of the value of `num`. Modifying the `value` parameter inside the method does not change the original `num` variable in the `testPrimitive` method.

When you pass a reference type to a method, the method receives a copy of the reference to the object. While the reference itself is passed by value, the method can still modify the state of the object that the reference points to.

Here's an example:

```
public void test() {
    Person person = new Person("John", 25);
    modifyPerson(person);
    System.out.println(person.getName()); // Output: Alice
    System.out.println(person.getAge()); // Output: 25
}

public void modifyPerson(Person p) {
    p.setName("Alice"); // Sets a new name
    p = new Person("Bob", 30); // Reassigns p to a new person
}
```

In this example, the `modifyPerson` method receives a copy of the reference to the `Person` object. Inside the method, the `setName()` method is called on the object referenced by `p`, which modifies the name of the original object. However, when `p` is reassigned to a new `Person` object, it does not affect the original person reference in the `main` method.

Let's explore a few more examples to clarify the difference between reassigning a reference and modifying the object itself.

First, consider this one about reassigning a reference:

```
public void test() {
    StringBuilder sb = new StringBuilder("Hello");
    modifyStringBuilder(sb);
    System.out.println(sb.toString()); // Output: Hello
```

```
}

public void modifyStringBuilder(StringBuilder builder) {
    builder = new StringBuilder("World");
}
```

In this example, the `modifyStringBuilder` method receives a copy of the reference to the `StringBuilder` object. Inside the method, the builder reference is reassigned to a new `StringBuilder` object, but this does not affect the original `sb` reference in the `main` method.

Contrast the previous example with the following, which demonstrates how modifying the state of an object differs from simply reassigning a reference:

```
public void test() {
    StringBuilder sb = new StringBuilder("Hello");
    appendToStringBuilder(sb);
    System.out.println(sb.toString()); // Output: Hello, World!
}

public void appendToStringBuilder(StringBuilder builder) {
    builder.append(", World!");
}
```

In this example, the `appendToStringBuilder` method receives a copy of the reference to the `StringBuilder` object. Inside the method, the `append()` method is called on the object referenced by builder, which modifies the state of the original object. The changes made to the object are visible outside the method.

Understanding the behavior of pass-by-value and the difference between reassigning a reference and modifying the object itself is important for writing correct and predictable code. Always consider whether you intend to modify the object or simply reassign the reference when passing reference types to methods.

Method Overloading

In Java, it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. This is called method overloading. Consider the methods of the following class:

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

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

When the `add` method is called, the Java compiler determines which version of the overloaded method to call based on the type of the arguments passed to it.

This is similar to ordering coffee at a coffee shop. The barista can prepare different variations of coffee based on your specifications, black coffee, coffee with milk, or coffee with milk and sugar. Each variation is ordered using the same word (coffee), but the ingredients you specify determine the exact type of coffee you'll receive. In the same way, when you call an overloaded method in Java, the arguments you pass determine which version of the method will be executed.

For example, if we call the `add` method with different arguments:

```
Calculator calc = new Calculator();

int result1 = calc.add(5, 10);
System.out.println(result1);  // Output: 15

double result2 = calc.add(5.5, 10.2);
System.out.println(result2);  // Output: 15.7

double result3 = calc.add(5, 10.2);
System.out.println(result3);  // Output: 15.2
```

This is what happens:

1. When `calc.add(5, 10)` is called, both arguments are of type `int`. The Java compiler matches this call with the `add` method that takes two `int` parameters, and the result is an `int` value of 15.
2. When `calc.add(5.5, 10.2)` is called, both arguments are of type `double`. The Java compiler matches this call with the `add` method that takes two `double` parameters, and the result is a `double` value of 15.7.
3. When `calc.add(5, 10.2)` is called, one argument is an `int`, and the other is a `double`. In this case, the Java compiler performs a conversion of the `int` argument to a `double` to match the `add` method that takes two `double` parameters. The result is a `double` value of 15.2.

This example demonstrates how the Java compiler uses the type of the arguments to determine which overloaded method to call. It matches the arguments with the most specific method signature available.

Java can only pick an overloaded method if it can find an exact match for the arguments or if it can find a version that is more specific through widening conversions.

Widening conversions are when you go from a smaller data type to a larger data type, for instance, from an `int` to a `long`, or, like in the above example, from an `int` to a `double`.

However, Java cannot apply narrowing conversions (going from a larger data type to a smaller one) automatically. If it doesn't find an exact match or a match through widening, it will give a compile error.

It's important to note that method overloading is not the same as method overriding. We'll talk more about overriding in the next chapter, but when overriding, you provide a different implementation for an inherited method. The overridden method must have the same name, return type, and parameters as the inherited method. On the other hand, overloaded methods must have the same name but different parameters.

So always keep this in mind, changing just the return type is not enough for method overloading. The parameter list must be different.

Also, a common misconception is that Java always choose the overloaded method with the most parameters. It doesn't. Java selects the method based on the most specific match to the argument types, not necessarily the method with the most parameters.

Consider this class that has multiple overloaded methods named `display`:

```
public class DisplayOverload {

    // Method with a single String argument
    public void display(String str) {
        System.out.println("Displaying a String: " + str);
    }

    // Overloaded method with a single int argument
    public void display(int num) {
        System.out.println("Displaying an integer: " + num);
```

```
    }

    // Overloaded method with two int arguments
    public void display(int num1, int num2) {
        System.out.println("Displaying two integers: " + num1 + " and " + num2);
    }
}

// ...

DisplayOverload obj = new DisplayOverload();

obj.display("Hello, World!"); // Calls the method with a String argument
obj.display(5); // Calls the method with a single int argument
obj.display(10, 20); // Calls the method with two int arguments
```

In this example: - When `display("Hello, World!");` is called, Java selects the `display(String str)` method because the argument is a String, which matches the parameter type of this specific method. - When `display(5);` is called, Java selects the `display(int num)` method because the argument is an integer, making it the most specific match among the overloaded methods. - When `display(10, 20);` is called, even though there are other `display` methods that could theoretically accept integers, Java picks `display(int num1, int num2)` because it most specifically matches the provided two integer arguments.

One last thing to note is that you cannot overload methods that differ only by a varargs parameter. For example, this will not compile:

```
public void sum(int[] numbers) { }
public void sum(int... numbers) { } // Compile-time error
```

The reason is that both `int[] numbers` and `int... numbers` are essentially the same from Java's perspective because `int...` is just syntactic sugar for an array of integers (`int[]`). When you try to overload a method with these two parameter types, Java sees them as identical signatures. But let's talk more about varargs.

Varargs

Varargs, short for variable-length arguments, are a feature that allow methods to accept an unspecified number of arguments of a specific type. Think of varargs like an all-you-can-eat buffet. At a buffet, you're not limited to a fixed number of dishes; you can choose to have as many different dishes as you want, and you can even go back for more. Similarly, with varargs, a method can be called with a varying number of arguments; you're not fixed to a specific number. This makes your methods more flexible and easier to use when the exact number of inputs may vary.

To define a method with varargs, you use an ellipsis (`...`) after the data type of the last parameter. Here's how it works:

```
public void display(String... words) {
    for (String word : words) {
        System.out.println(word);
    }
}
```

In this example, `display` can be called with any number of `String` arguments, including none at all. It's as if you're telling the method, "Here's what I have, take it all." This flexibility makes varargs extremely useful for creating methods that need to handle an unknown number of objects, like a list of names, numbers, or even complex objects.

Now, there are specific rules you must follow to use varargs effectively and correctly.

First, a varargs parameter must be the last parameter in a method's parameter list. This rule ensures that the method can accept a variable number of arguments without ambiguity regarding which arguments belong to the varargs parameter and which do not. For instance, consider the following method:

```
void printStrings(String title, String... strings) {
    System.out.println(title + ":");
    for (String str : strings) {
        System.out.println(str);
    }
}
```

In this example, `String... strings` is a varargs parameter that can accept any number of `String` arguments. Being the last parameter allows you to call `printStrings` with any number of strings, or even no strings at all.

Second, only one varargs parameter is allowed in a method's parameter list. This restriction prevents confusion over which arguments belong to which varargs parameter if more than one were allowed. For example, if you wanted to create a method that sums numbers, you might do the following:

```
double multiplyAndSum(double multiplier, int... numbers) {
    double sum = 0;
    for (int num : numbers) {
        sum += num;
    }
    return sum * multiplier;
}
```

This method correctly includes only one varargs parameter (`int... numbers`), ensuring clarity in how it should be called and how it operates on the passed arguments.

Third, a method with a varargs parameter can be overloaded, but you have to make sure to avoid ambiguity. This requires ensuring that each method signature is distinct enough to prevent compile-time errors. For example, you could have:

```
void display(String s, int... numbers) {
    System.out.println(s);
    for (int num : numbers) {
        System.out.print(num + " ");
    }
    System.out.println();
}

void display(String first, String second) {
    System.out.println(first + ", " + second);
}
```

Here, `display` is overloaded with one version accepting a string and a varargs integer parameter, and another accepting two strings. This overloading is valid because the method signatures are distinct, ensuring that the compiler can determine which method to call based on the arguments provided.

Inside the method, accessing the elements of a varargs parameter can be done in several ways, each suitable for different scenarios.

The simplest way to access elements in a varargs parameter is by treating it as an array and accessing its elements directly using an index. This method is useful when you know the exact number of arguments or need to access specific elements. For example, consider a method that prints the first, second, and last elements of a varargs parameter:

```
void printSelectedNumbers(int... numbers) {
```

```
    if (numbers.length >= 3) {
        System.out.println("First: " + numbers[0]);
        System.out.println("Second: " + numbers[1]);
        System.out.println("Last: " + numbers[numbers.length - 1]);
    } else {
        System.out.println("Insufficient arguments.");
    }
}
```

This method directly accesses elements by their indices, similar to an array access, making it straightforward to retrieve specific values.

For iterating over each element in a varargs parameter, the enhanced for loop provides a clean and concise way to process each argument. This approach is most beneficial when you need to perform operations on every element or when the number of arguments is variable. Here's an example that sums all numbers passed to the method:

```
int sumAll(int... numbers) {
    int sum = 0;
    for (int num : numbers) {
        sum += num;
    }
    return sum;
}
```

The enhanced `for` loop automatically iterates over each element in `numbers`, allowing for easy aggregation or processing.

Although similar to using an enhanced `for` loop, you might sometimes need to manually iterate over a varargs parameter using its `length` property for more complex logic, such as when you need to access the current index. Here's how you might print each element with its index:

```
void printWithIndices(String... strings) {
    for (int i = 0; i < strings.length; i++) {
        System.out.println("Element " + i + ": " + strings[i]);
    }
}
```

This method leverages the `length` property of the varargs parameter to manually control the iteration, offering flexibility for index-based operations.

For more complex operations, including filtering, mapping, or aggregating elements, Java's Stream API can work directly with varargs. This method is particularly powerful for processing elements in functional programming style. We'll cover streams in a later chapter, but, for instance, you can filter and sum only the even numbers as follows:

```
int sumEvenNumbers(int... numbers) {
    return Arrays.stream(numbers) // Convert varargs to a stream
                 .filter(n -> n % 2 == 0) // Filter even numbers
                 .sum(); // Sum them
}
```

Once you have defined a method that takes a vararg parameter, you can call it by passing individual arguments, by passing an array, or by calling it without any arguments.

The most straightforward way to call a method with varargs is by passing individual arguments to it. This approach is identical to calling a method with a fixed number of parameters, but with the added flexibility of specifying any number of arguments. Here's an example using a method that prints out each argument:

```
void printArgs(String... args) {
    for (String arg : args) {
        System.out.println(arg);
    }
}

// Calling the method
printArgs("Hello", "World", "Varargs", "are", "flexible");
```

In this example, the `printArgs` method is called with five string arguments, demonstrating the ease with which you can pass any number of arguments.

Alternatively, you can call a varargs method by passing an array of the specified type. This approach is useful when the arguments are already stored in an array, or when you wish to dynamically construct the list of arguments. Consider a method that sums an arbitrary number of integers:

```
int sumNumbers(int... numbers) {
    return Arrays.stream(numbers).sum();
}

// Calling the method with an array
int[] numberArray = {1, 2, 3, 4, 5};
int sum = sumNumbers(numberArray);
System.out.println("Sum is: " + sum);
```

Here, `sumNumbers` is called with an integer array, showcasing how an array matches the varargs signature, providing a compact way to pass multiple arguments.

Finally, a varargs method can also be called without passing any arguments. This feature is particularly useful when an operation is optional or when there is a valid default behavior in the absence of inputs. Here's a method that concatenates any number of strings, with a demonstration of calling it without arguments:

```
String concatenateStrings(String... strings) {
    return Stream.of(strings).collect(Collectors.joining(", "));
}

// Calling the method without arguments
String result = concatenateStrings();
System.out.println("Result: " + result);
```

The above example illustrates that calling `concatenateStrings` without any arguments is perfectly valid and that varargs provide a flexible method signature that accommodates a wide range of use cases.

The `main` Method

The `main` method is a special method in Java that serves as the entry point of a Java application. When you run a Java program, the JVM looks for this method and starts executing the code inside it. Every Java application must have a `main` method in at least one of its classes.

Here's the syntax for declaring a `main` method:

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

Let's break down each part:

- `public`: The `main` method must be declared as `public` to allow the JVM to call it from outside the class.

- `static`: The `main` method must be declared as `static` so that it can be called without creating an instance of the class.
- `void`: The `main` method doesn't return any value, so its return type is `void`.
- `main`: The name of the method must be "main" (all lowercase) for the JVM to recognize it as the entry point.
- `String[] args`: The `main` method accepts a single parameter, of type `String` array, conventionally named `args`. This parameter allows you to pass command-line arguments to the program.

Here's an example of a simple `main` method:

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

In this example, the `main` method simply prints `Hello, World!` to the console.

The arguments of the program are passed as a `String` array, where each element represents a separate argument:

```
public class CommandLineArguments {
    public static void main(String[] args) {
        if (args.length > 0) {
            System.out.println("Arguments:");
            for (String arg : args) {
                System.out.println(arg);
            }
        } else {
            System.out.println("No arguments provided.");
        }
    }
}
```

In this example, the `main` method checks if any arguments were passed, using `args.length`. If there are arguments, it iterates over the `args` array and prints each argument. If no arguments were provided, it prints a message indicating that.

You can run this program from the command line and pass arguments like this:

```
java CommandLineArguments arg1 arg2 arg3
```

This will be the output:

```
Arguments:
arg1
arg2
arg3
```

If you run the program without any arguments:

```
java CommandLineArguments
```

This will be the output:

```
No arguments provided.
```

It's possible to have multiple methods named `main` in a class, as long as they have different parameter lists. However, only the method defined as `public static void main(String[] args)` will be recognized as the entry point of the application:

```java
public class MainOverloading {
    public static void main(String[] args) {
        System.out.println("Main method with String[] args");
        main(42);
    }

    public static void main(int num) {
        System.out.println("Main method with int parameter: " + num);
    }
}
```

In this example, the class has two `main` methods: one with the standard signature and another with an int parameter, however, the `main(String[] args)` method is the entry point, and it calls the `main(int num)` method.

This is the output:

```
Main method with String[] args
Main method with int parameter: 42
```

Constructors and Initializers

Constructors

In Java, a constructor is a special method used to initialize objects. It is called when an instance of a class is created.

Imagine a constructor as a recipe for baking a specific cake. Just as the recipe contains the instructions and ingredients to make the cake, a constructor has the code to set up the initial state of an object.

The syntax to define a constructor is straightforward. It has the same name as the class and no return type, not even void.

Here's an example:

```java
class Cake {
    String flavor;
    double price;
    Cake() {
        flavor = "Vanilla";
        price = 9.99;
    }
}
```

To create an object, we use the `new` keyword followed by a call to the constructor:

```java
Cake myCake = new Cake();
```

The above line will create a new `Cake` object with the default `Vanilla` flavor and a price of `9.99`.

But what if you want the default flavor but a different price? Or customize both in certain cases?

Well, just as you can bake different varieties of cakes by tweaking the recipe, you can create objects with different initial states by providing multiple constructors.

For example, let's add another constructor to our `Cake` class:

```java
Cake(String flavor, double price) {
    this.flavor = flavor;
    this.price = price;
}
```

With this constructor, we can create a cake with any flavor and price we want:

```
Cake specialCake = new Cake("Chocolate", 12.99);
```

Having multiple constructors gives flexibility in object creation. We can provide different ways to initialize an object based on the data available at the time of creation.

The constructor with no parameters is called the default constructor. If you don't define any constructors in your class, the compiler will automatically provide a default constructor with an empty body.

However, if you define any constructor (like our parameterized one), the compiler will not provide a default constructor. In this case, if you still want the option to create an object without specifying parameters, you need to explicitly define the default constructor.

So in our `Cake` class, we could have both constructors:

```
class Cake {
    String flavor;
    double price;

    Cake() {
        flavor = "Vanilla";
        price = 9.99;
    }

    Cake(String flavor, double price) {
        this.flavor = flavor;
        this.price = price;
    }
}
```

Now we can create a default vanilla cake with `new Cake()` or a customized cake with `new Cake("Chocolate", 10.99)`.

Instance Initializers

Instance initializers are blocks of code that are executed when an object is created, just like constructors. However, while constructors are methods with a specific name and potentially parameters, instance initializers are just code blocks within a class.

Let's use an analogy to understand instance initializers.

Imagine moving into a new house. We all have our unique rituals to make a house feel like a home. Some might hang family photos, others might paint the walls in their favorite color. These rituals are specific to each person, just as instance initializers are specific to each object.

Here's the syntax for an instance initializer:

```
class House {
    String color;
    // instance initializer
    {
        color = "White";
        System.out.println("Performing move-in ritual");
    }
}
```

Whenever a new `House` object is created, the code inside the instance initializer block will run. It will set the color to `White` and print `"Performing move-in ritual"`.

So, how do instance initializers compare to constructors, and when might you use them?

Well, imagine you have a class with multiple constructors. Each constructor needs to perform some common initialization tasks. Instead of duplicating the code in each constructor, you can put it in an instance initializer. The initializer code will run regardless of which constructor is used.

```
class House {
    String color;
    int numberOfRooms;

    // instance initializer
    {
        color = "White";
        System.out.println("Performing move-in ritual");
    }

    House(int numberOfRooms) {
        this.numberOfRooms = numberOfRooms;
    }

    House(String color, int numberOfRooms) {
        this.color = color;
        this.numberOfRooms = numberOfRooms;
    }
}
```

In this case, regardless of which constructor is used to create a `House` object, the instance initializer will run, setting the default color to `White` and printing the move-in message.

However, it's important to note that in most cases, you can achieve the same result by simply moving the common initialization code into a separate method and calling that method from each constructor.

In fact, some argue that instance initializers are redundant since anything you can do with an instance initializer, you can also do with a constructor. The main difference is that constructors can take parameters, while instance initializers cannot.

That said, there are some scenarios where instance initializers can be useful. For example, if you're using anonymous classes (which we'll cover later), you can't define a constructor, so an instance initializer is your only option for initialization code.

Static Initializers

Static initializers are blocks of code that are executed when a class is loaded into memory, before any instances of the class are created. They are used to initialize static variables or perform actions that are common to all instances of the class.

Imagine a town hall meeting that happens once when a town is established. In this meeting, the town's leaders set up rules and guidelines that apply to everyone in the town. This one-time setup is similar to what a `static` initializer does for a class.

Here's the syntax for a `static` initializer:

```
class TownHall {
    static String townName;
    static int population;

    // static initializer
    static {
        townName = "JavaVille";
        population = 1000;
```

```
        System.out.println("Town established: " + townName);
    }
}
```

The `static` keyword before the opening brace denotes that this is a static initializer block. It will run once when the `TownHall` class is loaded, setting the `townName` to `JavaVille`, the initial `population` to `1000`, and printing `Town established: JavaVille`.

Now, you might think that static initializers are just another way to initialize static variables, and you could achieve the same result by directly initializing the variables at their declaration, like this:

```
static String townName = "JavaVille";
static int population = 1000;
```

And you'd be partially correct. For simple initializations, direct assignment is often clearer and more concise.

However, static initializers provide more flexibility. They allow you to write more complex initialization logic, such as: - Calling methods - Using control structures like loops and conditionals - Handling exceptions

Here's an example that demonstrates this:

```
static List<String> residents = new ArrayList<>();

static {
    Path path = Paths.get("residents.txt");
    try (Stream<String> lines = Files.lines(path)) {
        lines.forEach(residents::add);
    } catch (IOException e) {
        System.out.println("Residents file not found.");
    }
}
```

In this case, we're using the static initializer to read a list of residents from a file and populate the `residents` list. This kind of complex initialization would not be possible with a simple direct assignment.

Another key difference is that a class can have multiple static initializers, and they will be executed in the order they appear in the class. This can be useful for organizing complex initialization logic into readable chunks.

It's important to note that static initializers are executed before any instance of the class is created, and even before the `main` method is called. They are part of the class loading process.

In contrast, instance initializers and constructors are run every time a new instance of the class is created. They are part of the object creation process.

Initialization Order

We have reviewed constructors, instance initializers, and static initializers. However, if a class includes all three, which one executes first? What is the order of initialization?

When a class is loaded, the first things to be initialized are the static variables and static initializers, in the order they appear in the class. This happens once per class loading, before any instances are created.

After that, whenever a new instance of the class is created, the instance variables are initialized, and the instance initializers and constructors are run.

The order is as follows:

1. Instance variables are initialized to their default values (`0`, `false`, or `null`). This step ensures that all instance variables have a predictable starting state before any further initialization code is executed.
2. Instance initializers are run in the order they appear in the class.
3. The constructor is executed.

Here's an example that demonstrates this order:

```
class InitializationOrder {
    static int staticVar = 1;
    int instanceVar = 1;

    static {
        System.out.println("Static Initializer: staticVar = " + staticVar);
        staticVar = 2;
    }

    {
        System.out.println("Instance Initializer: instanceVar = " + instanceVar);
        instanceVar = 2;
    }

    InitializationOrder() {
        System.out.println("Constructor: instanceVar = " + instanceVar);
        instanceVar = 3;
    }

    public static void main(String[] args) {
        System.out.println("Creating new instance");
        InitializationOrder obj = new InitializationOrder();
        System.out.println("Created instance: instanceVar = " + obj.instanceVar);
    }
}
```

If you run this code, the output will be:

```
Static Initializer: staticVar = 1
Creating new instance
Instance Initializer: instanceVar = 1
Constructor: instanceVar = 2
Created instance: instanceVar = 3
```

Let's break this down: 1. When the `InitializationOrder` class is loaded, the static variable `staticVar` is initialized to `1`, and then the static initializer is run, which prints the current value of `staticVar` (`1`) and then sets it to `2`. 2. In the `main` method, we print `Creating new instance` to mark the start of instance creation. 3. A new `InitializationOrder` object is created. First, the instance variable `instanceVar` is initialized to its default value of `1`. 4. The instance initializer is run, which prints the current value of `instanceVar` (`1`) and then sets it to `2`. 5. The constructor is executed, which prints the current value of `instanceVar` (`2`) and then sets it to `3`. 6. Finally, back in the `main` method, we print the final value of `instanceVar` (`3`).

It's important to keep this order in mind, especially if your initializers and constructors depend on each other. Incorrect assumptions about initialization order can lead to subtle bugs.

Also, note that if a class has multiple static initializers, they will run in the order they appear in the class. The same is true for instance initializers.

Let's extend our previous example to demonstrate this:

```
class MultipleInitializers {
    static int staticVar1;
    static int staticVar2;
    int instanceVar1;
    int instanceVar2;
```

```
    static {
        System.out.println(
          "Static Initializer 1: staticVar1 = " + staticVar1
        );
        staticVar1 = 1;
    }

    static {
        System.out.println(
          "Static Initializer 2: staticVar2 = " + staticVar2
        );
        staticVar2 = 2;
    }

    {
        System.out.println(
          "Instance Initializer 1: instanceVar1 = " + instanceVar1
        );
        instanceVar1 = 1;
    }

    {
        System.out.println(
          "Instance Initializer 2: instanceVar2 = " + instanceVar2
        );
        instanceVar2 = 2;
    }

    MultipleInitializers() {
        System.out.println("Constructor");
    }

    public static void main(String[] args) {
        System.out.println("Creating new instance");
        MultipleInitializers obj = new MultipleInitializers();
        System.out.println(
          "Created instance: instanceVar1 = "
              + obj.instanceVar1
              + ", instanceVar2 = "
              + obj.instanceVar2
        );
    }
}
```

When we run this code, the output will be:

```
Static Initializer 1: staticVar1 = 0
Static Initializer 2: staticVar2 = 0
Creating new instance
Instance Initializer 1: instanceVar1 = 0
Instance Initializer 2: instanceVar2 = 0
Constructor
Created instance: instanceVar1 = 1, instanceVar2 = 2
```

Here's what's happening: 1. When the `MultipleInitializers` class is loaded, the static variables `staticVar1`

and `staticVar2` are initialized to their default value of `0`.

2. The first static initializer is run, which prints the current value of `staticVar1` (`0`) and then sets it to `1`.
3. The second static initializer is run, which prints the current value of `staticVar2` (`0`) and then sets it to `2`.
4. In the `main` method, we print `Creating new instance` to mark the start of instance creation.
5. A new `MultipleInitializers` object is created. First, the instance variables `instanceVar1` and `instanceVar2` are initialized to their default value of `0`.
6. The first instance initializer is run, which prints the current value of `instanceVar1` (`0`) and then sets it to `1`.
7. The second instance initializer is run, which prints the current value of `instanceVar2` (0) and then sets it to `2`.
8. The constructor is executed, which simply prints `Constructor`.
9. Finally, back in the `main` method, we print the final values of `instanceVar1` (`1`) and `instanceVar2` (`2`).

Remember, all static initializers will run before any instance initializers, and all initializers will run before the constructor. But within each category (static or instance), the initializers will run in the order they are defined in the class.

Extending from `java.lang.Object`

In Java, every class is implicitly a subclass of the `java.lang.Object` class, which is the root of the class hierarchy. Even if you don't explicitly extend any class, your class will automatically inherit from `Object`.

The `Object` class provides a set of fundamental methods that are common to all objects. When you create a new class, you automatically inherit these methods. Some of the commonly used methods inherited from `Object` include:

1. `toString()`: Returns a string representation of the object. By default, it returns a string consisting of the object's class name, an `@` symbol, and the object's hash code in hexadecimal format. You can override this method to provide a custom string representation of your object.
2. `equals(Object obj)`: Compares the object with another object for equality. By default, it compares the object references using the `==` operator. You can override this method to define custom equality logic based on the object's state.
3. `hashCode()`: Returns a hash code value for the object. The hash code is used in hash-based data structures such as `HashSet` and `HashMap`. By default, it returns a unique integer value for each object. If you override the `equals()` method, you should also override the `hashCode()` method to ensure that equal objects have the same hash code.
4. `getClass()`: Returns the runtime class of the object. It is a `final` method, which means it cannot be overridden.
5. `clone()`: Creates and returns a copy of the object. By default, it performs a shallow copy of the object. To use this method, your class must implement the `Cloneable` interface.

Here's an example that demonstrates some of the methods inherited from `Object`:

```
class MyClass {
    private int value;

    public MyClass(int value) {
        this.value = value;
    }
```

```
    @Override
    public String toString() {
        return "MyClass[value=" + value + "]";
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null || getClass() != obj.getClass())
            return false;
        MyClass other = (MyClass) obj;
        return value == other.value;
    }

    @Override
    public int hashCode() {
        return Objects.hash(value);
    }
}
```

In this example, the `MyClass` overrides the `toString()`, `equals()`, and `hashCode()` methods inherited from `Object`. The `toString()` method provides a custom string representation of the object, the `equals()` method defines equality based on the value field, and the `hashCode()` method generates a hash code based on the value field.

By leveraging these methods, you can provide meaningful string representations, define equality comparisons, and ensure proper behavior in hash-based data structures.

Nested Classes

In Java, it's possible to define a class within another class. Such classes are called nested classes. Similar to how a box can contain several smaller boxes inside it, a class (the outer or enclosing class) can have other classes (nested classes) defined within it.

There are four types of nested classes in Java:

1. Static nested class
2. Inner class (also known as a non-static nested class)
3. Local class
4. Anonymous class

Each type of nested class has its own characteristics and use cases:

- A static nested class is like a smaller box that doesn't depend on the larger box for its existence. It can access the static members of the enclosing class directly. However, to access non-static members, it needs an instance of the enclosing class, just like any other external class would.
- An inner class, on the other hand, is like a smaller box that is closely tied to the larger box. It can access both static and non-static members of the enclosing class directly. However, an instance of an inner class cannot exist without an instance of the enclosing class.
- A local class is like a temporary box created within a method or a block of the enclosing class. The scope of a local class is confined to the block in which it is defined. While local classes do not use traditional access modifiers like `public` or `private`, their accessibility is inherently limited to the enclosing block.

- An anonymous class is like a one-time, nameless box created for a specific purpose. It is defined and instantiated in a single statement, usually as an argument to a method call or as an initializer. The accessibility of an anonymous class is determined by the context in which it is used, such as within a method or as a field initializer, and does not use traditional access modifiers.

When deciding between a static nested class and an inner class, consider the relationship between the nested class and the enclosing class. If the nested class doesn't need access to the non-static members of the enclosing class, use a static nested class. This makes the class more independent and reusable. If the nested class requires access to the non-static members of the enclosing class or needs to be tied to an instance of the enclosing class, use an inner class.

It's important to note that while nested classes can help organize code better, they do impact how the code works. Each type of nested class has its own specific behavior and use cases. For example, an inner class has an implicit reference to an instance of the enclosing class, which can have implications for memory usage and serialization.

A common misconception is that static nested classes and inner classes are essentially the same since they are both defined within another class. However, this is not true. Static nested classes are semantically similar to any other top-level class and do not have an implicit reference to an instance of the enclosing class. Inner classes, on the other hand, are intimately tied to an instance of the enclosing class and cannot exist independently.

In terms of access modifiers, nested classes can be declared as `public`, package-private (default), `protected`, or `private`, unlike top-level classes that can't be declared as `protected` or `private`.

The accessibility of static and non-static nested classes depends on its access modifier and the accessibility of the enclosing class. For example, if the enclosing class is `public` and the nested class is `private`, the nested class can only be accessed within the enclosing class. If the nested class is public, it can be accessed from anywhere, provided the enclosing class is also accessible.

Here's a bit more detail on each type:

- **Public Nested Class:** A `public` nested class is accessible from any other class, but the accessibility of the nested class still depends on the accessibility of its outer class. If the outer class is not accessible in some context, then its nested public class won't be accessible there either.
- **Protected Nested Class:** A `protected` nested class is accessible within its own package and by subclasses of its outer class, regardless of the package the subclass is in. This allows for more controlled visibility compared to a public nested class, particularly useful when you want to expose certain functionality only to certain subclasses.
- **Private Nested Class:** A `private` nested class is accessible only within its outer class. This is useful for hiding the class from the outside world completely, making it accessible only to the outer class. This is often used for helper classes that are of no interest outside of the outer class.
- **Package-Private (Default) Nested Class:** A nested class with no access modifier is package-private, meaning it is accessible only within its own package. This is the default access level if no access modifier is specified. It's a middle ground in terms of accessibility, more restrictive than `public` but less restrictive than `private`.

Local classes are defined in a block, typically within a method body. The visibility of a local class is restricted to the block in which it is defined. Thus, while you cannot apply traditional access modifiers (`public`, `protected`, `private`) to the class itself because it is not visible outside the block, you can control the access to instances of this class from within the block.

And since anonymous classes are used within an expression, they don't allow access modifiers for the class itself. The context in which they are declared dictates their accessibility. However, the methods and fields within an anonymous class can have access modifiers, subject to normal scoping rules.

Here's a summary table of the allowed access modifiers for each type of nested class:

Nested Class Type	`public`	`protected`	`default`	`private`
Static Nested Class	Yes	Yes	Yes	Yes
Inner Class	Yes	Yes	Yes	Yes
Local Class	No	No	Yes*	No
Anonymous Class	No	No	Yes*	No

- `Yes` indicates that the access modifier is allowed.
- `No` indicates that the access modifier is not applicable.
- `Yes*` indicates that for local and anonymous classes the concept of traditional access modifiers does not apply to these classes because their visibility is inherently confined to the block in which they are declared. Therefore, they do not have access modifiers in the traditional sense.

Now let's go over each type in more detail.

Static Nested Classes

A static nested class is a class defined within another class and marked with the `static` keyword:

```
class OuterClass {
    static class StaticNestedClass {
        // members of the static nested class
    }
}
```

Static nested classes can be declared with any of the four access modifiers: `public`, `protected`, package-private (default), or `private`. The accessibility of the static nested class depends on the access modifier used and the accessibility of the enclosing class. Here's an example:

```
public class OuterClass {
    private static class PrivateNestedClass {
        // ...
    }

    protected static class ProtectedNestedClass {
        // ...
    }

    static class PackagePrivateNestedClass {
        // ...
    }

    public static class PublicNestedClass {
        // ...
    }
}
```

In this example, `PrivateNestedClass` is accessible only within `OuterClass`, `ProtectedNestedClass` is accessible within `OuterClass` and its subclasses, `PackagePrivateNestedClass` is accessible within the same package as `OuterClass`, and `PublicNestedClass` is accessible from anywhere.

Static nested classes can extend another class and implement interfaces, just like any other top-level class:

```
class BaseClass {
    // ...
}

interface MyInterface {
```

```
    // ...
}

class OuterClass {
    static class NestedClass extends BaseClass implements MyInterface {
        // ...
    }
}
```

Here, `NestedClass` extends `BaseClass` and implements `MyInterface`, demonstrating that a static nested class can extend another class and implement interfaces.

They can access the static members of the enclosing class directly, using the name of the enclosing class followed by the dot notation. However, to access non-static members of the enclosing class, a static nested class requires an instance of the enclosing class. This is because static nested classes do not inherently have access to the instance variables of the enclosing class.

Here's an example of a nested static class:

```
class OuterClass {
    private static int staticField = 10;
    private int instanceField = 20;

    static class NestedClass {
        void accessOuterMembers() {
            System.out.println(staticField); // Accessible directly
            System.out.println(instanceField); // Compilation error: cannot access non-static field
            System.out.println(new OuterClass().instanceField); // Accessible via an instance of OuterClass
        }
    }
}
```

In this example, `NestedClass` can directly access the `staticField` of `OuterClass`, but it cannot directly access the `instanceField`. To access `instanceField`, it needs an instance of `OuterClass`.

To create an instance of a static nested class, you don't need an instance of the enclosing class. You can instantiate it using the name of the enclosing class followed by the dot notation and the name of the static nested class.

```
OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();
```

When referencing static members of a static nested class from outside the enclosing class, use the enclosing class's name, followed by a dot, the static nested class's name, another dot, and then the member's name. This syntax highlights the nested structure while providing clear paths to access static members:

```
OuterClass.StaticNestedClass.staticField;
OuterClass.StaticNestedClass.staticMethod();
OuterClass.StaticNestedClass.StaticNestedNestedClass nestedNestedObject
                = new OuterClass.StaticNestedClass.StaticNestedNestedClass();
```

From within the enclosing class, you can directly access the members of the static nested class without the name of the enclosing class.

```
class OuterClass {
    static class StaticNestedClass {
        static void staticMethod() {
            // ...
        }
    }
```

```
    void outerMethod() {
        StaticNestedClass.staticMethod();
    }
}
```

As you can see, static nested classes are similar to regular, top-level classes in many ways:

1. They can have all types of access modifiers (`public`, `private`, `protected`, and package).
2. They can extend other classes and implement interfaces.
3. They can have static and non-static members.
4. They can be instantiated independently (without an instance of the enclosing class).

However, there are a few key differences:

1. Static nested classes are defined within another class, whereas top-level classes are defined independently.
2. Static nested classes have access to the static members of the enclosing class directly, while top-level classes need to use the enclosing class name to access its static members.
3. Static nested classes can be `private`, allowing for better encapsulation, whereas top-level classes can only be `public` or package-private.

In summary, static nested classes are essentially like regular top-level classes that have been nested within another class for organizational purposes. They do not have an implicit reference to an instance of the enclosing class and can be instantiated independently. This makes them useful for grouping related classes together and providing a level of encapsulation.

Non-static Nested Classes

Non-static nested classes, also known as inner classes, are classes that are defined within another class without the `static` keyword:

```
class OuterClass {
    class InnerClass {
        // members of the inner class
    }
}
```

An inner class can be declared with any of the four access modifiers: `public`, `protected`, `private`, or the default access level. The accessibility of the inner class depends on the access modifier used and the accessibility of the enclosing class. If the outer class is `public` and the inner class is `private`, the inner class can only be accessed within the outer class. Here's an example:

```
public class OuterClass {
    private class PrivateInnerClass {
        // ...
    }

    protected class ProtectedInnerClass {
        // ...
    }

    class PackagePrivateInnerClass {
        // ...
    }
```

```
    public class PublicInnerClass {
        // ...
    }
}
```

In this example, `PrivateInnerClass` is accessible only within `OuterClass`, `ProtectedInnerClass` is accessible within `OuterClass` and its subclasses, `PackagePrivateInnerClass` is accessible within the same package as `OuterClass`, and `PublicInnerClass` is accessible from anywhere, provided `OuterClass` is accessible.

An inner class can extend another class and implement interfaces, just like any other class. This allows inner classes to inherit behavior and conform to contracts defined by other classes and interfaces:

```
class BaseClass {
    // ...
}

interface MyInterface {
    // ...
}

class OuterClass {
    class InnerClass extends BaseClass implements MyInterface {
        // ...
    }
}
```

Here, `InnerClass` extends `BaseClass` and implements `MyInterface`, demonstrating that an inner class can inherit from another class and conform to an interface.

An inner class has access to all members (fields, methods, and nested classes) of the enclosing class, including `private` members. This is because an inner class is associated with an instance of the outer class and shares a special relationship with it. The inner class can directly access and manipulate the state of the outer class instance:

```
class OuterClass {
    private int privateField = 10;
    protected int protectedField = 20;
    int packagePrivateField = 30;
    public int publicField = 40;

    class InnerClass {
        void accessOuterMembers() {
            System.out.println(privateField);
            System.out.println(protectedField);
            System.out.println(packagePrivateField);
            System.out.println(publicField);
        }
    }
}
```

In this example, `InnerClass` has direct access to all members of `OuterClass`, including the private field `privateField`. The inner class can freely access and manipulate the state of the outer class instance.

To create an instance of an inner class, you typically need an instance of the outer class. The most common way to instantiate an inner class is from within a non-static method of the outer class:

```
class OuterClass {
    class InnerClass {
```

```
        // ...
    }

    void outerMethod() {
        InnerClass innerObject = new InnerClass();
    }
}
```

From outside the outer class, you can instantiate an inner class using the following syntax:

```
OuterClass outerObject = new OuterClass();
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

To reference members (fields, methods, nested classes) of an inner class from outside the outer class, you first need an instance of the outer class, then use the dot notation to access the inner class, followed by another dot and the member name:

```
OuterClass outerObject = new OuterClass();
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
innerObject.innerField;
innerObject.innerMethod();
```

From within the outer class, you can directly access the members of the inner class using an instance of the inner class:

```
class OuterClass {
    class InnerClass {
        void innerMethod() {
            // ...
        }
    }

    void outerMethod() {
        InnerClass innerObject = new InnerClass();
        innerObject.innerMethod();
    }
}
```

Inner classes differ from regular, top-level classes in several ways:

1. Inner classes are defined within another class, whereas top-level classes are defined outside of other classes.
2. Inner classes have access to all members of the enclosing class, including `private` members, while top-level classes can only access `public` and `protected` members of other classes.
3. Inner classes are associated with an instance of the outer class and cannot exist independently, while top-level classes can be instantiated independently.
4. Inner classes can be `private`, allowing for better encapsulation, whereas top-level classes can only be `public` or package-private.

Inner classes are useful when a class is closely tied to another class and needs access to its internals. They provide a way to organize related classes and maintain a tight coupling between them. Inner classes are commonly used for implementing event listeners, iterators, or other functionality that is specific to the enclosing class.

Local Classes

Local classes are defined within a block of code, typically within a method or a constructor. They have limited scope and are only accessible within the block where they are defined:

```
void someMethod() {
    class LocalClass {
        // members of the local class
    }
}
```

A local class cannot have any access modifiers. They cannot be accessed from outside the block or method in which they are defined. This is because local classes are not members of the enclosing class, but rather defined within a method or block.

However, they can extend another class and implement interfaces, just like any other class.

Example:

```
void someMethod() {
    class LocalClass extends BaseClass implements MyInterface {
        // ...
    }
}
```

Also, a local class has access to all members (fields, methods, and nested classes) of the enclosing class, including `private` members. Additionally, a local class can access `final` or effectively final local variables and parameters of the enclosing method:

```
class OuterClass {
    private int privateField = 10;

    void someMethod(final int parameter) {
        final int localVariable = 20;

        class LocalClass {
            void accessOuterMembers() {
                System.out.println(privateField);
                System.out.println(parameter);
                System.out.println(localVariable);
            }
        }

        LocalClass localObject = new LocalClass();
        localObject.accessOuterMembers();
    }
}
```

In this example, `LocalClass` has access to the `private` field `privateField` of `OuterClass`, as well as the `final` parameter `parameter` and the final local variable `localVariable` of the `someMethod()`.

To create an instance of a local class, you can instantiate it within the method or block where it is defined, using the `new` keyword:

```
void someMethod() {
    class LocalClass {
        // ...
    }
```

```
    LocalClass localObject = new LocalClass();
}
```

To reference members (fields, methods, nested classes) of a local class, you can directly access them using an instance of the local class within the method or block where it is defined:

```
void someMethod() {
    class LocalClass {
        int localField = 10;

        void localMethod() {
            System.out.println("Local method");
        }
    }

    LocalClass localObject = new LocalClass();
    System.out.println(localObject.localField);
    localObject.localMethod();
}
```

Local classes differ from regular, top-level classes in several ways:

1. Local classes are defined within a method or block, whereas top-level classes are defined independently.
2. Local classes have limited scope and are only accessible within the block where they are defined, while top-level classes have a broader scope.
3. Local classes cannot have access modifiers, while top-level classes can be `public` or package-private.
4. Local classes can access `final` or effectively final local variables and parameters of the enclosing method, while top-level classes cannot directly access local variables or parameters.

Local classes are useful when you need to define a class that is only used within a specific method or block and does not need to be accessed from other parts of the code. They provide a way to encapsulate behavior and state within a limited scope.

Anonymous Classes

Anonymous classes are a way to define and instantiate a class at the same time, without giving it a name. They are used for creating one-time implementations of interfaces or abstract classes.

To declare an anonymous class, you use the `new` keyword followed by the name of an interface or an abstract class, and then provide the class body in curly braces.

```
interface MyInterface {
    void myMethod();
}

MyInterface myObject = new MyInterface() {
    @Override
    public void myMethod() {
        // Implementation of myMethod()
    }
};
```

Since anonymous classes are not explicitly named and are defined at the point of use, they cannot have any explicit access modifiers. Their accessibility is determined by the context in which they are used. Specifically, the scope in which an anonymous class is defined determines its accessibility. For instance, if an anonymous class is defined within a method, it is accessible only within that method. If it is defined within a class, it follows the accessibility rules of that class.

An anonymous class can extend a class or implement an interface, however, it cannot do both at the same time:

```
class BaseClass {
    void baseMethod() {
        System.out.println("Base method");
    }
}

interface MyInterface {
    void myMethod();
}

BaseClass anonymousObject1 = new BaseClass() {
    @Override
    void baseMethod() {
        System.out.println("New implementation of base method");
    }
};

MyInterface anonymousObject2 = new MyInterface() {
    @Override
    public void myMethod() {
        System.out.println("Implementation of myMethod()");
    }
};
```

An anonymous class has access to all members (fields, methods, and nested classes) of the enclosing class, including `private` members. Additionally, an anonymous class can access `final` or effectively `final` local variables and parameters of the enclosing method:

```
class OuterClass {
    private int privateField = 10;

    void someMethod(final int parameter) {
        final int localVariable = 20;

        MyInterface anonymousObject = new MyInterface() {
            @Override
            public void myMethod() {
                System.out.println(privateField);
                System.out.println(parameter);
                System.out.println(localVariable);
            }
        };

        anonymousObject.myMethod();
    }
}
```

An anonymous class does not have a name, so you cannot directly reference its members from outside the class body. However, you can reference the members of the interface or abstract class that the anonymous class implements or extends:

```
interface MyInterface {
    void myMethod();
```

```
    int myField = 10;
}

MyInterface anonymousObject = new MyInterface() {
    @Override
    public void myMethod() {
        System.out.println("Implementation of myMethod()");
    }
};

anonymousObject.myMethod();
System.out.println(MyInterface.myField);
```

Anonymous classes differ from regular, top-level classes in several ways:

1. Anonymous classes are defined and instantiated at the same time, without an explicit name, whereas top-level classes are defined separately and instantiated using the `new` keyword.
2. Anonymous classes are defined at the point of use, typically as an argument to a method or as an initializer, while top-level classes are defined independently.
3. Anonymous classes cannot have explicit access modifiers, constructors, or static members, while top-level classes can have all of these.
4. Anonymous classes are used for creating one-time implementations or instances, while top-level classes are used for creating reusable and named classes.

In summary, anonymous classes are useful when you need to create a one-time implementation of an interface or abstract class without the need for a named class. They provide a concise way to define and instantiate a class in a single expression.

Finally, to wrap up this section, here's a table that summarizes many properties of each type of nested class:

Property	Static Nested Class	Inner Class	Local Class	Anonymous Class
Association with Outer Class	Loosely associated (can exist without an instance of the outer class)	Tightly coupled (cannot exist without an instance of the outer class)	Tightly coupled (associated with an instance of the enclosing block)	Tightly coupled (instantiated within an expression and associated with an instance of the enclosing block)
Can Declare Static Members	Yes (including static methods and fields)	No (except final static fields)	No (cannot declare static members, only final variables)	No (cannot declare static members, only final variables)
Access to Members of the Outer Class	Only static members	Both static and instance members	Both static and instance members	Both static and instance members
Requires Reference to Outer Class Instance	No	Yes	Yes (implicitly final or effectively final variables from the enclosing scope)	Yes (implicitly final or effectively final variables from the enclosing scope)

Property	Static Nested Class	Inner Class	Local Class	Anonymous Class
Typical Use Cases	Grouping classes that are used in only one place, enhancing encapsulation	Handling events, accessing private members of the outer class, providing more readable and maintainable code	Encapsulating complex code within a method without making it visible outside	Simplifying the instantiation of objects that are meant to be used once or where the class definition is unnecessary

Classes and Source Files

It's important to note that you can have one or more class definitions in one Java source file. However, you should follow these rules:

Public Class Rule If a Java class is declared as `public`, the name of the file must exactly match the name of the public class, including case sensitivity, with the addition of the `.java` extension. For example, if you have a `public` class named `MyClass`, then the source file must be named `MyClass.java`:

```
// File name: MyClass.java
public class MyClass {
    // class body
}
```

Single Public Class Per File A Java source file can contain multiple classes, but it can only have one `public` class. If there are multiple classes in a file and one of them is declared `public`, the file name must match the name of the `public` class. For example, if `PublicClass` is the `public` class, the file must be named `PublicClass.java`, and it can also contain `AnotherClass` which is not `public`:

```
// File name: PublicClass.java
public class PublicClass {
    // class body
}

class AnotherClass {
    // class body
}
```

No Public Class If there's no `public` class in the file, any name can be used. For example, the following file, `ManyClasses.java` contains multiple classes, none of which are `public`:

```
// File name: ManyClasses.java
class FirstClass {
    // class body
}

class SecondClass {
    // class body
}
```

Non-Public Classes If multiple non-public classes exist in a single file, then the file name does not need to match the name of any of the classes. For example, you can have a file named `UtilityClasses.java` containing multiple non-public classes that don't match this name:

```
// File name: UtilityClasses.java
class HelperClass {
    // class body
}

class AnotherHelperClass {
    // class body
}
```

Case Sensitivity Java is case-sensitive. If your class is named `CaseSensitiveClass`, the file name must match exactly (`CaseSensitiveClass.java`):

```
// File name: CaseSensitiveClass.java
public class CaseSensitiveClass {
    // class body
}
```

So the main restriction in Java is that a source file cannot contain more than one `public` class. This helps in organizing code and making it easier to manage. Each public class must be in its own source file, and the file name must match the class name (including case sensitivity) with the `.java` extension.

However, a single Java source file can contain any number of non-public classes. These classes are by default package-private, and the file can also contain `protected` or `private` nested classes within `public` or package-private classes. This flexibility allows for logically related classes to be grouped together within the same file if they are not intended for `public` use, aiding in encapsulation and modular design.

Key Points

- Object-oriented programming (OOP) organizes code into objects, which represent real-world entities containing data (attributes) and behaviors (methods).
- Classes are blueprints or templates that define the data and behaviors common to all objects of that type, while objects are distinct instances of a class containing unique data values.
- The main stages of an object's life-cycle in Java are creation using the `new` keyword, accessing via reference variables, and cleanup by Java's garbage collector when no longer referenced.
- Keywords are reserved words in Java that define the structure and syntax of Java programs. They cannot be used as identifiers.
- Comments are annotations in the code ignored by the compiler, used to describe or explain code. Java supports single-line (`//`), multi-line (`/* */`), and documentation (`/** */`) comments.
- Packages organize related classes, interfaces, and sub-packages into a single unit, providing a level of access control. The `package` keyword is used to create a package.
- Access modifiers (`public`, `protected`, `default`, `private`) control the visibility and accessibility of classes, methods, and variables from other parts of a Java application.
- A class is declared using the `class` keyword followed by the class name. It can optionally extend a superclass using `extends` and implement interfaces using `implements`.
- Fields are variables declared at the class level to hold the state of an object. They can have access modifiers, specifiers (`static`, `final`), a type, and an optional initial value.

- Methods are blocks of code that perform specific tasks and optionally return values. They are declared with an optional access modifier, specifiers, return type, name, parameters, and a method body.
- Method overloading is the practice of defining multiple methods with the same name but different parameter lists within the same class.
- The Java compiler determines which overloaded method to call based on the number, types, and order of the arguments passed during the method invocation.
- Java can only pick an overloaded method if it can find an exact match for the arguments or if it can find a more specific version through widening conversions (`int` to `long`, `int to double`, etc.).
- Varargs (variable-length arguments) allow methods to accept an unspecified number of arguments of a specific type. To define a method with varargs, use an ellipsis (…) after the data type of the last parameter in the method signature.
- A varargs parameter must be the last parameter in a method's parameter list, and only one varargs parameter is allowed per method.
- Methods with varargs can be overloaded, but you must avoid ambiguity by ensuring the method signatures are different.
- Constructors are special methods used to initialize objects, called when an instance of a class is created using the `new` keyword. They have the same name as the class and no return type.
- Instance initializers are blocks of code executed when an object is created, similar to constructors but without parameters. They are enclosed in `{}` within the class body.
- Static initializers are blocks of code executed when a class is loaded into memory, before any instances are created. They are defined using the `static` keyword followed by `{}`.
- Every class implicitly extends the `java.lang.Object` class, inheriting fundamental methods like `toString()`, `equals()`, and `hashCode()`.
- Nested classes are classes defined within another class. They can be static nested classes, inner classes (non-static nested classes), local classes, or anonymous classes.
- Static nested classes are associated with the outer class itself and can access its static members directly. They can be instantiated independently, without an instance of the outer class.
- Inner classes (non-static nested classes) are associated with an instance of the outer class and have access to both static and non-static members of the outer class. They require an instance of the outer class to be instantiated.
- Local classes are defined within a block, typically a method, and have access to final or effectively final variables from the enclosing scope. They cannot have access modifiers and are only visible within the defining block.
- Anonymous classes are defined within an expression and are used for creating one-time implementations of interfaces or abstract classes. They do not have a name and are instantiated at the point of declaration.
- If a class is declared `public`, the name of the Java source file must exactly match the name of the `public` class, including case sensitivity, with the `.java` extension.
- A Java source file can contain multiple class definitions, but only one of them can be declared `public`. If there is no `public` class, the file name can be different from the class names.

Practice Questions

1. Consider the following code snippet:

```java
public class Main {
    public static void main(String[] args) {
        StringBuilder sb1 = new StringBuilder("Java");
        StringBuilder sb2 = new StringBuilder("Python");
        sb1 = sb2;
        // More code here
    }
}
```

After the execution of the above code, which of the following statements is true regarding garbage collection?

A) Both `sb1` and `sb2` are eligible for garbage collection.
B) Only the `StringBuilder` object initially referenced by `sb1` is eligible for garbage collection.
C) Only the `StringBuilder` object initially referenced by `sb2` is eligible for garbage collection.
D) Neither of the `StringBuilder` objects are eligible for garbage collection.

2. Which of the following are reserved keywords in Java? (Choose all that apply.)

A) `implement`
B) `array`
C) `volatile`
D) `extends`

3. Consider the following code snippet:

```java
1. // calculates the sum of numbers
2. public class Calculator {
3.     /* Adds two numbers
4.      * @param a the first number
5.      * @param b the second number
6.      * @return the sum of a and b
7.      */
8.     public int add(int a, int b) {
9.         // return the sum
10.        return a + b;
11.    }
12.    //TODO: Implement subtract method
13.}
```

Which of the following statements are true about the comments in the above code? (Choose all that apply.)

A) Line 1 is an example of a single-line comment.
B) Lines 3-7 demonstrate the use of a javadoc comment.
C) Line 9 uses a javadoc comment to explain the `add` method.
D) Line 12 uses a special `TODO` comment, different from a single-line comment.
E) Lines 3-7 is a block comment that is used as if it were a javadoc comment.

4. Consider you have the following two Java files located in the same directory:

```java
// File 1: Calculator.java
package math;

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

// File 2: Application.java
```

```java
package app;

import math.Calculator;

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

Which of the following statements is true regarding the `package` and `import` statements in Java?

A) The `import` statement in `Application.java` is unnecessary because both classes are in the same directory.
B) The `import` statement in `Application.java` is necessary for using the `Calculator` class because they belong to different packages.
C) The `Calculator` class will not be accessible in `Application.java` due to being in a different directory.
D) Removing the `package` statement from both files will allow `Application.java` to use `Calculator` without an `import` statement, regardless of directory structure.

5. Consider the default access levels provided by Java's four access modifiers: `public`, `protected`, `default` (no modifier), and `private`. Which of the following statements correctly describe the access levels granted by these modifiers? (Choose all that apply.)

A) A `public` class or member can be accessed by any other class in the same package or in any other package.
B) A `protected` member can be accessed by any class in its own package, but from outside the package, only by classes that extend the class containing the `protected` member.
C) A member with `default` (no modifier) access can be accessed by any class in the same package but not from a class in a different package.
D) A `private` member can be accessed only by methods that are members of the same class or within the same file.
E) A `protected` member can be accessed by any class in the Java program, regardless of package.

6. Which of the following class declarations correctly demonstrates the use of access modifiers, `class` keyword, and class naming conventions in Java?

A) `class public Vehicle { }`
B) `public class vehicle { }`
C) `Public class Vehicle { }`
D) `public class Vehicle { }`
E) `classVehicle public { }`

7. Consider the following code snippet:

```java
public class Counter {
    public static int COUNT = 0;

    public Counter() {
        COUNT++;
    }

    public static void resetCount() {
        COUNT = 0;
    }

    public int getCount() {
        return COUNT;
```

```
    }
}
```

Which of the following statements are true about `static` and instance members within the `Counter` class? (Choose all that apply.)

A) The `COUNT` variable can be accessed directly using the class name without creating an instance of `Counter`.
B) The `getCount()` method is an example of a static method because it returns the value of a static variable.
C) Every time a new instance of `Counter` is created, the `COUNT` variable is incremented.
D) The `resetCount()` method resets the `COUNT` variable to 0 for all instances of `Counter`.

8. Which of the following are valid field name identifiers in Java? (Choose all that apply.)

A) `int _age;`
B) `double 2ndValue;`
C) `boolean is_valid;`
D) `String $name;`
E) `char #char;`

9. Consider the syntax used to declare methods in a class. Which of the following method declarations is correct according to Java syntax rules?

A) `int public static final computeSum(int num1, int num2)`
B) `private void updateRecord(int id) throws IOException`
C) `synchronized boolean checkStatus [int status]`
D) `float calculateArea() {}`

10. Given the method declarations below, which of them have the same method signature?

A) `public void update(int id, String value)`
B) `private void update(int identifier, String data)`
C) `public boolean update(String value, int id)`
D) `void update(String value, int id)`
E) `protected void update(int id, int value) throws IOException`

11. Given this class:

```java
public class AccountManager {
    private void resetAccountPassword(String accountId) {
        // Implementation code here
    }

    void auditTrail(String accountId) {
        // Implementation code here
    }

    protected void notifyAccountChanges(String accountId) {
        // Implementation code here
    }

    public void updateAccountInformation(String accountId) {
        // Implementation code here
    }
}
```

Which of the following statements correctly describe the accessibility of the methods within the `AccountManager` class from a class in the same package and from a class in a different package?

A) The `resetAccountPassword` method can be accessed from any class within the same package but not from a class in a different package.

B) The `auditTrail` method can be accessed from any class within the same package and from subclasses in different packages.
C) The `notifyAccountChanges` method can be accessed from any class within the same package and from subclasses in different packages.
D) The `updateAccountInformation` method can be accessed from any class, regardless of its package.

12. What will be the output of this program?

```
public class TestPassByValue {
    public static void main(String[] args) {
        int originalValue = 10;
        TestPassByValue test = new TestPassByValue();
        System.out.println("Before calling changeValue: " + originalValue);
        test.changeValue(originalValue);
        System.out.println("After calling changeValue: " + originalValue);
    }

    public void changeValue(int value) {
        value = 20;
    }
}
```

A)

```
Before calling changeValue: 10
After calling changeValue: 20
```

B)

```
Before calling changeValue: 10
After calling changeValue: 10
```

C)

```
Before calling changeValue: 20
After calling changeValue: 20
```

D)

```
Before calling changeValue: 20
After calling changeValue: 10
```

13. What will be the output of the following program?

```
public class Test {
    public static void main(String[] args) {
        print(null);
    }

    public static void print(Object o) {
        System.out.println("Object");
    }

    public static void print(String s) {
        System.out.println("String");
    }
}
```

A) `Object`
B) `String`

C) Compilation fails
D) A runtime exception is thrown

14. Which of the following method declarations correctly uses varargs? Choose all that apply.

A) `public void print(String... messages, int count)`
B) `public void print(int count, String... messages)`
C) `public void print(String messages...)`
D) `public void print(String[]... messages)`
E) `public void print(String... messages, String lastMessage)`

15. Given the class `Vehicle`:

```
public class Vehicle {
    private String type;
    private int maxSpeed;

    public Vehicle(String type) {
        this.type = type;
    }

    public Vehicle(int maxSpeed) {
        this.maxSpeed = maxSpeed;
    }

    // Additional methods here
}
```

Which of the following statements is true regarding its constructors?

A) The class `Vehicle` demonstrates constructor overloading by having multiple constructors with different parameter lists.
B) The class `Vehicle` will compile with an error because it does not provide a default constructor.
C) It is possible to create an instance of `Vehicle` with both `type` and `maxSpeed` initialized.
D) Calling either constructor will initialize both `type` and `maxSpeed` fields of the `Vehicle` class.

16. Consider the following class with an instance initializer block:

```
public class Library {
    private int bookCount;
    private List<String> books;

    {
        books = new ArrayList<>();
        books.add("Book 1");
        books.add("Book 2");
        // Instance initializer block
    }

    public Library(int bookCount) {
        this.bookCount = bookCount + books.size();
    }

    public int getBookCount() {
        return bookCount;
    }
```

```
    // Additional methods here
}
```

Given the `Library` class above, which of the following statements accurately describe the role and effect of the instance initializer block?

A) The instance initializer block is executed before the constructor, initializing the `books` list and adding two books to it.
B) The instance initializer block replaces the need for a constructor in the `Library` class.
C) Instance initializer blocks cannot initialize instance variables like `books`.
D) If multiple instances of `Library` are created, the instance initializer block will execute each time before the constructor, ensuring the `books` list is initialized and populated for each object.

17. Consider the following Java class with a `static` initializer block:

```java
public class Configuration {
    private static Map<String, String> settings;

    static {
        settings = new HashMap<>();
        settings.put("url", "https://eherrera.net");
        settings.put("timeout", "30");
        // Static initializer block
    }

    public static String getSetting(String key) {
        return settings.get(key);
    }

    // Additional methods here
}
```

Given the `Configuration` class above, which of the following statements accurately describe the role and effect of the `static` initializer block?

A) The `static` initializer block is executed only once when the class is first loaded into memory, initializing the `settings` map with default values.
B) The `static` initializer block allows instance methods to modify the `settings` map without creating an instance of the `Configuration` class.
C) `static` initializer blocks are executed each time a new instance of the `Configuration` class is created.
D) The `static` initializer block is executed before any instance initializer blocks or constructors, when an instance of the class is created.

18. Consider the following class definition:

```java
public class InitializationOrder {
    static {
        System.out.println("1. Static initializer");
    }

    private static int staticValue = initializeStaticValue();

    private int instanceValue = initializeInstanceValue();

    {
        System.out.println("3. Instance initializer");
    }
```

```
    public InitializationOrder() {
        System.out.println("4. Constructor");
    }

    private static int initializeStaticValue() {
        System.out.println("2. Static value initializer");
        return 0;
    }

    private int initializeInstanceValue() {
        System.out.println("3. Instance value initializer");
        return 0;
    }

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

When the `main` method of the `InitializationOrder` class is executed, what is the correct order of execution for the initialization blocks, method calls, and constructor?

A) `1. Static initializer  2. Static value initializer  3. Instance initializer  3. Instance value initializer  4. Constructor`

B)
`1. Static initializer  2. Static value initializer  3. Instance value initializer  3. Instance initializer  4. Constructor`

C)
`1. Static initializer  3. Instance initializer  2. Static value initializer  3. Instance value initializer  4. Constructor`

D)
`2. Static value initializer  1. Static initializer  3. Instance value initializer  3. Instance initializer  4. Constructor`

19. Consider a class `CustomObject` that does not explicitly override any methods from `java.lang.Object`:

```
public class CustomObject {
    // Class implementation goes here
}
```

Which of the following statements correctly reflect the outcomes when methods from `java.lang.Object` are used with instances of `CustomObject`? (Choose all that apply.)

A) Invoking `toString()` on an instance of `CustomObject` will return a `String` that includes the class name followed by the `@` symbol and the object's hashcode.
B) Calling `equals(Object obj)` on two different instances of `CustomObject` that have identical content will return `true` because they are instances of the same class.
C) Using `hashCode()` on any instance of `CustomObject` will generate a unique integer that remains consistent across multiple invocations within the same execution of a program.
D) The `clone()` method can be used to create a shallow copy of an instance of `CustomObject` without the need for `CustomObject` to implement the `Cloneable` interface.

20. Consider the code snippet below that demonstrates the use of a static nested class:

```
public class OuterClass {
```

```
    private static String message = "Hello, World!";

    static class NestedClass {
        void printMessage() {
            // Note: A static nested class can access the static members of its outer class.
            System.out.println(message);
        }
    }

    public static void main(String[] args) {
        OuterClass.NestedClass nested = new OuterClass.NestedClass();
        nested.printMessage();
    }
}
```

Which of the following statements is true regarding static nested classes in Java?

A) A static nested class can access both static and non-static members of its enclosing class directly.
B) Instances of a static nested class can exist without an instance of its enclosing class.
C) A static nested class can only be instantiated within the static method of its enclosing class.
D) Static nested classes are not considered members of their enclosing class and cannot access any members of the enclosing class.

21. Consider the following code snippet that demonstrates the use of a non-static nested (inner) class:

```
public class OuterClass {
    private String message = "Hello, World!";

    class InnerClass {
        void printMessage() {
            System.out.println(message);
        }
    }

    public static void main(String[] args) {
        OuterClass outer = new OuterClass();
        OuterClass.InnerClass inner = outer.new InnerClass();
        inner.printMessage();
    }
}
```

Which of the following statements is true regarding non-static nested (inner) classes in Java?

A) A non-static nested class can directly access both static and non-static members of its enclosing class.
B) Instances of a non-static nested class can exist independently of an instance of its enclosing class.
C) A non-static nested class cannot access the non-static members of its enclosing class directly.
D) Non-static nested classes must be declared static to access the static members of their enclosing class.

22. Consider the following code snippet demonstrating the use of a local class within a method:

```
public class LocalClassExample {
    public void printEvenNumbers(int[] numbers, int max) {
        class EvenNumberPrinter {
            public void print() {
                for (int number : numbers) {
                    if (number % 2 == 0 && number <= max) {
                        System.out.println(number);
```

```
                    }
                }
            }
        }
        EvenNumberPrinter printer = new EvenNumberPrinter();
        printer.print();
    }

    public static void main(String[] args) {
        LocalClassExample example = new LocalClassExample();
        int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        example.printEvenNumbers(numbers, 6);
    }
}
```

Which of the following statements correctly describe local classes in Java, based on the example provided?

A) Local classes can be declared within any block that precedes a statement.
B) Instances of a local class can be created and used outside of the block where the local class is defined.
C) Local classes are a type of static nested class and can access both static and non-static members of the enclosing class directly.
D) Local classes can access local variables and parameters of the enclosing block only if they are declared `final` or effectively final.

23. Consider the following Java code snippet demonstrating the use of an anonymous class:

```
public class HelloWorld {
    interface HelloWorldInterface {
        void greet();
    }

    public void sayHello() {
        HelloWorldInterface myGreeting = new HelloWorldInterface() {
            @Override
            public void greet() {
                System.out.println("Hello, world!");
            }
        };
        myGreeting.greet();
    }

    public static void main(String[] args) {
        new HelloWorld().sayHello();
    }
}
```

Which of the following statements is true about anonymous classes in Java?

A) Anonymous classes can implement interfaces and extend classes without the need to declare a named class.
B) An anonymous class must override all methods in the superclass or interface it declares it is implementing or extending.
C) Anonymous classes can have constructors as named classes do.
D) Instances of anonymous classes cannot be passed as arguments to methods.

24. Which of the following statements accurately reflects a valid rule regarding how classes and source files are organized?

A) A source file can contain multiple public classes.
B) Private classes can be declared at the top level in a source file.
C) A `public` class must be declared in a source file that has the same name as the class.
D) If a source file contains more than one class, none of the classes can be `public`.

Chapter ONE

Utilizing Java Object-Oriented Approach - Part 1

Answers

1. The correct answer is B.

Explanation:

- **A)** Both `sb1` and `sb2` are eligible for garbage collection.
  - This option is incorrect because `sb2` still holds a reference to the `StringBuilder` object it was initially assigned. Therefore, it is not eligible for garbage collection.
- **B)** Only the `StringBuilder` object initially referenced by `sb1` is eligible for garbage collection.
  - This option is correct. After `sb1` is reassigned to reference the same object as `sb2`, the original `StringBuilder` object created with `new StringBuilder("Java")` and initially referenced by `sb1` is no longer accessible. Since there are no references pointing to it, it becomes eligible for garbage collection.
- **C)** Only the `StringBuilder` object initially referenced by `sb2` is eligible for garbage collection.
  - This option is incorrect because after the assignment `sb1 = sb2;`, both `sb1` and `sb2` reference the same object (`new StringBuilder("Python")`). This object is still accessible through `sb2` (and now `sb1` as well), so it is not eligible for garbage collection.
- **D)** Neither of the `StringBuilder` objects are eligible for garbage collection.
  - This option is incorrect because, as explained, the object initially referenced by `sb1` becomes eligible for garbage collection after `sb1` is reassigned to `sb2`.

2. The correct answers are C and D.

Explanation:

- **A)** `implement` is incorrect. The correct keyword for implementing an interface in Java is `implements`.
- **B)** `array` is incorrect. Java does not have a reserved keyword named `array`. Arrays are declared with square brackets `[ ]`.
- **C)** `volatile` is correct. `volatile` is a reserved keyword that is used to indicate that a variable's value will be modified by different threads.
- **D)** `extends` is correct. `extends` is a reserved keyword used in class declarations to inherit from a superclass.

3. The correct answers are A and E.

Explanation:

- **A)** Line 1 is an example of a single-line comment.
  - This option is correct. Line 1 uses `//` to start a single-line comment, which is a common way to add notes or explain a part of code that does not affect the execution.
- **B)** Lines 3-7 demonstrate the use of a javadoc comment.
  - This option is incorrect. Lines 3-7 use a block comment. Javadoc comments start with `/**` and end with `*/`.
- **C)** Line 9 uses a javadoc comment to explain the `add` method.

  - This option is incorrect. Line 9 is a single-line comment, not a javadoc comment. Javadoc comments in Java are defined with `/**` at the beginning and `*/` at the end, and are specifically used to describe classes, methods, and fields.
- **D)** Line 12 uses a special `TODO` comment, different from a single-line comment.
  - This option is incorrect. Line 12 uses a `TODO` comment, which is a convention many developers follow to mark parts of the code that require further development or attention but is still a single-line comment.
- **E)** Lines 3-7 is a block comment that is used as if it were a javadoc comment.
  - This option is correct. Lines 3-7 use a block comment, which is not processed by javadoc tools and therefore not suitable for generating official documentation.

4. The correct answers are B and D.

Explanation:

- **A)** The `import` statement in `Application.java` is unnecessary because both classes are in the same directory.
  - This option is incorrect. In Java, the `import` statement is used to bring a class or an entire package into visibility, and its necessity is determined by the package membership of the classes, not their directory location. Even if classes are in the same directory, if they belong to different packages, the `import` statement is required to use one in the other.
- **B)** The `import` statement in `Application.java` is necessary for using the `Calculator` class because they belong to different packages.
  - This is the correct answer. The `Calculator` class is in the `math` package, and the `Application` class is in the `app` package. Despite being in the same directory, the different packages require an `import` statement to use `Calculator` in `Application`.
- **C)** The `Calculator` class will not be accessible in `Application.java` due to being in a different directory.
  - This option is incorrect. Java's access control is not based on the directory structure but on the `package` and `import` declarations. As long as the classes are correctly packaged and imported, they can be accessed across different directories.
- **D)** Removing the `package` statement from both files will allow `Application.java` to use `Calculator` without an `import` statement, regardless of directory structure.
  - This option is correct. Removing the `package` statement from both files will place them in the default package, and they will be able to access each other without an `import` statement. However, this is not recommended for anything beyond very simple or temporary code due to namespace management and readability concerns.

5. The correct answers are A, B, and C.

Explanation:

- **A)** A `public` class or member can be accessed by any other class in the same package or in any other package.
  - This is correct. The `public` modifier grants the highest level of access. A `public` class or member is accessible from any other class, regardless of the packages they belong to.
- **B)** A `protected` member can be accessed by any class in its own package, but from outside the package, only by classes that extend the class containing the protected member.
  - This is correct. The `protected` access level allows a member to be accessed within its own package and by subclasses in any package. It offers a more restrictive level of access than `public`.
- **C)** A member with `default` (no modifier) access can be accessed by any class in the same package but not from a class in a different package.
  - This is correct. If no access modifier (also known as `default` access level) is specified, the member is accessible only within classes in the same package. This is more restrictive than `protected` and `public`.
- **D)** A `private` member can be accessed only by methods that are members of the same class or within the same file.

  - This option is incorrect because `private` members can be accessed only within the same class. It's not about being within the same file, as Java allows only one public top-level class per file.
- **E)** A `protected` member can be accessed by any class in the Java program, regardless of package.
  - This is incorrect. `Protected` access does not grant universal access across all classes in a program. Access from outside the package is limited to subclasses only.

6. The correct answer is D:

Explanation:

- **A)** `class public Vehicle { }`
  - This option is incorrect because the syntax is wrong. The correct order is the access modifier followed by the `class` keyword, and then the class name.
- **B)** `public class vehicle { }`
  - This option is incorrect mainly due to the class naming convention. In Java, class names should start with an uppercase letter, so `vehicle` should be `Vehicle`.
- **C)** `Public class Vehicle { }`
  - This option is incorrect because `Public` is incorrectly capitalized. Java is case-sensitive, and the correct keyword is `public`.
- **D)** `public class Vehicle { }`
  - This is the correct answer. The syntax follows the proper order: the access modifier (`public`), followed by the `class` keyword, and then the class name (`Vehicle`), which correctly starts with an uppercase letter as per Java naming conventions.
- **E)** `classVehicle public { }`
  - This option is incorrect due to several reasons: the syntax order is wrong, there is no space between `class` and the class name, and the access modifier's position is incorrect.

7. The correct answers are A, C, and D.

Explanation:

- **A)** The `COUNT` variable can be accessed directly using the class name without creating an instance of `Counter`.
  - This option is correct. Static variables belong to the class and can be accessed directly with the class name, such as `Counter.COUNT`, without needing to instantiate the class.
- **B)** The `getCount()` method is an example of a static method because it returns the value of a static variable.
  - This option is incorrect. Although `getCount()` returns a static variable's value, it is not defined as a static method. Static methods are declared using the `static` modifier. The method's instance or non-static nature does not change based on the variables it accesses or returns.
- **C)** Every time a new instance of `Counter` is created, the `COUNT` variable is incremented.
  - This option is correct. The constructor increments the `COUNT` variable by 1 each time a new instance of `Counter` is created, demonstrating the shared nature of static variables across all instances.
- **D)** The `resetCount()` method resets the `COUNT` variable to 0 for all instances of `Counter`.
  - This option is correct. The `resetCount()` static method sets the `COUNT` variable to zero. Since `COUNT` is static, this change affects all instances of the class, as there is only one `COUNT` variable shared among them.

8. The correct answers are A, C, and D.

Explanation:

- **A)** `int _age;` is correct. Identifiers in Java can begin with a letter, an underscore (_), or a dollar sign ($). Therefore, `_age` is a valid identifier.
- **B)** `double 2ndValue;` is incorrect. Identifiers cannot start with a digit. The correct format would be to start with a letter or a non-digit character such as an underscore or a dollar sign.

- **C)** `boolean is_valid;` is correct. Similar to `_age`, `is_valid` is a valid identifier because it starts with a letter and can contain underscores.
- **D)** `String $name;` is correct. Identifiers can also start with a dollar sign (), *making*`name‘ a valid identifier.
- **E)** `char #char;` is incorrect. The hash (#) character is not allowed as a starting character in identifiers. Identifiers can only start with letters, `$`, or `_`.

9. The correct answer is B.

Explanation:

- **A)** `int public static final computeSum(int num1, int num2)` is incorrect because the return type in method declarations goes right before the name of the method, not at the beginning.
- **B)** `private void updateRecord(int id) throws IOException` is correct. This method declaration is syntactically correct in Java. It uses the `private` access modifier, specifies a return type (`void`), includes an exception (`IOException`) that this method might throw, and correctly defines the parameter list.
- **C)** `synchronized boolean checkStatus [int status]` Correct syntax requires parentheses for the parameter list, even when there are no parameters, making the correct declaration `synchronized boolean checkStatus(int status)`.
- **D)** `float calculateArea() {}` is incorrect because a method that returns `float` cannot have an empty method body.

10. The correct answers are A and B.

Explanation:

In Java, a method signature consists of the method name and the parameter list. The return type, access modifier, and exception list are not considered part of the method signature.

- **A)** (`public void update(int id, String value)`)
- **B)** (`private void update(int identifier, String data)`)
  - The above options have the same method signature (`update(int, String)`) because they both have the same method name and parameter list (an `int` and a `String`, in that order). The difference in parameter names (`id` vs. `identifier` and `value` vs. `data`) does not affect the method signature.
- **C)** `public boolean update(String value, int id)`
  - This option does not have the same signature as A or B because the parameter list is different.
- **D)** `void update(String value, int id)`
  - This option has a different method signature (`update(String, int)`) because the order of the parameters is reversed compared to A and B.
- **E)** `protected void update(int id, int value) throws IOException`
  - This option also has a different parameter list (`update(int, int)`).

11. The correct answers are C and D.

Explanation:

- **A)** The `resetAccountPassword` method can be accessed from any class within the same package but not from a class in a different package.
  - This option is incorrect. The `resetAccountPassword` method has `private` access, which means it is accessible only within the `AccountManager` class itself, not from any class, even within the same package. The initial statement was slightly incorrect in suggesting package-level access for a `private` method.
- **B)** The `auditTrail` method can be accessed from any class within the same package and from subclasses in different packages.

  - This option is incorrect because the `auditTrail` method has package-private access (no access modifier), which means it is accessible from any class within the same package but not from subclasses in different packages unless they are also within the same package.
- **C)** The `notifyAccountChanges` method can be accessed from any class within the same package and from subclasses in different packages.
  - This option is correct. The `notifyAccountChanges` method has `protected` access, meaning it can be accessed within the same package and by subclasses, even if the subclasses are in different packages.
- **D)** The `updateAccountInformation` method can be accessed from any class, regardless of its package.
  - This option is correct. The `updateAccountInformation` method is `public`, so it can be accessed from any class, regardless of the package it belongs to.

12. The correct answer is B.

Explanation:

Java is strictly pass-by-value. This means that when passing a variable to a method, Java passes a copy of the variable's value, not the variable itself. Changes to the parameter inside the method do not affect the original variable.

- **A)**

```
Before calling changeValue: 10
After calling changeValue: 20
```

- This option is incorrect because, although the `changeValue` method changes the `value` parameter to 20, this change does not affect the original variable `originalValue` outside the method. The change to `value` is made on its copy, not on `originalValue` itself.

- **B)**

```
Before calling changeValue: 10
After calling changeValue: 10
```

- This is the correct answer. `originalValue` is passed by value to the `changeValue` method. Thus, modifications to `value` inside `changeValue` do not affect `originalValue`. The output confirms that `originalValue` remains unchanged after the method call.

- **C)**

```
Before calling changeValue: 20
After calling changeValue: 20
```

- **D)**

```
Before calling changeValue: 20
After calling changeValue: 10
```

- These options are incorrect as they suggest changes to the method parameters can affect the original variables, which is not how Java's pass-by-value semantics work.

13. The correct answer is B.

Explanation:

- **A)** `Object`
  - This option is incorrect because Java uses the most specific method that is applicable to the parameters. In this case, `String` is more specific than `Object`, so the `print(String s)` method is called.
- **B)** `String`

  - This option is correct. Even though `null` can be assigned to any reference type, Java prefers the most specific method applicable to the method parameters. Since `String` is a more specific type than `Object`, the `print(String s)` method is chosen over the `print(Object o)` method.
- **C)** Compilation fails
  - Compilation does not fail because both `print` methods are correctly defined and can potentially match the call `print(null)`. Java's method overloading mechanism allows this to compile without any issues.
- **D)** A runtime exception is thrown
  - No runtime exception is thrown because the method call to `print` successfully resolves to the `print(String s)` method at compile time. Since the method is correctly invoked, and there is no other code that could cause a runtime exception, this program runs successfully.

14. The correct answers are B and D.

Explanation:

- **A)** `public void print(String... messages, int count)`
  - This option is incorrect because varargs (variable arguments) must be the last parameter in a method's parameter list. Having `int count` after `String... messages` violates this rule.
- **B)** `public void print(int count, String... messages)`
  - This option is correct. It correctly places the varargs parameter `String... messages` at the end of the method's parameter list, which is the required syntax for using varargs.
- **C)** `public void print(String messages...)`
  - This option is incorrect because the syntax `String messages...` is invalid. The correct syntax for varargs is to place the ellipsis (`...`) after the type and before the variable name, like `String... messages`.
- **D)** `public void print(String[]... messages)`
  - This option is correct. It demonstrates the use of varargs with an array type, which is allowed. Here, each argument passed to `messages` can itself be an array of `String`, and `messages` will be treated as an array of arrays (`String[][]`).
- **E)** `public void print(String... messages, String lastMessage)`
  - This option is incorrect, similar to option A, because varargs must be the last parameter in the method's parameter list. Having another parameter after the varargs parameter is not allowed.

15. The correct answer is A.

Explanation:

- **A)** The class `Vehicle` demonstrates constructor overloading by having multiple constructors with different parameter lists.
  - This option is correct. Constructor overloading in Java is a technique of having more than one constructor with different parameter lists in the same class. It allows objects of the class to be initialized in different ways. The `Vehicle` class has two constructors, one that takes a `String` (for the vehicle type) and another that takes an `int` (for the max speed), which is a perfect example of constructor overloading.
- **B)** The class `Vehicle` will compile with an error because it does not provide a default constructor.
  - This option is incorrect. Java does not require an explicit default constructor if the class provides any other constructors. The absence of a default constructor (one that takes no arguments) is not a compilation error; it simply means that the programmer cannot instantiate the class using a no-argument constructor unless it's explicitly defined.
- **C)** It is possible to create an instance of `Vehicle` with both `type` and `maxSpeed` initialized.
  - This option is incorrect because with the current constructors, it's not possible to create a `Vehicle` instance with both `type` and `maxSpeed` initialized through a single constructor call.
- **D)** Calling either constructor will initialize both `type` and `maxSpeed` fields of the `Vehicle` class.
  - This option is incorrect. Calling either constructor only initializes the parameter that is provided to it. The first constructor initializes the `type`, and the second initializes the `maxSpeed`. Without additional code, such as a constructor that accepts both parameters or setter methods, there's no

way for either constructor alone to initialize both fields.

16. The correct answer are A and D.

Explanation:

- **A)** The instance initializer block is executed before the constructor, initializing the `books` list and adding two books to it.
  - This option is correct. The instance initializer block is executed each time an instance of the class is created, before the constructor code runs. It initializes the `books` list and adds two books to it.
- **B)** The instance initializer block replaces the need for a constructor in the `Library` class.
  - This option is incorrect. The instance initializer block does not replace the need for a constructor. It is used in addition to constructors, often to initialize common parts of various constructors in a class.
- **C)** Instance initializer blocks cannot initialize instance variables like `books`.
  - This option is incorrect. Instance initializer blocks can indeed initialize instance variables. In this case, the `books` list is an instance variable that is being initialized and populated within the instance initializer block.
- **D)** If multiple instances of `Library` are created, the instance initializer block will execute each time before the constructor, ensuring the `books` list is initialized and populated for each object.
  - This option is correct. For each new instance of the `Library` class, the instance initializer block runs before the constructor is invoked. This ensures that the `books` list is initialized and populated with `"Book 1"` and `"Book 2"` for every `Library` object created.

17. The correct answer is A.

Explanation:

- **A)** The `static` initializer block is executed only once when the class is first loaded into memory, initializing the `settings` map with default values.
  - This option is correct. Static initializer blocks are executed a single time, when the class is first loaded into the JVM memory. In this case, it initializes the `settings` map with default configuration values.
- **B)** The `static` initializer block allows instance methods to modify the `settings` map without creating an instance of the `Configuration` class.
  - This option is misleading. While static methods like `getSetting` can access and modify static fields like `settings` without needing an instance of the class, this capability is not due to the static initializer block itself but rather the nature of static fields and methods.
- **C)** `static` initializer blocks are executed each time a new instance of the `Configuration` class is created.
  - This option is incorrect. Static initializer blocks are not executed each time a new instance of the class is created. They are executed only once: when the class is first loaded.
- **D)** The `static` initializer block is executed before any instance initializer blocks or constructors, when an instance of the class is created.
  - This statement is partially correct in that static initializer blocks are executed before any instance initializer blocks or constructors, but it's misleading as it implies a sequence with instance creation. The key point is that static initializer blocks run once upon class loading, irrespective of the creation of any instances.

18. The correct answer is B.

Explanation:

In Java, the order of initialization when a class is loaded and an instance of that class is created is as follows:

1. **Static fields and static initializers** are processed in the order they appear in the class definition. First, the static initializer block prints `"1. Static initializer"`. Then, the static field `staticValue` is initialized by calling `initializeStaticValue()`, which prints `"2. Static value initializer".`
2. **Instance fields and instance initializers** are processed in the order they appear when an instance

of the class is created. First, the instance field `instanceValue` is initialized by calling `initializeInstanceValue()`, which prints `"3. Instance value initializer"`. Then, the instance initializer block prints `"3. Instance initializer"`.

3. **Constructors** are executed after all fields and instance initializers have been processed. The constructor in this case prints `"4. Constructor"`.

The numbering of the output for `"3. Instance initializer"` and `"3. Instance value initializer"` in the question might seem to suggest they are executed simultaneously or out of order, but it's important to remember that instance fields and instance initializers execute in the order they appear in the class, before the constructor is executed. The duplicate numbering means that instance field initializers run first, followed by instance initializers, and finally, the constructor runs.

- **A)**

```
1. Static initializer
2. Static value initializer
3. Instance initializer
3. Instance value initializer
4. Constructor
```

- This option is incorrect.
- **B)**

```
1. Static initializer
2. Static value initializer
3. Instance value initializer
3. Instance initializer
4. Constructor
```

- This option is correct.
- **C)**

```
1. Static initializer
3. Instance initializer
2. Static value initializer
3. Instance value initializer
4. Constructor
```

- This option is incorrect.
- **D)**

```
2. Static value initializer
1. Static initializer
3. Instance value initializer
3. Instance initializer
4. Constructor
```

- This option is incorrect.

19. The correct answers are A and C.

Explanation:

- **A)** Invoking `toString()` on an instance of `CustomObject` will return a `String` that includes the class name followed by the `@` symbol and the object's hashcode.
  - This option is correct. The `toString()` method in `java.lang.Object` returns a string that includes the class name, the `@` symbol, and the object's hashcode in hexadecimal. If `CustomObject` does not override `toString()`, this default format is used.

- **B)** Calling `equals(Object obj)` on two different instances of `CustomObject` that have identical content will return `true` because they are instances of the same class.
  - This option is incorrect. The default implementation of `equals(Object obj)` in `java.lang.Object` checks for reference equality, meaning it returns `true` only if both references point to the exact same object. Without overriding `equals`, two different instances of `CustomObject`, even with identical content, would not be considered equal.
- **C)** Using `hashCode()` on any instance of `CustomObject` will generate a unique integer that remains consistent across multiple invocations within the same execution of a program.
  - This option is correct. The `hashCode()` method is designed to return an integer representation of the object's memory address or a value derived from it. While the exact implementation is not specified and can vary, it is consistent during the execution of a program for any given object.
- **D)** The `clone()` method can be used to create a shallow copy of an instance of `CustomObject` without the need for `CustomObject` to implement the `Cloneable` interface.
  - This option is incorrect. The `clone()` method in `java.lang.Object` is protected, and it throws a `CloneNotSupportedException` unless the class implements the `Cloneable` interface. Without `CustomObject` explicitly implementing `Cloneable` and overriding `clone()` to make it `public`, it cannot be used to clone instances of `CustomObject`.

20. The correct answer is B.

Explanation:

- **A)** A static nested class can access both static and non-static members of its enclosing class directly.
  - This option is incorrect because a static nested class cannot directly access non-static members of its enclosing class. It can only access static members directly.
- **B)** Instances of a static nested class can exist without an instance of its enclosing class.
  - This is the correct answer. A static nested class is associated with its outer class, and unlike inner classes, it does not need an instance of the outer class to be instantiated. This makes it useful for grouping classes that will be used in a static context.
- **C)** A static nested class can only be instantiated within the static method of its enclosing class.
  - This option is incorrect. A static nested class can be instantiated from any context (static or non-static) as long as it is accessible (i.e., visibility allows it).
- **D)** Static nested classes are not considered members of their enclosing class and cannot access any members of the enclosing class.
  - This option is incorrect. Static nested classes are indeed considered members of their enclosing class and can access its static members and static methods. However, they do not have access to non-static members of the enclosing class unless they instantiate the enclosing class.

21. The correct answer is A.

Explanation:

- **A)** A non-static nested class can directly access both static and non-static members of its enclosing clas
  - This option is correct. A non-static nested class, or inner class, has access to all members (including both static and non-static) of its enclosing class, as demonstrated in the code snippet where `InnerClass` accesses the non-static `message` field of `OuterClass`.
- **B)** Instances of a non-static nested class can exist independently of an instance of its enclosing class.
  - This option is incorrect. Instances of a non-static nested class (inner class) are implicitly associated with an instance of the enclosing class. Therefore, they cannot exist independently of an instance of the enclosing class. In the provided code snippet, the `InnerClass` instance is created through an instance of `OuterClass`.
- **C)** A non-static nested class cannot access the non-static members of its enclosing class directly.
  - This option is incorrect. As stated above, an inner class can directly access both static and non-static members of its enclosing class.
- **D)** Non-static nested classes must be declared static to access the static members of their enclosing class.

  - This option is incorrect. Non-static nested classes (inner classes) are designed to access members of their enclosing class directly without needing to be declared static. Declaring a nested class as static changes its type to a static nested class, which has different access properties from an inner class.

22. The correct answers are A and D.

Explanation:

- **A)** Local classes can be declared within any block that precedes a statement.
  - This option is correct. Local classes in Java can indeed be declared within any block that precedes a statement, such as a method body, a `for` loop, or an `if` statement.
- **B)** Instances of a local class can be created and used outside of the block where the local class is defined.
  - This option is incorrect. Instances of local classes cannot be created and used outside the block where they are defined. Their scope is limited to the block in which they are declared.
- **C)** Local classes are a type of static nested class and can access both static and non-static members of the enclosing class directly.
  - This option is incorrect. Local classes are not static; they are associated with an instance of the enclosing class and have access to its instance members. They do not have the static context that static nested classes have, and thus they can access both static and non-static members of the enclosing class.
- **D)** Local classes can access local variables and parameters of the enclosing block only if they are declared `final` or effectively final.
  - This is correct. Local classes can access local variables and parameters of the method (or any enclosing block) in which they are defined, but those variables must be declared `final` or effectively final (which means their values do not change after they are initialized).

23. The correct answer is A.

Explanation:

- **A)** Anonymous classes can implement interfaces and extend classes without the need to declare a named class.
  - This option is correct. Anonymous classes are a way to extend existing classes or implement interfaces on the spot without the need for a formal class declaration. This makes them useful for creating quick, one-off implementations.
- **B)** An anonymous class must override all methods in the superclass or interface it declares it is implementing or extending.
  - This option is incorrect. An anonymous class only needs to override abstract methods of the superclass or interface it extends or implements. If the superclass or interface has no abstract methods, then the anonymous class does not need to override any methods.
- **C)** Anonymous classes can have constructors as named classes do.
  - This option is incorrect. Anonymous classes do not have named constructors because they do not have names themselves. Instead, any initialization is done through an instance initializer block.
- **D)** Instances of anonymous classes cannot be passed as arguments to methods.
  - This option is incorrect. Instances of anonymous classes can indeed be passed as arguments to methods. They are useful for creating on-the-fly implementations for interfaces or subclasses that are required for a method call.

24. The correct answer is C.

Explanation:

- **A)** A source file can contain multiple public classes.
  - This option is incorrect. A Java source file cannot contain more than one `public` class. If a class is declared `public`, it must be the only `public` class in the file, and the file name must match the class name.

- **B)** Private classes can be declared at the top level in a source file.
  - This option is incorrect. Java does not allow classes to be declared as `private` at the top level. Only `public`, or package-private (no access modifier) classes can be defined at the top level. Inner classes can be `private`.
- **C)** A `public` class must be declared in a source file that has the same name as the class.
  - This is correct. According to Java's rules, if a class is declared `public`, the source file in which it is defined must have the same name as the class, followed by the `.java` extension. This is a strict rule that helps the Java compiler easily locate source files.
- **D)** If a source file contains more than one class, none of the classes can be `public`.
  - This is incorrect. While it is true that if a source file contains a `public` class, the source file must be named after that `public` class, it is not true that none of the classes can be `public` if a source file contains more than one class. A source file can contain multiple classes, but only one of them can be `public`, and the source file must be named after that `public` class. The statement could imply that multiple non-public top-level classes are a common scenario without the context of the `public` class naming rule.

Chapter TWO

Utilizing Java Object-Oriented Approach - Part 2

Chapter Content

Variables

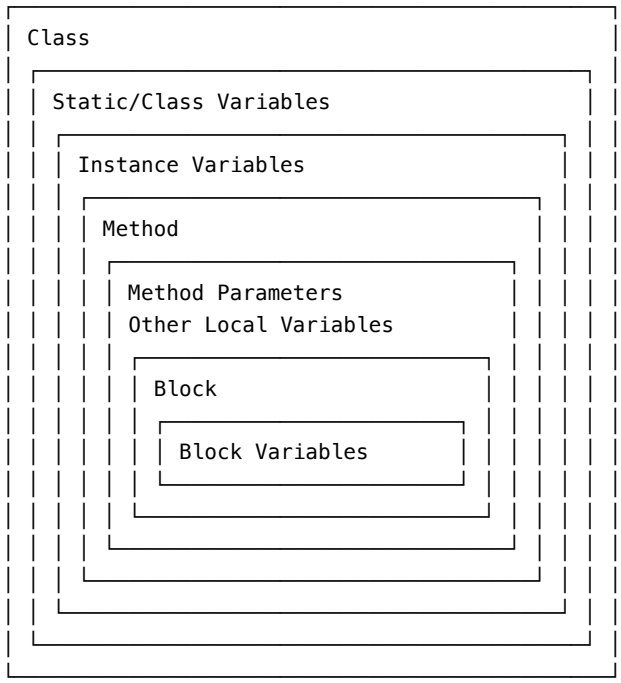
Variable Scopes

We can think of a variable's scope as its visibility, where it can be seen and accessed in our code. Properly managing variable scope helps us write cleaner, more maintainable code and avoid bugs related to accessing variables in the wrong context.

At the highest level, a variable's scope is determined by where it is declared. In Java, there are five main

scopes to be aware of: - Block variables - Local variables - Method parameters - Fields (instance variables) - Class variables (static fields)

Here's a diagram to visualize it:



Local variables are declared inside the method where they are defined, while block variables and are only accessible within the block where they are defined. They come into scope at their declaration and go out of scope at the end of the enclosing method/block:

```
void myMethod() {
    int x = 1;
    if (x > 0) {
        int y = 2;
        System.out.println(x + y); // x and y both in scope here
    }
    System.out.println(x); // Only x is in scope here
    System.out.println(y); // Compile error! y is out of scope
}
```

As you can see, `y` is only visible within the `if` block where it was declared. Attempting to access it outside that block results in a compile error.

If you declare a variable inside a loop, you can't access it outside the loop. Even if it's all in the same method, the scope still ends at the loop's closing `}`. For example:

```
void myLoopingMethod() {
    for (int i = 0; i < 10; i++) {
        System.out.println(i);
    }
    System.out.println(i); // Compile error! i is out of scope
}
```

Similarly, variables declared in a for-loop initializer, such as `int i` above, are scoped only to the loop body, not the entire enclosing method.

This concept applies to other blocks like `if/else` too. A variable declared inside an `if` is not visible in the corresponding `else`:

```
void myIfElseMethod(int x) {
    if (x > 0) {
        int y = 1;
    } else {
        System.out.println(y); // Compile error! y not in scope
    }
}
```

Then we have method parameters. These are also considered local variables, but with a scope that covers the entire method body. They come into scope when the method is called and go out of scope when the method completes.

Parameters are local to the method, no other methods can see them, even if the method is currently executing:

```
void methodA(int x) {
    methodB();
    System.out.println(x); // x is in scope
}

void methodB() {
    System.out.println(x); // Compile error! x is not in scope
}
```

Fields, or instance variables, are variables declared at the class level, outside any method. They come into scope when the object is instantiated and remain in scope as long as the object is in memory:

```
class MyClass {
    private int x; // Instance variable (field)

    void myMethod() {
        System.out.println(x); // x is in scope here
    }
}
```

Since instance variables belong to an object instance, they cannot be accessed from static contexts, but they can be accessed by any instance method in the class.

A common misconception is that instance variables are garbage collected as soon as the method that uses them finishes, which is not the case. An object's fields stay in memory until the object itself is eligible for garbage collection, which may be long after a particular method call completes.

Also, remember that if the variable or its class is declared `private`, then only the declaring class can access it. But if they have `public`, `protected`, or default (package) access, other classes can potentially access them too.

Finally, class variables, or static fields, are `static` variables declared at the class level. They come into scope when the class is loaded and stay in scope until the program ends. There is only one copy of a class variable shared across all instances of the class.

Class variables belong to the class itself, not a specific object instance. And unlike instance variables, class variables can be accessed from both static and instance contexts:

```
class MyClass {
    private static int x; // Class variable

    void myMethod() {
        System.out.println(x); // x is in scope
```

```
    }

    static void myStaticMethod() {
        System.out.println(x); // x is also in scope
    }
}
```

Class variables can be accessed from anywhere in your program, even without creating an instance of the class. But they are still subject to access controls like `private` and `public`.

An interesting case is when you have two variables with the same name but different scopes:

```
class MyClass {
    private int x; // Instance variable

    void myMethod() {
        int x = 1; // Local variable
        System.out.println(x); // Prints 1 (local variable)
        System.out.println(this.x); // Prints 0 (instance variable)
    }
}
```

In this situation, the local variable shadows the instance variable within its scope. To access the instance variable, we have to use the `this` keyword. We'll talk about this later in the chapter, but, as you can see, properly limiting scope is not about improving performance, but about organizing our code and controlling access to variables.

Variable Declarations

When you start learning Java, it's easy to think that fields and local variables are pretty much the same thing. After all, they're both just variables, right? You declare them, give them a type and a name, maybe assign them a value, and then use them in your code. What's the big deal?

Well, as it turns out, there are some pretty important differences between fields and local variables in Java.

Fields are declared directly inside a class, but outside any method or constructor. They're part of the class's state, and each instance of the class gets its own copy of these fields.

Local variables, on the other hand, are declared inside a method or constructor. They only exist for the duration of that method or constructor call, and they're not accessible from outside. Once the method has finished executing, the local variables disappear.

Here's an example:

```
public class MyClass {
    private int myField; // This is a field

    public void myMethod() {
        int myLocalVar = 25; // This is a local variable
        // Do something with myLocalVar...
    } // myLocalVar no longer exists after this point
}
```

Now, you might be thinking, "Okay, so fields are in the class, and local variables are in methods. But can't I just use them interchangeably otherwise?" Well, not quite. There are some key differences in how they behave.

For one thing, fields automatically get default values if you don't explicitly initialize them. For numeric types (like `int`, `long`, `float`, `double`) the default is `0`. For `boolean`, it's `false`. For reference types (like `String` or any object), it's `null`.

On the other hand, local variables don't get any default values. If you try to use a local variable before initializing it, you'll get a compile error. In other words, the Java compiler wants you to be explicit about your intentions with local variables:

```
public void myMethod() {
    int uninitialized;
    System.out.println(uninitialized); // Compile error!
}
```

So Java requires you to initialize a local variable before you use it. But when exactly do you need to do this initialization? The rule is simple: the initialization must happen on every possible execution path before the first use of the variable:

```
int myVar;
if (someCondition) {
    myVar = 1;
} else {
    myVar = 2;
}
System.out.println(myVar); // This is fine

int myOtherVar;
if (someCondition) {
    myOtherVar = 1;
}
System.out.println(myOtherVar); // Compile error! Not initialized on the else path.
```

In the first example, `myVar` is guaranteed to be initialized before it's used, regardless of which path the `if/else` takes. But in the second example, if `someCondition` is `false`, `myOtherVar` will not be initialized before its first use, hence the compile error.

In any case, fields or local variables, Java lets you declare several variables of the same type in a single line, separated by commas:

```
int a, b, c;
```

But this doesn't mean that these variables share the same value. They're completely independent variables that just happen to be declared together. You can assign them different values:

```
int a = 1, b = 2, c = 3;
```

In fact, you don't have to assign them all values right away. It's totally fine to do this:

```
int a, b, c;
a = 1;
b = 2;
// c remains uninitialized for now
```

Just remember that you can't use `c` until you initialize it with a value, or you'll get a compile error.

Now, what about when you want to declare multiple variables of different types? Well, you can't do that in a single line like you can with variables of the same type. You'll have to declare each one separately:

```
int a = 1;
String b = "hello";
// This won't compile: int a = 1, String b = "hello";
```

Another difference between local variables and fields is in how you use `final`. Marking a field as `final` means it must be initialized when the object is constructed, and then it can never be changed again. With a local variable, `final` just means you can only assign it a value once. But that assignment doesn't have to happen when the variable is declared:

```
public class MyClass {
    private final int myFinalField = 42; // Must initialize here

    public void myMethod(int arg) {
        final int myFinalVar; // Okay to initialize later
        if (arg > 0) {
            myFinalVar = arg;
        } else {
            myFinalVar = 0;
        }
        // Can't assign to myFinalVar again after this point
    }
}
```

The assignment must occur before the variable's first use, and can only happen once. This is often useful when you want to assign a value conditionally, like in the above example. Or when you want to assign a value in a loop but ensure it doesn't change after the loop:

```
final int myFinalVar;
for (int i = 0; i < 10; i++) {
    // Some calculation...
    myFinalVar = result;
    // Can't assign to myFinalVar again after this point
}
```

However, when working with references and objects, if you make a local variable `final`, you can change the properties of the object it references. `final` only prevents you from assigning a new value to the variable itself. If the variable is a reference to an object, you can still modify that object:

```
final StringBuilder sb = new StringBuilder();
sb.append("Hello"); // This is fine
sb = new StringBuilder(); // This won't compile
```

In this example, we can call methods on `sb` that modify the `StringBuilder` object, but we can't assign a new `StringBuilder` instance to `sb`.

Variable Type Inference

Java 10 and later versions introduced a new feature, `var`. It lets you declare a local variable without specifying its type:

```
var myVar = 42;
```

This is called local variable type inference. The compiler looks at the value you're assigning to the variable and figures out the appropriate type for you. In this case, it infers that `myVar` should be an `int`.

Traditionally, declaring local variables could often lead to verbose and repetitive code. For example:

```
HashMap<Integer, String> map = new HashMap<>();
List<String> list = new ArrayList<>();
AtomicInteger counter = new AtomicInteger(0);
```

In each case, the type is mentioned twice, once on the left-hand side and once on the right-hand side. This is where the `var` keyword comes into play.

By using `var`, the above code can be rewritten as:

```
var map = new HashMap<Integer, String>();
var list = new ArrayList<String>();
var counter = new AtomicInteger(0);
```

The types of `map`, `list`, and `counter` are inferred by the compiler based on the initializer expressions. This makes the code more concise and readable, while still maintaining type safety.

It's important to note that `var` behaves like a keyword in its context of use, even though it is technically a reserved type name for local variable type inference. This means code that uses `var` as a variable, method, or package name will not be impacted.

`var` is restricted to local variables within methods, constructors, or initializer blocks. It cannot be used to declare instance variables (fields) or class (static) variables. This restriction ensures that the type of class and instance variables is always clear from the class's API, not just from its implementation:

```
public class MyClass {
   var myVar = "Hello"; // This will not compile
}
```

Similar to instance and class variables, `var` cannot be used to declare method parameters. Method signatures are part of the class's public API and need to explicitly state their parameter types for clarity and to ensure contract stability:

```
public void myMethod(var param) { // This will not compile
   // ...
}
```

Apart from that, `var` can be used in other situations. For example in `for` loop indexes:

```
var numbers = Arrays.asList(1, 2, 3, 4, 5);
for (var num : numbers) {
    System.out.println(num);
}

// Or

for (var i = 1; i <= 10; i++) {
    System.out.println(i);
}
```

In `try-with-resources` statements:

```
try (var stream = Files.lines(Path.of("file.txt"))) {
    stream.forEach(System.out::println);
}
```

Or for the parameters of implicitly typed lambda expressions:

```
Function<Integer, String> toString = (var i) -> String.valueOf(i);
```

Keep in mind that in a lambda expression, either all parameters need to be declared with `var`, or none of them. Mixing `var` with manifest types or inferred types is not allowed.

However, be careful with `var`, it's not always the best choice. Sometimes explicitly declaring the type can make your code more readable and maintainable. You can only use `var` when you're initializing the variable right there in the declaration:

```
var myVar; // This won't compile
var myOtherVar = someMethodThatReturnsAnObject(); // Fine, as long as the method return type is clear
```

Similarly, `var` cannot be used when initializing a variable with a `null` value without specifying its type because the compiler cannot infer the type of the variable:

```
// This will not compile because the type cannot be inferred
var myVar = null;
```

However, once `var` has been used to declare a variable with a concrete type, it can be reassigned a `null` value:

```
var myString = "Hello, World!"; // Inferred as String
myString = null; // This is allowed
```

Finally, when using `var` with array initializers, explicit instantiation is required. You cannot use shorthand syntax because the type cannot be inferred: `java var numbers = new int[] {1, 2, 3}; // This works // var numbers = {1, 2, 3}; // This will not compile`

Inheritance

Introducing Inheritance

Inheritance is one of the core concepts in object-oriented programming. It allows you to define a new class based on an existing class. The new class inherits the attributes and methods of the existing class, allowing you to reuse code and build hierarchical relationships between your classes.

Do you remember the `Cookie` class from the beginning of the previous chapter?

```
public class Cookie {
    // Attributes
    String flavor;
    int size;

    // Behavior (Method)
    public void eat() {
        System.out.println("That was yummy!");
    }
}
```

How would you define a chocolate chip cookie class?

Well, chocolate chip cookies have a flavor, number of chips, and can be eaten like normal cookies. But they also have additional properties like number of chips per cookie. So our initial naive `ChocolateChipCookie` class might look like:

```
public class ChocolateChipCookie {

  String flavor;
  int size;

  void eat() {
    System.out.println("That was yummy!");
  }

  int chips;

}
```

We've duplicated the cookie attributes and methods! It is not a good design.

This is where the concept of inheritance enters in OOP.

All varieties of cookies share common properties like having a flavor and being edible. We can represent this with a parent `Cookie` class that contains `flavor`, `size`, and an `eat()` method.

Child classes, like `ChocolateChipCookie`, can then inherit these common cookie elements from the parent `Cookie` class. This way, we can create many specific varieties that inherit shared cookie properties. The child classes can still define their own specialized attributes, like the number of chocolate chips, but reuse the inherited parent code.

In Java, you use the `extends` keyword to create a subclass that inherits from a superclass. This is how the `ChocolateChipCookie` class can be defined using inheritance:

```
public class ChocolateChipCookie extends Cookie {

  int chips;

  public void addChips(int chipsPerCookie) {
    this.chips += chipsPerCookie;
  }

}
```

Here, `ChocolateChipCookie` is a subclass of `Cookie`. It inherits the `flavor` and `size` fields and the `eat()` method. The subclass can declare its own methods, like how `ChocolateChipCookie` declares the `addChips()` method.

However, a subclass cannot directly access `private` members of its superclass. Subclasses can only access `protected` and `public` members of the superclass directly. To access `private` fields, the superclass must provide `public` or `protected` accessors.

One important thing to know is that in Java, a class can only extend from one class due to the design choice to avoid the complexity and ambiguity associated with multiple inheritance. In other words, multiple inheritance, where a class can extend more than one class, can lead to:

1. **Diamond Problem:** This is a well-known complication where a class inherits from two classes that have a common base class. This scenario creates ambiguity in the inheritance hierarchy when two parent classes have methods with the same signature, as the system might not be able to determine which version of the inherited method to use.
2. **Increased Complexity:** Allowing multiple inheritance can make the design and maintenance of a program more complex. Understanding the flow of methods and variables becomes harder, especially in large codebases.

Some important class modifiers related to inheritance are `final`, `abstract`, and `sealed`.

Final classes cannot be subclassed. If you try to extend a `final` class, you'll get a compile error. Using the cookie example, if the `Cookie` class were declared as `final`:

```
public final class Cookie {
    // ...
}
```

The declaration of the `ChocolateChip` class will generate a compilation error.

Making a class `final` ensures that its implementation cannot be changed by subclassing. However, contrary to a common misconception, `final` classes are not more efficient at runtime just because they are `final`. The `final` modifier is about inheritance, not performance.

Abstract classes cannot be instantiated, only subclassed. They are incomplete on their own and need to be extended to be used. Abstract classes often contain abstract methods that have no implementation in the abstract class and must be implemented by concrete subclasses. Trying to create an instance of an abstract class with `new` will result in a compile error.

Sealed classes provide a middle ground between final and non-final classes. Sealed classes can be extended, but only by classes explicitly permitted to do so in the sealed class declaration. This gives you fine-grained control over inheritance. Subclasses of sealed classes must themselves be declared sealed, non-sealed or final. Sealed classes restrict but don't completely prohibit inheritance like final classes do.

Let's review in more detail abstract and sealed classes.

Abstract Classes

An abstract class is a class that cannot be instantiated, meaning you cannot create new instances of an abstract class. It serves as a base for subclasses:

```
abstract class Cookie {
    abstract void flavor();
}
```

You must use the `abstract` keyword to declare a class or a method as abstract. An abstract class may or may not include abstract methods.

Abstract methods are declared without an implementation (without braces, and followed by a semicolon):

```
abstract void flavor();
```

Abstract methods are similar to regular methods in the sense that you declare them with or without parameters, with a return value or `void`, and any access modifier like `public`, `protected`, or default. The only difference is that abstract methods do not have any implementation, they cannot have a body, therefore, they end with a semicolon (`;`) and not with brackets (`{}`).

To use an abstract class, you have to inherit it from another class using the `extends` keyword. Let's see an example:

```
class OatmealRaisinCookie extends Cookie {
    void flavor() {
        System.out.println("Oatmeal and raisin flavor");
    }
}
```

When inheriting from an abstract class, the subclass usually provides implementations for all of the abstract methods in its parent class. If it doesn't, then the subclass must also be declared abstract:

```
abstract class Cookie {
    abstract void flavor();

    public void bake() {
        System.out.println("Cookie is baking");
    }
}

abstract class OatmealRaisinCookie extends Cookie {
    // Abstract method which makes the class abstract
    // (Otherwise it will not compile)
    abstract void flavor();

    // Even if it defines concrete method(s)
    public void addRaisins() {
        System.out.println("Adding raisins");
    }
}
```

Why?

Because an abstract class is (or is intended to be) incomplete. Creating an object from an incomplete class would be wrong. An abstract class needs to be extended to be used, very much like a template.

Abstract classes are helpful to share code among closely related classes. Most importantly, abstract classes can define methods that the subclasses must implement, establishing a contract or a protocol that subclasses must follow.

It's good to think of concrete classes as specializations of abstract classes. The same way a compact car is a specialization of the general concept of a car, abstract classes are the general concept and concrete classes are a specific implementation of that concept.

Concrete classes have to implement all abstract methods but they can also define their own new methods. Not all methods have to be abstract in a concrete class, just the ones declared as abstract in the parent abstract class. Here's an example to illustrate this:

```
abstract class Cookie {
    abstract void flavor();

    public void bake() {
        System.out.println("Cookie is baking");
    }
}

class ChocolateCookie extends Cookie {
    // Implementing the abstract method
    void flavor() {
        System.out.println("Chocolate flavor");
    }

    // Defining its own new method
    public void addChocolateChips() {
        System.out.println("Adding chocolate chips");
    }
}

class OatmealRaisinCookie extends Cookie {
    // Implementing the abstract method
    void flavor() {
        System.out.println("Oatmeal and raisin flavor");
    }

    // Defining its own new method
    public void addRaisins() {
        System.out.println("Adding raisins");
    }
}
```

In this example, `Cookie` is an abstract class with one abstract method `flavor()` and one concrete method `bake()`.

The `ChocolateCookie` and `OatmealRaisinCookie` classes are concrete classes that extend the `Cookie` abstract class. They both implement the abstract method `flavor()` that they inherited from `Cookie`. Again, this is mandatory, otherwise they would have to be declared as abstract as well.

But `ChocolateCookie` and `OatmealRaisinCookie` also define their own new methods, `addChocolateChips()` and `addRaisins()` respectively. These methods are specific to each type of cookie and are not related to the abstract class.

When you create instances of `ChocolateCookie` and `OatmealRaisinCookie`, you can call all their methods:

```
ChocolateCookie chocolateCookie = new ChocolateCookie();
chocolateCookie.flavor();          // Output: Chocolate flavor
chocolateCookie.addChocolateChips();  // Output: Adding chocolate chips
chocolateCookie.bake();            // Output: Cookie is baking
```

```
OatmealRaisinCookie oatmealRaisinCookie = new OatmealRaisinCookie();
oatmealRaisinCookie.flavor();      // Output: Oatmeal and raisin flavor
oatmealRaisinCookie.addRaisins();  // Output: Adding raisins
oatmealRaisinCookie.bake();        // Output: Oatmeal Raisin Cookie is baking at a lower temperature
```

Abstract classes can have constructors. You need them to initialize attributes and execute any logic that needs to run when an instance of the concrete (sub)class is created. An abstract class is a class, and like any other class, it can have attributes and those attributes might need to be initialized when an instance (of the concrete class) is created. Here's an example:

```
abstract class Cookie {
    protected String name;

    public Cookie(String name) {
        this.name = name;
        System.out.println("Cookie constructor is called");
    }

    abstract void flavor();

    public void bake() {
        System.out.println(name + " is baking");
    }
}
```

In this updated example, the `Cookie` abstract class now has a constructor that takes a `name` parameter. It initializes the `name` attribute of the cookie. The `name` attribute is declared as `protected`, which means it is accessible to subclasses.

This way, the concrete classes `ChocolateCookie` and `OatmealRaisinCookie` can call the constructor of the abstract `Cookie` class using `super()`, passing in the specific name for each type of cookie. We'll see how to use `super()` later in this chapter.

When you think of abstract classes as a contract or a template that subclasses must follow and complete to ensure a common behavior, the following rules make sense:

- An abstract class cannot be final. The `final` modifier prevents a class from being subclassed, and this contradicts the essence of an abstract class, that it must be inherited to be used. So no, marking an abstract class as `final` would not make it more secure, it would make it useless.
- Abstract methods cannot be final either, for the same reason, they have to be overridden in a subclass.
- Abstract methods cannot be native or synchronized, for slightly different reasons. A native method is implemented in another language like C++ in the JVM, so it would already have an implementation. We'll talk about synchronized methods in a later chapter, but the `synchronized` modifier is used to coordinate multithreaded access, and for that, the method needs a body, an implementation.
- An abstract method cannot be `private`. It doesn't make sense that a method that must be implemented by a subclass in another class, cannot be seen by that class. So no, `private` abstract methods cannot exist.
- Finally, an abstract method cannot be static either. Static methods belong to the class itself, not to any instance. An abstract method is only useful when a subclass implements it, which means an instance uses it.

In summary, here are the rules for correctly declaring abstract classes and methods:

- If a class contains one or more abstract methods, the class must be declared as abstract.

- An abstract class can have both abstract and non-abstract (concrete) methods.
- An abstract class can extend another abstract or concrete class and an abstract class can be extended by another abstract or concrete class.
- A subclass can override a concrete method in a superclass and declare it as abstract.
- An abstract subclass can override some or none of the abstract methods in its superclass, but a first concrete subclass must implement all of them.

Now, before talking about sealed classes, let's review the topic of interfaces.

Interfaces

When it comes to object-oriented programming, in addition to classes, Java provides one powerful tool, interfaces. An interface in Java is essentially a contract that defines a set of methods a class must implement. It's similar to a menu at a restaurant. The menu lists the dishes available but doesn't provide details on how they're prepared. When you order a dish from the menu, the kitchen (the class) provides a specific implementation of that dish (the method).

So, what exactly is an interface and how does it differ from a regular class or even an abstract class?

An interface in Java is a reference type, similar to a class, that can contain only constants, method signatures, default methods, static methods, and nested types. Interfaces cannot be instantiated, they can only be implemented by classes or extended by other interfaces.

To declare an interface, you use the `interface` keyword instead of the `class` keyword. Here's an example:

```
public interface Drawable {
    void draw();
}
```

Any class that implements the `Drawable` interface must provide an implementation for the `draw()` method.

At first glance, interfaces might seem very similar to abstract classes. After all, both can contain abstract methods, methods without a body. However, there are some key differences: - An abstract class can have instance variables and constructors, while an interface cannot.

- An abstract class can have non-abstract methods, while all methods in an interface are implicitly abstract (with the exception of default and static methods, which we'll cover later).
- A class can extend only one abstract class, but it can implement multiple interfaces.

So while there is some overlap, interfaces and abstract classes serve different purposes and are not interchangeable.

To use an interface, a class must implement it. The `implements` keyword is used to implement an interface:

```
public class Circle implements Drawable {
    public void draw() {
        System.out.println("Drawing a circle");
    }
}
```

If a class implements an interface but does not implement all the methods, it must be declared as `abstract`.

```
public abstract class Shape implements Drawable {
    // Class content
}
```

All methods in an interface are implicitly `public` and `abstract`. You don't need to use the `public` or `abstract` keyword when declaring methods in an interface.

All variables declared in an interface are implicitly `public`, `static`, and `final`.

So this:

```
public interface MyInterface {
    int NUMBER = 10;
    void method();
}
```

It's equivalent to this:

```
public interface MyInterface {
    public static final int NUMBER = 10;
    public abstract void method();
}
```

It's important to note that because interface methods are `abstract`, they cannot be declared as `private`, `protected`, `final`, or `static` (with the exception of `static` methods, which we'll cover later).

An interface can extend another interface, similar to how a class can extend another class. The `extends` keyword is used for this:

```
public interface Moveable {
    void move();
}

public interface Drawable extends Moveable {
    void draw();
}
```

In this case, any class that implements `Drawable` must provide implementations for both `draw()` and `move()`.

A class can only extend from one class. However, a class can implement multiple interfaces. This is a way to achieve a form of multiple inheritance in Java:

```
public interface Moveable {
    void move();
}

public interface Drawable {
    void draw();
}

public class Circle implements Drawable, Moveable {
    public void draw() {
        System.out.println("Drawing a circle");
    }

    public void move() {
        System.out.println("Moving a circle");
    }
}
```

This doesn't violate Java's single inheritance rule because interfaces don't contain any implementation. If a class implements two interfaces that have the same method, it's not a problem. The class simply provides one implementation of the method, solving the ambiguity and complexity problems:

```
public interface A {
    void method();
}
```

```java
public interface B {
    void method();
}

public class C implements A, B {
    public void method() {
        System.out.println("Method implementation");
    }
}
```

Also, interfaces can have default methods. These are methods with a body that provide a default implementation if a class doesn't override them:

```java
public interface Drawable {
    void draw();
    default void print() {
        System.out.println("Printing...");
    }
}
```

Classes that implement `Drawable` can, but are not required to, override the `print()` method.

If a class implements two interfaces and both have the same default method, the class must override the method. If it wants to call the default method from one of the interfaces, it can do so using the `super` keyword:

```java
public interface A {
    default void method() {
        System.out.println("A's method");
    }
}

public interface B {
    default void method() {
        System.out.println("B's method");
    }
}

public class C implements A, B {
    public void method() {
        A.super.method();
    }
}
```

Interfaces can also have static methods, similar to static methods in classes:

```java
public interface Drawable {
    static void staticMethod() {
        System.out.println("Static method");
    }
}
```

Static methods in interfaces are not inherited by classes or interfaces that extend the interface.

For the above example, you would use the `Drawable` interface to call `staticMethod` like this:

```java
Drawable.staticMethod();
```

In addition to default and `static` methods, interfaces can also have `private` methods. These are helpful for sharing code between default methods in the interface:

```
public interface Drawable {
    default void print() {
        printLine();
        System.out.println("Printing...");
    }

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

Private methods in interfaces cannot be accessed by classes that implement the interface.

Sealed Classes

Imagine a royal family with a strict rule: only certain people can become future kings or queens, and this rule is unchangeable. In Java, sealed classes are like this royal family. They allow a class to strictly control which other classes can extend it, just like the royal family controls who can be in line for the throne.

So if a class is sealed, does that mean it's completely locked down and no one can extend it at all? Not quite. A sealed class simply restricts who can extend it, but it's not completely off limits. You get to specify a set of permitted subclasses.

This feature is useful for several reasons: - It allows library authors to evolve APIs over time while avoiding unintended extensions. - It enables modeling of hierarchies and state machines with a finite set of subclasses. - It provides compile-time safety by limiting the possibilities for external code.

To create a sealed class, you use the `sealed` modifier on the class declaration, along with the `permits` clause to specify the permitted subclasses:

```
public sealed class Vehicle permits Car, Truck, Motorcycle {
    public void startEngine() {
        System.out.println("Starting the vehicle's engine.");
    }
}

final class Car extends Vehicle {
    @Override
    public void startEngine() {
        System.out.println("Starting the car's engine.");
    }
}

final class Truck extends Vehicle {
    @Override
    public void startEngine() {
        System.out.println("Starting the truck's engine.");
    }
}

final class Motorcycle extends Vehicle {
    @Override
    public void startEngine() {
        System.out.println("Starting the motorcycle's engine." );
```

```
    }
}
```

The `sealed` modifier indicates that the class is sealed. The `permits` clause lists the classes that are allowed to extend the sealed class.

Sealed classes and their subclasses must be declared in the same package (or named module) as their direct subclasses. This ensures a close relationship between the sealed class and its permitted subclasses.

Every class that directly extends a sealed class must specify exactly one of the following three modifiers: `final`, `sealed`, or `non-sealed`:

- `final`: The subclass cannot be extended further. This is the most restrictive option.
- `sealed`: The subclass is also sealed and must specify its own permitted subclasses.
- `non-sealed`: The subclass is open for extension by unknown subclasses. This is the most permissive option.

If you don't specify one of these modifiers on a direct subclass of a sealed class, you'll get a compilation error. The compiler enforces this to ensure the hierarchy is well-defined.

Marking a subclass as `non-sealed` simply means it's open for extension. It doesn't require you to actually create new subclasses. Accidentally using `non-sealed` when you don't add more subclasses won't break anything, but it does signal to other developers that your intent is to allow the class to be extended.

The `permits` clause is optional if the sealed class and its direct subclasses are declared within the same file or the subclasses are nested within the sealed class. The compiler can infer the permitted subclasses in these cases, so you can omit the explicit listing.

Here's an example where the `permits` clause is omitted:

```java
// Beverage.java
public sealed class Beverage {
    void pour();
}

final class Coffee implements Beverage {
    public void pour() {
        System.out.println("Pouring coffee");
    }
}

final class Tea implements Beverage {
    public void pour() {
        System.out.println("Pouring tea");
    }
}
```

Since `Coffee` and `Tea` are declared in the same file as the sealed `Beverage` class (`Beverage.java`), the `permits` clause can be inferred by the compiler.

So sealed classes can only be used within the same file? No, sealed classes and their subclasses can be in different files, as long as they are in the same package or module. The same-file restriction is only relevant for omitting the `permits` clause.

And to answer another common question: "If I seal a class, I can't use it in another package, can I?" You can use a sealed class from another package, but you can't declare its subclasses in a different package. The usage is not restricted, only the extension.

In any case, once a class is sealed, the set of permitted subclasses is fixed. You can't add new subclasses outside of what's specified in the `permits` clause. If you need to extend the hierarchy later, you'd have to

modify the sealed class to permit additional subclasses. This requires recompiling the sealed class and its existing subclasses.

If you're wondering if there's a limit to how many subclasses a sealed class can permit, the answer is no, there's no hard limit on the number of subclasses you can permit. However, the intent of sealed classes is to have a finite and manageable set of subclasses. Allowing hundreds of subclasses would go against that spirit and likely indicate a design issue. Stick to a reasonable number that makes sense for your use case.

Sealing is not limited to just classes. You can seal interfaces too.

Interfaces can be sealed to limit the classes that implement them or the interfaces that extend them. Here's an example:

```
public sealed interface Shape permits Circle, Rectangle, Triangle, Polygon {
    double getArea();
}

final class Circle implements Shape {
    public double getArea() {
        // Implementation of getArea() for circles
    }
}

final class Rectangle implements Shape {
    public double getArea() {
        // Implementation of getArea() for rectangles
    }
}

final class Triangle implements Shape {
    public double getArea() {
        // Implementation of getArea() for triangles
    }
}

sealed interface Polygon extends Shape permits RegularPolygon, IrregularPolygon {
    int getNumberOfSides();
}

final class RegularPolygon implements Polygon {
    public double getArea() {
        // Implementation of getArea() for regular polygons
    }

    public int getNumberOfSides() {
        // Implementation of getNumberOfSides() for regular polygons
    }
}

final class IrregularPolygon implements Polygon {
    public double getArea() {
        // Implementation of getArea() for irregular polygons
    }

    public int getNumberOfSides() {
        // Implementation of getNumberOfSides() for irregular polygons
```

```
    }
}
```

In this example, the `Shape` interface is sealed and permits four classes to implement it: `Circle`, `Rectangle`, `Triangle`, and `Polygon`. This means that only these four classes can directly implement the `Shape` interface.

But the `Polygon` interface is also sealed and extends the `Shape` interface. It permits two classes to implement it: `RegularPolygon` and `IrregularPolygon`. This demonstrates how sealing can be used to control which interfaces can extend a sealed interface.

By sealing the `Polygon` interface, we restrict the classes that can implement it to just `RegularPolygon` and `IrregularPolygon`. No other class can directly implement `Polygon`. However, since `Polygon` extends `Shape`, the `RegularPolygon` and `IrregularPolygon` classes indirectly implement `Shape` as well.

This allows for a well-defined and constrained inheritance structure.

The above also applies to classes, you can change the `Shape` interface to a class and make the necessary modifications to the other classes to achieve a similar hierarchical structure.

To summarize, here are the key rules for sealed classes:

1. Sealed classes are declared with the `sealed` and `permits` modifiers.
2. Sealed classes must be declared in the same package or named module as their direct subclasses.
3. Direct subclasses of sealed classes must be marked `final`, `sealed`, or `non-sealed`.
4. The `permits` clause is optional if the sealed class and its direct subclasses are declared within the same file or the subclasses are nested within the sealed class.
5. Interfaces can be sealed to limit the classes that implement them or the interfaces that extend them.

The `this` Reference

When you're writing code in Java, you'll often see the keyword `this` sprinkled around in your methods and constructors. But what exactly is `this`, and why do we use it?

`this` is a reference to the current instance of a class. In other words, when you're inside a method or constructor of a class, `this` refers to the specific object that the method or constructor belongs to. Here's a simple example:

```
public class Person {
    private String name;

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

In the constructor, we use `this.name` to specify that we're talking about the `name` field of this particular `Person` object, not some other `name` variable.

But wait, you might be thinking, "So, `this` is just another variable I can change, right?" Well, not exactly. `this` is a final reference, which means you can't assign it to something else. It always points to the current object instance.

`this` cannot be used anywhere in the code, like in static methods, It is only relevant within the context of an instance method or constructor. Static methods belong to the class itself, not a specific instance, so `this` doesn't have any meaning there.

So, do you have to use `this` every time you refer to an attribute or method, no matter what? Not necessarily. If there's no ambiguity, you can often omit `this`. However, there are times when using `this` can make your code clearer and avoid confusion. For example:

```java
public class Person {
    private String name;

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

    public void introduce(Person other) {
        System.out.println("Hi " + other.name + ", I'm " + this.name);
    }
}
```

Here, using `this.name` makes it clear that we're referring to the `name` of the current `Person` instance, not the `other Person`.

Here are a few situations where `this` is necessary: - To disambiguate between local variables and instance variables with the same name - To pass the current instance as an argument to a method - To call another constructor from within a constructor

Speaking of constructors, you cannot use `this` to call a constructor from anywhere in my class. You can only use `this` to call another constructor from within a constructor, and it must be the first statement:

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

    public Person(String name) {
        this(name, 0);
    }

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

This is useful when you have multiple constructors and you want to avoid duplicating code.

One rule, however, is that if you're using `this` to invoke another constructor, it must be the first statement in the constructor. This rule ensures that another constructor is called before executing any code in the constructor that contains the this call, preventing the use of uninitialized fields or the duplication of initialization code. For example, the following will not compile:

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

    public Person(String name) {
        System.out.println("Person(String) Constructor Called");
        // The following line will cause a compilation error
        this(name, 0); // ERROR: Constructor call must be the first statement in a constructor
    }

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

```
}
```

Also, remember that `this` doesn't refer to the class itself. `this` refers to the current instance. Each instance gets its own `this` reference. It can't be `null`.

This also means that `this` is used for instance members. Static fields and methods belong to the class itself, not a specific instance, so `this` isn't applicable.

Also, when you use `this` inside a method, you're referring to the object instance that the method belongs to, not the method itself.

Finally, passing `this` as an argument is useful when you want to give another method access to the current instance. For example, you might pass `this` to a method of another class so that it can call back to the originating object:

```
public class Person {
    private String name;

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

    public void introduceYourselfTo(IntroductionService service) {
        service.introduce(this);
    }

    public String getName() {
        return name;
    }
}

public class IntroductionService {
    public void introduce(Person person) {
        System.out.println("Hello, my name is " + person.getName());
    }
}
```

In this example, we have two classes: `Person` and `IntroductionService`.

The `Person` class has a method called `introduceYourselfTo`, which takes an `IntroductionService` as a parameter. Inside this method, `this` (referring to the current `Person` instance) is passed as an argument to the `introduce` method of the `IntroductionService`.

The `IntroductionService` class has an `introduce` method that takes a `Person` as a parameter. This method can then access the `Person`'s `getName()` method to print out the introduction.

Here's how you might use these classes:

```
Person alice = new Person("Steve");
IntroductionService service = new IntroductionService();
alice.introduceYourselfTo(service);
```

And this is the output:

```
Hello, my name is Steve
```

The `super` Reference

So the `this` keyword is used to refer to the current instance of the class. But what if you want to refer to the superclass from which your current class inherits? That's where `super` comes in.

The `super` keyword acts as a reference to the parent class (superclass) of the current class. It allows access to the superclass's members (fields, methods, and constructors).

The main purpose of `super` is to differentiate between members of the superclass and members of the current class when they have the same name. By prefixing `super` to a member name, you specify that you wish to use the superclass's version of that member, rather than the current class's version.

The syntax for using `super` is straightforward:

```
super.memberName
```

Here, `memberName` can be a field, method, or constructor of the superclass.

Overriding in Java is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its superclasses or parent classes.

When you override a method in a subclass, you're not erasing or replacing the original method in the superclass. The superclass method is still there, but when you call the method on an object of the subclass, the overridden version in the subclass is executed instead. So, when overriding a method in a subclass, you might want to call the original implementation of the method from the superclass.

In that case, you can use `super` to invoke the superclass's version of the method:

```
@Override
public void someMethod() {
    super.someMethod(); // Calls the superclass's implementation
    // Additional code specific to the subclass
}
```

Another common use case for `super` is when you want to invoke the constructor of the superclass from the constructor of the current class. Just like with `this`, you must call `super()` as the first statement in the constructor:

```
public class SubClass extends SuperClass {
    public SubClass() {
        super(); // Invokes the superclass constructor
        // Other initialization code
    }
}
```

Otherwise, you'll get a compilation error.

If your superclass doesn't have a default (no-argument) constructor, you'll need to explicitly call a parameterized constructor using `super(arguments)`. You cannot use `super` without specifying the required arguments.

Consider this example:

```
// Superclass without a default constructor
public class Person {
    private String name;
    private int age;

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

    // Getter methods for name and age
    public String getName() {
        return name;
```

```
    }

    public int getAge() {
        return age;
    }
}

// Subclass that extends Person
public class Student extends Person {
    private String studentID;

    // Since Person does not have a default constructor, we must explicitly call a parameterized constructor
    public Student(String name, int age, String studentID) {
        super(name, age); // Calls the superclass constructor with arguments
        this.studentID = studentID;
    }

    // Getter method for studentID
    public String getStudentID() {
        return studentID;
    }
}
```

In the `Student` constructor, `super(name, age);` is used to explicitly call the parameterized constructor of the `Person` class. This is necessary because `Person` does not have a no-argument constructor. If this `super` call was omitted, the code would not compile, as Java would attempt to call a default constructor in the `Person` class, which does not exist in this case.

Now, you might be wondering, if I use `super`, does that mean I can't use `this` in the same method? The answer is no. You can use both `this` and `super` in the same method, as they serve different purposes. `this` refers to the current instance, while `super` refers to the superclass.

However, it's important to keep in mind is that `super` cannot be used to directly access private members (fields or methods) of the superclass. Private members are only accessible within the same class. If you need to access them, you'll have to rely on `public` or `protected` methods provided by the superclass.

Finally, it's also worth noting that while `super` is primarily used to call methods or access fields from the immediate parent class, it indirectly allows for interaction with the broader inheritance hierarchy. In particular, if the immediate parent class inherits methods from its ancestors (grandparent classes and beyond), `super` can indirectly access these methods as well. This is because the inherited methods from the parent class, which `super` can call, may themselves call methods from their ancestors within the inheritance chain. However, direct invocation of methods or access to fields from grandparent classes or higher, using `super`, is not possible. To access such methods directly, you would typically rely on the inherited methods that encapsulate this functionality within your immediate superclass.

Consider the following example, which extends the previous example by adding a new class, `GraduateStudent`, which inherits from `Student`, and a grandparent class, `Human`, from which `Person` inherits:

```
// Grandparent class
public class Human {
    private String nationality;

    public Human(String nationality) {
        this.nationality = nationality;
    }

    protected void sayHello() {
```

```
        System.out.println("Hello from Human!");
    }
}

// Parent class
public class Person extends Human {
    private String name;
    private int age;

    public Person(String name, int age, String nationality) {
        super(nationality); // Calls the Human constructor
        this.name = name;
        this.age = age;
    }

    // Overriding the sayHello method
    @Override
    protected void sayHello() {
        super.sayHello(); // Calls Human's sayHello
        System.out.println("Hello from Person!");
    }
}

// Current class
public class Student extends Person {
    private String studentID;

    public Student(String name, int age, String nationality, String studentID) {
        super(name, age, nationality); // Calls the Person constructor
        this.studentID = studentID;
    }

    // Overriding the sayHello method again
    @Override
    protected void sayHello() {
        super.sayHello(); // Calls Person's sayHello, which in turn calls Human's sayHello
        System.out.println("Hello from Student!");
    }
}

// New Subclass that extends Student
public class GraduateStudent extends Student {
    private String researchTopic;

    public GraduateStudent(String name, int age, String nationality, String studentID, String researchTopic) {
        super(name, age, nationality, studentID); // Calls the Student constructor
        this.researchTopic = researchTopic;
    }

    public void introduce() {
        super.sayHello(); // Calls Student's sayHello, which in turn calls Person's, and then Human's sayHello
        System.out.println("I am a graduate student working on " + researchTopic + ".");
    }
}
```

In this example, the `GraduateStudent` class uses `super.sayHello()` in its `introduce` method. This calls the `sayHello` method from the `Student` class, which itself overrides `Person`'s `sayHello` method. The `Person` class's `sayHello` method then calls `Human`'s `sayHello` method. This demonstrates how `super` can be used to indirectly access methods up the inheritance chain, from the `Human` class to the `GraduateStudent` class, even though direct access to `Human`'s methods from `GraduateStudent` using `super` is not possible.

Now let's talk more about overriding and polymorphism.

Polymorphism

Introducing Polymorphism

Polymorphism is one of the pillars of object-oriented programming, and it's a powerful concept in Java. In simple terms, polymorphism allows you to treat objects of different subclasses as if they were objects of the same superclass. It's like having a single remote control that can operate multiple types of devices, a TV, a stereo, and a DVD player. Just as the remote control sends signals to each device that performs different functions depending on the device it's communicating with, in Java, you can use a single reference type to interact with objects of different classes, allowing them to perform their own unique behaviors through a common interface.

However, polymorphism doesn't mean that methods can arbitrarily change their behavior. Instead, it allows subclasses to provide their own implementations of methods defined in the superclass, a concept known as method overriding.

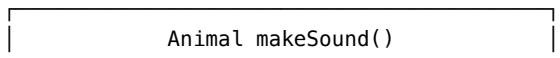
As mentioned before, when you override a method in a subclass, you're not erasing or replacing the original method in the superclass. The superclass method is still there, but when you call the method on an object of the subclass, the overridden version in the subclass is executed instead. It's important to note that overriding is not the same as hiding members, which we'll discuss later.

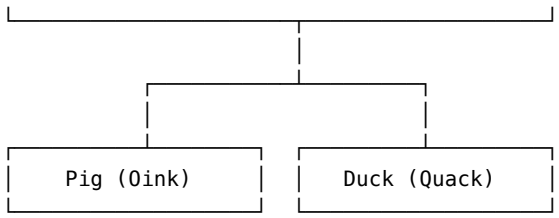
To properly override a method, the method in the subclass must have the same: - Name - Return type - Parameter list

As the method in the superclass. Here's an example:

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

class Pig extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Oink");
    }
}

class Duck extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Quack");
    }
}
```

And a diagram to visualize this hierarchy:





The `Animal` class has a method called `makeSound()`. The `Pig` and `Duck` classes, which extend `Animal`, override the `makeSound()` method to provide their own implementations. Now, let's see polymorphism in action:

```
Animal animal1 = new Pig();
Animal animal2 = new Duck();

animal1.makeSound(); // Output: Oink
animal2.makeSound(); // Output: Quack
```

Here, we create two variables of type `Animal`, but we assign them objects of the `Pig` and `Duck` classes. When we call the `makeSound()` method on each variable, the appropriate overridden method in the respective subclass is called. This is the power of polymorphism, the ability to treat objects of different subclasses as objects of a common superclass.

It's important to understand that overriding is not the same as overloading. Overloading refers to having multiple methods with the same name but different parameter lists within the same class. Overriding, on the other hand, is about providing a different implementation of a method in a subclass.

Another common misconception is that overriding applies to all members of a class, including variables. However, that's not true. Overriding specifically applies to methods. When you declare a variable with the same name in a subclass, you're actually hiding the variable from the superclass, not overriding it.

Let's explore some rules related to overriding.

Overriding Rules

There are several rules you need to follow when overriding methods from a superclass:

Rule #1: Method Signatures
The first and most important rule is that the method signature must match exactly between the superclass and subclass. This means the name, parameters, and return type need to be identical (with one exception we'll discuss later). You can't change the parameters or return type however you want:

```
// Superclass
class Cookie {
    // Define a method 'eat' in the superclass
    public String eat() {
        return "Eating a plain cookie";
    }
}

// Subclass
class ChocolateChipCookie extends Cookie {
    // Override the 'eat' method in the subclass
    @Override
    public String eat() {
        return "Eating a chocolate chip cookie";
    }
}
```

In this example: - The `Cookie` class defines a method named `eat` that returns a `String`.

- The `ChocolateChipCookie` class, which extends `Cookie`, overrides the `eat` method. The overriding method in `ChocolateChipCookie` has the same name, return type, and parameter list (in this case, none) as the method in `Cookie`.
- When an instance of `ChocolateChipCookie` calls `eat`, the overridden version of the method is executed, returning `"Eating a chocolate chip cookie"`.

Why does the method signature have to stay the same? Well, think of it like a contract between the superclass and subclass. The superclass is defining a specific method that subclasses can override if needed. If you change the signature, you're breaking that contract. The subclass method would no longer be a true override of the superclass method.

Rule #2: Access Modifiers
When overriding a method, you can make the access modifier more lenient, but not more restrictive. For example, you could override a `protected` method in the superclass and make it `public` in the subclass. But you can't do the opposite, like changing a `public` method to `private`:

```
// Superclass
class Cookie {
    // Define a method with 'protected' access modifier in the superclass
    protected String recipe() {
        return "Default cookie recipe";
    }
}

// Subclass
class ChocolateChipCookie extends Cookie {
    // Override the 'recipe' method in the subclass and change the access modifier to 'public'
    @Override
    public String recipe() {
        return "Chocolate chip cookie recipe";
    }
}
```

In this example: - The `Cookie` class defines a `recipe` method with a `protected` access modifier. This means the `recipe` method can only be accessed within its own class, subclasses, or within the same package.

- The `ChocolateChipCookie` class, which extends `Cookie`, overrides the `recipe` method, changing the access modifier of the overriding method to `public`, which is less restrictive than `protected`.
- Trying to access the `recipe` method directly from a `Cookie` instance would lead to a compile-time error, due to the `protected` access control. However, accessing the `recipe` method through an instance of `ChocolateChipCookie` is possible because it's `public`.

This often confuses people. They think: "Since it's my subclass, shouldn't I be able to limit access to the method if I want?" However, this goes against the idea that a subclass should always work wherever its superclass is used. If you make the method more restricted in the subclass, you disrupt this compatibility.

Rule #3: Checked Exceptions
We'll review exceptions in more detail in a later chapter, but if the superclass method declares any checked exceptions in its `throws` clause, the overridden method in the subclass can only declare exceptions that are the same or more specific. It can't add any new checked exceptions that aren't a subclass of those declared by the superclass method:

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

```
class OverBakingException extends BakingException {
    public OverBakingException(String message) {
        super(message);
    }
}

// Superclass
class Cookie {
    // Define a method that declares throwing a general BakingException
    public String bake() throws BakingException {
        return "Cookie is baked";
    }
}

// Subclass
class ChocolateChipCookie extends Cookie {
    // Override the 'bake' method, declaring a more specific exception, OverBakingException
    @Override
    public String bake() throws OverBakingException {
        return "Chocolate chip cookie is baked";
    }
}
```

In this example: - `BakingException` is a checked exception that represents a general baking error.

- `OverBakingException` is a more specific checked exception, indicating the cookie has been overbaked, and it extends `BakingException`.
- The `Cookie` class has a `bake` method that declares it might throw a `BakingException`.
- The `ChocolateChipCookie` class overrides the `bake` method and declares it might throw an `OverBakingException`, which is a subclass of `BakingException`.

People often think they are allowed to throw any checked exception they want in an overridden method, especially if the superclass doesn't declare any. But that's not the case. Again, it comes down to the contract defined by the superclass method. The subclass can't suddenly introduce new checked exceptions that the caller wasn't expecting to handle.

Rule #4: Covariant Return Types
Here's the one exception to the rule about method signatures. An overridden method is allowed to have a covariant return type. That means the return type can be a subclass of the return type declared in the superclass method. It doesn't have complete freedom to return just anything loosely related though.

For example, if the superclass method returns a `Number`, the subclass could return an `Integer`, since `Integer` is a subclass of `Number`. However, it cannot return a `String`, despite any perceived loose relation to the original `Number`. The return types need that direct hierarchical relationship.

Here's an example to illustrate this rule:

```
class Cookie {
    // A method in the superclass that returns an instance of Cookie
    public Cookie getCookie() {
        return new Cookie();
    }
}

class ChocolateChipCookie extends Cookie {
```

```
    // An overriding method with a covariant return type
    // It returns ChocolateChipCookie, a subclass of Cookie
    @Override
    public ChocolateChipCookie getCookie() {
        return new ChocolateChipCookie();
    }
}
```

In this example: - The `Cookie` class has a method `getCookie` that returns an instance of `Cookie`.

- The `ChocolateChipCookie` class extends `Cookie` and overrides the `getCookie` method. The return type of the overridden method is `ChocolateChipCookie`, which is a subclass of `Cookie`. This change in return type is an example of using covariant return types.
- When `getCookie` is called on an instance of `ChocolateChipCookie`, it returns an instance of `ChocolateChipCookie`, demonstrating the overridden method with a covariant return type in action.

All right.

Have you notice the `@Override` annotation in all these examples?

The `@Override` annotation explicitly marks methods that are intended to override a superclass method. But what's the point of using it? Is it just for clarity, or does it have a real purpose?

While it's true that `@Override` can make your code more readable by clearly indicating overridden methods, it provides a safeguard against accidental errors. Consider this scenario:

```
class Cookie {
    public String recipe() {
        return "Default cookie recipe";
    }
}

class ChocolateChipCookie extends Cookie {
    @Override
    public String recipes() { // Oops, typo in the method name!
        return "Chocolate chip cookie recipe";
    }
}
```

In this case, the subclass intended to override `recipe`, but accidentally introduced a typo, naming it `recipes` instead. Without the `@Override` annotation, this would compile just fine. The subclass would simply have two separate methods: the inherited `recipe` and the new `recipes`.

But with `@Override`, the compiler will catch the mistake and produce an error, indicating that `recipes` does not override any method. The annotation forces the compiler to verify that the method truly overrides a superclass method, providing an extra layer of safety.

Now, what happens if you redeclare a private method from the superclass in a subclass? Is that considered overriding? The answer is no. Private methods are not inherited at all, so there's nothing to override.

If you redeclare a private method in the subclass, it's essentially a completely separate method that just happens to have the same name. It doesn't interact with the superclass method in any way. For example:

```
class Cookie {
    private String recipe() {
        return "Default cookie recipe";
    }
}
```

```
class ChocolateChipCookie extends Cookie {
    private String recipe() {
        return "Chocolate chip cookie recipe";
    }
}
```

In this case, `Cookie` and `ChocolateChipCookie` each have their own separate `recipe`. Calling `recipe` on a `ChocolateChipCookie` instance will always invoke the subclass version, never the superclass one.

Another source of confusion is the difference between hiding and overriding static methods. When you redeclare a `static` method in a subclass, it's called hiding, not overriding. The subclass method hides the superclass method, but doesn't actually override it.

The key difference is that overriding is a runtime concept, while hiding is a compile-time concept. With overriding, the specific method invoked is determined by the actual object type at runtime. But with hiding, the method invoked is determined by the reference type at compile-time.

Here's an example to illustrate this:

```
class Cookie {
    public static String bake() {
        return "Cookie is baked";
    }
}

class ChocolateChipCookie extends Cookie {
    public static String bake() {
        return "Chocolate chip cookie is baked";
    }
}
```

Now, consider the following code:

```
Cookie obj1 = new Cookie();
System.out.println(obj1.bake());  // Output: "Cookie is baked"

ChocolateChipCookie obj2 = new ChocolateChipCookie();
System.out.println(obj2.bake());  // Output: "Chocolate chip cookie is baked"

Cookie obj3 = new ChocolateChipCookie();
System.out.println(obj3.bake());  // Output: "Cookie is baked"
```

In the last case, even though `obj3` is actually a `ChocolateChipCookie` instance at runtime, the reference type is `Cookie`. So it invokes the hidden `Cookie` method, not the overridden `ChocolateChipCookie` one.

Similar to static methods, variables can be hidden in subclasses. If a subclass declares a variable with the same name as a variable in the superclass, it hides the superclass variable within the scope of the subclass.

Here's an example:

```
class Cookie {
    protected int size = 10;
}

class ChocolateChipCookie extends Cookie {
    private int size = 20;
}
```

In this case, the `size` variable in `ChocolateChipCookie` hides the `size` variable from `Cookie`. Any reference to `size` within `ChocolateChipCookie` will access the subclass variable, not the superclass one.

But here's the tricky part. The hidden superclass variable doesn't go away. It's still there, and can be accessed through a superclass reference. Consider this:

```
Cookie cookie = new ChocolateChipCookie();
System.out.println(cookie.size);  // Output: 10
```

Even though `cookie` is actually a `ChocolateChipCookie` instance, the variable is declared as type `Cookie`. So it accesses the hidden `Cookie` variable, not the `ChocolateChipCookie` one.

This can lead to a lot of confusion and subtle bugs. In general, it's best to avoid hiding variables altogether. If you need to override a superclass variable, consider using a getter/setter method instead, which can be properly overridden.

Finally (pun intended), let's talk about the `final` keyword. When applied to a method, `final` prevents that method from being overridden in subclasses. It essentially locks the method, ensuring that its implementation remains constant throughout the hierarchy.

A common misconception is that `final` methods can't be accessed by subclasses at all. That's not true. Subclasses can still call and use `final` methods; they just can't override them.

For example:

```
class Cookie {
    public final void bake(int temp) {
        System.out.println("Baking at " + temp);
    }
}

class ChocolateChipCookie extends Cookie {
    // Attempting to override bake() will cause a compile error
    // @Override
    // public void bake(int temp) { ... }

    public void extras() {
        bake(350);  // Calling the final bake() method is allowed
    }
}
```

The `bake()` method in `Cookie` is `final`, so `ChocolateChipCookie` can't override it. But it can still call `bake()` whenever needed.

So when should you use `final` methods? Only when you have a critical reason to prevent overriding. Overuse of `final` can make your code rigid and hard to extend. In most cases, it's better to leave methods open for overriding, as it promotes flexibility and reusability.

Accessing Java Objects

In the previous chapter, you learned that when declaring a field or a variable, one thing is the reference type, and another thing is the object type.

Taking this into account, there are three main ways to access an object in Java: 1. Using a reference with the same type as the object

2. Using a reference that is a superclass of the object's type
3. Using a reference that defines an interface the object's class implements or inherits

Let's dive into each of these in more detail.

Using a Reference with the Same Type as the Object.

The most straightforward way to access an object is by using a reference variable that matches the object's type exactly.

Consider this class:

```
class Dog {
    public void bark() {
        System.out.println("Woof!");
    }
}
```

And this code:

```
Dog myDog = new Dog();
myDog.bark(); // Can access all public methods of Dog
```

Here, `myDog` is a reference variable of type `Dog`, and it's referring to a `Dog` object. With this setup, we can access any `public` method or variable defined in the `Dog` class directly through the `myDog` reference.

If you're wondering if polymorphism is happening when a reference type and object type are the same, the answer is yes. Even with matching types, polymorphism is still at play under the hood. The reference type determines what methods you can call, but the actual object type determines which implementation of those methods gets used at runtime.

Using a Reference that is a Superclass of the Object.

Things get a bit more interesting when we bring inheritance into the picture. In Java, it's perfectly valid to have a reference variable with a type that is a superclass of the actual object type.

Consider this class and its subclass:

```
class Animal {
    public void eat() {
        System.out.println("Animal is eating.");
    }
}

class Dog extends Animal {
    public void eat() {
        System.out.println("Dog is eating.");
    }

    public void bark() {
        System.out.println("Woof!");
    }
}
```

We can have something like this:

```
Animal myAnimal = new Dog();
```

Here, we have a reference of type `Animal` referring to a `Dog` object. Since `Dog` extends `Animal`, this is allowed. But what does this mean for accessing the object's functionality?

When you have a superclass reference to a subclass object, you can access any methods defined in the superclass, but not methods that are unique to the subclass. So in the above example, we could call `myAnimal.eat()` as `eat()` is defined in `Animal`, but we couldn't call `myAnimal.bark()` as `bark()` is only defined in `Dog`. The reference type restricts you to the methods that type defines. However, Java does give us a way around this: casting.

If you're certain your superclass reference is pointing to a specific subclass object, you can cast the reference to that subclass type and then call the subclass methods:

```
Dog myDog = (Dog) myAnimal; // Casting from Animal to Dog
myDog.bark(); // Now we can call Dog-specific methods
```

Casting essentially says, "I know this seems to be an `Animal`, but trust me, it's really a `Dog`." Of course, you need to be careful, if you try to cast to the wrong subclass, you'll get a `ClassCastException` at runtime.

We'll continue looking at casting in the next section, but in summary, superclass references give you flexibility (you can use a `Dog` anywhere an `Animal` is expected) but they restrict direct access to subclass-specific functionality. This is a key aspect of polymorphism in Java.

Using a Reference that Defines an Interface the Object Implements.

The third way to access an object in Java is through an interface reference. If a class implements an interface, you can refer to instances of that class using a reference variable of the interface type.

Consider this interface and its implementations:

```
interface Pet {
    void play();
}

class Dog implements Pet {
    public void play() {
        System.out.println("Dog is playing!");
    }

    public void bark() {
        System.out.println("Woof!");
    }
}

class Cat implements Pet {
    public void play() {
        System.out.println("Cat is playing!");
    }

    public void meow() {
        System.out.println("Meow!");
    }
}
```

This way, we can have something like this:

```
Pet myPet = new Dog();
```

In this example, `Dog` implements the `Pet` interface, so we can create a `Pet` reference and point it to a `Dog` object.

Now, you might be thinking, does creating an interface reference to an object mean I can only use the methods defined in the interface? And the answer is yes. When you have an interface reference, you can only directly call methods that are defined in that interface, even if the actual object has other methods available.

```
myPet.play(); // Valid, play() is defined in Pet
myPet.bark(); // Not valid, bark() is not part of Pet
```

This might seem limiting, but it's actually a powerful feature. By programming to an interface, you can write more flexible, maintainable code. You can change the actual object type (for example, from `Dog` to `Cat`) without having to change any code that uses the interface reference:

```
Pet myPet = new Dog();
myPet.play(); // Output: Dog is playing!

myPet = new Cat();
myPet.play(); // Output: Cat is playing!
```

The key point in this example is that the `myPet` reference doesn't care whether it's dealing with a `Dog` or a `Cat`. It just knows it's working with some `Pet`. We can change the actual object type from `Dog` to `Cat`, and the `play` method still works without any changes.

But what if you need to access methods that are specific to the actual object type? Just like with superclass references, you can use casting:

```
Dog myDog = (Dog) myPet; // Casting from Pet to Dog
myDog.bark(); // Now we can call Dog-specific methods
```

Again, you need to be certain that your interface reference is actually pointing to a `Dog` object before you perform this casting, or you'll get a runtime exception.

And remember, interfaces do not have instances, you can't create an object of an interface type directly. However, any object of a class that implements the interface can be referred to using the interface type. In that sense, the object *is-a* form of the interface type.

It's also worth remembering that a single class can implement multiple interfaces. If a class implements multiple interfaces, you can use a reference of any of those interface types to refer to instances of the class:

```
interface Trainable {
    void doTrick();
}

class Dog implements Pet, Trainable {
    // Implement methods from both interfaces
}

Pet myPet = new Dog();
Trainable myStudent = (Trainable) myPet;
```

In this example, a single `Dog` object can be referred to as both a `Pet` and as a `Trainable`, because `Dog` implements both interfaces.

So, interface references provide a way to write more abstract, flexible code. They allow you to focus on a specific set of behaviors that an object can perform, regardless of its actual class type. This is a fundamental principle of object-oriented design.

One final note, remember that interfaces are not part of an object's inheritance hierarchy. They are a separate construct. So, while a `Dog` object can be referred to as `Pet`, a `Pet` reference is not a superclass of `Dog`. It's a distinct type of relationship.

Type Casting

To understand type casting, you can think of variables as actors. Each variable has a specific role to play, determined by its data type. But sometimes, just like in a movie, a variable needs to take on a new role temporarily to fit the needs of a particular scene in your code. This is where type casting comes in.

For primitive types, type casting allows you to assign a value of one primitive data type to another type. In the case of objects, it allows you to treat an object of one class as an object of another class, as long as there

is an inheritance relationship between the two classes.

So, does casting an object change its actual type? Not exactly. When you cast an object, you're not altering its underlying type, instead, you're merely treating it as a different type temporarily for a specific context. It's like an actor putting on a costume for a scene. Underneath, they're still the same person, but they're playing a different role for that moment. Once the cast is over, the variable reverts back to its original type. It's like an actor taking off the costume after the scene is done. They're back to being themselves.

Now, you might be wondering, can you cast any type to any other type? After all, it's all just data, right? Well, not quite. Java is a strongly-typed language, which means it has strict rules about type compatibility. You can't arbitrarily cast between unrelated types, like trying to cast an `int` to a `String`. The compiler will give you an error if you try to do something like that.

The rules for type casting in Java are as follows:

1. Casting a reference from a subtype to a supertype doesn't require an explicit cast.
2. Casting a reference from a supertype to a subtype requires an explicit cast.
3. At runtime, an invalid cast of a reference to an incompatible type results in a `ClassCastException` being thrown.
4. The compiler disallows casts to unrelated types.

Let's break these down one by one.

The first rule says that casting a reference from a subtype to a supertype doesn't require an explicit cast. This is known as upcasting. If you have a class hierarchy where class `B` extends class `A`, you can assign a reference of type `B` to a variable of type `A` without an explicit cast:

```
class A {}
class B extends A {}

B b = new B();
A a = b; // upcasting, no explicit cast needed
```

Upcasting is safe because a subclass always contains all the features of its superclass. So treating a subclass object as a superclass object will never cause a problem.

The second rule says that casting a reference from a supertype to a subtype requires an explicit cast. This is known as downcasting. If you have a variable of the supertype and you want to treat it as the subtype, you need to explicitly cast it:

```
A a = new B(); // upcasting
B b = (B) a; // downcasting, explicit cast needed
```

Downcasting is necessary when you want to access methods or variables that are specific to the subclass and not available in the superclass.

However, downcasting comes with a risk. What if the object being referenced is not actually an instance of the subclass you're trying to cast it to? This leads us to the third rule.

At runtime, an invalid cast of a reference to an incompatible type results in a `ClassCastException` being thrown:

```
A a = new A();
B b = (B) a; // Compiles but throws ClassCastException at runtime
```

In this example, `a` is referring to an instance of class `A`, not class `B`. When we try to cast it to `B`, it compiles without error because the compiler allows the possibility that `a` might be referring to a `B` object. But at runtime, when the cast is actually attempted, Java realizes that `a` is not in fact a `B`, and it throws a `ClassCastException`.

This is an important point: casting doesn't magically transform an object into something it's not. If you try to cast an object to an incompatible type, it will result in a runtime exception. Explicit casting basically tells the compiler, "Trust me, I know what I'm doing." But if you're wrong, Java will let you know at runtime.

However, the fourth rule states that the compiler disallows casts to unrelated types. If you try to cast between classes that are not in the same inheritance hierarchy, the compiler will give you an error:

```
class A {}
class C {}

A a = new A();
C c = (C) a; // Compilation error
```

Classes `A` and `C` are not related through inheritance, so the compiler knows that it's impossible for an `A` object to ever be a `C` object. It won't even let this code compile.

So, if casting doesn't work, is it a compile-time problem or a runtime problem? It can be either, depending on the situation. If you try to cast to an unrelated type, it's a compile-time error. If you try to cast to a related type but the object is not actually an instance of that type, it's a runtime exception.

Now, you might be thinking, isn't all this casting dangerous? Doesn't it basically bypass Java's type checking system? Not exactly. Java's type system is still in effect, and the compiler won't let you do anything too unsafe. Explicit casting is a way of telling the compiler that you have additional knowledge about the type of an object, but it's still checked at runtime.

That said, it's generally a good idea to avoid excessive casting, especially downcasting. If you find yourself downcasting a lot, it might be a sign that your class hierarchy needs to be redesigned.

So when is casting actually useful? Upcasting is very common and is an important part of polymorphism in Java. It allows you to treat a more specific type as a more general type, which is safe and often necessary.

For example, let's say you have a method that takes a parameter of type `List`. You can pass in an `ArrayList`, a `LinkedList`, or any other subclass of `List`, and it will work fine due to upcasting.

```
void processNames(List<String> names) {
    // code here
}

ArrayList<String> nameList = new ArrayList<>();
processNames(nameList); // upcasting from ArrayList to List
```

Downcasting is less common and should be used more sparingly. It's necessary when you have a reference to a superclass but you need to access methods or variables that are only available in a subclass.

```
class Shape {
    void draw() { /* ... */ }
}

class Circle extends Shape {
    void drawCircle() { /* ... */ }
}

Shape shape = new Circle();
shape.draw(); // Fine, draw() is defined in Shape
((Circle)shape).drawCircle(); // Downcast to access drawCircle()
```

In this case, the downcast is safe because we know that `shape` is actually referring to a `Circle` object.

In summary, type casting in Java allows you to temporarily treat an object as a different type, either a superclass (upcasting) or a subclass (downcasting), as long as there is an inheritance relationship. Upcasting

is safe and common, while downcasting requires an explicit cast and should be used carefully. The compiler checks for invalid casts to unrelated types, while invalid casts to related types result in a runtime exception. And always remember, underneath the cast, the object itself doesn't change, it's just being viewed through a different lens.

But to be extra safe, you can use the `instanceof` operator to check the type before casting. Let's talk about it next.

The `instanceof` Operator

In Java, the `instanceof` operator is used to test whether an object is an instance of a particular class or implements a specific interface. It returns a `boolean` value: `true` if the object is an instance of the class/interface, `false` otherwise.

The syntax for using `instanceof` is:

```
objectReference instanceof ClassName/InterfaceName
```

For example:

```
Object obj = "Hello";
if(obj instanceof String) {
    System.out.println("obj is a String");
}
```

This will print `"obj is a String"` since the object referenced by `obj` is an instance of the `String` class.

It's important to note that using `instanceof` does not actually change the object or its type in any way. It simply checks the object against the specified class or interface and returns a `boolean` result. `instanceof` cannot be used with primitive types like `int` or `double`, it only works with object references.

Passing the `instanceof` test for a class indicates that the object is an instance of either that class itself or one of its subclasses. All objects in Java inherit from the `Object` class, so `instanceof Object` will always return `true`:

```
String str = "abc";
if(str instanceof Object) {
    System.out.println("This will always print");
}
```

An exception to this rule is when the reference is `null`:

```
String str = null;
if(str instanceof String) {
    System.out.println("This will never be executed");
}
```

`instanceof` can also check if an object implements a particular interface. If a class implements an interface either directly or through inheritance, `instanceof` will return `true` for that interface:

```
interface Trainable {
    void doTrick();
}

interface Pet extends Trainable {
    void play();
}

class Dog implements Pet {
    // Implement methods from both interfaces
}
```

```
Pet dog = new Dog();
if(dog instanceof Pet) {
    System.out.println("A Dog is a Pet");
}
if(dog instanceof Trainable) {
    System.out.println("A Dog is a Trainable");
}
```

Both of these print statements will execute, since `Dog` directly implements `Pet`, and `Pet` extends `Trainable`.

One common use case for `instanceof` is to safely downcast an object before calling a subclass-specific method. Remember, a downcast is when you cast a reference from a superclass type to a subclass type:

```
Object obj = getSomeObject();
if(obj instanceof String) {
    String str = (String) obj;
    System.out.println(str.toUpperCase());
}
```

Here we first check if `obj` is actually a `String` before downcasting and calling the `String` specific `toUpperCase()` method. The explicit cast `(String)` is required even though we already confirmed the type with `instanceof`.

However, we can use pattern matching for the `instanceof` operator to streamline the process of checking and casting object types.

So instead of an explicit cast, you can combine the type check and cast in a single operation using the following syntax:

```
if (objectReference instanceof ClassName variableName) {
    // Use variableName here, which is automatically cast to ClassName
}
```

This syntax checks whether `objectReference` is an instance of `ClassName`. If it is, `objectReference` is cast to `ClassName`, and the cast object is assigned to `variableName` within the scope of the `if` statement. If the check fails, no exception is thrown. The code within the block simply doesn't execute, and the pattern variable remains inaccessible. This eliminates the need for an explicit cast and reduces boilerplate code.

Here's the previous downcast example rewritten to use pattern matching:

```
Object obj = getSomeObject();
if(obj instanceof String str) {
    System.out.println(str.toUpperCase());
}
```

In this example, `str` is the pattern variable that is automatically cast to `String` if `obj` is an instance of `String`. Pattern variables are implicitly initialized upon a successful match. No additional casting is required.

Pattern variables have a limited scope. They are only accessible where their matching is guaranteed. `str` in the above example is not available outside the `if` block. This design choice ensures that pattern variables are only used in contexts where their types are assured, eliminating a common source of errors.

However, this does not always mean that the scope is the `if` block where they are defined. When using pattern matching with `instanceof`, if the condition is `true`, meaning the object is an instance of the specified type, the pattern variable is indeed scoped to and accessible within the block that follows the condition. However, consider this example, where the pattern matching is used with a negation:

```
Object obj = getSomeObject();
if (!(obj instanceof String str)) {
    // The pattern variable str is NOT accessible here
    return "";
```

```
}
// But, because the execution only reaches this point if str IS an instance of String,
// the pattern variable str is accessible here.
return str.toUpperCase();
```

In this example, the `if` statement checks if `obj` is not an instance of `String`. If `obj` is not a `String`, the method returns `false` immediately, and the pattern variable `str` is not accessible within the `if` block because the condition for its instantiation (`obj` being an instance of `String`) is `false`.

However, immediately after this `if` block, the code execution continues only if `obj` is indeed an instance of `String`, which means `str` was successfully matched and is now accessible and usable outside of, but directly after, the if block that contains the pattern matching. This is a specific scenario where the flow of the program ensures that the pattern variable `str` is instantiated and can be used safely because the method would have exited early if the condition were `false`.

You can also use a pattern variable this way:

```
Object obj = getSomeObject();
if(obj instanceof String str && str.length() > 3) {
    System.out.println(str.toUpperCase());
}
```

Because, being the conditional-AND operator (`&&`) short-circuiting, the program can reach the `str.length() > 3` expression only if the `instanceof` expression returns `true`.

However, you can't use an OR operator (`||`):

```
Object obj = getSomeObject();
if(obj instanceof String str || str.length() > 3) { // Error
    System.out.println(str.toUpperCase());
}
```

This will result in an error because the `str.length() > 3` expression may execute when `obj` is not an instance of `String`, leading to an attempt to access `str` when it may not have been initialized.

Also, pattern matching with `instanceof` is designed for one type at a time. It simplifies the process for a single type check and cast but doesn't extend to multiple types simultaneously:

```
Object obj = getSomeObject();

if (obj instanceof String str) {
    // obj is a String, use str here
    System.out.println("String length: " + str.length());
} else if (obj instanceof Integer intVal) {
    // obj is an Integer, use intVal here
    System.out.println("Integer value: " + intVal);
} else if (obj instanceof List<?> list) {
    // obj is a List, use list here
    System.out.println("List size: " + list.size());
}
```

In this example, `obj` is checked against multiple types: `String`, `Integer`, and `List`. Depending on the actual type of `obj`, the corresponding block of code executes. Within each block, the object `obj` is automatically cast to the type being checked, and you can use the cast object directly without an explicit cast.

This approach keeps your code clean and type-safe, allowing for more readable and maintainable code when dealing with multiple possible types for a single object reference.

It's generally good practice to use `instanceof` sparingly and prefer polymorphism where possible. Frequent `instanceof` checks can be a sign of poor object-oriented design. But it does have valid uses for safely

downcasting, reflective code, and some equality comparisons.

Finally, here are two other key facts about `instanceof`:

- Child classes are considered instances of their parent classes, but parent classes are not considered instances of their child classes.
- It can check for implementations of interfaces, but cannot distinguish between direct implementation in a class vs inherited implementation from a parent class.

Encapsulation

What is Encapsulation?

Encapsulation is one of the fundamental principles of object-oriented programming in Java. It involves bundling data (attributes) and methods (behavior) that operate on that data within a single unit (like a class) and restricting access to the inner workings of the class from the outside.

Encapsulation in Java can be thought of like a vending machine. Just as you interact with a vending machine using the provided buttons to select your snack or drink, without needing to understand or access the internal mechanisms that actually dispense the item, encapsulation allows you to interact with an object through its public methods, while the internal state and implementation details remain hidden and protected from external interference.

The main purpose of encapsulation is to protect the data from unauthorized access and modification, and to separate the interface of a class (how it can be used) from the implementation (how it actually works internally). By encapsulating the internal state of an object, we ensure that it cannot be put into an invalid or inconsistent state by external code.

Some programmers might wonder, "Can't I just make everything public to simplify the coding process? Why bother hiding class internals?"

While this approach may seem simpler in the short term, it quickly leads to inflexible, fragile, and hard-to-maintain code. Encapsulation helps manage complexity by reducing interdependencies between different parts of a program. When a class is well encapsulated, changes to its internal implementation do not affect the rest of the codebase, allowing for easier maintenance, refactoring, and updating of the class without causing ripple effects throughout the program.

So how exactly do we implement encapsulation in Java? The primary mechanism is through the use of access modifiers on class members.

Remember, there are four access modifiers that determine the visibility and accessibility of classes, fields, and methods: - `private`: Only accessible within the same class.

- `default` (package-private): Accessible within the same class and from any other class in the same package.
- `protected`: Accessible within the same class, from any other class in the same package, and from subclasses (even in different packages).
- `public`: Accessible from anywhere.

You can apply these modifiers to classes, attributes, and methods according to the following table:

Access Modifier	Class/Interface	Class Attribute	Class Method	Interface Attribute	Interface Method
public					
private					
protected					
default					

And here's the summary of the rules of access modifiers:

Access Modifier	Same Class	Subclass (Same Package)	Subclass (Different Package)	Another Class (Same Package)	Another Class (Different Package)
public					
private					
protected					
default					

To encapsulate a class, we typically: 1. Declare the fields (instance variables) of the class as `private`. This prevents direct access to the fields from outside the class.

2. Provide `public` getter methods to retrieve the values of the fields, and setter methods to modify them, if needed. These methods provide controlled access to the fields and allow for adding validation, logging, or any other logic when the field values are accessed or modified.

Here's an example of a well-encapsulated `BankAccount` class:

```
public class BankAccount {
    private String accountNumber;
    private double balance;

    public String getAccountNumber() {
        return accountNumber;
    }

    public double getBalance() {
        return balance;
    }

    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        } else {
            throw new IllegalArgumentException("Deposit amount must be positive.");
        }
    }

    public void withdraw(double amount) {
        if (amount > balance) {
            throw new IllegalArgumentException("Insufficient funds.");
        } else if (amount < 0) {
            throw new IllegalArgumentException("Withdrawal amount must be positive.");
        } else {
            balance -= amount;
        }
    }
}
```

And a diagram to visualize it:

```
┌──────────────────────────────────────┐
│              BankAccount             │
├──────────────────────────────────────┤
│ - accountNumber: String              │
```

```
| - balance: double                          |
|--------------------------------------------|
| + getAccountNumber(): String               |
| + getBalance(): double                     |
| + deposit(amount: double): void            |
| + withdraw(amount: double): boolean        |
└--------------------------------------------┘
```

In this example, the `accountNumber` and `balance` fields are declared `private`, so they cannot be directly accessed or modified from outside the `BankAccount` class. The public `getAccountNumber()` and `getBalance()` methods allow for controlled retrieval of these field values, while the `deposit()` and `withdraw()` methods enable controlled modification of the `balance` field with added validation logic.

Now, you might wonder, "If I use getters and setters for all my fields, does that automatically mean my class is well-encapsulated?"

Not necessarily. While using getters and setters is a common way to encapsulate fields, simply having these methods does not guarantee good encapsulation. Encapsulation is about more than just hiding data. It's about ensuring that the internal state of an object is always valid and consistent. Getters and setters are just one tool for achieving this.

For example, consider this `Rectangle` class:

```java
public class Rectangle {
    private double width;
    private double height;

    public double getWidth() {
        return width;
    }

    public void setWidth(double width) {
        this.width = width;
    }

    public double getHeight() {
        return height;
    }

    public void setHeight(double height) {
        this.height = height;
    }

    public double getArea() {
        return width * height;
    }
}
```

While this class uses getters and setters, it's not really well-encapsulated. The `width` and `height` can be set to any value, including negative numbers, which doesn't make sense for a rectangle. A better approach would be to validate the input in the setters:

```java
public void setWidth(double width) {
    if (width > 0) {
        this.width = width;
    } else {
        throw new IllegalArgumentException("Width must be positive.");
```

```
    }
}

public void setHeight(double height) {
    if (height > 0) {
        this.height = height;
    } else {
        throw new IllegalArgumentException("Height must be positive.");
    }
}
```

By adding this validation logic, we ensure that the internal state of the `Rectangle` object is always valid, thus achieving better encapsulation.

In summary, encapsulation is about managing complexity, protecting data integrity, and separating the interface of a class from its implementation. It is achieved primarily through the use of access modifiers, with `private` fields and `public` getters and setters being a common pattern. However, good encapsulation goes beyond just using getters and setters; it requires carefully designing the `public` interface of a class and ensuring that its internal state is always valid and consistent.

Immutable Objects

In object-oriented programming, immutability is the ability to create objects whose state cannot be changed after they are created.

Immutable objects in Java are like a printed book: once the content is published (or the object is created), it cannot be altered. Just as you can't change the words on a printed page without creating a new book, you can't modify an immutable object without creating a new instance with the desired changes.

So what makes an object immutable in Java? It's not as simple as just omitting setter methods. There are several key requirements:

1. Mark the class as `final` or make all of the constructors `private`. This prevents subclassing, which could otherwise allow mutability to sneak in.
2. Mark all the instance variables `private` and `final`. This ensures the state can't be modified directly from outside the class. But is this alone sufficient for immutability?
3. Don't define any setter methods. Any method that modifies state, even indirectly, breaks immutability.
4. Don't allow referenced mutable objects to be modified. If your class holds a reference to a mutable object (like a `Date` or a `Collection`), you must ensure that reference can't be used to change the object's state.
5. Use a constructor to set all properties of the object, making a defensive copy if needed. Once an immutable object is constructed, its state can never change. The constructor must establish the invariants.

Let's dive deeper into each of these requirements.

Marking the class as `final` prevents it from being subclassed. If we allowed subclassing, a subclass could add mutable state or override methods to be mutable, breaking the immutability contract.

```
public final class ImmutableExample {
    // class definition here
}
```

Alternatively, we can make the constructors `private` and control instantiation through factory methods:

```
public class ImmutableExample {
    private ImmutableExample() {
```

```
        // private constructor
    }

    public static ImmutableExample create() {
        return new ImmutableExample();
    }
}
```

But making a class `final` doesn't automatically make it immutable. We also need to ensure all its fields are `private` and `final`:

```
public final class ImmutableExample {
    private final int value;

    public ImmutableExample(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}
```

By making the fields `private`, we prevent direct access from outside the class. And by making them `final`, we ensure they can only be set once, in the constructor.

But even with private `final` fields, immutability can still be violated if the class has methods that change state:

```
public final class NotActuallyImmutable {
    private final int value;

    public NotActuallyImmutable(int value) {
        this.value = value;
    }

    public void setValue(int value) {
        this.value = value; // Mutates state - not okay!
    }
}
```

To be truly immutable, a class must not have any setter methods or any other methods that change its fields after construction.

However, immutability goes beyond just the immediate state of the object. An immutable object's state includes the state of any other objects it holds references to.

Consider this class:

```
public final class NotImmutable {
    private final Date start;

    public NotImmutable(Date start) {
        this.start = start;
    }

    public Date getStart() {
        return start;
```

```
    }
}
```

At first glance, it might seem immutable, the start field is `private` and `final`, and there are no setters. But the `Date` class is mutable. Someone could do this:

```
NotImmutable example = new NotImmutable(new Date());
example.getStart().setTime(0); // Mutates the internal state of example!
```

To fix this, we need to make a defensive copy of the `Date` in the constructor:

```
public final class ActuallyImmutable {
    private final Date start;

    public ActuallyImmutable(Date start) {
        this.start = new Date(start.getTime()); // Defensive copy
    }

    public Date getStart() {
        return new Date(start.getTime()); // Defensive copy
    }
}
```

Now the state of the `ActuallyImmutable` instance cannot be changed through the reference it holds.

The same principle applies to collections and arrays, if an immutable class holds a reference to a mutable collection or array, it must defensively copy it and provide no way for the internal collection to be modified.

Proper use of constructors is also key to immutability. An immutable object's state should be fully defined by the arguments passed to its constructor. And the constructor must establish all invariants of the object.

This means that an immutable class shouldn't have a no-arg constructor, because then its state wouldn't be fully defined at the end of construction. All properties should be set via constructor arguments.

Here's an example of an immutable class with a collection:

```
public final class ImmutableCollection {
    private final List<String> strings;

    public ImmutableCollection(List<String> strings) {
        this.strings = List.copyOf(strings); // Immutable copy
    }

    public List<String> getStrings() {
        return strings;
    }
}
```

By following these rules, making the class and fields final, providing no mutator methods, defensively copying mutable components, and setting all state in the constructor, we can create truly immutable objects in Java.

Immutable objects have many advantages, especially in concurrent contexts. Because their state never changes, they are inherently thread-safe. They can be freely shared between threads without synchronization.

They are also simpler to reason about, because you know their state will always remain the same. And they can serve as building blocks for more complex thread-safe structures.

However, immutability does come with some costs. Immutable objects can be more expensive to create, because they often require making defensive copies. And if you need to make any changes, you have to create a new instance, which can be costly for large objects.

Key Points

- Variable scope refers to the visibility and accessibility of a variable in code. The four main scopes in Java are local variables, method parameters, fields (instance variables), and class variables (static fields).
- Local variables are declared inside a method or block and are only accessible within that block. They come into scope at their declaration and go out of scope at the end of the enclosing block.
- Method parameters are also considered local variables, with a scope that covers the entire method body. They come into scope when the method is called and go out of scope when the method completes.
- Fields, or instance variables, are variables declared at the class level. They come into scope when the object is instantiated and remain in scope as long as the object is in memory.
- Class variables, or static fields, are static variables declared at the class level. They come into scope when the class is loaded and stay in scope until the program ends.
- Fields automatically get default values if not explicitly initialized, while local variables must be explicitly initialized before use.
- The `var` keyword allows for local variable type inference. The compiler infers the type based on the initializer expression.
- Inheritance allows a new class to be based on an existing class, inheriting its attributes and methods. The `extends` keyword is used to create a subclass.
- Abstract classes cannot be instantiated and are intended to be subclassed. They can contain abstract methods, which have no implementation in the abstract class and must be implemented by concrete subclasses.
- Interfaces define a contract of methods that a class must implement. The `implements` keyword is used to implement an interface. A class can implement multiple interfaces.
- Sealed classes restrict which other classes can extend them. Permitted subclasses are specified using the `permits` keyword. Subclasses of a sealed class must be declared `final`, `sealed`, or `non-sealed`.
- The `this` keyword is a reference to the current instance of a class. It's used to disambiguate between local variables and instance variables, to pass the current instance as a method argument, and to call another constructor from within a constructor.
- The `super` keyword is a reference to the parent class (superclass) of the current class. It's used to access the superclass's members and to invoke the superclass constructor from the subclass constructor.
- Polymorphism allows you to treat objects of different subclasses as if they were objects of the same superclass.
- Method overriding is a key concept in polymorphism, where a subclass provides its own implementation of a method defined in the superclass.
- To properly override a method, the method in the subclass must have the same name, return type, and parameter list as the method in the superclass.
- When overriding methods, you can make the access modifier more lenient in the subclass, but not more restrictive. The overridden method can also only declare exceptions that are the same or more specific than those declared by the superclass method.
- An overridden method is allowed to have a covariant return type, meaning the return type can be a subclass of the return type declared in the superclass method.
- The `@Override` annotation explicitly marks methods that are intended to override a superclass method and provides a safeguard against accidental errors.

- Redeclaring a private method from the superclass in a subclass is not considered overriding. Private methods are not inherited.
- Redeclaring a static method in a subclass is called hiding, not overriding. The subclass method hides the superclass method, but doesn't actually override it.
- Variables can also be hidden in subclasses if a subclass declares a variable with the same name as a variable in the superclass.
- There are three main ways to access an object in Java: using a reference with the same type as the object, using a reference that is a superclass of the object's type, and using a reference that defines an interface the object's class implements or inherits.
- Type casting allows you to assign a value of one primitive data type to another type or treat an object of one class as an object of another class, as long as there is an inheritance relationship between the two classes.
- Casting a reference from a subtype to a supertype (upcasting) doesn't require an explicit cast, while casting a reference from a supertype to a subtype (downcasting) requires an explicit cast.
- The `instanceof` operator is used to test whether an object is an instance of a particular class or implements a specific interface. It returns a `boolean` value.
- Pattern matching for the `instanceof` operator allows you to combine the type check and cast in a single operation, reducing boilerplate code.

Practice Questions

1. What is the result of compiling and executing the following code?

```
void myMethod() {
    int x = 1;
    if (x > 0) {
        int y = 2;
        System.out.println(x + y);
    }
    System.out.println(x);
    System.out.println(y);
}
```

A) The code compiles and outputs `3` followed by `1`.
B) The code compiles and outputs `3` followed by `1` and an undefined value for `y`.
C) The code does not compile because `y` is accessed outside of its scope.
D) The code compiles but throws a runtime exception when trying to print `y`.

2. Which of the following variable declarations statements are valid? (Choose all that apply.)

A) `double x, double y;`
B) `int i = 0, String s = "hello";`
C) `float f1 = 3.14f, f2 = 6.28f;`
D) `char a = 'A', b, c = 'C';`

3. Which of the following statements are true regarding the use of `var` in Java? (Choose all that apply.)

A) `var` can be used to declare both local variables within methods and instance variables within classes.
B) The use of `var` is restricted to local variables within methods, constructors, or initializer blocks.
C) `var` can be used to declare method parameters.
D) `var` enhances readability by inferring types where it's clear from the context, but it's not allowed in method signatures to maintain clarity.
E) `var` can be used to declare class (static) variables.

4. Which of the following statements correctly describe the use of inheritance in Java? (Choose all that apply.)

A) Subclasses can only access `protected` and `public` members of their superclass directly.
B) In Java, a class can extend multiple classes to achieve multiple inheritance.
C) The `extends` keyword is used in Java to create a subclass that inherits from a superclass.
D) A subclass in Java can directly access `private` members of its superclass.

5. Consider the following code snippet:

```
abstract class Animal {
    abstract void eat();
}

class Dog extends Animal {
    void eat() {
        System.out.println("Dog eats");
    }
}

class Cat extends Animal {
    void eat() {
        System.out.println("Cat eats");
    }
}

public class Test {
    public static void main(String[] args) {
        Animal myAnimal = new Dog();
        myAnimal.eat();
    }
}
```

Which of the following statements is true regarding the above code? Choose all that apply.

A) The code will compile and print `"Dog eats"` when executed.
B) The `Animal` class can be instantiated.
C) Removing the `eat` method from the `Dog` class will cause a compilation error.
D) The `Cat` class is necessary for the code to compile and run.

6. Consider the following interfaces:

```
interface Walkable {
    int distance = 10;
    void walk();
}

interface Runnable {
    void run();
    default void getSpeed() {
        System.out.println("Default speed");
    }
}

class Person implements Walkable, Runnable {
    public void walk() {
        System.out.println("Walking...");
```

```
    }
    public void run() {
        System.out.println("Running...");
    }
}
```

Which of the following statements is true?

A) The `Person` class must override the `getSpeed` method.
B) The `distance` variable in the `Walkable` interface is implicitly `public`, `static`, and `final`.
C) A `Person` object can call the `getSpeed` method without any implementation in the `Person` class.
D) The `Runnable` interface causes a compilation error due to a naming conflict with `java.lang.Runnable`.

7. Consider the following code snippet related to sealed classes:

```
sealed abstract class Shape permits Circle, Square {
    abstract double area();
}

final class Circle extends Shape {
    private final double radius;

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

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

non-sealed class Square extends Shape {
    private final double side;

    Square(double side) {
        this.side = side;
    }

    public double area() {
        return side * side;
    }
}

public class TestShapes {
    public static void main(String[] args) {
        Shape shape = new Circle(10);
        System.out.println("Area: " + shape.area());
    }
}
```

Which of the following statements is true?

A) The `Shape` class is correctly defined as a sealed class, allowing only specified classes to extend it.
B) The `Square` class does not correctly extend the `Shape` class because it is not marked as `final`.
C) The `Circle` class can be further extended by other classes.
D) The `area` method in the `Shape` class must provide a default implementation.

8. Consider the following class:

```
public class Widget {
    private int size;

    public Widget() {
        this(10); // Line 5
    }

    public Widget(int size) {
        this.size = size;
    }

    public void resize(int size) {
        if (size > this.size) {
            this.size = size; // Line 14
            updateWidget();
        }
    }

    private void updateWidget() {
        System.out.println("Widget updated to size " + this.size);
    }

    public static void main(String[] args) {
        Widget widget = new Widget();
        widget.resize(15);
    }
}
```

In line 114, what does the `this` keyword represent in the context of the `Widget` class?

A) A reference to the `static` context of the class, allowing access to static methods and fields.
B) A special variable that stores the return value of a method.
C) An optional keyword that can always be omitted without affecting the functionality of the code.
D) A reference to the current object, whose instance variable is being called.

9. Consider the following classes:

```
class Animal {
    String name;

    Animal(String name) {
        this.name = name;
    }

    protected void eat() {
        System.out.println("Animal eats");
    }
}

class Dog extends Animal {
    Dog(String name) {
        super(name);
    }
```

```
    @Override
    protected void eat() {
        super.eat();
        System.out.println(name + " (Dog) eats");
    }
}

public class TestAnimal {
    public static void main(String[] args) {
        Animal myDog = new Dog("Buddy");
        myDog.eat();
    }
}
```

Which of the following statements are true regarding the use of `super` in the above code? (Choose all that apply.)

A) The `super` keyword is used in the `Dog` constructor to call the superclass constructor.
B) The `eat` method in the `Dog` class uses `super` to invoke the superclass's `eat` method.
C) Removing the `super.eat();` call in the `Dog` class's `eat` method will prevent the `Dog` class from compiling.
D) The `super` keyword can be used to access `static` methods from the superclass.

10. Consider the following classes:

```
class Vehicle {
    public void drive(int speed) {
        System.out.println("Vehicle driving at speed: " + speed);
    }
}

class Car extends Vehicle {
    @Override
    public void drive(long speed) {
        System.out.println("Car driving at speed: " + speed);
    }
}

public class TestDrive {
    public static void main(String[] args) {
        Vehicle myCar = new Car();
        myCar.drive(60);
    }
}
```

What is the result of compiling and executing the above code?

A) It compiles and prints `"Car driving at speed: 60"`.
B) It does not compile because the `drive` method cannot be called using a `Vehicle` reference.
C) It does not compile because the `drive` method in the `Car` class does not properly override the `drive` method in the `Vehicle` class.
D) It compiles and prints `"Vehicle driving at speed: 60"` because the `drive` method in the `Car` class is an overload, not an override.

11. Consider the following code snippet:

```
class Fruit {
    public void flavor() {
        System.out.println("Fruit flavor");
```

```
    }
}

class Apple extends Fruit {
    @Override
    public void flavor() {
        System.out.println("Apple flavor");
    }

    public void color() {
        System.out.println("Red");
    }
}

public class TestFruit {
    public static void main(String[] args) {
        Fruit myFruit = new Apple();
        myFruit.flavor();
        // myFruit.color();
    }
}
```

If the commented line `// myFruit.color();` is uncommented, what will be the result of compiling and executing the above code?

A) It compiles and prints `"Apple flavor"` followed by `"Red"`.
B) It compiles and prints `"Fruit flavor"`.
C) It compiles but throws a runtime exception when attempting to call `color()`.
D) It does not compile because `Apple` is not a valid type of `Fruit`.
E) It does not compile because the `color` method is not defined in the `Fruit` class.

12. Consider the following code snippet:

```
class Animal {}

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

class Cat extends Animal {
    public void meow() {
        System.out.println("Meow");
    }
}

public class TestCasting {
    public static void main(String[] args) {
        Animal animal = new Dog();
        ((Dog)animal).bark();

        Animal anotherAnimal = new Animal();
        // Line 1
    }
}
```

Which of the following lines of code, if inserted independently at Line 1, will compile without causing a runtime exception? (Choose all that apply.)

A) `((Dog)anotherAnimal).bark();`
B) `if (anotherAnimal instanceof Dog) ((Dog)anotherAnimal).bark();`
C) `((Cat)animal).meow();`
D) `if (anotherAnimal instanceof Cat) ((Cat)anotherAnimal).meow();`

13. Consider the following code snippet:

```
public class AdvancedPatternMatching {
    public static void process(Object input) {
        if (input instanceof String s && s.contains("Java")) {
            System.out.println("String with Java: " + s);
        } else if (input instanceof Integer i && i > 10) {
            System.out.println("Integer greater than 10: " + i);
        }
    }

    public static void main(String[] args) {
        process("Hello Java!");
        process(15);
        process("Just a string");
        process(5);
    }
}
```

Given the above code, which statement accurately describes its execution result?

A) It compiles and prints `"String with Java: Hello Java!"` followed by `"Integer greater than 10: 15"`.
B) It compiles but only prints `"String with Java: Hello Java!"` because integers are not supported with pattern matching.
C) It does not compile because pattern matching in `instanceof` cannot be combined with logical operators like `&&`.
D) It compiles but prints all four lines due to incorrect use of pattern matching that always evaluates to `true`.

14. Consider the encapsulation practices in the following class structure:

```
package store;

public class Product {
    private String name;
    private double price;
    private int stock;

    public Product(String name, double price, int stock) {
        setName(name);
        setPrice(price);
        setStock(stock);
    }

    public String getName() {
        return name;
    }

    private void setName(String name) {
```

```
        this.name = name;
    }

    public double getPrice() {
        return price;
    }

    private void setPrice(double price) {
        if (price >= 0) {
            this.price = price;
        }
    }

    public int getStock() {
        return stock;
    }

    private void setStock(int stock) {
        if (stock >= 0) {
            this.stock = stock;
        }
    }
}
```

Which statement is true regarding the encapsulation of the `Product` class?

A) Making the `setName`, `setPrice`, and `setStock` methods `public` would enhance the class's encapsulation.
B) The class is not encapsulated because the `Product` class's fields are `private`.
C) Encapsulation is weakened because the constructor allows direct setting of fields without validation.
D) The `Product` class should have package-private getters to improve encapsulation.
E) The class is properly encapsulated by providing `public` getters for all fields and `private` setters with validation, ensuring control over the state of its objects.

15. Consider the following classes defined in the same package:

```
class Account {
    private double balance;

    Account(double initialBalance) {
        if (initialBalance > 0) {
            balance = initialBalance;
        }
    }

    void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }

    protected double getBalance() {
        return balance;
    }
}

public class SavingsAccount extends Account {
```

```
    private double interestRate;

    public SavingsAccount(double initialBalance, double interestRate) {
        super(initialBalance);
        this.interestRate = interestRate;
    }

    public void applyInterest() {
        double interest = getBalance() * interestRate / 100;
        deposit(interest);
    }
}
```

Which statement(s) about encapsulation principles and the use of access modifiers accurately describes the code above? Choose all tha apply.

A) The `SavingsAccount` class cannot access the `balance` field directly due to its `private` access modifier in the `Account` class.
B) The `getBalance` method should be `public` to allow `SavingsAccount` to access the account balance.
C) The `deposit` method in the `Account` class should be marked as `final` to prevent overriding.
D) The `interestRate` field in the `SavingsAccount` class violates encapsulation principles by being `private`.
E) The `Account` class correctly encapsulates the `balance` field, and `SavingsAccount` adheres to encapsulation by accessing `balance` through `getBalance` and `deposit`.

16. Consider the following class:

```
public final class Contact {
    private final String name;
    private final String email;
    private final Address address;

    public Contact(String name, String email, Address address) {
        this.name = name;
        this.email = email;
        this.address = new Address(address.getStreet(), address.getCity());
    }

    public String getName() {
        return name;
    }

    public String getEmail() {
        return email;
    }

    public Address getAddress() {
        return new Address(address.getStreet(), address.getCity());
    }

    public static class Address {
        private final String street;
        private final String city;

        public Address(String street, String city) {
            this.street = street;
            this.city = city;
```

```
        }

        public String getStreet() {
            return street;
        }

        public String getCity() {
            return city;
        }
    }
}
```

Given the above implementation, which statement accurately describes the `Contact` object?

A) The `Contact` object is mutable because the `Address` class is not `final`.
B) The `Contact` object is immutable, but only because it does not provide setters.
C) The `Contact` object is immutable, and it properly prevents leakage of mutable internal state through defensive copying.
D) The `Contact` object is mutable because the `Address` object can be changed via the `getAddress` method.
E) The `Contact` object is immutable but fails to prevent access to its mutable internal state.

Chapter TWO

Utilizing Java Object-Oriented Approach - Part 2

Answers

1. The correct answer is C.

Explanation:

- **A)** The code compiles and outputs `3` followed by `1`.
  - This option is incorrect because, although the code prints `3` followed by `1` due to `x` being in scope, attempting to access `y` outside of its declaring block (the `if` block) will cause a compile-time error.
- **B)** The code compiles and outputs `3` followed by `1` and an undefined value for `y`.
  - This option is incorrect because Java does not allow access to local variables (`y` in this case) outside of their scope. The notion of an "undefined value for `y`" is not applicable here; the compiler will simply not compile the code.
- **C)** The code does not compile because `y` is accessed outside of its scope.
  - This is the correct option. Local variable `y` is declared inside the `if` block, and thus, it is only accessible within that block. Trying to access it outside of its scope, as done in the last `System.out.println(y);`, causes a compile-time error, specifically saying that `y` cannot be found.
- **D)** The code compiles but throws a runtime exception when trying to print `y`.
  - This option is incorrect because the issue with the code is at compile time, not runtime. The compiler will not allow the code to compile due to the scope violation of the local variable `y`, hence a runtime exception regarding `y` is out of the question.

2. The correct answers are C and D.

Explanation:

- **A)** `double x, double y;`
  - This option is incorrect because when declaring multiple variables of the same type in a single statement, you do not repeat the type before each variable. The correct syntax would be `double x, y;`.
- **B)** `int i = 0, String s = "hello";`

  - This option is incorrect for the same reason as A; you cannot declare variables of different types (`int` and `String` in this case) in a single statement.
- **C)** `float f1 = 3.14, f2 = 6.28f;`
  - This is correct. You can declare multiple variables of the same type (`float` in this case) in a single statement, and it's also fine to initialize them with values in the same statement.
- **D)** `char a = 'A', b, c = 'C';`
  - This is correct. It's valid to declare multiple variables of the same type (`char` in this case), and initialize some, all, or none of them in the same statement.

3. The correct answers are B and D.

Explanation:

- **A)** `var` can be used to declare both local variables within methods and instance variables within classes.
  - This option is incorrect because `var` cannot be used to declare instance variables. It is specifically restricted to local variables within methods, constructors, or initializer blocks, as using `var` for fields would reduce the clarity of a class's public API.
- **B)** The use of `var` is restricted to local variables within methods, constructors, or initializer blocks.
  - This option is correct. `var` is intended for local variable type inference, significantly reducing the verbosity of Java code in scenarios where the compiler can easily determine the type of the local variable from its initializer. Its use is restricted to ensure clarity and prevent ambiguity in more complex constructs like class fields or method parameters.
- **C)** `var` can be used to declare method parameters.
  - This option is incorrect. The example clearly demonstrates that `var` cannot be used to declare method parameters. This limitation ensures that method signatures remain explicit in their type requirements, a critical aspect of a class's contract with its callers.
- **D)** `var` enhances readability by inferring types where it's clear from the context, but it's not allowed in method signatures to maintain clarity.
  - This option is correct. While `var` is primarily used to improve code readability by reducing the need for explicit type declarations where the type can be inferred from the context, it is not allowed in method signatures. This restriction ensures that the types of parameters in methods are always explicitly defined, aiding in the readability and maintainability of public APIs.
- **E)** `var` can be used to declare class (static) variables.
  - This option is incorrect. Similar to instance variables, `var` is not permissible for declaring class (`static`) variables. The rationale behind this restriction aligns with the goal of maintaining explicit type declarations in the class's structure, ensuring the class's design remains clear and unambiguous to both the compiler and developers.

4. The correct answers are A and C.

Explanation:

- **A)** The `extends` keyword is used in Java to create a subclass that inherits from a superclass.
  - This option is correct. Subclasses can directly access `protected` and `public` members of their superclass. This accessibility allows subclasses to leverage and extend the functionality provided by the superclass while maintaining encapsulation of `private` members.
- **B)** In Java, a class can extend multiple classes to achieve multiple inheritance.
  - This option is incorrect. Java does not support multiple inheritance for classes. A class in Java can only extend one other class, preventing complications like the diamond problem and the complexity associated with multiple inheritance.
- **C)** Subclasses can only access `protected` and `public` members of their superclass directly.
  - This option is correct. The `extends` keyword is indeed used to define a subclass that inherits properties and behaviors from a single superclass, establishing an *is-a* relationship between the subclass and the superclass. This is a fundamental concept in Java's implementation of inheritance.
- **D)** A subclass in Java can directly access `private` members of its superclass.
  - This option is incorrect. A subclass cannot directly access `private` members of its superclass. Instead, it can access them through `public` or `protected` accessors provided by the superclass.

This encapsulation principle ensures a controlled interaction with the superclass's state.

5. The correct answers are A and C.

Explanation:

- **A)** The code will compile and print `"Dog eats"` when executed.
  - This option is correct. The `Dog` class has provided an implementation for the `eat` method, which is abstract in the superclass `Animal`. Since `myAnimal` is of type `Animal` but instantiated as a `Dog`, it will call the overridden `eat` method in the `Dog` class, printing `"Dog eats"`.
- **B)** The `Animal` class can be instantiated.
  - This option is incorrect. The `Animal` class is abstract and cannot be instantiated. Attempting to create an instance of `Animal` directly (`new Animal()`) would result in a compilation error.
- **C)** Removing the `eat` method from the `Dog` class will cause a compilation error.
  - This option is correct. Since `Dog` extends the abstract class `Animal` and `Animal` has an abstract `eat` method, `Dog` must provide an implementation for `eat`. Failing to do so will prevent the code from compiling because `Dog` would also be considered abstract.
- **D)** The `Cat` class is necessary for the code to compile and run.
  - This option is incorrect. The `Cat` class is not referenced in the `main` method or anywhere else in the provided code snippet. Thus, it is unnecessary for the compilation and execution of the given code segment.

6. The correct answers are B and C.

Explanation:

- **A)** The `Person` class must override the `getSpeed` method.
  - This option is incorrect. The `Person` class is not required to override the `getSpeed` method because it is a default method in the `Runnable` interface. Default methods provide an implementation that can be used or overridden by implementing classes, but overriding is not mandatory.
- **B)** The `distance` variable in the `Walkable` interface is implicitly `public`, `static`, and `final`.
  - This option is correct. In Java, all variables declared in an interface are implicitly `public`, `static`, and `final`. This means the `distance` variable in the `Walkable` interface is a constant and must be initialized at the point of declaration. It is accessible with the interface name, like `Walkable.distance`.
- **C)** A `Person` object can call the `getSpeed` method without any implementation in the `Person` class.
  - This option is correct. Since the `Runnable` interface provides a default implementation for the `getSpeed` method, a `Person` object can call the `getSpeed` method without any additional implementation in the `Person` class itself. The default implementation from the interface will be used.
- **D)** The `Runnable` interface causes a compilation error due to a naming conflict with `java.lang.Runnable`.
  - This option is incorrect because Java fully supports namespace resolution. The `Runnable` interface declared in the code snippet and `java.lang.Runnable` exist in different packages. There is no compilation error unless there's an attempt to import both in the same file without using a fully qualified name. Plus, this situation does not directly relate to the functionality or declaration of interfaces per the exam's focus.

7. The correct answer is A.

Explanation:

- **A)** The `Shape` class is correctly defined as a sealed class, allowing only specified classes to extend it.
  - This option is correct. The `Shape` class is declared as a sealed class, which means it can be extended only by the classes it explicitly permits through the `permits` clause. In this case, `Shape` permits `Circle` and `Square` to extend it, and both classes are correctly defined as permitted subclasses.
- **B)** The `Square` class does not correctly extend the `Shape` class because it is not marked as `final`.
  - This option is incorrect. There's no requirement for classes extending a sealed class to be marked as `final` if they are non-sealed. The keyword `non-sealed` explicitly allows the `Square` class to extend the sealed `Shape` class without being final, indicating it can be further extended.

- **C)** The `Circle` class can be further extended by other classes.
  - This option is incorrect. The `Circle` class is declared as `final`, which means it cannot be extended further and aligning with the constraints of extending a sealed class where the permitted subclass can be final, sealed, or non-sealed.
- **D)** The `area` method in the `Shape` class must provide a default implementation.
  - This option is incorrect. Abstract classes like `Shape` are not required to provide implementations for their abstract methods. The purpose of an abstract class is to define a template that its subclasses will follow, which includes implementing any abstract methods declared in the abstract class.

8. The correct answer is D.

Explanation:

- **A)** A reference to the `static` context of the class, allowing access to static methods and fields.
  - This option is incorrect. The `this` keyword does not refer to the static context of the class. It specifically refers to the current instance of the class. Static methods and fields belong to the class itself and are not part of any instance, so they cannot be accessed through `this`.
- **B)** A special variable that stores the return value of a method.
  - This option is incorrect. The `this` keyword does not store the return value of a method. It is used within an instance method or a constructor to refer to the current object the method or constructor is being invoked upon.
- **C)** An optional keyword that can always be omitted without affecting the functionality of the code.
  - This option is incorrect. While it is true that in some cases `this` can be omitted (for example, when accessing instance fields or methods without any naming conflict), its use is necessary for situations like constructor chaining (`this()` call) or when the method parameter names shadow the instance field names. In such scenarios, `this` clarifies to which variable the code is referring.
- **D)** A reference to the current object, whose instance variable is being called.
  - This option is correct. The `this` keyword in Java is used to refer to the current object—the object whose instance variable, method, or constructor is being called. You can see its usage in line 5 to call another constructor within the same class, in line 14 to differentiate between the method parameter `size` and the instance variable `size`, and in the `updateWidget` method to access the instance variable `size`. This usage demonstrates `this` as a way to refer explicitly to properties or methods of the current object.

9. The correct answers are A and B.

Explanation:

- **A)** The `super` keyword is used in the `Dog` constructor to call the superclass constructor.
  - This optional is correct. In the `Dog` constructor, `super(name);` is used to call the superclass (`Animal`) constructor with the `name` parameter. This is necessary to initialize the `name` field inherited from the `Animal` class in the `Dog` instance.
- **B)** The `eat` method in the `Dog` class uses `super` to invoke the superclass's `eat` method.
  - This optional is correct. The `eat` method in the `Dog` class calls `super.eat();` to invoke the `eat` method defined in the superclass (`Animal`). This allows the `Dog` class to extend the functionality of the `eat` method beyond what is defined in the superclass, demonstrating method overriding and use of `super` to access the overridden method.
- **C)** Removing the `super.eat();` call in the `Dog` class's `eat` method will prevent the `Dog` class from compiling.
  - This optional is incorrect. Removing the `super.eat();` call from the `Dog` class's `eat` method would not prevent the class from compiling. It would simply mean that the `Dog` class's `eat` method no longer calls the superclass's `eat` method, altering the program's behavior but not its compilability.
- **D)** The `super` keyword can be used to access `static` methods from the superclass.
  - This optional is incorrect. While `super` can indeed be used to access superclass methods, it's not specifically used or necessary for accessing static methods. Static methods belong to the class, not to instances, and should be invoked using the class name. `super` is used primarily for instance

methods and constructors.

10. The correct answer is D.

Explanation:

- **A)** It compiles and prints `"Car driving at speed: 60"`.
  - This option is incorrect because the `drive` method in the `Car` class has a different parameter type (`long`) than the method in the `Vehicle` class (`int`). Due to the difference in parameter types, the `Car` class's `drive` method does not override but rather overloads the `Vehicle` class's `drive` method. Since the method is called on a `Vehicle` reference, the `Vehicle` class's `drive` method is invoked.
- **B)** It does not compile because the `drive` method cannot be called using a `Vehicle` reference.
  - This option is incorrect because `Vehicle` defines the `drive` method correctly.
- **C)** It does not compile because the `drive` method in the `Car` class does not properly override the `drive` method in the `Vehicle` class.
  - This option is incorrect because the code does compile. The `@Override` annotation does not cause a compile-time error here because it is not strictly enforced in terms of method overloading (changing the parameter type creates a new method signature, making this a valid overload).
- **D)** It compiles and prints `"Vehicle driving at speed: 60"` because the `drive` method in the `Car` class is an overload, not an override.
  - This option is correct. The `drive` method in the `Car` class has a different signature from the `drive` method in the `Vehicle` class due to the parameter type (`int` vs. `long`). Therefore, the `drive` method in the `Car` class overloads the superclass method rather than overriding it. When a `Vehicle` reference calls the `drive` method with an `int` argument, it invokes the `Vehicle` class's `drive` method, not the `Car` class's method.

11. The correct answer is E.

Explanation:

- **A)** It compiles and prints `"Apple flavor"` followed by `"Red"`.
  - This option is incorrect because, while the `flavor` method will indeed print `"Apple flavor"` due to polymorphism (the `Apple` class overrides the `flavor` method of `Fruit`), the code will not compile if the `color()` method is called on a `Fruit` reference. This is because the `color` method is not part of the `Fruit` class's interface.
- **B)** It compiles and prints `"Fruit flavor"`.
  - This option is incorrect for a similar reason to A. The `flavor` method would print `"Apple flavor"` because of the overridden method in the `Apple` class, not `"Fruit flavor"`. However, the presence of the `color()` method call would still prevent compilation.
- **C)** It compiles but throws a runtime exception when attempting to call `color()`.
  - This option is incorrect because the issue occurs at compile time, not runtime. The Java compiler will not allow a method to be called on a reference type if that method is not defined in the reference type's class or its superclass hierarchy.
- **D)** It does not compile because `Apple` is not a valid type of `Fruit`.
  - This option is incorrect. `Apple` is a valid type of `Fruit` due to inheritance (`Apple extends Fruit`). This relationship allows an `Apple` object to be referenced by a `Fruit` variable.
- **E)** It does not compile because the `color` method is not defined in the `Fruit` class.
  - This option is correct. The `color` method is only defined in the `Apple` class and not in the `Fruit` class. Since the reference type of `myFruit` is `Fruit`, which does not have a `color` method, attempting to call `myFruit.color()` will result in a compilation error. This illustrates a key principle of polymorphism: the type of the reference (not the object) determines what methods can be called.

12. The correct answers are B and D.

Explanation:

- **A)** `((Dog)anotherAnimal).bark();`

  - This option is incorrect because it tries to cast `anotherAnimal` to `Dog` without checking its actual type first. Since `anotherAnimal` is an instance of `Animal` (not `Dog`), attempting this cast will compile, but it will cause a `ClassCastException` at runtime.
- **B)** `if (anotherAnimal instanceof Dog) ((Dog)anotherAnimal).bark();`
  - This option is correct. It uses `instanceof` to check whether `anotherAnimal` is an instance of `Dog` before attempting the cast and calling `bark()`. In this case, since `anotherAnimal` is not an instance of `Dog`, the check prevents the cast and method call, avoiding a `ClassCastException`.
- **C)** `((Cat)animal).meow();`
  - This option is incorrect because it casts `animal` to `Cat` and attempts to call `meow()`. Since `animal` is actually an instance of `Dog`, this cast will compile but will result in a `ClassCastException` at runtime.
- **D)** `if (anotherAnimal instanceof Cat) ((Cat)anotherAnimal).meow();`
  - This option is correct. It checks if `anotherAnimal` is an instance of `Cat` before casting it to `Cat` and calling `meow()`.

13. The correct answer is A.

Explanation:

- **A)** It compiles and prints `"String with Java: Hello Java!"` followed by `"Integer greater than 10: 15"`.
  - This option is correct. The code snippet effectively demonstrates the use of pattern matching with the `instanceof` operator for both `String` and `Integer` types. The pattern matching feature checks if `input` is an instance of `String` or `Integer` and binds it to a variable (`s` for `String` and `i` for `Integer`) within the scope of the `if` and `else if` blocks. The logical operator `&&` is correctly used to further conditionally check properties of the variables (`s.contains("Java")` and `i > 10`). Thus, the method `process` prints output for inputs that are a `String` containing `"Java"` and an `Integer` greater than `10`, respectively.
- **B)** It compiles but only prints `"String with Java: Hello Java!"` because integers are not supported with pattern matching.
  - This option is incorrect because pattern matching works for any reference type, including `Integer`. The code does support integers and performs additional checks using pattern matching correctly.
- **C)** It does not compile because pattern matching in `instanceof` cannot be combined with logical operators like `&&`.
  - This option is incorrect. The code will compile and run as expected. Pattern matching in `instanceof` can indeed be combined with logical operators like `&&` for additional checks in the same conditional statement, as demonstrated in the code snippet.
- **D)** It compiles but prints all four lines due to incorrect use of pattern matching that always evaluates to `true`.
  - This option is incorrect because the use of pattern matching in the provided code is correct and does not always evaluate to `true`. The code correctly prints specific messages only for the inputs that match the given conditions.

14. The correct answer is E.

Explanation:

- **A)** Making the `setName`, `setPrice`, and `setStock` methods public would enhance the class's encapsulation.
  - This option is incorrect. Making the setters public would actually reduce the class's encapsulation by allowing external classes to modify the fields without restriction, potentially bypassing any validation logic contained within the setters.
- **B)** The class is not encapsulated because the `Product` class's fields are `private`.
  - This option is incorrect. The use of `private` fields is a fundamental aspect of encapsulation. It prevents external classes from directly accessing and modifying the object's state, thus enforcing encapsulation.
- **C)** Encapsulation is weakened because the constructor allows direct setting of fields without validation.

  - This option is incorrect. The constructor does not weaken encapsulation; instead, it uses `private` setters that contain validation logic. This ensures that the object's state is correctly managed and validated upon creation.
- **D)** The `Product` class should have package-private getters to improve encapsulation.
  - This option is incorrect. Making getters package-private would limit the class's usability and does not inherently improve encapsulation. Public getters are necessary for external classes to view (but not modify) the object's state.
- **E)** The class is properly encapsulated by providing public getters for all fields and private setters with validation, ensuring control over the state of its objects.
  - This option is correct. The `Product` class demonstrates proper encapsulation practices by making its fields `private` and controlling access to them through `public` getters and `private` setters. The setters include validation logic, ensuring that only valid states are assigned to the fields. This design pattern ensures that the internal state of `Product` instances is both protected and correctly managed.

15. The correct answers are A and E.

Explanation:

- **A)** The `SavingsAccount` class cannot access the `balance` field directly due to its `private` access modifier in the `Account` class.
  - This option is correct. The design intentionally restricts direct access to the `balance` field to maintain encapsulation.
- **B)** The `getBalance` method should be `public` to allow `SavingsAccount` to access the account balance.
  - This option is incorrect. Making `getBalance public` would increase its visibility unnecessarily. `protected` is sufficient for subclass access, and this change is not required for `SavingsAccount` to function correctly, making this statement incorrect.
- **C)** The `deposit` method in the `Account` class should be marked as `final` to prevent overriding.
  - This option is incorrect. Marking `deposit` as `final` would prevent it from being overridden in subclasses, which is not a requirement or suggestion indicated by the given code. The decision to make a method `final` should be based on the specific design needs rather than a general principle of encapsulation.
- **D)** The `interestRate` field in the `SavingsAccount` class violates encapsulation principles by being `private`.
  - This option is incorrect. Using a `private` access modifier for `interestRate` in `SavingsAccount` is an example of proper encapsulation. It restricts access to the field from outside the class, which is aligned with encapsulation principles, making this option incorrect.
- **E)** The `Account` class correctly encapsulates the `balance` field, and `SavingsAccount` adheres to encapsulation by accessing `balance` through `getBalance` and `deposit`.
  - This option is correct. The `Account` class uses `private` access for the `balance` field to encapsulate its state, providing `protected` and package-private methods (`getBalance` and `deposit`) for controlled access and modification. `SavingsAccount` respects this encapsulation by using these methods to interact with the `balance` field, demonstrating a proper understanding and application of encapsulation principles. This design allows `SavingsAccount` to leverage functionality provided by `Account` without breaking encapsulation, which is a key objective in object-oriented design.

16. The correct answer is C.

Explanation:

- **A)** The `Contact` object is mutable because the `Address` class is not `final`.
  - This option is incorrect because the `Address` class does not directly impact the immutability of the `Contact` object. The `Contact` class ensures its immutability by not providing setters and by making deep copies of mutable objects, such as `Address`, both in the constructor and the getter.
- **B)** The `Contact` object is immutable, but only because it does not provide setters.
  - This option is incorrect. While it's true that it does not provide setters, this option does not fully capture the essence of immutability. Thus, it doesn't highlight the fact that all fields in `Contact`

are `final` and the defensive copying strategy.

- **C)** The `Contact` object is immutable, and it properly prevents leakage of mutable internal state through defensive copying.
  - This is the correct option. The `Contact` class is immutable because it meets all criteria for immutability: the class is declared as `final` (preventing subclassing), all its fields are `private` and `final`, and it does not provide any setters. Also, it implements defensive copying for the mutable `Address` field to ensure that the internal state cannot be altered by external changes to `Address` objects passed in or returned. This prevents the leakage of its mutable internal state.
- **D)** The `Contact` object is mutable because the `Address` object can be changed via the `getAddress` method.
  - This option is incorrect because the `Contact` object's immutability is maintained through defensive copying. The `getAddress` method returns a new `Address` instance each time it is called, ensuring that the original `Address` object's state cannot be altered from outside the `Contact` object.
- **E)** The `Contact` object is immutable but fails to prevent access to its mutable internal state.
  - This option is incorrect because the `Contact` object does implement a strategy to prevent access to its mutable internal state: it uses defensive copying for the `Address` object in both the constructor and the getter method, which ensures that the internal state remains unchanged from outside modifications.

Chapter THREE

Working with Records and Enums

Chapter Content

- Records
  - Introducing Records
  - Record Immutability
  - Initializing Records
  - Customizing Records
- Enums
  - Introducing Enums
  - Declaring an Enum
  - Special Methods of an Enum
  - Customizing Enums
- Key Points
- Practice Questions

Records

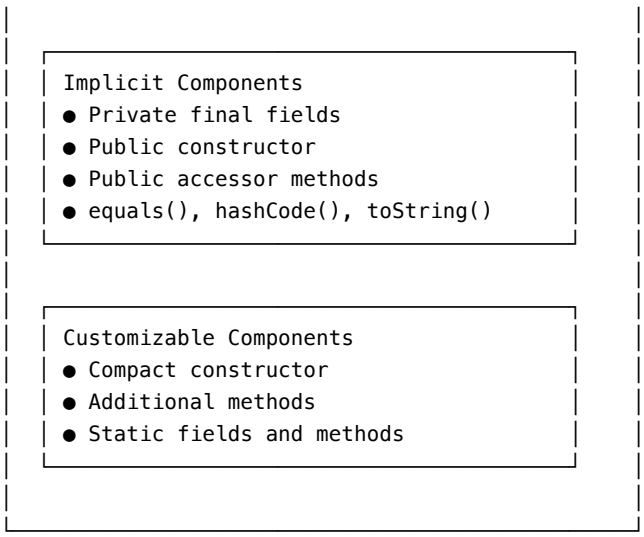
Introducing Records

Records provide a more concise way to declare classes that are primarily intended as simple data carriers. You can think of records as a special type of class that is specifically designed to store immutable data, kind of like a sturdy, tamper-proof safe for your information.

But what exactly are records? Well, essentially, a record is a final class that automatically generates a constructor, `private final` fields for the parameters you define, and implementations of the `equals()`, `hashCode()`, and `toString()` methods based on those fields. This means that records give you a shorthand way to create a class that encapsulates data, without having to write a lot of repetitive boilerplate code.

Here's a diagram that shows the basic structure and components of a record declaration:

```
┌──────────────────────────────────────────────┐
| public record Person(String name, int age) {  |
```



And here's an example of a record definition:

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

With just this one line, we've defined a `Person` record that has two fields: `name` and `age`. The record automatically generates a constructor that takes those fields as parameters, so we can create instances of the record like this:

```
Person john = new Person("John Doe", 30);
```

One important thing to understand about records is that they are not just a shorthand for writing classes. While they do provide a more concise syntax, records have some unique characteristics that set them apart from regular classes. One of the most significant is that records are implicitly final, which means they cannot be extended by other classes. This reinforces their purpose as simple, immutable data carriers.

Additionally, records are implicitly static when they are declared as nested types. This means that they do not have a reference to the instance of the enclosing class:

```
public class OuterClass {

    // Nested record
    public record NestedRecord(int value) {
    }

    // ...
}
```

In the example, `NestedRecord` is a record nested inside `OuterClass`. It is implicitly static, meaning it can be instantiated without an instance of `OuterClass`:

```
OuterClass.NestedRecord nestedRecord = new OuterClass.NestedRecord(8);
```

So, when should you use a record instead of a class? Records are ideal for situations where you need to represent a simple, immutable data structure, like a point with x and y coordinates, or a person with a name and age. In these cases, using a record can save you a lot of time and reduce the verbosity of your code:

```
record Point(int x, int y) {}
```

On the other hand, if you need a more complex data structure that requires additional behavior or mutable state, a regular class is still the way to go. Records are not meant to replace classes entirely, but rather to complement them by providing a streamlined solution for a specific use case.

Record Immutability

One of the defining characteristics of records is their immutability. When we say that records are immutable, it means that once an instance of a record is created, its state cannot be changed. This is enforced by the fact that all fields in a record are implicitly final, which means they must be initialized when the record is instantiated and cannot be modified afterward.

```
record Person(String name, int age) {
    void birthday() {
        age++; // Compile-time error: Cannot assign a value to final variable age
    }
}
```

Since records are designed to be immutable, there's no way to make individual fields mutable. If you find yourself needing to modify the values of fields after instantiation, it's a good indication that a record might not be the right choice for your use case, and a regular class would be more appropriate.

Regarding immutability, there are some reasons why records are often preferred over mutable objects:

1. Records are inherently thread-safe because their state cannot be modified after creation, eliminating the risk of concurrent access issues.
2. Records are simpler to reason about and less prone to bugs because their state remains constant throughout their lifetime.
3. Records can be safely shared and reused without the need for defensive copying.

However, it's important to note that immutability in records only applies to the record itself and its fields. If a record contains a reference to a mutable object, such as a list or an array, that object can still be modified even though the record itself is immutable:

```
record Numbers(List<Integer> values) {}

Numbers numbers = new Numbers(new ArrayList<>(List.of(1, 2, 3)));
numbers.values().add(4); // The list can still be modified
```

In this example, even though the `Numbers` record is immutable, the `List` stored in its `values` field can still be modified because it is a mutable object.

So, when designing your records, it's important to consider the immutability of the objects they contain. If you want to ensure complete immutability, you should use immutable objects or defensive copying techniques when storing mutable objects inside your records.

Initializing Records

Previously, you saw how records automatically generate a constructor based on the record components. This default constructor is sufficient for many use cases, but there are times when you might need more control over the initialization process. Fortunately, records provide several ways to customize the constructor and add your own initialization logic.

The long constructor, also known as the canonical constructor, is the default constructor generated by the record. It takes all the record components as parameters in the order they are declared.

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

Person john = new Person("John Doe", 30);
```

In this example, the `Person` record has a default constructor that takes a `String` for the `name` and an `int` for the `age`.

If you need to validate or preprocess any of the fields before they are assigned, you can use a compact constructor. This constructor does not specify parameters explicitly. Instead, you write the constructor without parameters, and the compiler understands that it should use the record's parameters. Inside the compact

constructor, you can add validation or transformation logic. However, unlike the canonical constructor, you don't assign values to the fields directly, this is handled automatically.

Here's an example of a compact constructor for the `Person` record:

```
record Person(String name, int age) {
    public Person {
        if (age < 0) {
            throw new IllegalArgumentException("Age cannot be negative");
        }
    }
}
```

The constructor body contains a validation check to ensure that the `age` is not negative. If an invalid age is provided, an `IllegalArgumentException` is thrown.

Records also support constructor overloading, which means you can define multiple constructors with different parameter lists. This can be useful when you want to provide alternative ways to initialize a record.

However, each of these constructors must delegate to the canonical constructor (either directly or indirectly through another custom constructor) to ensure all fields are initialized. This is usually done with the `this()` call, passing the necessary parameters.

Here's an example of a custom/overloaded constructor:

```
record Person(String name, int age) {
    public Person(String name) {
        this(name, 0);
    }
}

Person john = new Person("John Doe", 30);
Person jane = new Person("Jane Smith");
```

In this example, we've added an overloaded constructor that takes only the `name` parameter. Inside the constructor, we call the canonical constructor using `this()`, passing the provided `name` and a default `age` of `0`.

This approach allows you to: - Define custom constructors to initialize the record in different ways, giving you flexibility in how you create instances of the record. - Add your own initialization logic and validation checks using compact constructors or overloaded constructors.

Customizing Records

While records are straightforward to use out of the box, Java provides a few ways to customize them to fit your needs.

Instance Methods Although records are primarily designed to carry data, this doesn't mean they can't have behavior. Just like regular classes, you can add instance methods to records to encapsulate logic that operates on the record's components. Here's an example:

```
public record Point(int x, int y) {
    public double distance(Point other) {
        int dx = x - other.x;
        int dy = y - other.y;
        return Math.sqrt(dx * dx + dy * dy);
    }
}
```

In this case, the `Point` record has an instance method `distance()` that calculates the Euclidean distance between itself and another `Point`. The method can access the record's components `x` and `y` directly.

You can also override methods inherited from the `Object` class, such as `equals()`, `hashCode()`, and `toString()`. By default, records provide sensible implementations of these methods based on the record's components, but you can customize them if needed:

```
public record Person(String name, int age) {
    @Override
    public String toString() {
        return name + " (" + age + " years old)";
    }
}
```

In this example, `toString()` is overridden to offer a more human-readable representation of a `Person` record.

However, when overriding `equals()` and `hashCode()`, be careful to maintain consistency with the automatically generated implementations. The record's components should be included in the equality comparison and hash code computation to ensure that two records with the same component values are considered equal and have the same hash code.

Nested Types Records can contain nested classes, interfaces, annotations, enums, and even other records. This allows you to group related types together within the record, enhancing encapsulation and readability. For example:

```
public record Employee(String name, Department department) {
    public class Department {
        // Implementation of the class
    }

    public static record Manager(String name) {
        // Additional fields and methods for managers
    }
}
```

In this example, the `Employee` record has a nested class `Department` representing, for example, the different departments an employee can belong to. It also has a nested static record `Manager`, which may have additional fields and methods specific to managers.

Nested types declared within a record are implicitly static, so they can be accessed using the record name followed by the type name, like `Employee.Department` or `Employee.Manager`.

Generics and Type Parameters Records can be generic and accept type parameters, just like classes and interfaces. This allows you to create records that can work with different data types while still maintaining type safety. Here's an example of a generic `Pair` record:

```
public record Pair<T, U>(T first, U second) { }
```

You can then create instances of the `Pair` record with specific types:

```
Pair<String, Integer> nameAge = new Pair<>("Alice", 30);
```

Generic records work seamlessly with Java's type system, including wildcards, bounded type parameters, and type inference. We'll talk more about generics in another chapter.

Local Records In addition to being declared at the class level, records can also be declared locally within methods. This can be handy when you need a temporary data structure with a limited scope. Here's an example:

```
public void processCoordinates() {
    record Coordinate(int x, int y) { }

    Coordinate point1 = new Coordinate(10, 20);
    Coordinate point2 = new Coordinate(30, 40);

    // Process the coordinates...
}
```

The `Coordinate` record is declared inside the `processCoordinates()` method and is only accessible within that method.

Implementing Interfaces Although records are primarily designed for data encapsulation, they can still implement interfaces. This allows records to satisfy contracts and be used in contexts where a specific interface is required. Here's an example:

```
public interface Drawable {
    void draw();
}

public record ColoredPoint(int x, int y, String color) implements Drawable {
    @Override
    public void draw() {
        System.out.println("Drawing a " + color + " point at (" + x + ", " + y + ")");
    }
}
```

In this case, the `ColoredPoint` record implements the `Drawable` interface and provides an implementation for the `draw()` method.

Restrictions First of all, records cannot extend classes or be extended by other classes. This restriction enforces the idea that records are standalone data carriers and not part of an inheritance hierarchy. However, records can implement interfaces, as shown earlier.

Another important restriction is that records do not allow additional instance fields outside of the ones defined in the record declaration. The record's components are the only instance fields allowed. For example:

```
public record Point(int x, int y) {
    private int z; // Compilation error: field declaration must be static
}
```

Adding extra instance fields like `z` in this example will result in a compilation error. The purpose of this restriction is to maintain the record's immutability and keep its state tied solely to its components.

The need for additional instance fields indicates that a regular class may be more suitable than a record. Records are meant to be lightweight data carriers, not complex objects with mutable state.

However, it's important to note that the error message specifically mentions that the field declaration must be static. So, if we modify the example to make `z` a `static` field:

```
public record Point(int x, int y) {
    private static int z; // Compiles successfully
}
```

This version of the `Point` record will compile without issues. However, keep in mind that static fields are shared across all instances of the record, so they don't contribute to the record's individual state.

Another thing to keep in mind is that records do not support instance initializers. If you try to add an instance initializer block to a record, like this:

```
public record Point(int x, int y) {
    // Instance initializer block
    {
        System.out.println("Initializing Point...");
    } // Compiler error: instance initializers not allowed in records
}
```

The Java compiler will throw an error. The reason behind this restriction is that records are designed to be simple and immutable, and instance initializers can introduce complex initialization logic that may violate these principles.

If you need to perform additional initialization logic, you can use a compact constructor instead:

```
public record Point(int x, int y) {
    public Point {
        System.out.println("Initializing Point...");
    }
}
```

A compact constructor allows you to execute code at the time of the record's instantiation while still ensuring that the record's components are properly initialized.

However, static initializers are allowed. The following will compile without errors:

```
public record Point(int x, int y) {
    // Static initializer block
    static {
        System.out.println("Initializing Point...");
    }
}
```

Why?

Static initializers are allowed in records for the same reasons they are allowed in other classes: to initialize static fields or to perform static initialization blocks that run when the class is loaded.

So while you can add instance methods and static fields and static initializer blocks, you can't add instance fields or instance initializer blocks, because these could break immutability.

Remember, records are not a replacement for regular classes but rather a complementary feature for specific use cases where immutable data carriers are needed.

Enums

Introducing Enums

In Java, an enumeration (or enum) is a special type of class used to define a set of predefined constants. It's a way to give names to numeric values, making your code more readable and maintainable.

Think of an enum like a VIP list for an exclusive event. The list (enum) defines who's allowed in (the predefined constants), but each person on the list can also have their own unique attributes (fields) and actions they can perform (methods). The process of adding someone to the list with their specific attributes is similar to using a constructor in an enumeration.

Let's say you're creating an application to manage a pet shop. You might have a variable to represent the type of animal:

```
String animalType;
//...
if(animalType.equals("DOG")) {
    // process dog
```

```
} else if(animalType.equals("CAT")) {
    // process cat
} else if(animalType.equals("BIRD")) {
    // process bird
}
```

But this approach has some problems. First, it's error-prone. What if you mistype "DOG" as "DIG" somewhere? The compiler won't catch that. Second, it's not very readable. Someone reading this code might not immediately know what "BIRD" means in the context of your application.

Here's where enums come in:

```
enum AnimalType {
    DOG, CAT, BIRD
}
```

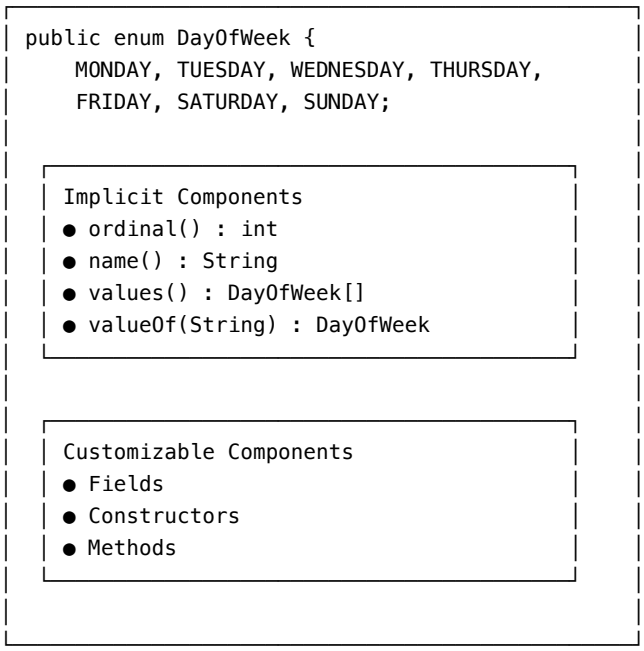
Now you can use the enum like this:

```
AnimalType animalType;
//...
if(animalType == AnimalType.DOG) {
    // process dog
} else if(animalType == AnimalType.CAT) {
    // process cat
} else if(animalType == AnimalType.BIRD) {
    // process bird
}
```

If you mistype `DOG`, the compiler will catch it. And it's much more readable.

So in essence, enums provide a way to define a set of named constants, which can make your code more readable, maintainable, and less error-prone.

Here's a diagram that shows the basic structure and components of an enum declaration:



Declaring an Enum

Declaring an enum is similar to declaring a class, but you use the `enum` keyword instead of `class`:

```
public enum AnimalType {
    DOG, CAT, BIRD
}
```

Each constant (`DOG`, `CAT`, `BIRD`) is implicitly `public`, `static`, and `final`. The convention is to use all caps for their names.

It's important to note that enums can only have `public` or default (package-private) access when declaring outside of a class, they cannot be declared with `protected` or `private` access. If an enum is defined within a class, it can have any access level that a regular inner class can have.

Here's an example:

```
public class PetStore {
    // This is okay
    private enum EmployeeLevel {
        TRAINEE, MANAGER, DIRECTOR
    }

    // This is okay
    protected enum AnimalBreed {
        LABRADOR, SIAMESE, PARROT
    }
}

// This is okay
enum AnimalType {
    DOG, CAT, BIRD
}

// This will not compile
private enum FoodType {
    KIBBLE, CANNED, SEEDS
}
```

As you can see, enums declared within a class (`EmployeeLevel`) can have any access modifier that a regular inner class can have. And when an enum is declared outside a class, it must be public or have default access, it cannot be private (`FoodType`).

Also, if you declare an enum in its own file, the enum name should match the filename.

But enums aren't just a list of constants. They can have constructors, methods, and fields, just like a regular class. However, the constructor of an enum is always private, either explicitly or implicitly. By default, if no access modifier is specified, the constructor is implicitly private. Enum constructors cannot be `public` or `protected`. This is because you don't create instances of an enum using `new`. Instead, the instances are predefined.

```
public enum AnimalType {
    DOG("Dog"), CAT("Cat"), BIRD("Bird");

    private String displayName;

    AnimalType(String displayName) {
        this.displayName = displayName;
    }
```

```
    public String getDisplayName() {
        return displayName;
    }
}
```

In this example, each constant is created with a display name, which is passed to the constructor. The constructor is private, which is the default for enums. Each constant is essentially an instance of the enum class.

This answers a few common questions about enums: - Enums can have methods, constructors, and fields in addition to the predefined constants.
- The constructors in an enum are always private (or package-private), even if not explicitly declared so. That's why you can't instantiate an enum using `new`. If you mark the constructor as `public` or `protected`, the compiler will generate an error. - Enums aren't just a list of integer constants. Each enum constant is actually an instance of the enum class, which can have its own state (fields) and behavior (methods).

Another important thing to note is that all enums implicitly extend `java.lang.Enum`. This is a special class in Java that provides some built-in methods for enums.

Because of this implicit extension, an enum can't extend any other class. However, it can implement interfaces.

Special Methods of an Enum

An enum class implicitly declares some `public static` methods that are quite useful and that are not obvious at first sight, like the `values()` and `valueOf()` methods.

For example, assuming we have this enum:

```
enum Season {
    WINTER, SPRING, SUMMER, FALL;
}
```

The `public static T[] values()` method returns an array containing all the constants of the enum class, in the same order they are declared. This method is commonly used to iterate over all the constants. For example:

```
for(Season s : Season.values()) {
    System.out.println(s);
}
```

Outputs:

```
WINTER
SPRING
SUMMER
FALL
```

You might be wondering where this method comes from, as it is not mentioned on the javadoc for the enum class. The answer is that the Java compiler automatically adds it to the enum class during compilation. So in a way, it's like syntactic sugar provided by the language.

The `public static T valueOf(String)` method returns the enum constant with the specified name. The name must match exactly an identifier used to declare the constant in the enum class. For example:

```
Season s = Season.valueOf("SUMMER");
```

Apart from those, each enum constant also has a `name()` method to get the name of the constant as declared in the enum, and an `ordinal()` method to get its position in the declaration order (starting from `0`). For example:

```
Season.WINTER.name();    // "WINTER"
Season.SPRING.ordinal(); // 1
```

The `compareTo(E o)` method is another important method available for all enum types. This method compares the enum constant with another enum constant of the same enum type based on their ordinal values. It returns a negative integer, zero, or a positive integer if this enum constant is considered less than, equal to, or greater than the specified enum constant, respectively. This method allows enum constants to be used in sorted collections or for any comparison-based operations. For example:

```
Season.WINTER.compareTo(Season.SUMMER); // Returns a negative number
Season.FALL.compareTo(Season.SPRING);   // Returns a positive number
Season.SPRING.compareTo(Season.SPRING); // Returns 0
```

It's worth noting that the natural ordering provided by `compareTo()` for enum constants is based on their declaration order, which may not always be the most meaningful ordering for your specific use case. In such situations, you might need to implement a custom `Comparator` for your enum type.

Here's a table that not only summarizes all these methods but also provides a bit more depth into how they can be used and what to be aware of when using them:

Method	Description	Return Type	Remarks
`values()`	Returns an array containing all of the enum constants in the order they're declared.	`EnumType[]`	Useful for iterating over all constants in an enum.
`valueOf(String name)`	Returns the enum constant of the specified name.	`EnumType`	Throws `IllegalArgumentException` if the specified name doesn't match any of the enum constants.
`name()`	Returns the name of this enum constant, exactly as declared in its enum declaration.	`String`	Identical to calling `toString()`, but `name()` is final and cannot be overridden.
`ordinal()`	Returns the ordinal of this enumeration constant (its position in the enum declaration, where the initial constant is assigned an ordinal of zero).	`int`	Can be used to associate array or list indices directly with enum constants. If you have an array where each position corresponds to a specific enum constant, `ordinal()` helps in directly accessing these array elements based on the enum constants' order.
`compareTo(E o)`	Compares this enum with the specified object for order.	`int`	Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object. The natural ordering is based on the ordinal values of the enum constants.

Customizing Enums

As mentioned before, you can add your own constructors to an enum class. The only requisite is that the constructors must be `private` or package-private. However, you can also add fields and methods to customize the enum class.

Let's say we want to associate a minimum and maximum average temperature to each season:

```
public enum Season {
    WINTER(-5, 10),
```

```
    SPRING(11, 20),
    SUMMER(21, 35),
    FALL(5, 20);

    private int minTemp;
    private int maxTemp;

    Season(int minTemp, int maxTemp) {
        this.minTemp = minTemp;
        this.maxTemp = maxTemp;
    }

    public int getMinTemp() { return minTemp; }
    public int getMaxTemp() { return maxTemp; }
}
```

The example adds a constructor that receives the temperatures. It's package-private, as required. Also, it declares the fields to store the values and public getters for them.

With this, we can consult the temperatures associated with a season:

```
Season.WINTER.getMaxTemp(); // 10
```

We can add any other fields and methods we want to make our enum more interesting.

The only thing we have to remember is to declare the enum constants first in the class. We can declare fields and constructors in the middle, but no other constants below them, or we'll get a compile error.

The following example attempts to declare fields in the middle of enum constants. This will lead to a compile error:

```
public enum Season {
    WINTER(-5, 10),
    SPRING(11, 20),

    private int minTemp; // Compile error: enum constant expected here
    private int maxTemp;

    SUMMER(21, 35),
    FALL(5, 20);

    Season(int minTemp, int maxTemp) {
        this.minTemp = minTemp;
        this.maxTemp = maxTemp;
    }

    public int getMinTemp() { return minTemp; }
    public int getMaxTemp() { return maxTemp; }
}
```

So be careful, declaring enum constants after any fields or constructors is a common pitfall when defining enums with customized constructors and fields.

Key Points

- Records provide a concise way to declare classes that are primarily intended as simple, immutable data carriers.

- A record automatically generates a constructor, private final fields for the parameters, and implementations of `equals()`, `hashCode()`, and `toString()` based on those fields.
- Records are implicitly final and cannot be extended by other classes.
- All fields in a record are implicitly final, enforcing immutability. The state of a record cannot be changed after instantiation.
- Records provide a default canonical constructor that takes all record components as parameters.
- Compact constructors allow adding validation or preprocessing logic without explicitly specifying parameters.
- Records support constructor overloading, but each constructor must delegate to the canonical constructor to ensure field initialization.
- Instance methods can be added to records to encapsulate behavior that operates on the record's components.
- Records can contain nested classes, interfaces, annotations, enums, and other records.
- Records can be generic and accept type parameters, allowing them to work with different data types while maintaining type safety.
- Local records can be declared within methods for temporary data structures with limited scope.
- Records can implement interfaces to satisfy contracts and be used where a specific interface is required.
- Records cannot extend classes or be extended, cannot have additional instance fields beyond the record components, and do not support instance initializers.
- Static fields and static initializers are allowed in records.
- An enum is a special type of class used to define a set of predefined constants, making code more readable and maintainable.
- Each enum constant is implicitly `public`, `static`, and `final`, and by convention, their names are in all caps.
- Enum constructors are always `private` (or package-private), so enum instances cannot be created using `new`.
- All enums implicitly extend `java.lang.Enum`, which provides built-in methods like `valueOf()`.
- The `values()` method returns an array of all the enum constants in the order they are declared.
- The `valueOf()` method returns the enum constant with the specified name.
- Each enum constant also has a `name()` method to get its declared name and an `ordinal()` method to get its position.
- Enums can be customized with fields, constructors, and methods to associate additional data and behavior with each constant.
- When defining an enum with custom fields and constructors, all enum constants must be declared before any fields or constructors.

Practice Questions

1. Consider the following record definition:

```java
public record Employee(String name, int age) {}
```

Which of the following statements is true about the `Employee` record?

A) The `Employee` record explicitly defines a public constructor that initializes its fields.
B) The fields `name` and `age` can be reassigned to new values after an `Employee` object is created.
C) The `Employee` record implicitly creates a public constructor and private final fields for `name` and `age`.
D) It is mandatory to define getters for the fields `name` and `age` in the `Employee` record.

2. Given the record definition below:

```
public record Account(String id, double balance) {}
```

Which statement accurately describes the immutability of records?

A) The `balance` field can be modified using a public setter method within the `Account` record.
B) Once an `Account` object is created, its `id` and `balance` cannot be changed.
C) Immutability of records can be bypassed by you define custom setter methods for the `id` and `balance` fields.
D) Records allow field values to be modified if accessed directly, without using setter methods.

3. Consider the following record declaration:

```
public record Product(int id, String name, double price) {}
```

How can you correctly initialize an instance of the `Product` record?

A) `Product p = new Product();`
B) `Product p = Product(101, "Coffee", 15.99);`
C) `Product p = {101, "Coffee", 15.99};`
D) `Product p = new Product(101, "Coffee", 15.99);`

4. Consider a record that needs to implement the `Comparable` interface to allow sorting based on one of its fields. Given the following record definition:

```
public record Item(int id, String name, double price) implements Comparable<Item> {
    public int compareTo(Item other) {
        return Double.compare(this.price, other.price);
    }
}
```

Which statement correctly describes how records can be customized by implementing interfaces?

A) Records cannot implement interfaces because they are `final` and immutable by design, which prevents any form of behavior customization.
B) This record correctly implements the `Comparable` interface, allowing `Item` objects to be sorted based on their `price`.
C) Implementing interfaces in records is restricted only to functional interfaces due to their immutable nature.
D) The `compareTo` method cannot be overridden in records because method overriding is not supported in record types.

5. Consider the ways to declare enums in Java. Which of the following declarations are valid? (Choose all that apply.)

A)

```
public enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
```

B)

```
enum Month {
    private JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER;
}
```

C)

```
protected enum Season {
    WINTER, SPRING, SUMMER, FALL
}
```

D)

```
enum Status {
    ACTIVE, INACTIVE, DELETED;

    public void printStatus() {
        System.out.println("Current status: " + this);
    }
}
```

6. Consider the following enum declaration:

```
public enum Color {
    RED, GREEN, BLUE;
}
```

What is the result of calling `Color.GREEN.ordinal()`?

A) `1`
B) `2`
C) `0`
D) `Color.GREEN`

7. Consider an enum that needs to provide a custom method to display a message based on the enum constant. Which of the following implementations correctly defines such an enum?

A)

```
public enum Size {
    SMALL, MEDIUM, LARGE;
    public static void printSize() {
        System.out.println("The size is " + this.name());
    }
}
```

B)

```
enum Flavor {
    CHOCOLATE, VANILLA, STRAWBERRY;
    void printFlavor() {
        System.out.println("Flavor: " + Flavor.name);
    }
}
```

C)

```
protected enum Direction {
    NORTH, SOUTH, EAST, WEST;
    private printDirection() {
        System.out.println("Going " + this.toString());
    }
}
```

D)

```
public enum Season {
    WINTER, SPRING, SUMMER, FALL;
    public void printSeason() {
        System.out.println("The season is " + this.name());
    }
}
```

Chapter THREE

Working with Records and Enums

Answers

1. The correct answer is C.

Explanation:

- **A)** The `Employee` record explicitly defines a public constructor that initializes its fields.
  - This option is incorrect because the record `Employee` does not explicitly define a `public` constructor. Records automatically generate a `public` constructor with the same parameters as the record's declaration.
- **B)** The fields `name` and `age` can be reassigned to new values after an `Employee` object is created.
  - This option is incorrect as the fields within a record are `final`, which means they cannot be reassigned to new values after an `Employee` object has been created. This immutability is one of the key characteristics of records.
- **C)** The `Employee` record implicitly creates a `public` constructor and `private final` fields for `name` and `age`.
  - This is the correct option. Records implicitly create a public constructor for the record's fields and also make these fields `private` and `final`. This means you don't have to manually write boilerplate code for constructor, getters, or to ensure immutability.
- **D)** It is mandatory to define getters for the fields `name` and `age` in the `Employee` record.
  - This option is incorrect because records automatically generate public methods to access the fields, known as accessor methods, which essentially act as getters. Therefore, it is not mandatory (or even possible) to define separate getters for the fields.

2. The correct answer is B.

Explanation:

- **A)** The `balance` field can be modified using a public setter method within the `Account` record.
  - This option is incorrect because records in Java do not support public setter methods for their fields. The fields of a record are `final` and cannot be modified after the object's construction, which is a key aspect of their design to enforce immutability.
- **B)** Once an `Account` object is created, its `id` and `balance` cannot be changed.
  - This is the correct option. Records are immutable by design, meaning that once a record object is created, the values of its fields (`id` and `balance` in this case) cannot be changed. This immutability is ensured by making the fields `private` and `final`, and by not providing setter methods.
- **C)** Immutability of records can be bypassed by you define custom setter methods for the `id` and `balance` fields.
  - This option is incorrect. Custom setter methods cannot be defined for the record fields because records do not allow defining mutators for their components.
- **D)** Records allow field values to be modified if accessed directly, without using setter methods.
  - This option is incorrect because the fields in a record are implicitly `final` and private, which means they cannot be modified directly or through setter methods. The design of records enforces this immutability to ensure that instances of records act as true carriers of immutable data.

3. The correct answer is D.

Explanation:

- **A)** `Product p = new Product();`
  - This option is incorrect because the default constructor without parameters does not exist for records in Java. Records require all their fields to be specified at the time of instantiation.
- **B)** `Product p = Product(101, "Coffee", 15.99);`
  - This option is incorrect because the syntax used here is not valid for creating a new instance of a record in Java. The correct syntax for instantiating a record involves using the `new` keyword followed by the record name and the parameters in parentheses.
- **C)** `Product p = {101, "Coffee", 15.99};`
  - This option is incorrect as it mistakenly uses the syntax for array initialization. In Java, objects, including records, cannot be instantiated using curly braces without the `new` keyword and proper constructor.
- **D)** `Product p = new Product(101, "Coffee", 15.99);`
  - This is the correct option. Records in Java are instantiated using the `new` keyword followed by the record's constructor, which requires passing all the fields defined in the record. This syntax correctly creates a new `Product` record with the given `id`, `name`, and `price`.

4. The correct answer is B.

Explanation:

- **A)** Records cannot implement interfaces because they are `final` and immutable by design, which prevents any form of behavior customization.
  - This option is incorrect. Records in Java can implement interfaces. The finality and immutability of records do not preclude them from implementing interfaces, which can be used to add behaviors or contractual obligations to a record.
- **B)** This record correctly implements the `Comparable` interface, allowing `Item` objects to be sorted based on their `price`.
  - This is the correct option. The provided record definition correctly implements the `Comparable<Item>` interface by overriding the `compareTo` method. This customization allows instances of the `Item` record to be sorted based on the `price` field, demonstrating that records can indeed implement interfaces and override their methods as needed.
- **C)** Implementing interfaces in records is restricted only to functional interfaces due to their immutable nature.
  - This option is incorrect. There is no such restriction that limits records to implementing only functional interfaces. Records can implement any interface, including those with multiple abstract methods, as long as the record provides implementations for the abstract methods defined in the interface.
- **D)** The `compareTo` method cannot be overridden in records because method overriding is not supported in record types.
  - This option is incorrect. Records can override methods from the interfaces they implement, including the `compareTo` method from the `Comparable` interface in this example. Method overriding is a key aspect of implementing interfaces and is fully supported by record types in Java.

5. The correct answers are A and D.

Explanation:

- **A)**

```
public enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
```

- This option is correct. It demonstrates a valid declaration of an enum in Java. Enums are used to define a set of named constants, and this syntax is the standard way to declare them. The `public` access modifier makes this enum accessible from any other class.

- **B)**

```
enum Month {
    private JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER;
}
```

- This option is incorrect. Enums cannot have `private` access modifiers for their constants. Enum constants are implicitly `public`, `static`, and `final` and should be declared without access modifiers.

- **C)**

```
protected enum Season {
    WINTER, SPRING, SUMMER, FALL
}
```

- This option is incorrect because enums cannot be declared with `protected` or `private` access levels. Enums are implicitly `public` if they are defined outside of a class. If defined within a class, they can have any access level, but the `protected` keyword cannot be used at the enum level itself.

- **D)**

```
enum Status {
    ACTIVE, INACTIVE, DELETED;

    public void printStatus() {
        System.out.println("Current status: " + this);
    }
}
```

- This option is correct. It shows an enum `Status` with a method `printStatus()`. Enums in Java can contain methods, fields, constructors, and implement interfaces. This demonstrates the ability of enums to have methods, making this declaration valid.

6. The correct answer is A.

Explanation:

- **A)** `1`
  - This option is correct. The `ordinal()` method returns the ordinal of this enumeration constant (its position in its enum declaration, where the initial constant is assigned an ordinal of zero). Since `GREEN` is the second enum constant declared in the `Color` enum, its ordinal value is 1.
- **B)** `2`
  - This option is incorrect. The ordinal value of `BLUE` would be 2, not `GREEN`, because `BLUE` is the third declared constant in the `Color` enum.
- **C)** `0`
  - This option is incorrect. The ordinal value of `RED` is 0, as it is the first declared constant in the `Color` enum.
- **D)** `Color.GREEN`
  - This option is incorrect. The `ordinal()` method returns an integer representing the position of the enum constant in the declaration, not the enum constant itself.

7. The correct answer is D.

Explanation:

- **A)**

```
public enum Size {
    SMALL, MEDIUM, LARGE;
    public static void printSize() {
        System.out.println("The size is " + this.name());
```

```
    }
}
```

- This option is incorrect because the method `printSize()` is defined as `static`, which means it cannot access the `this` reference. Static methods in enums can't directly access the enum constants without specifying the constant explicitly or being passed a reference.
- **B)**

```
enum Flavor {
    CHOCOLATE, VANILLA, STRAWBERRY;
    void printFlavor() {
        System.out.println("Flavor: " + Flavor.name);
    }
}
```

- This option is incorrect because the `name` property of an enum constant is `private`. You can only access it using the `this` reference and the `name()` method (`this.name()`).
- **C)**

```
protected enum Direction {
    NORTH, SOUTH, EAST, WEST;
    private printDirection() {
        System.out.println("Going " + this.toString());
    }
}
```

- This option is incorrect for two reasons. First, `protected` is not a valid access modifier for a top-level enum, top-level enums can only be `public` or package-private (no modifier). Second, `printDirection()` method is missing a return type (e.g., `void`).
- **D)**

```
public enum Season {
    WINTER, SPRING, SUMMER, FALL;
    public void printSeason() {
        System.out.println("The season is " + this.name());
    }
}
```

- This is the correct option. The `printSeason()` method is properly defined: it's `public`, non-static, and utilizes the `this` reference to access the name of the current enum constant. This method correctly provides custom behavior for each enum constant, allowing it to print a message indicating the current season.

Chapter FOUR

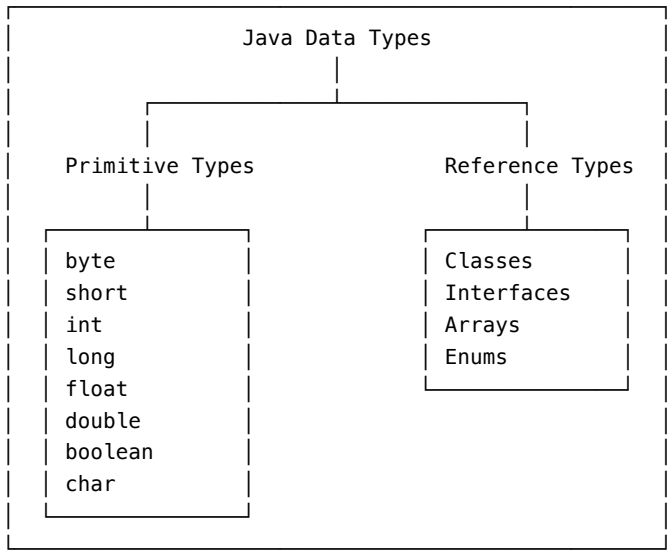
Working with Data

Chapter Content

  - Binary Operators
  - Bitwise and Shift Operators
  - Assignment Operators
  - Equality Operators
  - Relational Operators
  - Logical Operators
- `String` and `StringBuilder`
  - Creating Strings
  - String Concatenation
  - Important String Methods
  - Overriding `toString()`
  - Formatting Strings
  - Using the `StringBuilder` Class
  - Important `StringBuilder` Methods
- Text Blocks
  - Features of Text Blocks
- The Math API
  - Finding the Minimum and Maximum
  - Rounding Numbers
  - Generating Random Numbers
- Key Points
- Practice Questions

Understanding Data Types

There are two main data types, primitive and reference:



Let's review each one in more detail.

Primitive Types

Java is a statically typed language, which means that all variables must first be declared before they can be used. A variable's type determines the values it may contain and what operations can be performed on it.

In Java, primitive types are the most basic data types available. They are not objects and do not belong to

any class. Instead, they are defined by the language itself. Primitive types are used to store simple values like integers, floating point numbers, booleans, and characters.

Primitive types are stored directly in memory and are accessed by their values. This is in contrast to reference types (objects), which are accessed by their reference. Because of this direct storage, primitives are faster and require less memory than objects.

There are eight primitive data types in Java:

Type	Size (bits)	Minimum Value	Maximum Value	Default
`byte`	8	-128	127	0
`short`	16	-32,768	32,767	0
`int`	32	-2,147,483,648	2,147,483,647	0
`long`	64	-9,223,372,036,854,775,808	9,223,372,036,854,775,807	0L
`float`	32	1.4E-45	3.4028235E38	0.0f
`double`	64	4.9E-324	1.7976931348623157E308	0.0d
`boolean`	n/a	n/a	n/a	false
`char`	16	`'\u0000'` (0)	`'\uffff'` (65,535)	`'\u0000'`

Let's break this down.

The integer types (`byte`, `short`, `int`, `long`) are for whole number values. They differ by the range of values they can hold. A `byte` is `8` bits and can hold values from `-128` to `127`. A `short` is `16` bits and can hold values from `-32,768` to `32,767`. An `int` is `32` bits and can hold values from `-2,147,483,648` to `2,147,483,647`. And a `long` is 64 bits, holding values from `-9,223,372,036,854,775,808` to `9,223,372,036,854,775,807`.

You might wonder, aren't all numbers in Java treated the same way? Why do we even need different types like `int`, `long`, etc.? The reason is efficiency and memory usage. If you know your values will always be within a certain range, you can use a smaller type to save memory. For example, an `int` takes up half the memory of a `long`. In large-scale applications with lots of data, this can make a big difference.

The floating point types (`float` and `double`) are for numbers with decimal points. A `float` is 32 bits and a `double` is 64 bits. This gives `double` much more precision than `float`. Many developers opt to use `double` for all decimal values to avoid precision issues, but there are scenarios where `float` can be used to conserve memory if high precision is not needed.

In Java, the size of a boolean is not explicitly defined by the Java Language Specification. The actual memory size of a boolean variable is implementation-dependent and varies based on the Java Virtual Machine being used. However, for the exam, you only need to know that the `boolean` type has only two possible values: `true` and `false`, and that it's used for conditional logic.

The `char` type is for single characters. It uses 16 bits because it uses Unicode encoding, allowing it to represent a wide variety of characters from different languages.

An important thing to remember is that the default values only apply to fields. Local variables, on the other hand, must be explicitly initialized before use; otherwise, your code won't compile.

You can assign values to variables using literals. In addition to standard decimal notation, Java allows you to assign integer literals using hexadecimal (prefix `0x` or `0X`), octal (prefix `0`) and binary (prefix `0b` or `0B`) notation.

Here's an example:

```
// Decimal notation
int decimalNum = 42;

// Hexadecimal notation
int hexNum = 0x2A; // Equivalent to decimal 42
```

```
// Octal notation
int octalNum = 052; // Equivalent to decimal 42

// Binary notation
int binaryNum = 0b101010; // Equivalent to decimal 42
```

When assigning literals to variables, it's important to note that the literal's type must match the variable's type. If they don't match, you may need to use a suffix to specify the literal's type.

For integer literals: - `long` literals use the suffix `L` or `l`: `long longNum = 1000L;` - `int` literals don't require a suffix, as `int` is the default value for integers: `int intNum = 1000;`

For floating point literals: - `float` literals use the suffix `F` or `f`: `float floatNum = 3.14f;` - `double` literals use the suffix `D` or `d`, although this suffix is optional as double is the default type for decimal literals: `double doubleNum = 3.14D;`

Here are some examples:

```
long longNum = 1000L; // Suffix L is required
float floatNum = 3.14f; // Suffix f is required
double doubleNum1 = 3.14; // Suffix d is optional
double doubleNum2 = 3.14d; // Suffix d is optional
```

If you don't use the correct suffix, you might encounter a compilation error. For example:

```
byte longNum = 1000; // Compilation error: integer literal is too large
float floatNum = 3.14; // Compilation error: incompatible types
```

In these cases, Java assumes the literal is of type `int` or `double` respectively, which can't be directly assigned to a `byte` or `float` variable without an explicit cast.

You can also use underscores in numeric literals to improve readability, such as `1_000_000`. Here are some examples illustrating the use of underscores in numeric literals:

```
// Valid use of underscores
int million = 1_000_000;
long creditCardNumber = 1234_5678_9012_3456L;
float pi = 3.14_15F;
double avogadro = 6.022_140_857e23;
```

However, there are some restrictions. You cannot place an underscore: - At the beginning or end of a number - Adjacent to a decimal point in a floating point literal - Prior to an `F` or `L` suffix - In positions where a string of digits is expected

Here are some additional examples of invalid underscore placements:

```
// Invalid use of underscores
int x1 = _1000; // Compilation error: illegal underscore
int x2 = 1000_; // Compilation error: illegal underscore
float y1 = 3_.14F; // Compilation error: illegal underscore
float y2 = 3._14F; // Compilation error: illegal underscore

float y3 = 3.14__F; // Compilation error: consecutive underscores
long z1 = 1000_L; // Compilation error: underscore before L suffix

int x3 = 0_x42; // Compilation error: underscore in position where digits are expected
int x4 = 0b_101010; // Compilation error: underscore in position where digits are expected
```

These rules are in place to prevent ambiguity and to ensure that the use of underscores does not conflict with other parts of the language syntax.

Reference Types

In the previous section, we explored the concept of primitive types in Java. However, as we know, Java is an object-oriented language, and nearly everything is treated as an object. While primitives provide the basic building blocks, reference types allow us to work with objects and take full advantage of Java's object-oriented features.

So, what exactly are reference types? Unlike primitives, which hold their values directly, reference types store the memory address where the actual object resides. In other words, a reference variable *refers* to the location of the object rather than containing the object itself.

```
String myString = "Hello"; // Reference type
```

In this example, `myString` is a reference variable of type `String`. It doesn't hold the actual string value but rather a reference to the memory location where the `"Hello"` object is stored.

This contrasts with how primitive types work:

```
int myNumber = 42; // Primitive type
```

Here, `myNumber` directly holds the integer value `42` rather than referring to an object.

But what if we want to treat primitives as objects? This is where wrapper classes come into play. Java provides a set of wrapper classes that correspond to each primitive type, allowing them to be used in scenarios that require objects:

Primitive Type	Wrapper Class	Inherits from Number
`boolean`	`Boolean`	No
`byte`	`Byte`	Yes
`short`	`Short`	Yes
`int`	`Integer`	Yes
`long`	`Long`	Yes
`float`	`Float`	Yes
`double`	`Double`	Yes
`char`	`Character`	No

Each primitive type has a corresponding wrapper class, most of which inherit from the `Number` class. The `Boolean` and `Character` classes are exceptions, as they don't represent numeric values.

Wrapper classes provide methods for creating instances from many representations and for converting between different data types. Here are some examples:

1. Parsing methods:
   - `Integer.parseInt(String s)`: Parses a string argument as a signed decimal integer.
   - `Double.parseDouble(String s)`: Parses a string argument as a double-precision floating-point number.
   - `Boolean.parseBoolean(String s)`: Parses a string argument as a boolean value.

   For example:

   ```
   int num = Integer.parseInt("42");
   double value = Double.parseDouble("3.14");
   boolean flag = Boolean.parseBoolean("true");
   ```

2. Conversion methods:
   - `Integer.valueOf(String s)`: Returns an `Integer` object holding the value of the specified string representation.
   - `Long.valueOf(long l)`: Returns a `Long` object holding the specified primitive long value.

- `Double.valueOf(double d)`: Returns a `Double` object holding the specified primitive double value.

For example:

```
Integer myInt = Integer.valueOf("100");
Long myLong = Long.valueOf(1234567890L);
Double myDouble = Double.valueOf(2.71828);
```

3. Conversion between numeric types:

   - `Integer.byteValue()`: Returns the value of an `Integer` as a byte.
   - `Long.intValue()`: Returns the value of a `Long` as an int.
   - `Float.doubleValue()`: Returns the value of a `Float` as a double.

   For example:

   ```
   byte myByte = myInt.byteValue();
   long myLong = myInt.longValue();
   ```

4. Character methods:

   - `Character.isDigit(char ch)`: Determines if the specified character is a digit.
   - `Character.isLetter(char ch)`: Determines if the specified character is a letter.
   - `Character.toUpperCase(char ch)`: Converts the character argument to uppercase.

   For example:

   ```
   char myChar = '7';
   boolean isDigit = Character.isDigit(myChar);
   boolean isLetter = Character.isLetter(myChar);
   char upperCase = Character.toUpperCase(myChar);
   ```

Notice that these methods are `static`, allowing you to use them without creating an instance of the wrapper class.

These are just a few examples of the methods provided by wrapper classes for creating instances and converting between different representations. Each wrapper class offers a range of methods specific to its corresponding primitive type, providing flexibility and convenience when working with different data formats and conversions.

So, do wrapper classes make primitives objects? Not quite. Wrapper classes are separate from primitives but provide a way to wrap primitives in an object form. This allows primitives to be used in contexts that expect objects, such as collections or when using generics.

To bridge the gap between primitives and their wrapper classes, Java introduced autoboxing and unboxing. As mentioned before, autoboxing is the automatic conversion of a primitive type to its corresponding wrapper class, while unboxing is the reverse process:

```
int num = 42;
Integer objNum = num; // Autoboxing
int num2 = objNum;    // Unboxing
```

In this example, `num` is automatically boxed into an `Integer` object when assigned to `objNum`. Similarly, `objNum` is unboxed back to an `int` when assigned to `num2`. This happens implicitly, making it convenient to switch between primitives and their wrapper classes.

Autoboxing and unboxing work with all primitive types and their corresponding wrapper classes, not just with `int` and `Integer`. Java handles these conversions automatically based on the context in which they are used.

It's important to note that while autoboxing and unboxing simplify code readability, they could have some performance implications. Each conversion between a primitive and its wrapper class involves creating or

discarding an object, which adds a small overhead. In most cases, this overhead is negligible, but it can add up in performance-sensitive scenarios with frequent autoboxing/unboxing operations.

One advantage of wrapper classes is their ability to represent the absence of a value using `null`. While primitives cannot be `null`, wrapper objects can.

```
Integer num = null;
int value = num; // NullPointerException
```

Here, assigning `null` to `num` is valid because it's an `Integer` object. However, attempting to unbox `num` to an `int` throws a `NullPointerException`. This behavior allows for more explicit handling of `null` values and can be useful in scenarios where a variable might not have a value assigned.

Also, it's worth noting that wrapper classes are immutable, meaning their values cannot be changed once assigned. When you perform operations on a wrapper object, a new object is created with the updated value rather than modifying the existing object.

```
Integer num = 42;
num++;
```

In this example, the `++` operation on `num` creates a new `Integer` object with the value 43 rather than modifying the original object. This behavior ensures thread safety and avoids unexpected side effects when sharing wrapper objects across multiple parts of the program.

Operators

Introducing Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical operations.

Java provides a rich set of operators to perform operations on variables and values: - Unary operators: Operators that act on a single operand, such as `++` to increment a value or `!` to negate a boolean. - Binary operators: Operators that act on two operands, like arithmetic operators (`+`, `-`, `*`, `/`, `%`) and comparison operators (`>`, `<`, `>=`, `<=`, `==`, `!=`). - Ternary operator: The conditional operator that takes three operands (`condition ? value_if_true : value_if_false`). - Assignment operators: Operators used to assign values to variables (`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `^=`, `|=`, `<<=`, `>>=`, `>>>=`). - Logical operators: Operators used to determine the logic between variables or values (`&&`, `||`).

Many developers new to Java mistakenly believe operators are just for math operations. But operators also play an important role in controlling program flow, performing logical operations, manipulating bits, and more. For example:

```
int a = 10;
int b = 5;

// Arithmetic operator
System.out.println(a + b);  // 15

// Comparison operator
System.out.println(a > b);  // true

// Logical operator
System.out.println((a > b) && (a != b)); // true
```

As you can see, operators in Java go well beyond basic arithmetic.

Operator Precedence

An important concept to grasp with operators is precedence. Just like in mathematics, some operators in Java have higher precedence than others, meaning they get evaluated first in an expression.

For instance, consider this code:

```
int result = 10 + 5 * 2;
System.out.println(result);
```

You might expect `result` to be `30` (`10 + 5` is `15`, then `15 * 2`). But actually, it prints out `20`. That's because the `*` operator has higher precedence than `+`. So `5 * 2` is evaluated first, giving `10`, and then `10` is added to the original `10`.

Here's a table showing operator precedence in Java, from highest to lowest:

Category	Operator	Associativity
Postfix	`expr++ expr--`	Left to right
Unary	`++expr --expr +expr -expr ~ !`	Right to left
Multiplicative	`* / %`	Left to right
Additive	`+ -`	Left to right
Shift	`<< >> >>>`	Left to right
Relational	`< > <= >= instanceof`	Left to right
Equality	`== !=`	Left to right
Bitwise AND	`&`	Left to right
Bitwise XOR	`^`	Left to right
Bitwise OR	`|`	Left to right
Logical AND	`&&`	Left to right
Logical OR	`||`	Left to right
Ternary	`? :`	Right to left
Assignment	`= += -= *= /= %= &= ^= |= <<= >>= >>>=`	Right to left

As the table shows, most operators evaluate left-to-right. So in an expression like `a + b - c`, `a + b` happens first, then `- c`.

But the assignment operators and the unary operators actually evaluate right-to-left. Consider this code:

```
int a = 10;
int b = 20;
int c = (a = 3) + (b = 5);
System.out.println(a + ", " + b + ", " + c); // 3, 5, 8
```

Here, `a` is assigned `3`, `b` is assigned `5`, and because assignment evaluates right-to-left, the assignments happen before the addition. So `c` ends up as `8` (`3 + 5`), while `a` is `3` and `b` is `5`.

This right-to-left evaluation allows chained assignments, like `a = b = c = 5`. The `5` is assigned to `c`, then that result is assigned to `b`, and finally to `a`, right-to-left.

It's a lot to remember, and memorizing the entire precedence table isn't necessary. The key points are: 1. Postfix operations like `x++` happen before prefix ones like `++x`. 2. Multiplicative operations (`*`, `/`, `%`) happen before additive ones (`+`, `-`). 3. Bitwise operations (`&`, `|`, `^`) happen after comparisons (`>`, `==`, etc.) but before logical ones (`&&`, `||`). 4. Assignments evaluate last, and right-to-left.

When in doubt, parentheses can always be used to make the order explicit. The expressions `(a + b) * c` and `a + (b * c)` are unambiguously different.

In general, parentheses should be used whenever the precedence is unclear or to improve readability. But they shouldn't be overused to the point of clutter. With a solid grasp of operator precedence, many parentheses become unnecessary, leading to cleaner, more readable code.

Let's look at a few more examples so you can better understand operators and precedence in Java:

```
int x = 10;
int y = 20;
```

```
int z = 30;

System.out.println(x + y - z);  // 10 + 20 - 30 = 0
System.out.println(x - y + z);  // 10 - 20 + 30 = 20
System.out.println(x * y / z);  // 10 * 20 / 30 = 6
System.out.println(x / y * z);  // 10 / 20 * 30 = 0
```

In the first two statements, the operators have the same precedence (`+` and `-`), so they're evaluated left-to-right. In the third and fourth statements, `*` and `/` have higher precedence and are evaluated first, left-to-right, before the results are added. In the fourth example, `10 / 20` is zero (because we are using integers).

Now let's mix in some assignments and unary operators:

```
int a = 5;
int b = 10;
int c = ++a * b--;
System.out.println(a + ", " + b + ", " + c);  // 6, 9, 60
```

Here, `++a` increments `a` to 6 before the multiplication. Then `6 * 10` gives 60, which is assigned to `c`. Finally, `b--` decrements `b` to 9, but after the multiplication. So we end up with `a` as 6, `b` as 9, and `c` as 60.

The right-to-left nature of assignments is critical to understand:

```
int x = 2;
int y = 3;
int z = 1;
x += y -= z;
System.out.println(x + ", " + y + ", " + z);  // 4, 2, 1
```

First, `z` (1) is subtracted from `y` (3), giving 2, which is then assigned back to `y`. Then this value (2) is added to `x` (2), giving 4, which is assigned back to `x`. So `x` ends up as 4, `y` as 2, and `z` remains 1.

Logical and bitwise operators can add further complexity:

```
int a = 10;  // 1010 in binary
int b = 6;   // 0110 in binary

System.out.println(a & b);  // 1010 & 0110 = 0010 (2 in decimal)
System.out.println(a | b);  // 1010 | 0110 = 1110 (14 in decimal)
System.out.println(a ^ b);  // 1010 ^ 0110 = 1100 (12 in decimal)

System.out.println(a > 5 && b < 10); // true && true = true
System.out.println(a > 5 || b < 5);  // true || false = true
```

The bitwise operators `&`, `|`, and `^` perform AND, OR, and XOR operations on each bit of the numbers. The logical operators `&&` and `||` perform AND and OR on boolean conditions, with `&&` having higher precedence.

Lastly, let's not forget the ternary operator, which is like a compact `if-else` statement:

```
int x = 10;
int y = 20;
int max = (x > y) ? x : y;
System.out.println(max);  // 20
```

Here, `(x > y)` is false, so the value after the colon (`y`, which is 20) is assigned to `max`.

In the next sections, we'll review each type of operators in more detail.

Unary Operators

Unary operators are operators that work on only one operand. You have already seen some unary operators in action, such as the logical complement operator (`!`) used with boolean values. However, in this section, we'll cover the unary operators that are primarily used with numeric types.

Complement and Negation Operators The unary complement operator (`~`), also known as the bitwise complement operator, inverts all the bits in a number, effectively changing each 0 to 1 and each 1 to 0. In Java, integers are represented using 32 bits in two's complement format.

For positive numbers, the bitwise complement will flip all the bits, and the resulting number is the negation of the original number minus one. This is because inverting all bits and then interpreting the result in two's complement yields `-(n + 1)`.

Here's how it works:

For a positive number like 5, the binary representation is:

```
0000 0000 0000 0000 0000 0000 0000 0101
```

When you apply the bitwise complement operator, it flips all the bits:

```
1111 1111 1111 1111 1111 1111 1111 1010
```

In two's complement, this is the representation of `-6`. Therefore, `~5` in Java equals `-6`.

For negative numbers, the bitwise complement operator also flips all the bits. The result is the positive version of the original number minus one, because flipping all bits of a negative number and interpreting it in two's complement yields the positive counterpart decreased by 1.

For example, let's take -5. In two's complement, it's represented as:

```
1111 1111 1111 1111 1111 1111 1111 1011
```

Applying the bitwise complement operator:

```
0000 0000 0000 0000 0000 0000 0000 0100
```

This binary number represents 4 in decimal. So, `~(-5)` equals 4.

In summary: - `~n` for a positive number `n` results in `-(n + 1)`. - `~(-n)` results in `n - 1`.

On the hand, the unary negation operator (`-`) is simpler to understand. This operator is used to negate a numeric value, effectively changing its sign. For example, if `x` is 5, then `-x` would be -5.

A common misconception is that the negation operator is the same as subtracting the number from zero. While the end result may be the same, the negation operator works differently under the hood. It directly changes the sign bit of the number, rather than performing a subtraction operation.

Increment and Decrement Operators Java also provides increment (`++`) and decrement (`--`) operators, which are used to increment or decrement a variable's value by 1. These operators can be used in either prefix or postfix form.

Here's a table summarizing the different increment and decrement operators:

Operator	Name	Description	Example
`++x`	Prefix increment operator	Increments `x` by 1, then returns the new value of `x`	`++x`
`x++`	Postfix increment operator	Returns the current value of `x`, then increments `x` by 1	`x++`
`--x`	Prefix decrement operator	Decrements `x` by 1, then returns the new value of `x`	`--x`

Operator	Name	Description	Example
`x--`	Postfix decrement operator	Returns the current value of `x`, then decrements `x` by 1	`x--`

One question is whether you can use increment and decrement operators with boolean values. The answer is no. These operators are only applicable to numeric types like `int`, `long`, `float`, `double`, etc.

Another point of confusion is whether `x++` increments `x` before or after the expression it's used in. The postfix increment operator (`x++`) returns the original value of `x`, and then increments `x` after that value is returned. So if you have an expression like `y = x++;`, y will be assigned the original value of `x`, and then `x` will be incremented.

On the flip side, if you use `--x`, it decreases the value of `x` before the expression is evaluated. So `y = --x;` would first decrement `x`, and then assign the new value of `x` to `y`.

Also, you might wonder if there's a difference between `++x` and `x++` if they are the only operations in a statement. In this case, there is no difference. Both will increment `x` by 1. The difference only comes into play when the increment operation is part of a larger expression.

Summary of Unary Operators Here's a comprehensive table of the unary operators in Java:

Operator	Name	Description	Example
`+`	Unary plus	Indicates a positive value (rarely used)	`+x`
`-`	Unary minus	Negates a value	`-x`
`++`	Increment	Increments a value by 1	`++x (prefix) x++ (postfix)`
`--`	Decrement	Decrements a value by 1	`--x (prefix) x-- (postfix)`
`~`	Bitwise complement	Inverts all bits	`~x`

Note that the complement operator (`~`) only works on integer types, not on `float` or `double`.

A few more nuanced points to consider:

- When using `++x` and `x++`, the order of operations and side effects come into play. Consider this example:

  ```
  int x = 5;
  int y = ++x + x++; // y = 12, x = 7
  ```

  Here's what's happening:

  1. The prefix increment operator (`++x`) is applied first. This increments `x` to 6, and the value of the expression `++x` is 6.
  2. Then, the postfix increment operator (`x++`) is applied. This returns the current value of `x`, which is 6, and then increments `x` to 7.
  3. The value of `y` is the sum of the prefix increment expression (6) and the postfix increment expression (6), which is 12.
  4. After the statement is executed, `x` is 7 (due to the postfix increment), and `y` is 12.

- It is possible to use increment and decrement operators inside a complex expression without affecting the outcome, but it can make your code much harder to read and understand. For example:

  ```
  int x = 5;
  int y = 3 * x++ + 2; // y = 17, x = 6
  ```

  This works, but it's clearer to do the increment operation on a separate line before the expression.

- The increment and decrement operators cannot be used with boolean values because boolean values can only be `true` or `false`. They don't have a "next" or "previous" value like numbers do.
- If you use multiple increment or decrement operators on the same variable in a single statement, the order of operations is from left to right:

  ```
  int x = 5;
  System.out.println(++x + x++ + x--); // Output: 19
  ```

  Here's what's happening:

  1. `++x` increments `x` to 6 and returns 6
  2. `x++` returns 6 (the current value of `x`), then increments `x` to 7
  3. `x--` returns 7 (the current value of `x`), then decrements `x` to 6
  4. The sum is $6 + 6 + 7 = 19$
- The complement operator (`~`) is useful for low-level bit manipulation tasks, often used in systems programming, embedded systems, networking protocols, cryptography, and more.
- The Java compiler recognizes the increment and decrement operators and generates the appropriate bytecode based on whether the operator is used as a prefix or postfix. It's not something you need to worry about as a programmer, but it's handled at the bytecode level.
- The increment and decrement operators can be used on variables of type float and double. The same prefix/postfix rules apply as with integer types.
- If you use `++x` versus `x++` inside a loop, the difference in the final value of `x` after the loop depends on when the increment happens. Consider:

  ```
  int x = 0;
  for(int i = 0; i < 5; i++) {
      System.out.println(++x);
  }
  // Output: 1 2 3 4 5
  // x is 5 after the loop

  x = 0;
  for(int i = 0; i < 5; i++) {
      System.out.println(x++);
  }
  // Output: 0 1 2 3 4
  // x is 5 after the loop
  ```

  In both cases, `x` ends up being 5, but when the values are printed is different. With `++x`, x is incremented before its value is printed, while with `x++`, the original value of `x` is printed before being incremented.

Binary Operators

Binary operators are operators that work on two operands. Java provides a set of binary arithmetic operators for performing basic mathematical operations on numeric operands. These operators include addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), and modulus (`%`).

Arithmetic Operators The addition (`+`), subtraction (`-`), and multiplication (`*`) operators work as you'd expect:

```
int a = 10;
int b = 20;
int sum = a + b; // 30
int difference = b - a; // 10
int product = a * b; // 200
```

The division operator (`/`) performs division between two numeric operands. It's important to note that when used with integer operands, the division operator performs integer division, which means it returns the quotient and discards any remainder.

```
int a = 10;
int b = 3;
int quotient = a / b; // 3
```

If you want to perform floating-point division and get a fractional result, at least one of the operands needs to be a floating-point type (`float` or `double`).

```
int a = 10;
double b = 3.0;
double quotient = a / b; // 3.3333333333333335
```

The modulus operator (`%`) returns the remainder after performing integer division.

```
int a = 10;
int b = 3;
int remainder = a % b; // 1
```

Numeric Promotion When performing arithmetic operations on operands of different types, Java automatically promotes the operands according to a set of rules known as numeric promotion.

Numeric promotion is the automatic conversion of a smaller numeric type to a larger numeric type to prevent loss of precision during arithmetic operations. This allows you to perform arithmetic operations on mixed types without having to explicitly cast the operands.

Java follows these rules for numeric promotion:

1. If either operand is of type `double`, the other is promoted to `double`.
2. Otherwise, if either operand is of type `float`, the other is promoted to `float`.
3. Otherwise, if either operand is of type `long`, the other is promoted to `long`.
4. Otherwise, both operands are promoted to `int`.

Here are some examples:

```
int a = 10;
double b = 20.0;
double result1 = a + b; // a is promoted to double

float c = 10.0f;
long d = 20L;
float result2 = c + d; // d is promoted to float

short e = 10;
short f = 20;
int result3 = e + f; // e and f are promoted to int
```

Adding Parentheses to Change the Order of Operation You can use parentheses to change the default order of operations in an arithmetic expression. Expressions inside parentheses are evaluated first.

```
int a = 10;
int b = 20;
int c = 30;
int result = a + b * c; // 610 (multiplication happens first)
int result2 = (a + b) * c; // 900 (addition happens first)
```

However, when using parentheses, it's important to ensure that they are properly balanced. Each opening parenthesis must have a corresponding closing parenthesis. Mismatched parentheses will result in a compilation error.

```
int result = (a + b) * c; // correct
int result2 = (a + b * c; // compilation error (mismatched parentheses)
```

Summary of Binary Operators Here's a table summarizing the binary operators in Java:

Operator	Name	Description	Example
`+`	Addition	Adds two operands	`a + b`
`-`	Subtraction	Subtracts the second operand from the first	`a - b`
`*`	Multiplication	Multiplies two operands	`a * b`
`/`	Division	Divides the first operand by the second	`a / b`
`%`	Modulus	Returns the remainder of division	`a % b`

Some nuanced points to consider are:

- When working with floating-point numbers (`float` and `double`), it's important to remember that they have limited precision. This can lead to small inaccuracies in calculations. `java double a = 0.1; double b = 0.2; double sum = a + b; // 0.30000000000000004 (not exactly 0.3)` This is due to how floating-point numbers are represented in binary. For precise decimal calculations, you should use the `BigDecimal` class.
- When the result of an arithmetic operation exceeds the maximum or minimum value that can be represented by the target type, overflow or underflow can occur. In Java, integer overflow and underflow do not throw an exception; the value simply wraps around. `java int a = Integer.MAX_VALUE; int b = 1; int sum = a + b; // -2147483648 (minimum int value)` For floating-point types, overflow results in Infinity and underflow results in 0.
- Attempting to divide an integer by zero will throw an `ArithmeticException`. `java int a = 10; int b = 0; int result = a / b; // throws ArithmeticException` However, dividing a floating-point number by zero does not throw an exception. It results in `Infinity` or `NaN` (Not-a-Number). `java double a = 10.0; double b = 0.0; double result = a / b; // Infinity or NaN (Not-a-Number)`

Bitwise and Shift Operators

In addition to the arithmetic operators, Java provides a set of bitwise and shift operators that allow you to manipulate the individual bits of integer values. These operators are particularly useful when working with flags, masks, and low-level system operations.

Bitwise Operators Java has four bitwise operators: AND (`&`), OR (`|`), XOR (`^`), and complement (`~`).

The bitwise AND operator (`&`) returns a 1 in each bit position for which the corresponding bits of both operands are 1s.

```
int a = 0b1010; // 10
int b = 0b1100; // 12
int result = a & b; // 0b1000 = 8
```

The bitwise OR operator (`|`) returns a 1 in each bit position for which the corresponding bits of either or both operands are 1s.

```
int a = 0b1010; // 10
int b = 0b1100; // 12
int result = a | b; // 0b1110 = 14
```

The bitwise XOR (exclusive OR) operator (^) returns a 1 in each bit position for which the corresponding bits of either but not both operands are 1s.

```
int a = 0b1010; // 10
int b = 0b1100; // 12
int result = a ^ b; // 0b0110 = 6
```

The bitwise complement operator (~) is a unary operator that inverts all the bits of its operand.

```
int a = 0b1010; // 10
int result = ~a; // 0b11111111111111111111111111110101 = -11
```

Shift Operators Java provides three shift operators: left shift (<<), signed right shift (>>), and unsigned right shift (>>>).

The left shift operator (<<) shifts the bits of the first operand left by the number of positions specified by the second operand. The new rightmost bits are filled with 0s.

```
int a = 0b1010; // 10
int result = a << 1; // 0b10100 = 20
```

Each left shift effectively doubles the number.

The signed right shift operator (>>) shifts the bits of the first operand right by the number of positions specified by the second operand. The new leftmost bits are filled with the sign bit (0 for positive numbers, 1 for negative numbers), preserving the sign of the number.

```
int a = 0b1010; // 10
int result = a >> 1; // 0b0101 = 5
```

Each signed right shift effectively halves the number, rounding down.

The unsigned right shift operator (>>>) is similar to the signed right shift, but the new leftmost bits are always filled with 0s, regardless of the sign.

```
int a = 0b11111111111111111111111111110110; // -10
int result = a >>> 1; // 0b01111111111111111111111111111011 = 2147483643
```

Summary of Bitwise and Shift Operators Here's a summary of the bitwise and shift operators in Java:

Operator	Name	Description	Example
&	Bitwise AND	Returns 1 if both bits are 1	a & b
|	Bitwise OR	Returns 1 if at least one bit is 1	a | b
^	Bitwise XOR	Returns 1 if exactly one bit is 1	a ^ b
~	Bitwise Complement	Inverts all bits	~a
<<	Left Shift	Shifts bits left, filling with 0s	a << b
>>	Signed Right Shift	Shifts bits right, filling with sign bit	a >> b
>>>	Unsigned Right Shift	Shifts bits right, filling with 0s	a >>> b

A few more nuanced points to consider:

- The left operand of a shift operator determines the type of the result. The right operand (the shift distance) is always promoted to int. `java  byte a = 10;  byte b = a << 1; // Compilation error: the result is int  int c = a << 1; // OK`

- There is no separate *unsigned left shift operator* in Java. The left shift operator (`<<`) inherently shifts bits to the left and fills the rightmost bits with zeros, making it effectively unsigned. The notion of signed or unsigned does not apply to left shift in the same way it does for right shift because shifting left does not involve the sign bit.
- Bitwise and shift operators have lower precedence than arithmetic operators but higher precedence than comparison and logical operators. Use parentheses to make the precedence explicit and the code more readable.

Assignment Operators

Assignment operators are used to assign values to variables. In addition to the simple assignment operator (=), Java provides compound assignment operators that combine an arithmetic or bitwise operation with assignment.

Compound assignment operators combine an arithmetic or bitwise operation with the assignment operation. They provide a concise way to modify the value of a variable based on its current value.

The general syntax for compound assignment operators is:

```
variable op= expression;
```

Where `op` is one of the arithmetic or bitwise operators (`+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `<<`, `>>`, `>>>`).

This is equivalent to:

```
variable = variable op expression;
```

The compound assignment operators are: - `+=` (addition assignment) - `-=` (subtraction assignment) - `*=` (multiplication assignment) - `/=` (division assignment) - `%=` (modulus assignment) - `&=` (bitwise AND assignment) - `|=` (bitwise OR assignment) - `^=` (bitwise XOR assignment) - `<<=` (left shift assignment) - `>>=` (signed right shift assignment) - `>>>=` (unsigned right shift assignment)

Here are examples of each compound assignment operator:

```
int a = 10;

a += 5;  // equivalent to a = a + 5; a is now 15
a -= 3;  // equivalent to a = a - 3; a is now 12
a *= 2;  // equivalent to a = a * 2; a is now 24
a /= 4;  // equivalent to a = a / 4; a is now 6
a %= 5;  // equivalent to a = a % 5; a is now 1

int b = 0b1010; // binary representation of 10

b &= 0b1100;  // equivalent to b = b & 0b1100; b is now 0b1000 (8 in decimal)
b |= 0b0101;  // equivalent to b = b | 0b0101; b is now 0b1101 (13 in decimal)
b ^= 0b1001;  // equivalent to b = b ^ 0b1001; b is now 0b0100 (4 in decimal)
b <<= 2;      // equivalent to b = b << 2; b is now 0b10000 (16 in decimal)
b >>= 1;      // equivalent to b = b >> 1; b is now 0b01000 (8 in decimal)
b >>>= 2;     // equivalent to b = b >>> 2; b is now 0b00010 (2 in decimal)
```

Compound assignment operators are not only more concise but can also be more efficient than their expanded equivalents. This is because the variable is only evaluated once in the compound form, while it's evaluated twice in the expanded form.

For example, consider the following code:

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

```
array[index++] += 10; // More efficient
array[index++] = array[index++] + 10; // Less efficient
```

In the first line, `index` is only incremented once after its value has been used to access the array element. In the second line, `index` is incremented twice, leading to unexpected behavior and less efficient code.

Changing the Primitive Type with Suffixes When assigning a value to a variable of a different primitive type, you can use the suffixes `f`, `l`, and `d` to specify the type of the literal value.

- `f` is used for float literals
- `l` or `L` is used for long literals
- `d` or `D` is used for double literals

```
float a = 3.14f;
long b = 100L;
double c = 3.14d; // d is optional here, as double is the default for decimal literals
```

Casting Values When assigning a value of one type to a variable of another type, you may need to use casting to explicitly convert the value to the target type.

```
int a = 10;
byte b = (byte) a;
```

In this example, the `int` value is cast to a byte before assignment.

When assigning a value that is too large or too small for the target type, overflow or underflow can occur.

```
byte a = 127;
a++; // a is now -128 (underflow)

byte b = -128;
b--; // b is now 127 (overflow)
```

In these examples, incrementing the maximum value of a byte causes underflow, and decrementing the minimum value causes overflow.

To avoid unexpected behavior due to overflow or underflow, it's important to review your assignments and ensure that the values are appropriate for the target types.

```
int a = 1000;
byte b = (byte) a; // b is now -24 (overflow)
```

Here, casting the int value 1000 to a byte results in overflow, as 1000 is outside the range of a byte (-128 to 127).

Summary of Assignment Operators Here's a summary table of the assignment operators in Java:

Operator	Name	Example
`=`	Simple assignment	`a = 10`
`+=`	Addition assignment	`a += 5`
`-=`	Subtraction assignment	`a -= 5`
`*=`	Multiplication assignment	`a *= 5`
`/=`	Division assignment	`a /= 5`
`%=`	Modulus assignment	`a %= 5`
`&=`	Bitwise AND assignment	`a &= 5`
`|=`	Bitwise OR assignment	`a |= 5`
`^=`	Bitwise XOR assignment	`a ^= 5`
`<<=`	Left shift assignment	`a <<= 2`

Operator	Name	Example
`>>=`	Signed right shift assignment	`a >>= 2`
`>>>=`	Unsigned right shift assignment	`a >>>= 2`

There are a few nuanced points to consider: - When casting a floating-point value to an integer type, the fractional part is truncated (not rounded). `java double a = 3.9999; int b = (int) a; // b is now 3`

- The compound assignment operators have lower precedence than the arithmetic operators but higher precedence than the simple assignment operator. `java int a = 10; a *= 5 + 2; // a is now 70 ((10 * 5) + 2) a = 10; a = a * 5 + 2; // a is now 52 ((10 * 5) + 2)`
- The simple assignment operator (`=`) can be chained to assign the same value to multiple variables. `java int a, b, c; a = b = c = 10; // a, b, and c are all 10`
- When assigning a variable to itself using a compound assignment operator, the operation is performed using the original value of the variable. `java int a = 5; a += a++; // a is now 10 (5 + 5, then a is incremented)`
- When using compound assignment operators with expressions, be aware of operator precedence to avoid unexpected results. `java int a = 10; a *= 2 + 5; // a is now 70, not 25! (a = a * (2 + 5))`

Equality Operators

Equality operators in Java are used to compare two values for equality or inequality. They return a `boolean` result (`true` or `false`) based on the comparison.

Java provides two equality operators: - `==` (equal to) - `!=` (not equal to)

Here are examples of using these operators:

```
int a = 10;
int b = 20;

boolean result1 = (a == b); // false
boolean result2 = (a != b); // true
```

Understanding Equality When using equality operators, it's important to understand how Java compares values for equality.

For primitive types, the `==` operator compares the actual values:

```
int a = 10;
int b = 10;
boolean result = (a == b); // true
```

However, for objects, the `==` operator compares the object references, not the contents of the objects:

```
String s1 = new String("Hello");
String s2 = new String("Hello");
boolean result = (s1 == s2); // false
```

In this case, `s1` and `s2` are two different objects in memory, even though they contain the same string value.

To compare the contents of objects, you should use the `equals()` method:

```
String s1 = new String("Hello");
String s2 = new String("Hello");
boolean result = s1.equals(s2); // true
```

However, when comparing objects using the `equals()` method, it's important to ensure that the class of the objects has overridden the `equals(Object)` method to provide a meaningful comparison.

The default implementation of `equals(Object)` in the `Object` class simply compares object references, just as the == operator does. To compare the contents of objects, you need to override `equals(Object)` in your class:

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

    // Constructor, getters, setters...

    @Override
    public boolean equals(Object obj) {
        // Implementation...
    }
}
```

To override the `equals` method in Java, you need to follow certain rules to ensure the method works correctly and adheres to the contract defined by the `Object` class:

1. **Symmetry**: If `a.equals(b)` is `true`, then `b.equals(a)` must also be `true`.
2. **Reflexivity**: An object must be equal to itself; that is, `a.equals(a)` must be `true`.
3. **Transitivity**: If `a.equals(b)` is `true` and `b.equals(c)` is `true`, then `a.equals(c)` must be `true`.
4. **Consistency**: If `a.equals(b)` returns `true` once, it must continue to return `true` as long as neither object is modified. Similarly, if it returns `false`, it must consistently return `false`.
5. **Non-nullity**: `a.equals(null)` must always return `false`.

Here is an example of overriding the `equals` method in the `Person` class:

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

    // Constructor, getters, and setters

    @Override // 1.
    public boolean equals(Object obj) {
        // 2. Check if obj is the same as this object
        if (this == obj) {
            return true;
        }
        // 3. Check if obj is null or not an instance of Person
        if (obj == null || getClass() != obj.getClass()) {
            return false;
        }
        // 4. Cast obj to Person and compare significant fields
        Person person = (Person) obj;

        // 5. Compare significant fields
        return age == person.age && (name != null ? name.equals(person.name) : person.name == null);
    }

    @Override
    public int hashCode() {
```

```
        // Ensure consistency with the equals method
        int result = name != null ? name.hashCode() : 0;
        result = 31 * result + age;
        return result;
    }
}
```

The above example shows how to correctly override the `equals` method:

1. **Use the `@Override` annotation**: This ensures you are correctly overriding the method and helps with readability.
2. **Check for `null`**: The first check should be to see whether the object being compared is `null`.
3. **Check for type**: Use the `instanceof` operator to ensure the objects being compared are of the same type.
4. **Cast the object**: Cast the object to the correct type after checking.
5. **Compare significant fields**: Compare the fields that determine equality using the `==` operator for primitive fields and the `equals` method for object fields.

Also, always override `hashCode` when overriding `equals` to maintain the general contract that equal objects must have equal hash codes.

Summary of Equality and Inequality Operators Here's a summary table of the equality and inequality operators in Java:

Operator	Name	Example
`==`	Equal to	`a == b`
`!=`	Not equal to	`a != b`

There are some nuanced points to consider: - When comparing floating-point values (float and double) using `==` or `!=`, be aware that the results may not be as expected due to the imprecise nature of floating-point representation. `java double a = 0.1 + 0.2; double b = 0.3; boolean result = (a == b); // false`

- The `==` and `!=` operators have higher precedence than the logical operators (`&&`, `||`) but lower precedence than the relational operators (`<`, `>`, `<=`, `>=`).
- The `!=` operator is the negation of the `==` operator, so `a != b` is equivalent to `!(a == b)`.

Relational Operators

Relational operators in Java are used to compare two values and determine their relationship. They return a boolean result (`true` or `false`) based on the comparison.

Java provides four relational operators: - `<` (less than) - `>` (greater than) - `<=` (less than or equal to) - `>=` (greater than or equal to)

These operators can be used with numeric primitive types and `char`.

Here are examples of using relational operators with integer values:

```
int a = 10;
int b = 20;

boolean result1 = (a < b);  // true
boolean result2 = (a > b);  // false
boolean result3 = (a <= b); // true
boolean result4 = (a >= b); // false
```

And here's another with char values, which compares the Unicode values of the characters:

```
char c1 = 'a';
char c2 = 'b';
boolean result = (c1 < c2); // true
```

Summary of Relational Operators Here's a summary table of the relational operators in Java:

Operator	Name	Example
`<`	Less than	`a < b`
`>`	Greater than	`a > b`
`<=`	Less than or equal to	`a <= b`
`>=`	Greater than or equal to	`a >= b`

Also, there are two nuanced points to consider: - When comparing floating-point values (`float` and `double`) using relational operators, be aware that the results may not be as expected due to the imprecise nature of floating-point representation. `java double a = 0.1 + 0.2; double b = 0.3; boolean result = (a <= b); // false`

- Relational operators have higher precedence than equality operators (`==`, `!=`) and logical operators (`&&`, `||`), but lower precedence than arithmetic operators (`+`, `-`, `*`, `/`, `%`).

Logical Operators

Logical operators in Java are used to perform logical operations on boolean expressions. They return a boolean result (`true` or `false`) based on the operands and the specific operator used.

Java provides six logical operators: - `&` (logical AND) - `|` (logical OR) - `^` (logical XOR) - `&&` (short-circuit logical AND) - `||` (short-circuit logical OR) - `!` (logical NOT)

These operators can be used with `boolean` values or expressions that evaluate to `boolean` values.

Here are examples of using logical operators:

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

boolean result1 = a & b;  // false
boolean result2 = a | b;  // true
boolean result3 = a ^ b;  // true
boolean result4 = a && b; // false
boolean result5 = a || b; // true
boolean result6 = !a;     // false
```

And these are the truth tables for the logical AND (`&`), logical OR (`|`), and logical XOR (`^`) operators:

Logical AND (`&`):

a	b	a & b
`false`	`false`	`false`
`false`	`true`	`false`
`true`	`false`	`false`
`true`	`true`	`true`

Logical OR (`|`):

a	b	a | b
`false`	`false`	`false`
`false`	`true`	`true`
`true`	`false`	`true`
`true`	`true`	`true`

Logical XOR (`^`):

a	b	a ^ b
`false`	`false`	`false`
`false`	`true`	`true`
`true`	`false`	`true`
`true`	`true`	`false`

In summary: - `&` (AND) is `true` only if both operands are `true`. - `|` (OR) is `true` if at least one operand is `true`. - `^` (XOR) is `true` if exactly one operand is `true`.

Short-circuit The `&&` and `||` operators are short-circuit operators. They evaluate the second operand only if it is necessary, based on the result of the first operand.

For `&&`, if the first operand is `false`, the entire expression will be `false`, regardless of the second operand. Therefore, the second operand is not evaluated.

For `||`, if the first operand is `true`, the entire expression will be `true`, regardless of the second operand. Therefore, the second operand is not evaluated.

Short-circuiting can be useful for avoiding a `NullPointerException` when checking for `null` before accessing an object's methods or fields:

```
String str = null;
if (str != null && str.length() > 0) {
    // This code will not throw a NullPointerException
}
```

However, be careful when using short-circuit operators with expressions that have side effects (for example, method calls that modify data or have other consequences):

```
int a = 10;
if (a > 5 || ++a > 10) {
    // a will be 11 if a > 5, but will remain 10 if a <= 5
}
```

Summary of Logical Operators Here's a summary table of the logical operators in Java:

Operator	Name	Example
`&`	Logical AND	`a & b`
`|`	Logical OR	`a | b`
`^`	Logical XOR (exclusive OR)	`a ^ b`
`&&`	Short-circuit logical AND	`a && b`
`||`	Short-circuit logical OR	`a || b`
`!`	Logical NOT	`!a`

Here are some nuanced points to consider: - The `&`, `|`, and `^` operators can also be used as bitwise operators when applied to integer types (`byte`, `short`, `int`, `long`). In this context, they perform bitwise AND, OR, and XOR operations on the individual bits of the operands.

- The `!` operator has higher precedence than the `&`, `|`, `^`, `&&`, and `||` operators.
- The `&`, `|`, and `^` operators have lower precedence than the `&&` and `||` operators.
- The `&`, `|`, and `^` operators always evaluate both operands, even if the result can be determined from the first operand. This can be less efficient than using the short-circuit operators `&&` and `||` when the second operand is expensive to evaluate or has side effects.
- The `^` operator returns `true` if and only if exactly one of its operands is `true`. This is different from the behavior of the `!=` operator, which returns true if the operands are not equal.
- Logical operators can be combined to form complex boolean expressions. It's important to use parentheses to clearly specify the intended order of evaluation when multiple operators are used. `java boolean result = (a && b) || (c && d);`

`String` and `StringBuilder`

A string is simply a sequence of characters. However, under the hood, strings have some unique properties and optimizations that are important to understand.

```
String greeting = "Hello World!";
```

The `String` class in Java is immutable, meaning once a string object is created, its value cannot be changed. This may seem counterintuitive at first, after all, we often modify strings in our programs. But what's really happening is that a new string object is being created each time, while the original remains unchanged.

This immutability brings some advantages. Strings can be shared safely between multiple parts of a program without worrying about one part accidentally modifying the string for everyone else. The JVM can also optimize memory by reusing common strings.

However, immutability also means that operations which modify a string (like concatenation) are less efficient, because a new string must be created each time.

Creating Strings

There are a few ways to create a string in Java:

```
String literalString = "I am a literal string";
String objectString = new String("I am a String object");
```

Both achieve the same end result, a string with the specified value. However, there's a slight difference in how the JVM handles these.

When you create a string literal, the JVM first checks the **string pool**, a special area of memory reserved just for strings. If an equivalent string already exists in the pool, the JVM simply returns a reference to that existing string, rather than allocating new memory.

```
String s1 = "Hello";
String s2 = "Hello";
System.out.println(s1 == s2);  // Prints 'true'
```

Here, `s1` and `s2` actually refer to the same string object in memory, because `"Hello"` was already in the string pool.

In contrast, using the `new` keyword always creates a new object, even if an equivalent string already exists in the pool.

```
String s3 = new String("Hello");
System.out.println(s1 == s3);  // Prints 'false'
```

Here, despite `s1` and `s3` having the same content, they refer to different objects in memory.

If you have a string object and you want to ensure it's using the memory-optimized string from the pool, you can use the `intern()` method.

```
String s4 = s3.intern();
System.out.println(s1 == s4);  // Prints 'true'
```

After interning `s3`, `s4` now refers to the same pooled string as `s1`.

However, it's important to use `intern()` judiciously. Overuse can actually lead to performance issues, as the string pool is a finite resource. It's best used for strings that you expect to be frequently reused throughout your program.

String Concatenation

Concatenating strings is a common operation, and Java provides two main ways to do it.

```
String s1 = "Hello";
String s2 = "World";
String s3 = s1 + " " + s2;  // Using the + operator
String s4 = s1.concat(" ").concat(s2);  // Using the concat() method
```

Both approaches yield the same result. However, there are some differences to consider.

The `+` operator is often more readable and is optimized by the Java compiler into a `StringBuilder` operation (which we'll cover shortly). When you use `+`, the compiler actually transforms it into something like this:

```
String s3 = new StringBuilder(s1).append(" ").append(s2).toString();
```

So, even though it looks like you're creating a new string with each `+`, the compiler is smart enough to use a `StringBuilder` under the hood to optimize it.

On the other hand, the `concat` method is a direct method of the `String` class. It concatenates the specified string to the end of the current string and returns a new string. This is the signature of the method:

```
String concat(String str)
```

And here's another example:

```
String s4 = s1.concat(" ");  // s4 is "Hello "
s4 = s4.concat(s2);  // s4 is now "Hello World"
```

One advantage of using `concat` is that it's more explicit about what's happening, you're calling a method to concatenate strings, rather than using an operator. This can make the code more readable, especially for developers who are new to Java and might not be familiar with how the `+` operator is optimized.

However, `concat` can only concatenate one string at a time, so for concatenating multiple strings, you'd need multiple calls to `concat`, which can get cumbersome. The `+` operator allows for concatenating multiple strings in a single expression, which is often more convenient.

Ultimately, the choice between `+` and `concat` often comes down to personal preference and coding style. Many developers prefer `+` for its conciseness and readability, while others prefer the explicitness of `concat`.

However, it's important to be cautious when using either approach in loops, as it can lead to performance issues due to the creation of many intermediate string objects. In such cases, `StringBuilder` should be used directly.

```
String result = "";
for (int i = 0; i < 100; i++) {
```

```
    result = result.concat(Integer.toString(i));  // Inefficient!
}
```

This code will create a new string on each iteration of the loop. For large numbers of iterations, this can be very inefficient in terms of both time and memory. A `StringBuilder` should be used in such cases (which we'll discuss later).

Important String Methods

The `String` class provides a rich set of methods for examining and manipulating string content. Here are some of the most commonly used ones:

- `int length()`: Returns the number of characters in the string.
- `char charAt(int index)`: Returns the character at the specified index.
- `int indexOf(String str)`: Returns the index within the string of the first occurrence of the specified substring.
- `String substring(int beginIndex, int endIndex)`: Returns a new string that is a substring of this string.
- `String toLowerCase()`: Converts all of the characters in this string to lower case.
- `String toUpperCase()`: Converts all of the characters in this string to upper case.
- `boolean equals(Object anObject)`: Compares this string to the specified object.
- `boolean equalsIgnoreCase(String anotherString)`: Compares this string to another string, ignoring case considerations.
- `boolean startsWith(String prefix)`: Tests if this string starts with the specified prefix.
- `boolean endsWith(String suffix)`: Tests if this string ends with the specified suffix.
- `boolean contains(CharSequence s)`: Returns `true` if and only if this string contains the specified sequence of `char` values.
- `String replace(char oldChar, char newChar)`: Returns a new string resulting from replacing all occurrences of `oldChar` in this string with `newChar`.
- `String strip()`: Returns a string whose value is this string, with all leading and trailing white space removed.
- `String trim()`: Returns a string whose value is this string, with all leading and trailing space removed, where space is defined as any character whose codepoint is less than or equal to `'U+0020'` (the space character).
- `String indent(int n)`: Adjusts the indentation of each line of this string based on the value of `n`.
- `String stripIndent()`: Returns a string whose value is this string, with incidental whitespace removed from the beginning and end of every line.
- `boolean isEmpty()`: Returns `true` if, and only if, `length()` is 0.
- `boolean isBlank()`: Returns `true` if the string is empty or contains only white space codepoints, otherwise `false`.

Each of these methods provides a specific utility, and together they form a powerful toolkit for working with strings. However, remember that due to the immutability of strings, the methods that have `String` as return type return a new string rather than modifying the original. For example:

```
String s1 = "  Hello World   ";
String s2 = s1.strip();
```

```
System.out.println(s1);  // Still prints "  Hello World   "
System.out.println(s2);  // Prints "Hello World"
```

This also allows for a technique known as method chaining, where multiple methods are invoked in a single expression.

```
String result = "  Hello World  ".trim().toUpperCase().replace('O', '0');
System.out.println(result);  // Prints "HELL0 W0RLD"
```

Here, the original string is trimmed of whitespace, then converted to uppercase, and finally all `'O'` characters are replaced with `'0'`. Each method returns a new string that becomes the base for the next method in the chain.

Chaining can make your code more concise and readable, but it's important not to overdo it. Excessively long chains can be hard to understand and debug.

Also, some of these methods work with indexes. An index ranges from 0 to `length()` - 1. The first char value of the sequence is at index 0, the next at index 1, and so on, just like it works with arrays.

Overriding `toString()`

One special method to be aware of is `toString()`. This method is defined in the `Object` class, which all classes in Java inherit from. It returns a string representation of the object.

By default, this string is not very informative (it includes the object's class name and hash code). However, we can override `toString()` in our own classes to provide a more useful representation.

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

    // Constructor and other methods...

    @Override
    public String toString() {
        return "Person[name=" + name + ",age=" + age + "]";
    }
}
```

Now, when we print a `Person` object, we'll get a nicely formatted string:

```
Person alice = new Person("Alice", 25);
System.out.println(alice);  // Prints "Person[name=Alice,age=25]"
```

This is especially useful for logging and debugging purposes.

Formatting Strings

In addition to manipulating strings, Java provides powerful tools for formatting them. The `String` class includes `format()` and `formatted()` methods which allow you to create a formatted string using a format string and arguments.

```
String name = "Alice";
int age = 25;
String city = "Florida";
String formatted = String.format("My name is %s, I'm %d years old, and I live in %s.", name, age, city);
System.out.println(formatted);
// Prints "My name is Alice, I'm 25 years old, and I live in Florida."
```

The format string includes placeholders (`%s` for strings, `%d` for integers, etc.) which are replaced by the corresponding arguments.

Here are some of the most common formatting placeholders:

Format Specifier	Description
`%s`	String
`%c`	Character
`%d`	Decimal integer
`%f`	Floating-point number
`%t`	Date/time
`%n`	Newline

These are just a few examples, the full list of formatting options is quite extensive, allowing for precise control over the output format.

Using the `StringBuilder` Class

While the `String` class is powerful, its immutability can lead to performance issues when you need to make many modifications to a string. This is where `StringBuilder` comes in.

`StringBuilder` is a mutable sequence of characters. It provides similar methods to `String` for appending, inserting, and deleting characters. However, these methods modify the `StringBuilder` itself rather than creating a new object.

This mutability allows for more efficient code when you need to make multiple modifications to a string. With `String`, each modification creates a new string object, which can be costly in terms of time and memory if done frequently, such as in a loop. `StringBuilder` avoids this by modifying its internal character sequence directly.

Furthermore, `StringBuilder` methods can be chained together, similar to `String` methods. However, because `StringBuilder` is mutable, each method in the chain modifies the same `StringBuilder` instance and returns a reference to it, allowing for further chaining:

```
StringBuilder sb = new StringBuilder("Hello");
sb.append(" World").insert(0, "Hey, ").delete(4, 9);
System.out.println(sb);  // Prints "Hey, World"
```

In this example, we start with a `StringBuilder` containing `"Hello"`. We then append `" World"` to it, insert `"Hey, "` at the beginning, and delete the characters from index 4 to 8 (inclusive). Each of these operations modifies the same `StringBuilder` instance.

You can create a `StringBuilder` in a few ways:

```
StringBuilder sb1 = new StringBuilder();  // Creates an empty StringBuilder
StringBuilder sb2 = new StringBuilder(10);  // Creates a StringBuilder with initial capacity of 10
StringBuilder sb3 = new StringBuilder("Hello");  // Creates a StringBuilder initialized with the string "Hello"
```

When you create a `StringBuilder` without specifying an initial string, it starts with a default capacity of 16 characters. If you know that you'll be building a larger string, you can specify a higher initial capacity to avoid automatic resizing later, which can be costly.

Important `StringBuilder` Methods

`StringBuilder` provides many of the same methods as `String` for examining and modifying the character sequence, however, it's worth mentioning two things: - These methods modify the `StringBuilder` instance itself rather than creating a new one. - `StringBuilder` does not extend from `String`, however, both classes extend from the CharSequence interface.

Common methods with `String`: - `int length()`: Returns the number of characters in the `StringBuilder`. - `char charAt(int index)`: Returns the character at the specified position. - `int indexOf(String str)`: Returns

the index within this string of the first occurrence of the specified substring. - `String substring(int start)` and `substring(int start, int end)`: Returns a new string that is a substring of this sequence.

Appending values: - `StringBuilder append(...)`: Appends the string representation of the argument to the sequence. There are overloads for all primitive types, `char` arrays, `CharSequence`, and `Object`.

Inserting data: - `StringBuilder insert(int offset, ...)`: Inserts the string representation of the second argument into the sequence at the position specified by the first argument. There are overloads for all primitive types, `char` arrays, `CharSequence`, and `Object`.

Deleting contents: - `StringBuilder delete(int start, int end)`: Removes the characters in a substring of this sequence. - `StringBuilder deleteCharAt(int index)`: Removes the char at the specified position.

Replacing portions: - `StringBuilder replace(int start, int end, String str)`: Replaces the characters in a substring of this sequence with characters in the specified string. - `setCharAt(int index, char ch)`: Sets the character at the specified index to `ch`.

Reversing: - `void StringBuilder reverse()`: Causes this character sequence to be replaced by the reverse of the sequence.

Converting to `String`: - `String toString()`: Returns a string representing the data in this sequence.

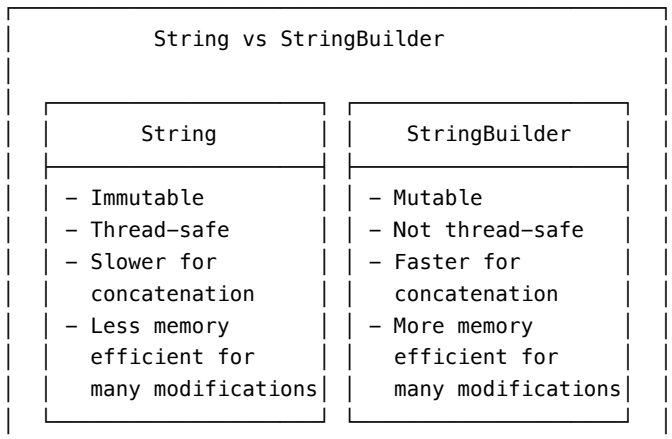
Here's an example demonstrating some of these methods:

```
StringBuilder sb = new StringBuilder("Hello");
sb.append(" there");  // Now contains "Hello there"
sb.insert(5, ",");  // Now contains "Hello, there"
sb.replace(7, 12, "world");  // Now contains "Hello, world"
sb.delete(5, 7);  // Now contains "Helloworld"
sb.reverse();  // Now contains "dlrowolleH"

String finalString = sb.toString();
System.out.println(finalString);  // Prints "dlrowolleH"
```

In this example, we start with a `StringBuilder` containing `"Hello"`, then append `" there"` to it, insert a comma after `"Hello"`, replace `"there"` with `"world"`, delete the comma and space, reverse the whole string, and finally convert it to a `String`.

It's important to note that while `StringBuilder` is mutable, it's not synchronized. If multiple threads are accessing the same `StringBuilder` instance concurrently and at least one of the threads is modifying it, you should ensure proper synchronization in your code to avoid data corruption. If you need a thread-safe version, you can use `StringBuffer`, which is like `StringBuilder` but is synchronized (at the cost of some performance overhead).

Finally, here's a diagram that summarizes the differences between `String` and `StringBuilder`:



```
|                                                    |
|  Use for:                Use for:                  |
|  - Constant strings      - Building strings        |
|  - Simple concatenation  - Many modifications      |
|  - Thread safety needed  - Performance critical    |
|                            string operations       |
└────────────────────────────────────────────────────┘
```

Text Blocks

A text block provides a more concise and intuitive syntax for representing strings that preserves newlines and indentation without the need for explicit escape sequences or concatenation:

```
String traditional = "{\n" +
                     "  \"name\": \"John Doe\",\n" +
                     "  \"age\": 30\n" +
                     "}";

String textBlock = """
                   {
                     "name": "John Doe",
                     "age": 30
                   }
                   """;
```

As you can see, the text block version is much cleaner and easier to read. It gets rid of all the noise of newline characters (`\n`) and escaped quotes (`\"`) that clutter the traditional string literal. With text blocks, what you see is what you get, the string appears in your code exactly as it will be outputted.

To define a text block, you use three double-quotes (`"""`) as the opening and closing delimiters. The content of the text block appears between these delimiters and can span multiple lines:

```
String textBlock = """
                   This is a Text Block.
                   It can contain multiple lines,
                     indentation,
                   and "special" characters.
                   """;
```

Note that the closing delimiter (`"""`) must appear on a line by itself and be followed by a semicolon. Any whitespace after the closing delimiter on that line will be ignored.

A common misconception is that you can use a single quote to close a text block that was opened with triple quotes. However, this is not the case. The opening and closing delimiters for a text block must always be three double-quotes (`"""`).

The compiler treats all content between the delimiters as part of the string literal, including newlines, indentation, and any other whitespace. However, there are a few rules around indentation and escaping that we'll discuss shortly.

Features of Text Blocks

One of the key features of text blocks is their ability to represent multi-line strings naturally, without resorting to explicit newline characters or string concatenation.

```
String multiLine = """
                   First line
                   Second line
```

```
                    Third line
                    """;
```

This text block will preserve the newlines and indentation exactly as written, resulting in a string with three lines of text. You don't need to manually add `\n` characters or worry about lining up concatenated strings.

Text blocks also provide automatic indentation handling based on the position of the closing delimiter. The compiler will determine a common whitespace prefix from the lines between the delimiters and automatically strip that prefix from each line.

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

// Equivalent to:
// "Line 1\n  Line 2\nLine 3\n"
```

In this example, the closing delimiter is aligned with the least indented line (`Line 1`). Therefore, the common whitespace prefix is four spaces, which gets stripped from each line. The resulting string will have `Line 2` indented by two spaces relative to the other lines.

It's important to note that text blocks do not automatically trim all leading and trailing whitespace. The compiler only removes the common whitespace prefix based on the closing delimiter's position. Any additional leading or trailing whitespace will be preserved in the final string.

Consider this example:

```
String traditional = "  \n  ";  // This evaluates to two spaces, a newline, and two more spaces.
String textBlock = """
                     \n
                    """;        // Evaluates to two spaces, a newline, two spaces,
                                // and an additional final newline added by the text block syntax.
```

Here, the `traditional` string will include spaces before and after the newline as they are explicitly part of the string. In the `textBlock`, all spaces and the newline are preserved as they appear, and a newline is added at the end because of how text blocks handle the closing delimiter.

Another important aspect of text blocks is escaping special characters. The rules for escaping in text blocks are mostly the same as in traditional string literals, with a few caveats: - A single double quote or a pair of double quotes within a text block does not need to be escaped. - Triple double quotes within the text block need to be escaped to prevent them from prematurely ending the block. - To include a backslash character (\) in the text block, you need to escape it with another backslash (\\).

Here's an example:

```
String escaped = """
                 This is a "quoted" text with \\ and \u0040.
                 """;
```

A common misconception is that backslashes are ignored in text blocks since they are multi-line literals. However, this is not the case. Backslashes still have their special meaning in text blocks and need to be escaped if you want to include a literal backslash in the string.

And, as you can see in the above example, text blocks also support the use of Unicode escapes (`\uXXXX`) for representing characters by their Unicode code points.

Lastly, text blocks can be combined with traditional string literals and even other text blocks using the `+` operator, just like regular strings:

```
String name = "John";
String greeting = """
                  Hello, """ + name + """
                  . How are you?
                  """;
```

Here, we concatenate a text block with a traditional string literal (`name`) to create a personalized greeting. The `+` operator appears on the same line as the opening and closing delimiters of the text blocks. Placing it on a separate line might lead to unintended whitespace in the resulting string.

The Math API

The `Math` API provides a rich set of static methods for performing mathematical operations. It includes methods for finding the minimum and maximum of two values, rounding numbers, determining the ceiling and floor of a value, and generating random numbers. Let's review each of these.

Finding the Minimum and Maximum

The `Math` class provides `min` and `max` methods to find the minimum and maximum of two values respectively. These methods are overloaded to accept `int`, `long`, `float`, and `double` arguments:

```
static double max(double a, double b)
static float max(float a, float b)
static int max(int a, int b)
static long max(long a, long b)

static double min(double a, double b)
static float min(float a, float b)
static int min(int a, int b)
static long min(long a, long b)
```

Here are some examples:

```
int min = Math.min(5, 10);  // min is 5
int max = Math.max(5, 10);  // max is 10

double min2 = Math.min(5.7, 10.2);  // min2 is 5.7
double max2 = Math.max(5.7, 10.2);  // max2 is 10.2
```

These methods are useful when you need to enforce a range on a value.

Rounding Numbers

The `Math` class provides several methods for rounding numbers:

- `int round(float)` and `long round(double)`: These methods return the closest `int` or `long` to the argument. Halfway values (like 0.5) are rounded up, following the round half up convention.
- `double rint(double)`: Returns the `double` value that is closest in value to the argument and is equal to a mathematical integer. If two `double` values that are mathematical integers are equally close, the even one is chosen.
- `double floor(double)`: Returns the largest (closest to positive infinity) `double` value that is less than or equal to the argument and is equal to a mathematical integer.
- `double ceil(double)`: Returns the smallest (closest to negative infinity) `double` value that is greater than or equal to the argument and is equal to a mathematical integer.

Here are some examples:

```
long roundedLong = Math.round(5.7);  // roundedLong is 6
int roundedInt = Math.round(5.4f);  // roundedInt is 5
```

```java
double rintValue = Math.rint(5.5);  // rintValue is 6.0 (ties round to even)
double rintValue2 = Math.rint(6.5);  // rintValue2 is 6.0

double floorValue = Math.floor(5.7);  // floorValue is 5.0
double ceilingValue = Math.ceil(5.2);  // ceilingValue is 6.0
```

As you can see, the floor is the largest integer less than or equal to the value, while the ceiling is the smallest integer greater than or equal to the value.

Generating Random Numbers

The `Math` class includes a `random()` method that returns a `double` value with a positive sign, greater than or equal to 0.0 and less than 1.0:

```java
static double random()
```

This method is useful for generating random numbers.

```java
double randomValue = Math.random();  // randomValue is a random double between 0.0 and 1.0
```

You can use `Math.random()` in combination with other `Math` methods to generate random numbers in a specific range. For example, to generate a random integer between 1 and 10 (inclusive), you can do this:

```java
int randomInt = (int)(Math.random() * 10) + 1;
```

Here's how this works: 1. `Math.random()` generates a random double between 0.0 and 1.0, let's call it `r`. 2. `r * 10` is then a random double between 0.0 and 10.0. 3. `(int)(r * 10)` casts this double to an int, effectively rounding it down. So now we have a random integer between 0 and 9. 4. Finally, we add 1 to shift the range to between 1 and 10.

You can adjust this formula to generate random numbers in any integer range. For example, to generate a random number between `min` and `max` (inclusive), you can use:

```java
int randomNum = (int)(Math.random() * (max - min + 1)) + min;
```

Key Points

- Java has 8 primitive data types: `byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, and `char`.
- Primitive types are the most basic data types and are not objects. They store simple values directly in memory.
- Integer literals can be assigned using decimal, hexadecimal (prefix `0x` or `0X`), octal (prefix `0`), or binary (prefix `0b` or `0B`) notation.
- Underscores can be used in numeric literals for improved readability but have restrictions on their placement.
- Reference types store the memory address where an object resides, rather than the object itself.
- Wrapper classes (`Boolean`, `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Character`) allow primitives to be used as objects.
- Autoboxing automatically converts a primitive to its wrapper class, while unboxing converts a wrapper object to its primitive type.
- Wrapper classes provide methods for parsing, converting between types, and more.
- Wrapper objects can be `null`, while primitives cannot. Unboxing a `null` wrapper object throws a `NullPointerException`.
- Java provides a rich set of operators for mathematical, logical, and bitwise operations.

- Operator precedence determines the order of evaluation in expressions. Parentheses can change the default precedence.
- Unary operators (`++`, `--`, `+`, `-`, `~`, `!`) operate on a single operand. Increment and decrement operators (`++` and `--`) can be used in prefix or postfix form.
- Binary operators (`+`, `-`, `*`, `/`, `%`) operate on two operands. Numeric promotion automatically converts operands to a larger type to prevent precision loss.
- Bitwise operators (`&`, `|`, `^`, `~`) and shift operators (`<<`, `>>`, `>>>`) manipulate individual bits of integer values.
- Assignment operators (`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `^=`, `|=`, `<<=`, `>>=`, `>>>=`) assign values to variables. Compound assignment operators combine an operation with assignment.
- Equality operators (`==` and `!=`) compare values for equality. For objects, `==` compares references, while equals() compares contents.
- Relational operators (`<`, `>`, `<=`, `>=`) compare values and determine their relationship.
- Logical operators (`&`, `|`, `^`, `&&`, `||`, `!`) perform logical operations on boolean expressions. Short-circuit operators (`&&` and `||`) can skip evaluating the second operand based on the first operand's value.
- Strings in Java are immutable, meaning their value cannot be changed once created. Any operation that appears to modify a string actually creates a new string.
- String literals are stored in the string pool, a special area of memory. If an equivalent string already exists in the pool, a reference to that string is returned instead of creating a new object.
- Strings can be concatenated using the `+` operator or the `concat()` method. The `+` operator is optimized by the compiler into a `StringBuilder` operation.
- The `String` class provides many methods for examining and manipulating string content, such as `length()`, `charAt()`, `substring()`, `toLowerCase()`, `equals()`, `startsWith()`, `endsWith()`, `replace()`, `trim()`, and more.
- The `toString()` method, inherited from the `Object` class, returns a string representation of an object. It can be overridden in custom classes to provide a more informative representation.
- The `String` class provides `format()` and `formatted()` methods for creating formatted strings using placeholders.
- `StringBuilder` is a mutable sequence of characters. It provides similar methods to `String` for appending, inserting, and deleting characters, but these methods modify the `StringBuilder` itself rather than creating a new object.
- `StringBuilder` is more efficient than `String` when many modifications need to be made, as it avoids creating a new object for each modification.
- Important `StringBuilder` methods include `append()`, `insert()`, `delete()`, `replace()`, `reverse()`, and `toString()`.
- Text blocks provide a more concise and intuitive syntax for representing multi-line strings. They are defined using triple double-quotes (`"""`) as delimiters.
- Text blocks automatically handle newlines, indentation, and common whitespace prefixes, making them easier to read and write than traditional string literals.
- The `Math` class provides many static methods for performing mathematical operations, including `min()`, `max()`, `round()`, `floor()`, `ceil()`, and `random()`.
- The `random()` method can be used in combination with other `Math` methods to generate random numbers in a specific range.

Practice Questions

1. Which of the following statements about Java primitive and reference data types is true?

A) A `double` can be directly assigned to a `float` without casting.
B) A `boolean` can be cast to an `int`.
C) A `String` can be assigned to an `Object` reference variable.
D) A `char` is a reference data type.
E) An `int` can store a `long` value without any explicit casting.

2. What is the output of the following code snippet?

```
public class OperatorTest {
    public static void main(String[] args) {
        int a = 5;
        int b = 10;
        int c = 15;
        int result = a + b * c / a - b;
        System.out.println(result);
    }
}
```

A) `25`
B) `35`
C) `20`
D) `15`

3. Which of the following statements about `String` and `StringBuilder` is true?

A) `StringBuilder` objects are immutable.
B) `String` objects can be modified after they are created.
C) `StringBuilder` is synchronized and thread-safe.
D) `StringBuilder` provides methods for mutable sequence of characters.
E) `String` and `StringBuilder` have the same performance characteristics for string manipulation.

4. Which of the following statements about text blocks are true? (Choose all that apply.)

A) Text blocks can span multiple lines without needing escape sequences for new lines.
B) Text blocks preserve the exact format, including whitespace, of the code as written.
C) Text blocks can only be used within methods.
D) Text blocks automatically trim leading and trailing whitespace from each line.
E) Text blocks require a minimum indentation level of one space.

5. Which of the following statements about the `Math` class is true?

A) The `Math.round()` method returns a `double`.
B) The `Math.random()` method returns a random integer.
C) The `Math.max()` method can only be used with integers.
D) The `Math.pow()` method returns the result of raising the first argument to the power of the second argument.
E) The `Math.abs()` method can only be used with positive numbers.

Chapter FOUR

Working with Data

Answers

1. The correct answer is C.

Explanation:

- **A)** A `double` can be directly assigned to a `float` without casting.
  - This option is incorrect. A `double` cannot be directly assigned to a `float` without casting because `double` has a larger range and precision than a `float`.
- **B)** A `boolean` can be cast to an `int`.
  - This option is incorrect. `boolean` values cannot be cast to `int` in Java. They are not compatible types.
- **C)** A `String` can be assigned to an `Object` reference variable.
  - This option is correct. A `String` is an instance of the `Object` class, and hence it can be assigned to an `Object` reference variable.
- **D)** A `char` is a reference data type.
  - This option is incorrect. `char` is a primitive data type, not a reference data type.
- **E)** An `int` can store a `long` value without any explicit casting.
  - This option is incorrect. an `int` cannot store a `long` value without explicit casting because `long` has a larger range than `int`.

2. The correct answer is A.

Explanation:

Let's break down the expression `a + b * c / a - b` step-by-step according to the order of operations:

1. **Multiplication and Division** are performed first from left to right:
   - `b * c` = `10 * 15` = `150`
   - `150 / a` = `150 / 5` = `30`
2. **Addition and Subtraction** are performed next from left to right:
   - `a + 30` = `5 + 30` = `35`
   - `35 - b` = `35 - 10` = `25`

So, the value of `result` is `25`, and the program prints `25`.

- **A)** `25`
  - This option is correct.
- **B)** `35`
  - This option is incorrect.
- **C)** `20`
  - This option is incorrect.
- **D)** `15`
  - This option is incorrect.

3. The correct answer is D.

Explanation:

- **A)** `StringBuilder` objects are immutable.
  - This option is incorrect. `StringBuilder` objects are mutable, meaning they can be changed after they are created.
- **B)** `String` objects can be modified after they are created.
  - This option is incorrect. `String` objects are immutable, meaning once a `String` object is created, it cannot be modified. Any modification results in a new `String` object.
- **C)** `StringBuilder` is synchronized and thread-safe.
  - This option is incorrect. `StringBuilder` is not synchronized and is not thread-safe. If synchronization is required, `StringBuffer` should be used instead.
- **D)** `StringBuilder` provides methods for mutable sequence of characters.
  - This option is correct. `StringBuilder` provides methods for a mutable sequence of characters, allowing for modification of the object without creating new instances.
- **E)** `String` and `StringBuilder` have the same performance characteristics for string manipulation.

  - This option is incorrect. `String` and `StringBuilder` do not have the same performance characteristics for string manipulation. `StringBuilder` is generally more efficient for such operations because it is mutable and does not create new instances with each modification.

4. The correct answers are A and B.

Explanation:

- **A)** Text blocks can span multiple lines without needing escape sequences for new lines.
  - This option is correct. Text blocks can indeed span multiple lines without needing escape sequences for new lines, making it easier to work with multi-line strings.
- **B)** Text blocks preserve the exact format, including whitespace, of the code as written.
  - This option is correct. Text blocks preserve the exact format, including whitespace, of the code as written. This is useful for maintaining the original layout of the text.
- **C)** Text blocks can only be used within methods.
  - This option is incorrect. Text blocks can be used anywhere a regular `String` can be used, not just within methods. They can be part of class fields, method parameters, etc.
- **D)** Text blocks automatically trim leading and trailing whitespace from each line.
  - This option is incorrect. Text blocks do not automatically trim leading and trailing whitespace from each line. They preserve the exact whitespace as written in the code.
- **E)** Text blocks require a minimum indentation level of one space.
  - This option is incorrect. Text blocks do not require a minimum indentation level of one space. The leading indentation common to all lines is removed automatically, but lines within the text block can have zero or more leading spaces.

5. The correct answer is D.

Explanation:

- **A)** The `Math.round()` method returns a `double`.
  - This option is incorrect. The `Math.round()` method returns a `long` when given a `double` argument and an `int` when given a `float` argument.
- **B)** The `Math.random()` method returns a random integer.
  - This option is incorrect. The `Math.random()` method returns a `double` value between 0.0 (inclusive) and 1.0 (exclusive).
- **C)** The `Math.max()` method can only be used with integers.
  - This option is incorrect. The `Math.max()` method can be used with various numeric types, including `int`, `long`, `float`, and `double`.
- **D)** The `Math.pow()` method returns the result of raising the first argument to the power of the second argument.
  - This option is correct. The `Math.pow()` method returns the result of raising the first argument to the power of the second argument. Both arguments are of type `double`.
- **E)** The `Math.abs()` method can only be used with positive numbers.
  - This option is incorrect. The `Math.abs()` method can be used with negative numbers to return their absolute value, and it works with various numeric types including `int`, `long`, `float`, and `double`.

Chapter FIVE

Controlling Program Flow

Chapter Content

- The `if` Statement
  - Pattern Matching in `if` Statements
  - Flow Scoping
- The `switch` Statement

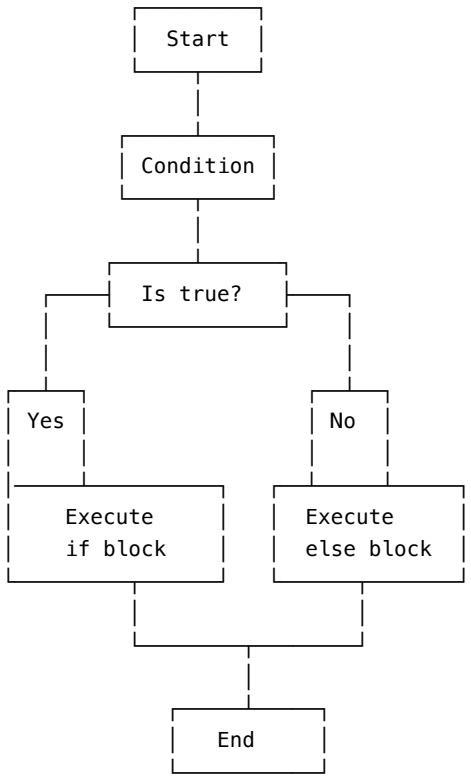
  - Types in `case` Statements
  - Values in `case` Statements
  - The `switch` Expression
  - Pattern Matching in `switch` Statements
- The `while` Loop
  - Nested Loops
  - The `break` and `continue` Statements
  - Adding Labels
- The `for` Loop
  - The Traditional `for` Loop
  - The `for-each` Loop
  - Nested `for` Loops
  - The `break` and `continue` Statements
  - Adding Labels
- Key Points
- Practice Questions

The `if` Statement

One of the most fundamental control flow statements in Java and many other programming languages is the `if` statement. It allows your program to make decisions and execute different code paths based on whether certain conditions are met.

In essence, the purpose of an `if` statement is to conditionally execute a block of code. If the specified condition evaluates to `true`, the code block will run. If not, that block is skipped over and the program continues with the next statement after the if block.

Here's the flowchart diagram for the `if` statement:



The basic syntax of an `if` statement looks like this:

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

The condition goes inside parentheses and must evaluate to a boolean value, either `true` or `false`. The code to conditionally execute goes between curly braces. If the code block contains only a single statement, you can omit the curly braces:

```
if (x > 10)
    System.out.println("x is greater than 10");
```

However, using curly braces is considered good practice even for single statements, as it makes your code clearer and less prone to errors if you later add more statements to the block.

You can chain multiple conditions together using the `else if` construct:

```
if (condition1) {
    // Code to execute if condition1 is true
} else if (condition2) {
    // Code to execute if condition1 is false and condition2 is true
} else {
    // Code to execute if both condition1 and condition2 are false
}
```

Here, each `else if` condition will only be checked if all the prior `if`/`else if` conditions evaluated to `false`. As soon as one condition is found to be `true`, its corresponding block executes and the rest of the `if`/`else if`/`else` chain is skipped. The final `else` block runs if none of the conditions were `true`.

There's no hard limit to how many `else if` statements you can have, but if you find yourself with very long `if`/`else if` chains, you may want to consider refactoring to a cleaner approach, such as a `switch` statement or polymorphism.

A common point of confusion is attempting to access variables declared inside an `if` block from the corresponding `else` block:

```
if (condition) {
    int x = 10;
} else {
    System.out.println(x); // Compile error - x is not in scope!
}
```

This fails because variables declared inside an `if` or `else` block are only in scope within that block. To use a variable in both the `if` and `else` sections, you must declare it outside (before) the `if` statement.

Pattern Matching in `if` Statements

Java has been expanding its pattern matching capabilities, making it easier to work with complex data structures. Let's explore how pattern matching works with `if` statements.

Type Patterns Type patterns allow you to test if an object is an instance of a particular type and, if so, create a variable of that type in one step:

```
if (obj instanceof String s) {
    System.out.println(s.toUpperCase());
}
```

Here, `obj` is tested to see if it's an instance of `String`. If so, it's cast to `String` and assigned to the pattern variable `s`, which can then be used in the `if` block.

There are a few rules when using type patterns in `if` statements: - The type of the pattern variable must be a subtype of the variable on the left side of `instanceof`. - The pattern variable is only usable when the compiler can definitively say its type, if there's ambiguity, it may not be considered initialized. - Pattern matching can use any valid expression, even method calls, not just simple variable checks.

For example:

```
if (getObject() instanceof String s) {
    System.out.println(s); // s in scope here
} else {
    System.out.println(s); //Compile error! s is definitely not a String
}

// ...

Object getObject() {
    return "hi";
}
```

Record Patterns Java 21 introduced record patterns, which allow you to destructure record instances directly in the if statement. This provides a more declarative and composable way to work with data. Here's an example:

```
record Book(String title, String author) {}

static void printDetails(Object obj) {
    if (obj instanceof Book(String title, String author)) {
        System.out.println("Title: " + title);
        System.out.println("Author: " + author);
    }
}
```

In this code, `Book(String title, String author)` is a record pattern. It not only checks if `obj` is an instance of `Book`, but also extracts the `title` and `author` components directly into pattern variables. This eliminates the need for separate accessor method calls.

Record patterns can also be nested, allowing you to destructure complex object graphs in a single step:

```
record Book(String title, String author) {}
record Library(String name, Book bestSeller) {}

Library myLibrary = new Library("City Library", new Book("Java Programming", "John Doe"));

if (myLibrary instanceof Library(var name, Book(var title, var author))) {
    System.out.println("Best seller at " + name + " is '" + title + "' by " + author);
}
```

In this example, we're using a nested record pattern to match against the `Library` record and its `Book` component simultaneously. If the pattern matches, we get direct access to the `name`, `title`, and `author` components without needing to use accessor methods.

Record patterns also work with generic records. The compiler will infer the type arguments when possible:

```
record Box<T>(T t) {}

public class GenericRecord {
    static void unbox(Box<Box<Integer>> box) {
        if (box instanceof Box(Box(var u))) {
```

```
            System.out.println("Unboxed Integer: " + u);
        }
    }

    public static void main(String[] args) {
        unbox(new Box<>(new Box<>(8))); // Prints Unboxed Integer: 8
    }
}
```

Here, the compiler infers that `u` is an `Integer`, based on the type of `box`.

It's important to note that record patterns don't match against `null`. If you need to handle potential `null` values, you should do so before the pattern matching:

```
if (obj != null && obj instanceof Book(String title, String author)) {
    System.out.println("Title: " + title);
    System.out.println("Author: " + author);
}
```

In summary, here are some key points about record patterns:

1. They consist of a record class type followed by a parenthesized list of patterns for each component.
2. They can be nested, allowing you to destructure complex record hierarchies in a single pattern.
3. The `var` keyword can be used to infer the type of a component.
4. `null` values do not match any record pattern.
5. For generic record classes, type arguments are inferred if not explicitly provided.

Flow Scoping

An important concept to understand when using pattern matching in `if` statements is flow scoping. This refers to how the compiler reasons about the scope and availability of pattern variables based on the flow of control through your code.

Consider this example:

```
if (obj instanceof String s) {
    System.out.println(s); // s is definitely a String here
} else {
    System.out.println(s); // Compiler error: s might not be initialized
}
System.out.println(s); // Compiler error: s is not in scope here
```

Inside the `if` block, s is definitely a `String`, the compiler knows this because the `instanceof` check must have succeeded for that block to execute. Therefore, it's safe to use `s` as a `String` within this scope.

However, in the corresponding `else` block, `s` is not considered initialized. The compiler doesn't assume the opposite of the `if` condition, it reasons that if the `else` block is executing, the `instanceof` check must have failed, and so `s` was never assigned a value. Attempting to use `s` here results in a compile error.

Outside the `if-else` statement entirely, `s` is not in scope at all. Pattern variables are only accessible within the `if` block where they're declared, and in subsequent `else if` or `else` blocks if the compiler can prove they were definitely assigned.

Flow scoping becomes more complex when you have multiple pattern variables in play:

```
if (obj instanceof String s || obj instanceof Integer i) {
    // s or i is in scope, but not both
} else {
    // neither s nor i are in scope
}
```

In this case, inside the `if` block, only one of `s` or `i` will be in scope, depending on which instanceof check succeeded. The compiler doesn't let you use a pattern variable unless it can definitively say it was assigned.

If you need to use a pattern variable in multiple scopes, you must assign it separately:

```
String s = null;
if (obj instanceof String temp) {
    s = temp;
}
// s is now in scope, but may be null if the if block didn't execute
```

This may feel like a limitation, but it is actually a powerful safety feature. By tightly controlling the scope of pattern variables, Java helps prevent common bugs and makes your code more robust.

It's worth noting that flow scoping only applies to the declared pattern variables themselves, not the original variables. In the example above, `obj` remains in scope throughout, because it was declared before the `if` statement.

The `switch` Statement

Sometimes, you need to check the value of a variable or expression and execute different code depending on what that value is. If there are only a couple of options, an `if-else` statement works fine:

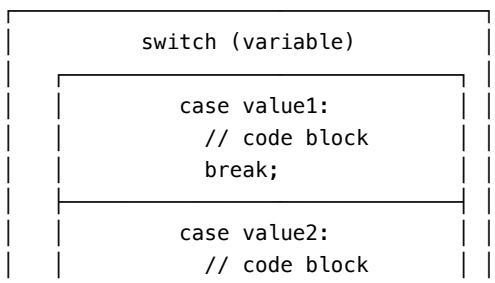
```
String animal = "cat";
if(animal.equals("dog")) {
    System.out.println("Woof!");
} else {
    System.out.println("Meow!");
}
```

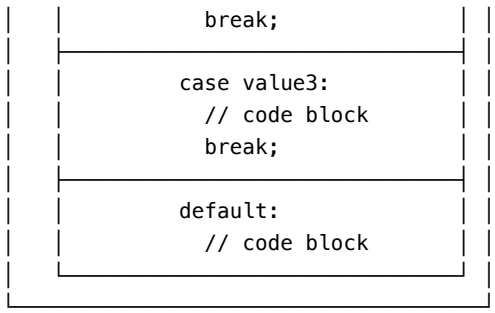
But what if there are many possible values to check? You could chain a bunch of `if-else` statements together:

```
String animal = "horse";
if(animal.equals("dog")) {
    System.out.println("Woof!");
} else if(animal.equals("cat")) {
    System.out.println("Meow!");
} else if(animal.equals("pig")) {
    System.out.println("Oink!");
} else if(animal.equals("horse")) {
    System.out.println("Neigh!");
} else {
    System.out.println("Unknown animal!");
}
```

However, this can get cumbersome and messy fast. That's where the `switch` statement comes in. It allows you to define separate code blocks for different values of a variable or expression.

Here's the diagram for the `switch` statement:





And this is its basic syntax:

```
switch(variable) {
    case value1:
        // code to run if variable == value1
        break;
    case value2:
        // code to run if variable == value2
        break;
    default:
        // code to run if no case matches
}
```

So the animal example could be rewritten more cleanly as:

```
String animal = "horse";
switch(animal) {
    case "dog":
        System.out.println("Woof!");
        break;
    case "cat":
        System.out.println("Meow!");
        break;
    case "pig":
        System.out.println("Oink!");
        break;
    case "horse":
        System.out.println("Neigh!");
        break;
    default:
        System.out.println("Unknown animal!");
}
```

Each `case` defines a value to compare the switch variable against. If there's a match, the code for that case executes. The `break` causes execution to jump to the end of the `switch` block. If no case matches, the `default` block runs.

It's important to include a `break` (or `return`) statement for each case, otherwise execution *falls through* to the next case, which is rarely what you want. The default case doesn't need an explicit `break` since it's the last one.

Types in `case` Statements

Not just any type can be used in a `switch`. Historically, switches could only work with these integral types and their wrapper classes: - `int`/`Integer` - `byte`/`Byte` - `short`/`Short` - `char`/`Character`

Then, in later versions of Java, `String`, records, and the constants of an `enum` were added as an option.

Also, you can use `var` in a `switch` statement as long as the type resolves to one of the other permitted types:

```
var animal = "horse";
switch(animal) {
    case "dog":
        System.out.println("Woof!");
        break;
    case "cat":
        System.out.println("Meow!");
        break;
    case "pig":
        System.out.println("Oink!");
        break;
    case "horse":
        System.out.println("Neigh!");
        break;
    default:
        System.out.println("Unknown animal!");
}
```

In this case, `animal` is inferred to be a `String` based on the value assigned to it. Since `String` is a valid type for a switch, using `var` here is perfectly fine.

However, if you try to do something like this:

```
var data = 3.14;
switch(data) {
    // ...
}
```

You will get a compilation error because `data` is inferred to be a `double`, which is not a permitted type for `switch` statements.

About enums, let's consider an interface `Season` and an enum `Weather` that implements this interface:

```
sealed interface Season permits Weather {}
enum Weather implements Season { SPRING, SUMMER, FALL, WINTER }
```

In older versions of Java, the switch statement required you to use only the simple names of the enum constants:

```
void oldEnumSwitch(Weather w) {
    switch (w) {
        case SPRING -> {
            System.out.println("It's spring!");
        }
        case SUMMER -> {
            System.out.println("It's summer!");
        }
        case FALL -> {
            System.out.println("It's fall!");
        }
        case WINTER -> {
            System.out.println("It's winter!");
        }
    }
}
```

This restriction worked fine for basic use cases but became cumbersome when dealing with more complex scenarios, such as combining enum types or using sealed interfaces.

In Java 21, you can now use fully qualified names of enum constants and mix them with other case labels, providing greater flexibility and allowing for more complex switch expressions:

```
void newEnumSwitch1(Season s) {
    switch (s) {
        case Weather.SPRING -> {  // Qualified name of enum constant
            System.out.println("It's spring!");
        }
        case Weather.SUMMER -> {
            System.out.println("It's summer!");
        }
        case Weather.FALL -> {
            System.out.println("It's fall!");
        }
        case Weather.WINTER -> {
            System.out.println("It's winter!");
        }
    }
}
```

Additionally, the requirement that the selector expression be of an enum type is relaxed, allowing you to use qualified names of enum constants even if the selector expression is not of the enum type, as long as it is assignment compatible, like in the above example.

However, an invalid use case would be when the enum constant is not fully qualified:

```
void invalidEnumSwitch(Season s) {
    switch (s) {
        case SPRING -> { // Error: SPRING must be qualified as Weather.SPRING
            System.out.println("It's spring!");
        }
        case Weather.SUMMER -> {
            System.out.println("It's summer!");
        }
        case Weather.FALL -> {
            System.out.println("It's fall!");
        }
        case Weather.WINTER -> {
            System.out.println("It's winter!");
        }
        default -> {
            System.out.println("Unknown season");
        }
    }
}
```

Values in `case` Statements

When defining the values for each `case`, there are some important rules to keep in mind. The value must be a compile-time constant, meaning it has to be known at the time the code is compiled, not determined at runtime.

So you can use literal values like `"dog"` or `3`, `final` variables (as long as they're initialized with a constant value), and `enum` constants. But you can't use a regular variable or a method call, even if the method always

returns the same value. For example:

```
final int NUMBER = 2;

int getSome() {
    return 1;
}

int x = 3;

switch(value) {
    case NUMBER: // OK, NUMBER is final and initialized with a constant
    case getSome(): // Error! Method calls aren't allowed
    case x: // Error! x is not final
    ...
}
```

Sometimes, you might want to run the same code for multiple `case` values. Rather than duplicating the code, you can simply list the values together for a single case:

```
int dayNumber;
switch(dayName) {
    case "Monday":
        dayNumber = 1;
        break;
    case "Tuesday":
        dayNumber = 2;
        break;
    case "Saturday", "Sunday": // Runs the same code for "Saturday" and "Sunday"
        dayNumber = 0;
        break;
    default:
        throw new IllegalArgumentException("Invalid day: " + dayName);
}
```

I mentioned this earlier, but it's worth reiterating, don't forget to `break` out of each case block (or use `return`), unless you specifically want execution to fall through to the next case. Forgetting a `break` is a common source of bugs in switch statements.

The `switch` Expression

Java 14 officially introduced a new form of `switch`, known as the `switch` expression. It has a few key differences from the traditional `switch` statement. First, here's the syntax:

```
variable = switch(anotherVariable) {
    case value1 -> expression1;
    case value2 -> { statements; yield expression2; }
    default -> expression3;
};
```

Instead of `case:` and `break`, the switch expression uses `->` to map each case to a value or block of code. If you need multiple statements for a case, use curly braces and the `yield` keyword to specify the value to return.

Note the semicolons. Each case needs one at the end, as does the entire `switch` expression.

The `switch` expression must always return a value, and each case must cover all possibilities (either explicitly or with a `default`). The data types of all the `case` results must also be consistent with each other.

Here's a more concrete example:

```
String animal = "horse";
String sound = switch(animal) {
    case "dog" -> "Woof!";
    case "cat" -> "Meow!";
    case "pig" -> "Oink!";
    case "horse" -> "Neigh!";
    case "human" -> {
        String greeting = "Hello!";
        yield greeting; // Use yield when there are multiple statements
    }
    default -> throw new IllegalArgumentException("Unknown animal: " + animal);
};
```

In this case, each animal is mapped directly to the sound it makes, except for `human` which has a block of code. The `default` case throws an exception since the `switch` expression must cover all possible input values.

Pattern Matching in `switch` Statements

Java 21 introduced a powerful new feature: pattern matching in `switch` statements and expressions. This allows you to test the structure of an object directly in the `switch`, making your code more expressive and less error-prone.

Let's start with a simple example:

```
Object obj = "Hello, World!";
String result = switch (obj) {
    case Integer i -> "It's an integer: " + i;
    case String s -> "It's a string: " + s;
    case Double d -> "It's a double: " + d;
    default -> "It's something else";
};
System.out.println(result); // Outputs: It's a string: Hello, World!
```

In this example, we're switching on an `Object`, and each case checks if the object is of a specific type. If it matches, we can use the variable declared in the pattern (like `s` for String) directly in the case body.

We can add guards to our case labels for even more precise matching:

```
Object obj = 42;
String category = switch (obj) {
    case Integer i when i < 0 -> "Negative integer";
    case Integer i when i > 0 -> "Positive integer";
    case Integer i -> "Zero";
    case String s when s.length() > 5 -> "Long string";
    case String s -> "Short string";
    default -> "Something else";
};
System.out.println(category); // Outputs: Positive integer
```

The `when` clause allows us to add additional conditions to our pattern matching.

However, when using pattern matching, the order of cases matters. More specific patterns should come before more general ones:

```
Object obj = "Hello";
String result = switch (obj) {
    case String s when s.length() > 5 -> "Long string";
    case String s -> "Short string";
    case CharSequence cs -> "Some other CharSequence";
```

```
    default -> "Not a CharSequence";
};
System.out.println(result); // Outputs: Long string
```

If we were to put the `case String s` before `case String s when s.length() > 5`, the guard would never be reached, and the compiler would warn us about an unreachable case.

Pattern matching in switch also introduces a more elegant way to handle `null` values:

```
String str = null;
String description = switch (str) {
    case null -> "It's null!";
    case String s -> "It's a string of length " + s.length();
};
System.out.println(description); // Outputs: It's null!
```

In traditional switches, a `null` value would throw a `NullPointerException`. With pattern matching, we can explicitly handle the `null` case.

However, you have to be careful about having only one match-all case label in a `switch` block. If, for example, you add a `default` case to the above example:

```
String str = null;
String description = switch (str) {
    case null -> "It's null!";
    case String s -> "It's a string of length " + s.length();
    default -> "default";
};
System.out.println(description); // Outputs: It's null!
```

You'll get a compilation error: `switch has both an unconditional pattern and a default label`.

Having more than one match-all case labels in a `switch` statement or expression generates a compile-time error. The match-all case labels are: - A case label with a pattern that unconditionally matches the selector expression - The `default` case label

However, the following compiles:

```
Object obj = null; // Notice the Object type
String description = switch (obj) {
    case String s -> "It's a string of length " + s.length();
    case null, default  -> "It's null or not a string!";
};
System.out.println(description); // Outputs: It's null or not a string!
```

If a selector expression evaluates to `null` and the switch block does not have `null` case label, like in the following case:

```
Object obj = null;
String description = switch (obj) { // Throws NullPointerException
    case String s -> "It's a string of length " + s.length();
    default  -> "It's null or not a string";
};
System.out.println(description);
```

Then a `NullPointerException` is thrown.

Another key benefits of pattern matching in `switch` is exhaustiveness checking. The compiler ensures that all possible cases are covered:

```java
sealed interface Shape permits Circle, Rectangle, Triangle {}
record Circle(double radius) implements Shape {}
record Rectangle(double width, double height) implements Shape {}
record Triangle(double base, double height) implements Shape {}

public class SwitchExhaustiveness {
    public static void main(String[] args) {
        Shape shape = new Circle(5);
        double area = switch (shape) {
            case Circle c -> Math.PI * c.radius() * c.radius();
            case Rectangle r -> r.width() * r.height();
            case Triangle t -> 0.5 * t.base() * t.height();
        };
        System.out.println("Area: " + area);
    }
}
```

In this example, because `Shape` is a sealed interface and we've covered all its permitted subclasses, the compiler knows we've exhaustively covered all possibilities. If the expression is a `sealed` type, only the classes declared in the `permits` clause of the `sealed` type need to be handled by the `switch`.

However, if you don't cover all the possibilities:

```java
double area = switch (shape) {
    case Circle c -> Math.PI * c.radius() * c.radius();
    case Rectangle r -> r.width() * r.height();
};
```

A compile-time error is generated: `switch expression does not cover all possible input values`.

The issue can be fixed simply by adding a `default` case:

```java
double area = switch (shape) {
    case Circle c -> Math.PI * c.radius() * c.radius();
    case Rectangle r -> r.width() * r.height();
    default -> 0;
};
```

Finally, in a `switch` statement, the compiler can also infer the type arguments for a generic record pattern. For example, taking into account the following record declaration:

```java
record Point<T, U>(T x, U y) { }
```

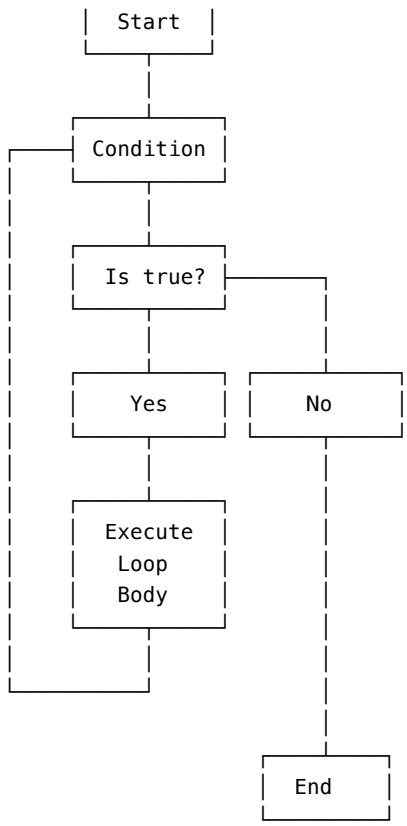
The compiler can infer `Point(var x, var y)` as `Point<Long, Long>(Long x, Long x)`:

```java
Point<Long, Long> p = new Point(1L, 2L);

switch (p) {
    case Point(var x, var y) ->
        System.out.println(x + ", " + y);
}
```

The `while` Loop

A `while` loop allows you to repeatedly execute a block of code as long as a specified `boolean` condition remains `true`.

Here's the flowchart diagram for the `while` statement:



There are actually two variants of the `while` loop in Java: 1. The standard `while` loop 2. The `do-while` loop

The standard `while` loop has the following structure:

```
while(condition) {
    // code block to be executed
}
```

The condition is a `boolean` expression that is evaluated before each iteration of the loop. If the condition is `true`, the code block is executed. This process repeats until the condition becomes `false`.

It's important to note that if the condition is false when the loop is first reached, the code block will not be executed at all. The loop will be skipped entirely.

Here's an example that prints the numbers 0 through 9:

```
int count = 0;
while(count < 10) {
    System.out.println(count);
    count++;
}
```

The loop will continue executing until count is no longer less than 10.

The `do-while` loop is similar to the standard `while` loop but with one key difference: the condition is evaluated after the code block has executed. This means the code block will always execute at least once, even if the condition is initially false.

Here's the syntax of a `do-while` loop:

```
do {
    // code block to be executed
```

```
} while(condition);
```

As you can see, the code block comes before the `while` keyword and condition. The condition is checked after each iteration, determining whether the loop should continue or terminate.

The following example is functionally equivalent to the previous `while` loop example:

```
int count = 0;
do {
    System.out.println(count);
    count++;
} while(count < 10);
```

Even though the structure is different, this `do-while` loop achieves the same result as the standard `while` loop, printing the numbers 0 through 9.

So why would you choose a `do-while` loop over a standard `while` loop? It really depends on the specific problem you're trying to solve. If you know you always want the code block to execute at least once regardless of the initial condition state, a `do-while` can be a good choice and can make your intention clearer. However, in many cases, a standard `while` loop is sufficient and more commonly used.

Nested Loops

It's possible to place one loop inside the body of another loop. This is known as loop nesting. Nested loops allow you to iterate over multiple dimensions, such as the rows and columns of a 2D array.

Here's an example that uses nested `while` loops to print out a multiplication table:

```
int i = 1;
while(i <= 10) {
    int j = 1;
    while(j <= 10) {
        System.out.print(i * j + "\t");
        j++;
    }
    System.out.println();
    i++;
}
```

The outer loop iterates from 1 to 10, representing the rows of the multiplication table. For each iteration of the outer loop, the inner loop also iterates from 1 to 10, representing the columns. The product of the current row and column values is printed, followed by a tab (`\t`) character for formatting. After each row is complete, a newline is printed to move to the next row.

While this example uses `while` loops, you can also nest `do-while` loops in a similar manner. The choice of loop type depends on the specific requirements of your use case.

The `break` and `continue` Statements

The `break` statement is used to immediately terminate a loop or switch statement. When encountered inside a loop, `break` causes program control to transfer to the next statement after the loop.

Here's an example of using `break` in a `while` loop:

```
int count = 0;
while(true) {
    System.out.println(count);
    count++;
    if(count >= 5) {
        break;
```

```
    }
}
```

This loop will continue infinitely because the condition is always true. However, the `break` statement inside the loop will cause it to terminate once count reaches 5.

On the other hand, the `continue` statement is used to skip the rest of the current loop iteration and immediately move on to the next iteration.

Here's an example that uses `continue`:

```
int i = 0;
while(i < 10) {
    if(i % 2 == 0) {
        i++;
        continue;
    }
    System.out.println(i);
    i++;
}
```

This loop iterates from 0 to 9. However, when `i` is even (divisible by 2), the `continue` statement is executed, causing the rest of the iteration to be skipped. As a result, only the odd numbers are printed.

However, it's important to note that using `break` or `continue` can sometimes lead to unreachable code, which will cause a compilation error.

Consider this example:

```
while(condition) {
    // code block
    break;
    // more code
}
```

The code after the `break` statement will never be executed because `break` always causes the loop to terminate. The Java compiler will detect this and raise an "unreachable code" compilation error.

The same applies to `continue`. Any code placed after a `continue` statement in the same loop iteration will be unreachable.

To avoid these errors, make sure that any code placed after a `break` or `continue` has a chance to execute under some condition.

Adding Labels

Finally, you can associate a label with a loop. Labels provide a way to break out of or continue a specific outer loop from within a nested loop. Here's the syntax for adding a label to a loop:

```
label:
while(condition) {
    // code block
}
```

The label is an identifier followed by a colon. It's placed just before the loop declaration.

Here's an example that demonstrates the use of labels:

```
int i = 0;
outerLoop:
while(i < 10) {
    int j = 0;
```

```
    while(j < 10) {
        if(j == 5) {
            break outerLoop;
        }
        System.out.println("i: " + i + ", j: " + j);
        j++;
    }
    i++;
}
```

In this case, the outer loop is labeled `outerLoop`. Inside the nested loop, there's a condition that checks if `j` is equal to 5. When this condition is met, the `break` statement is used with the `outerLoop` label, causing execution to jump out of both the inner and outer loops. Without the label, the break would only exit the inner loop.

Like `break`, `continue` can also be used with a label to skip to the next iteration of an outer loop.

Here's an example that demonstrates this:

```
int i = 0;
outerLoop:
while(i < 3) {
    int j = 0;
    while(j < 3) {
        if(i == 1 && j == 1) {
            i++;
            continue outerLoop;
        }
        System.out.println("i: " + i + ", j: " + j);
        j++;
    }
    i++;
}
```

In this example, the outer loop is labeled `outerLoop`. The outer loop iterates over the values of `i` from 0 to 2, and the inner loop iterates over the values of `j` from 0 to 2.

Inside the nested loops, there's a condition that checks if both `i` and `j` are equal to 1. When this condition is met, the `continue` statement is used with the `outerLoop` label. This causes the program control to immediately jump to the next iteration of the outer loop, skipping the rest of the inner loop.

As a result, the output of this code will be:

```
i: 0, j: 0
i: 0, j: 1
i: 0, j: 2
i: 1, j: 0
i: 2, j: 0
i: 2, j: 1
i: 2, j: 2
```

Notice that the output `i: 1, j: 1` is missing because when `i` and `j` are both 1, the `continue outerLoop` statement is executed, causing the program to skip to the next iteration of the outer loop, bypassing the print statement.

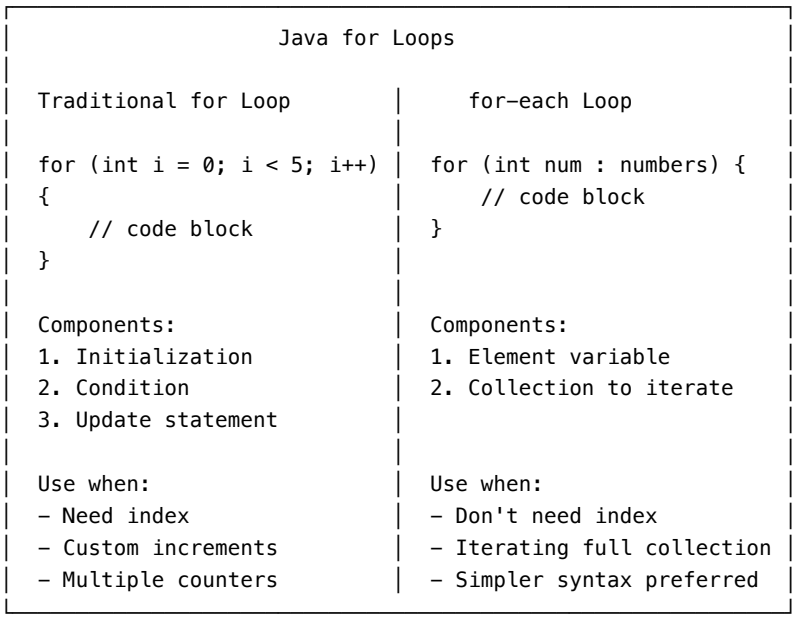
Using `continue` with a label is less common than using `break` with a label, but it can be useful in situations where you want to skip multiple levels of nested loops based on a certain condition.

The `for` Loop

Like `while` loops, `for` loops are used to repeatedly execute a block of code. However, `for` loops provide a more concise syntax for iterating over a range of values or elements in a collection.

In Java, there are two types of `for` loops: - The traditional `for` loop - The `for-each` loop (also known as the enhanced `for` loop)

Here's a diagram with the key points of the `for` loops:



Let's start by examining in more detail the traditional `for` loop.

The Traditional `for` Loop

The traditional `for` loop has the following structure:

```
for(initialization; booleanExpression; updateStatement) {
    // code block to be executed
}
```

The loop consists of three parts separated by semicolons: 1. **Initialization:** This is where you initialize the loop variable(s). It's executed only once at the beginning of the loop.

2. **Boolean Expression:** This is the condition that's checked before each iteration. If it evaluates to `true`, the loop continues. If it's `false`, the loop terminates.
3. **Update Statement:** This is where you specify how the loop variable(s) should be updated after each iteration. It's executed at the end of each iteration.

Here's a simple example that prints the numbers 0 to 4:

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

The loop initializes `i` to 0, checks if `i` is less than 5, and if so, executes the code block (printing the value of `i`). After each iteration, `i` is incremented by 1. The loop continues until `i` is no longer less than 5.

You can also use the `var` keyword in the initialization part:

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

If you omit the boolean expression, it defaults to `true`, creating an infinite loop:

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

This loop will continue indefinitely because there's no condition to make it `false`. To stop an infinite loop, you'd need to use a `break` statement or some other means of interrupting the loop.

You can initialize multiple variables and include multiple update statements in a `for` loop by separating them with commas:

```
for(int i = 0, j = 10; i < j; i++, j--) {
    System.out.println("i: " + i + ", j: " + j);
}
```

This loop initializes `i` to 0 and `j` to 10, checks if `i` is less than `j`, and if so, executes the code block. After each iteration, `i` is incremented, and `j` is decremented.

It's important to note that you cannot redeclare a variable in the initialization block of a `for` loop:

```
int i = 0;
for(int i = 0; i < 5; i++) { // Doesn't compile
    System.out.println(i);
}
```

This code will not compile because `i` is declared twice. If you need to use a variable that's already declared, simply omit the data type in the initialization block:

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

Also, all variables declared in the initialization block must be of the same data type or compatible types:

```
for(int i = 0, long j = 10; i < j; i++, j--) { // Doesn't compile
    System.out.println("i: " + i + ", j: " + j);
}
```

This code will not compile because `i` is of type `int`, and `j` is of type `long`. Here is the corrected example:

```
for(int i = 0, j = 10; i < j; i++, j--) {
    System.out.println("i: " + i + ", j: " + j);
}
```

Alternatively, if you need to use different data types, you should declare them before the loop:

```
int i = 0;
long j = 10;
for(; i < j; i++, j--) {
    System.out.println("i: " + i + ", j: " + j);
}
```

Regarding the scope of a variable declared in the initialization block, it is limited to the `for` loop. You cannot use it outside the loop:

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

```
}
System.out.println(i); // Doesn't compile
```

Once again, if you need to use the final value of the loop variable after the loop, you must declare it before the loop:

```
int i;
for(i = 0; i < 5; i++) {
    System.out.println(i);
}
System.out.println(i); // OK, prints 5
```

In many cases, you may need to compare the current loop variable with other elements in the loop. The traditional `for` loop makes this possible by allowing you to read elements forward or backward:

```
int[] arr = {1,2,3,4,5};
for(int i = 0; i < arr.length; i++) {
    // Read forward
    if(i < arr.length - 1) {
        System.out.println("Current: " + arr[i] + ", Next: " + arr[i+1]);
    }

    // Read backward
    if(i > 0) {
        System.out.println("Current: " + arr[i] + ", Previous: " + arr[i-1]);
    }
}
```

The forward reading if condition checks if the current element is not the last one and if so, it prints the current element and the next one.

The backward reading if condition checks if the current element is not the first one and if so, it prints the current element and the previous one.

The `for-each` Loop

The `for-each` loop, also known as the enhanced `for` loop, provides a simpler way to iterate over arrays and collections. It eliminates the need to explicitly declare and update loop variables.

The structure of a `for-each` loop is as follows:

```
for(dataType item : collection) {
    // code block to be executed
}
```

The loop consists of two parts with three elements: 1. **dataType:** The data type of the elements in the collection.

2. **item:** A variable that will hold the current element during each iteration.
3. **collection:** The array or collection to be iterated over.

Here's an example that prints the elements of an array using a `for-each` loop:

```
int[] numbers = {1, 2, 3, 4, 5};
for(int num : numbers) {
    System.out.println(num);
}
```

In each iteration, the loop assigns the next element of the numbers array to the `num` variable and executes the code block.

The `for-each` loop can be used with with arrays and with any object that implements the `Iterable` interface, which includes most collection classes, such as `ArrayList` and `HashSet`.

If you're wondering if everything that applies to `for` loops applies to `for-each` loops, the answer is not quite. While `for` loops and `for-each` loops share some similarities, there are a few key differences in how they behave and what they can do:

1. **Iteration**: A `for-each` loop automatically iterates over all elements in an array or collection, from the first to the last. You don't have control over the index or the order of iteration. A traditional `for` loop, on the other hand, gives you full control over the initialization, condition, and update statements, allowing you to iterate in any order or skip elements.
2. **Modification**: A `for-each` loop does not prevent you from modifying the elements of the array or collection within the loop, but it does not provide direct access to the index. You can modify the elements if the underlying collection supports modification. In contrast, a traditional `for` loop allows you to modify elements by accessing them via their index.
3. **Iterating over arrays and collections**: A `for-each` loop can be used to iterate over arrays and any object that implements the `Iterable` interface, which includes most collection classes. A traditional `for` loop can be used to iterate over arrays and collections, but you need to use an explicit index or iterator.
4. **Accessing index**: In a `for-each` loop, you don't have direct access to the index of the current element. If you need the index, you'll have to use a traditional `for` loop, which gives you access to the index through the loop variable.
5. **Performance**: For arrays, the performance difference between a `for-each` loop and a traditional for loop is generally negligible. For collections, the performance is similar as a `for-each loop` is syntactic sugar for using an iterator.

Here's an example that demonstrates a situation where a `for-each` loop cannot be used:

```
int[] numbers = {1, 2, 3, 4, 5};
for(int i = 0; i < numbers.length; i++) {
    if(numbers[i] % 2 == 0) {
        numbers[i] *= 2; // Double even numbers
    }
}
```

In this case, we need to modify the elements of the array based on a condition. We also need access to the index to perform the modification. This cannot be done with a `for-each` loop.

However, if we just needed to print the doubled even numbers, a `for-each` loop would be suitable:

```
int[] numbers = {1, 2, 3, 4, 5};
for(int num : numbers) {
    if(num % 2 == 0) {
        System.out.println(num * 2); // Print doubled even numbers
    }
}
```

In summary, `for-each` loops provide a concise syntax for iterating over all elements, while for loops offer more control and flexibility, allowing you to access indexes and iterate in custom ways.

Nested for Loops

Just like `while` loops, `for` loops can also be nested. This allows you to iterate over multidimensional arrays or perform complex iterations.

```
int[][] matrix = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };
for(int[] row : matrix) {
```

```
    for(int cell : row) {
        System.out.print(cell + " ");
    }
    System.out.println();
}
```

This code uses two nested `for-each` loops to iterate through a 2D array. The outer loop iterates over each row, and the inner loop iterates over each cell in the current row.

The `break` and `continue` Statements

The `break` statement can be used in `for` loops to prematurely terminate the loop.

```
int[] numbers = {1, 2, 3, 4, 5};
for(int num : numbers) {
    if(num == 3) {
        break;
    }
    System.out.println(num);
}
```

In this example, the loop will terminate when `num` is equal to 3. The output will be:

```
1
2
```

On the other hand, the `continue` statement can be used in `for` loops to skip the rest of the current iteration and move on to the next one.

```
int[] numbers = {1, 2, 3, 4, 5};
for(int num : numbers) {
    if(num % 2 == 0) {
        continue;
    }
    System.out.println(num);
}
```

This loop will print only the odd numbers in the array. When `num` is even, the `continue` statement is executed, and the rest of the iteration is skipped.

Also, using `break` or `continue` in `for` loops can sometimes lead to unreachable code, resulting in compilation errors.

```
for(int i = 0; i < 10; i++) {
    System.out.println(i);
    break;
    System.out.println("Unreachable"); // Unreachable code
}
```

In this example, the code after the `break` statement is unreachable because `break` always causes the loop to terminate. The Java compiler will detect this and throw a compilation error.

The same principle applies to `continue`. Any code after a `continue` statement in the same iteration will be unreachable.

To avoid these errors, ensure that any code placed after a `break` or `continue` statement has a chance to execute under some condition.

Adding Labels

Labels can be added to `for` loops in the same way as `while` loops. They are useful for breaking out of or continuing outer loops from within nested loops:

```
int[][] matrix = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };
outerLoop:
for(int[] row : matrix) {
    for(int cell : row) {
        if(cell == 5) {
            break outerLoop;
        }
        System.out.print(cell + " ");
    }
    System.out.println();
}
```

In this example, the outer loop is labeled as `outerLoop`. When the value of `cell` is 5, the `break` statement is used with the `outerLoop` label, causing the program to terminate both the inner and outer loops. This is what happens: 1. The outer loop starts with the first row `(1, 2, 3)` of `matrix`. - The inner loop prints "1", then "2", then "3". - The inner loop completes, and a newline is printed.

2. The outer loop moves to the second row `(4, 5, 6)`.
    - The inner loop prints "4".
    - The inner loop encounters 5, and `break outerLoop;` is executed.
    - Both the inner and outer loops are terminated.

The program ends at this point. The output is:

```
1 2 3
4
```

The third row `(7, 8, 9)` is never processed because the loops were terminated early.

Key Points

- The `if` statement allows your program to conditionally execute a block of code based on a boolean condition.
- The basic syntax of an `if` statement is: `if (condition) { code }`. The code block executes if the condition is true.
- You can chain multiple conditions using `else if`. The conditions are checked in order until one is true or the `else` block is reached.
- Variables declared inside an `if` or `else` block are only in scope within that block.
- `if` statements can use pattern matching with the `instanceof` operator, assigning the matched object to a pattern variable for use in the `if` block.
- Java 21 introduced record patterns, allowing destructuring of record instances directly in `if` statements.
- Record patterns can be nested, enabling the destructuring of complex object graphs in a single step.
- Pattern matching doesn't match against `null` values.
- The scope of pattern variables is tightly controlled by the compiler based on flow scoping rules to prevent bugs.
- The `switch` statement allows executing different code blocks based on the value of a variable or expression.

- Java 21 allows the use of fully qualified names of enum constants in `switch` statements.
- Enum constants can now be mixed with other case labels in the same `switch`.
- The requirement for the selector expression to be of an enum type is relaxed, allowing use of qualified names of enum constants even if the selector is not of the enum type (but is assignment compatible).
- Each `case` in a `switch` defines a value to compare against. If there's a match, that case's code block executes.
- Include a `break` statement at the end of each case block to prevent fall-through, unless fall-through is desired.
- `switch` statements can work with `String`, `enum` constants, and integral types like `int`, `char`, etc.
- Case values must be compile-time constants.
- Java 14 officially introduced `switch` expressions, which use `->` to map cases to result values and must cover all input possibilities.
- Java 21 introduced pattern matching in `switch` statements and expressions. This allows testing the structure of an object directly in the `switch`.
- You can use type patterns, record patterns, and add guards with `when` clauses for more precise matching.
- The order of cases matters; more specific patterns should come before more general ones.
- Pattern matching in `switch` introduces a way to handle `null` values explicitly.
- The compiler ensures exhaustiveness in `switch` statements and expressions: The `switch` block must have clauses that deal with all possible values of the selector expression.
- The `while` loop repeatedly executes a code block as long as its boolean condition remains true.
- If the condition is initially `false`, the code block will not execute at all.
- The `do-while` loop is similar but the condition is checked after each iteration, so the code block always executes at least once.
- You can nest one loop inside another to iterate over multiple dimensions.
- The `break` statement immediately terminates a loop, while `continue` skips to the next iteration.
- You can give a loop a label and then use `break` or `continue` with that label to break out of or continue an outer labeled loop.
- The `for` loop provides a concise syntax for iterating over a range of values.
- The traditional `for` loop has an initialization, condition, and update statement. The code block executes repeatedly until the condition is `false`.
- Variables declared in the initialization block are limited in scope to the `for` loop.
- The `for-each` loop (enhanced `for` loop) simplifies iterating over arrays/collections, eliminating the need for explicit indexing.
- `for-each` can't be used if you need the index or want to iterate in a custom order. Use a traditional `for` loop in those cases.
- You can use `break`/`continue` and labels with `for` loops just like with `while` loops.
- Avoid unreachable code after `break` or `continue` statements.

Practice Questions

1. What will be the output of the following program?

```
public class IfStatementTest {
    public static void main(String[] args) {
        int x = 10;
        if (x > 5) {
            if (x < 20) {
                System.out.println("x is between 5 and 20");
            }
        } else {
            System.out.println("x is 5 or less");
        }
    }
}
```

A) `x is between 5 and 20`
B) `x is 5 or less`
C) `x is greater than 20`
D) The program does not compile
E) The program compiles but does not produce any output

2. Given the following code:

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

public class RecordPattern {
    public static void main(String[] args) {
        Employee emp = new Employee(1001, new Person("Alice", 30));

        // Insert code here
    }
}
```

Which of the following options correctly uses record pattern matching in an `if` statement to extract and print the name and age of a `Person` record in Java 21?

A)

```
if (emp instanceof Employee) {
    var (id, Person(name, age)) = emp;
    System.out.println(name + " is " + age + " years old.");
}
```

B)

```
if (emp instanceof Employee(_, Person(var name, var age))) {
    System.out.println(name + " is " + age + " years old.");
}
```

C)

```
if (emp instanceof Employee e) {
    System.out.println(e.person().name() + " is " + e.person().age() + " years old.");
}
```

D)

```
if (emp instanceof Employee(var id, Person(var name, var age))) {
    System.out.println(name + " is " + age + " years old.");
}
```

E)

```
if (emp instanceof Employee(var id, var person)) {
    System.out.println(person.name() + " is " + person.age() + " years old.");
}
```

3. Which of the following code snippets compile without error?

```
public class FlowScopingTest {
    public static void main(String[] args) {
        int x = 10;
        if (x > 5) {
            int y = x * 2;
        }
        // Code snippet 1
        System.out.println(y);

        if (x < 20) {
            int z = x + 5;
        }
        // Code snippet 2
        z += 5;

        int a = 5;
        if (a > 0) {
            a = 15;
        }
        // Code snippet 3
        System.out.println(a);

        if (x > 0) {
            int b = x + 3;
            if (b > 15) {
                b -= 2;
            }
        }
        // Code snippet 4
        System.out.println(b);
    }
}
```

A) Code snippet 1
B) Code snippet 2
C) Code snippet 3
D) None of the above

4. What will be the output of the following program?

```
public class SwitchTest {
    public static void main(String[] args) {
        int dayOfWeek = 3;
        String dayType;
        switch (dayOfWeek) {
```

```
            case 1:
            case 7:
                dayType = "Weekend";
                break;
            case 2:
            case 3:
            case 4:
            case 5:
            case 6:
                dayType = "Weekday";
                break;
            default:
                dayType = "Invalid day";
        }
        System.out.println(dayType);
    }
}
```

A) `Weekend`
B) `Invalid day`
C) `Weekday`
D) The program does not compile
E) The program compiles but does not produce any output

5. What will be the output of the following program?

```
public class SwitchExpressionTest {
    public static void main(String[] args) {
        int score = 85;
        String grade = switch (score) {
            case 90, 100 -> "A";
            case 80, 89 -> "B";
            case 70, 79 -> "C";
            case 60, 69 -> "D";
            default -> "F";
        };
        System.out.println(grade);
    }
}
```

A) `A`
B) `F`
C) The program does not compile
D) `B`
E) The program compiles but does not produce any output

6. Given the following code:

```
public class SwitchEnums {
    sealed interface Vehicle permits CarType {}
    enum CarType implements Vehicle { SEDAN, SUV, HATCHBACK, CONVERTIBLE }

    void processVehicle(Vehicle v) {
        switch(v) {
            // Insert case statements here
        }
    }
```

```
}
```

Which of the following `case` statements are valid in Java 21 when inserted in the `switch` expression?

A)

```
case CarType.SEDAN, CarType.HATCHBACK -> System.out.println("Compact vehicle");
case CarType.SUV -> System.out.println("Large vehicle");
case CarType.CONVERTIBLE -> System.out.println("Open-top vehicle");
```

B)

```
case SEDAN, HATCHBACK -> System.out.println("Compact vehicle");
case SUV -> System.out.println("Large vehicle");
case CONVERTIBLE -> System.out.println("Open-top vehicle");
```

C)

```
case CarType.SEDAN || CarType.HATCHBACK -> System.out.println("Compact vehicle");
case CarType.SUV -> System.out.println("Large vehicle");
case CarType.CONVERTIBLE -> System.out.println("Open-top vehicle");
```

D)

```
case Vehicle.SEDAN, Vehicle.HATCHBACK -> System.out.println("Compact vehicle");
case Vehicle.SUV -> System.out.println("Large vehicle");
case Vehicle.CONVERTIBLE -> System.out.println("Open-top vehicle");
```

7. Given the following code:

```
sealed interface Shape permits Circle, Square, Triangle {}
record Circle(double radius) implements Shape {}
record Square(double side) implements Shape {}
record Triangle(double base, double height) implements Shape {}

Shape shape = new Circle(5);
double area = switch (shape) {
    // Insert case statements here
};
```

Which of the following `case` statements correctly implements pattern matching for the `Shape` hierarchy when inserted in the `switch` expression?

A)

```
case Circle c -> Math.PI * c.radius() * c.radius();
case Square s -> s.side() * s.side();
case null -> 0;
```

B)

```
default -> 0;
case Circle c -> Math.PI * c.radius() * c.radius();
case Square s -> s.side() * s.side();
case Triangle t -> 0.5 * t.base() * t.height();
```

C)

```
case Shape s when s instanceof Circle ->
        Math.PI * ((Circle)s).radius() * ((Circle)s).radius();
case Shape s when s instanceof Square ->
        ((Square)s).side() * ((Square)s).side();
```

```
case Shape s when s instanceof Triangle ->
        0.5 * ((Triangle)s).base() * ((Triangle)s).height();
```

D)

```
case Circle c -> Math.PI * c.radius() * c.radius();
case Square s -> s.side() * s.side();
case Triangle t -> 0.5 * t.base() * t.height();
```

8. What will be the output of the following program?

```
public class LabeledBreakTest {
    public static void main(String[] args) {
        int count = 0;
        outerLoop:
        while (count < 5) {
            while (true) {
                count++;
                if (count == 3) {
                    break outerLoop;
                }
            }
        }
        System.out.println(count);
    }
}
```

A) 2
B) 3
C) 4
D) 5
E) The program does not compile

9. What will be the output of the following program?

```
public class ForLoopTest {
    public static void main(String[] args) {
        int sum = 0;
        for (int i = 1; i <= 5; i++) {
            sum += i;
        }
        System.out.println(sum);
    }
}
```

A) 5
B) 10
C) 15
D) 20
E) The program does not compile

10. What will be the output of the following program?

```
public class EnhancedForLoopTest {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3, 4, 5};
        int sum = 0;
        for (int num : numbers) {
            if (num % 2 == 0) {
```

```
                continue;
            }
            sum += num;
        }
        System.out.println(sum);
    }
}
```

A) `9`
B) `10`
C) `12`
D) `15`
E) The program does not compile

Chapter FIVE

Controlling Program Flow

Answers

1. The correct answer is A.

Explanation:

- **A)** `x is between 5 and 20`
  - This option is correct. The value of `x` is 10, which satisfies both conditions in the nested `if` statements (`x > 5` and `x < 20`). Therefore, the program prints `"x is between 5 and 20"`.
- **B)** `x is 5 or less`
  - This option is incorrect. The value of `x` is 10, which does not satisfy the condition `x <= 5` in the `else` block. Therefore, this message will not be printed.
- **C)** `x is greater than 20`
  - This option is incorrect. The value of `x` is 10, which does not satisfy the condition `x > 20`. Therefore, this message will not be printed.
- **D)** The program does not compile
  - This option is incorrect. The program compiles successfully without any errors.
- **E)** The program compiles but does not produce any output
  - This option is incorrect. The program produces output because the value of `x` satisfies the conditions within the nested `if` statements, leading to the output `"x is between 5 and 20"`.

2. The correct answer is D.

Explanation:

- **A)**

```
if (emp instanceof Employee) {
    var (id, Person(name, age)) = emp;
    System.out.println(name + " is " + age + " years old.");
}
```

- This option is incorrect. While it attempts to use destructuring, this syntax is not valid in Java. Java doesn't support destructuring assignment in this way.

- **B)**

```
if (emp instanceof Employee(_, Person(var name, var age))) {
    System.out.println(name + " is " + age + " years old.");
}
```

- This option is incorrect. It uses the underscore (_) to ignore the `id` field, which is not a valid technique in Java 21.

- **C)**

```
if (emp instanceof Employee e) {
    System.out.println(e.person().name() + " is " + e.person().age() + " years old.");
}
```

- This option is incorrect. It uses traditional `instanceof` without pattern matching, relying on accessor methods to extract the data.

- **D)**

```
if (emp instanceof Employee(var id, Person(var name, var age))) {
    System.out.println(name + " is " + age + " years old.");
}
```

- This option is correct. It uses nested record pattern matching to extract both the `Employee` and `Person` data in a single step. It uses `var` for type inference and correctly names the variables `name` and `age` as required.

- **E)**

```
if (emp instanceof Employee(var id, var person)) {
    System.out.println(person.name() + " is " + person.age() + " years old.");
}
```

- This option is incorrect. While it uses pattern matching for the `Employee` record, it doesn't nest the pattern matching for the `Person` record, so it still requires calling accessor methods on `person`.

3. The correct answer is C.

Explanation:

- **A)** Code snippet 1
  - This option is incorrect. The variable `y` is declared inside the first `if` statement and is not accessible outside its block. Therefore, trying to print `y` outside its scope results in a compilation error.
- **B)** Code snippet 2
  - This option is incorrect. The variable `z` is declared inside the second `if` statement and is not accessible outside its block. Therefore, attempting to use `z` outside its scope results in a compilation error.
- **C)** Code snippet 3
  - This option is correct. The variable `a` is declared outside the `if` statement, so it is accessible both inside and outside the `if` block. Reassigning a inside the `if` block is allowed.
- **D)** None of the above
  - This option is incorrect. While it's true that code snippets 1, 2 and 4 will not compile, code snippet 3 does compile without any errors. Therefore, the answer cannot be "none of the above".

4. The correct answer is C.

Explanation:

- **A)** `Weekend`
  - This option is incorrect. The value of `dayOfWeek` is 3, which does not match cases 1 or 7, so it does not print `"Weekend"`.
- **B)** `Invalid day`
  - This option is incorrect. The default case is not executed because the value of `dayOfWeek` matches one of the specific cases (2, 3, 4, 5, or 6).
- **C)** `Weekday`

- This option is correct. The value of `dayOfWeek` is 3, which matches case 3. Therefore, the variable `dayType` is set to `"Weekday"`, and this value is printed.

- **D)** The program does not compile
  - This option is incorrect. The program compiles without errors.
- **E)** The program compiles but does not produce any output
  - This option is incorrect. The program compiles and produces output, which is `"Weekday"` based on the given `dayOfWeek` value.

5. The correct answer is B.

Explanation:

- **A)** `A`
  - This option is incorrect. The value of `score` is 85, which does not match the cases for 90 or 100. Therefore, it does not print `"A"`.
- **B)** `F`
  - This option is correct. The default case is executed because the value of `score` doesn't match any of the other `case` statements.
- **C)** The program does not compile
  - This option is incorrect. The program uses a `switch` expression correctly, compiling without errors.
- **D)** `B`
  - This option is incorrect. The value of `score` is 85, which doesn't match the case for 80 or 89.
- **E)** The program compiles but does not produce any output
  - This option is incorrect. The program compiles and produces output, which is `"F"` based on the given `score` value.

6. The correct answer is A.

Explanation:

- **A)**

```
case CarType.SEDAN, CarType.HATCHBACK -> System.out.println("Compact vehicle");
case CarType.SUV -> System.out.println("Large vehicle");
case CarType.CONVERTIBLE -> System.out.println("Open-top vehicle");
```

- This option is correct. In Java 21, you can use fully qualified names of enum constants in switch statements, even when the selector expression is of a type that's assignment-compatible with the enum type (in this case, `Vehicle` is assignment-compatible with `CarType`).

- **B)**

```
case SEDAN, HATCHBACK -> System.out.println("Compact vehicle");
case SUV -> System.out.println("Large vehicle");
case CONVERTIBLE -> System.out.println("Open-top vehicle");
```

- This option is incorrect. When using an interface type (`Vehicle`) as the selector expression, you must use fully qualified names for the enum constants. Using unqualified names (`SEDAN`, `HATCHBACK`, etc.) will result in a compilation error.

- **C)**

```
case CarType.SEDAN || CarType.HATCHBACK -> System.out.println("Compact vehicle");
case CarType.SUV -> System.out.println("Large vehicle");
case CarType.CONVERTIBLE -> System.out.println("Open-top vehicle");
```

- This option is incorrect. It attempts to use the logical OR operator (`||`) in the case label, which is not valid syntax for switch statements. Multiple case labels should be separated by commas, not logical operators.

- **D)**

```
case Vehicle.SEDAN, Vehicle.HATCHBACK -> System.out.println("Compact vehicle");
case Vehicle.SUV -> System.out.println("Large vehicle");
case Vehicle.CONVERTIBLE -> System.out.println("Open-top vehicle");
```

- This option is incorrect. Although it uses fully qualified names, it incorrectly prefixes the enum constants with `Vehicle` instead of `CarType`. The enum constants belong to the `CarType` enum, not the `Vehicle` interface, so this will result in a compilation error.

7. The correct answer is D.

Explanation:

- **A)**

```
case Circle c -> Math.PI * c.radius() * c.radius();
case Square s -> s.side() * s.side();
case null -> 0;
```

- This option is incorrect. It doesn't compile because the `switch` expression is not exhaustive, it does not cover all possible `Shape` values.

- **B)**

```
default -> 0;
case Circle c -> Math.PI * c.radius() * c.radius();
case Square s -> s.side() * s.side();
case Triangle t -> 0.5 * t.base() * t.height();
```

- This option is incorrect. It doesn't compile because the (unnecessary) `default` case comes before the rest of the `case` statements.

- **C)**

```
case Shape s when s instanceof Circle ->
       Math.PI * ((Circle)s).radius() * ((Circle)s).radius();
case Shape s when s instanceof Square ->
       ((Square)s).side() * ((Square)s).side();
case Shape s when s instanceof Triangle ->
       0.5 * ((Triangle)s).base() * ((Triangle)s).height();
```

- This option is incorrect. It doesn't compile because it's not exhaustive. Since it uses verbose `instanceof` checks instead of leveraging pattern matching, it's missing a `default` branch.

- **D)**

```
case Circle c -> Math.PI * c.radius() * c.radius();
case Square s -> s.side() * s.side();
case Triangle t -> 0.5 * t.base() * t.height();
```

- This option is correct. It covers all possible subtypes of the sealed `Shape` interface without an unnecessary `default` case.

8. The correct answer is B.

Explanation:

- **A)** `2`
  - This option is incorrect. The value of `count` is incremented until it reaches 3. The labeled `break` statement breaks out of the outer loop when `count` equals 3.
- **B)** `3`
  - This option is correct. The value of `count` is incremented inside the inner `while` loop. When `count` reaches 3, the labeled `break` statement (`break outerLoop`) is executed, causing the control to exit the outer loop. Therefore, `count` is 3 when printed.

- **C)** `4`
  - This option is incorrect. The loop does not continue incrementing `count` to 4 because the labeled `break` statement exits the loop when `count` is 3.
- **D)** `5`
  - This option is incorrect. The loop does not continue incrementing `count` to 5 because the labeled `break` statement exits the loop when `count` is 3.
- **E)** The program does not compile
  - This option is incorrect. The program compiles successfully and runs without errors.

9. The correct answer is C.

Explanation:

- **A)** `5`
  - This option is incorrect. The value 5 is just the upper limit of the loop and not the sum of the integers from 1 to 5.
- **B)** `10`
  - This option is incorrect. The value 10 is less than the sum of the integers from 1 to 5.
- **C)** `15`
  - This option is correct. The loop iterates from 1 to 5, adding each value of `i` to `sum`. The calculations are as follows: $1 + 2 + 3 + 4 + 5 = 15$.
- **D)** `20`
  - This option is incorrect. The value 20 is more than the sum of the integers from 1 to 5.
- **E)** The program does not compile
  - This option is incorrect. The program compiles successfully and runs without errors.

10. The correct answer is A.

Explanation:

- **A)** `9`
  - This option is correct. The `continue` statement skips the current iteration when the number is even (`num % 2 == 0`). The odd numbers in the array are 1, 3, and 5. Their sum is $1 + 3 + 5 = 9$.
- **B)** `10`
  - This option is incorrect. The sum of the odd numbers (1, 3, and 5) is 9, not 10.
- **C)** `12`
  - This option is incorrect. The sum of the odd numbers (1, 3, and 5) is 9, not 12.
- **D)** `15`
  - This option is incorrect. The sum of all the numbers in the array ($1 + 2 + 3 + 4 + 5$) is 15, but the `continue` statement causes the loop to skip adding the even numbers.
- **E)** The program does not compile
  - This option is incorrect. The program compiles successfully and runs without errors.

Chapter SIX

Arrays, Generics, and Collections

Chapter Content

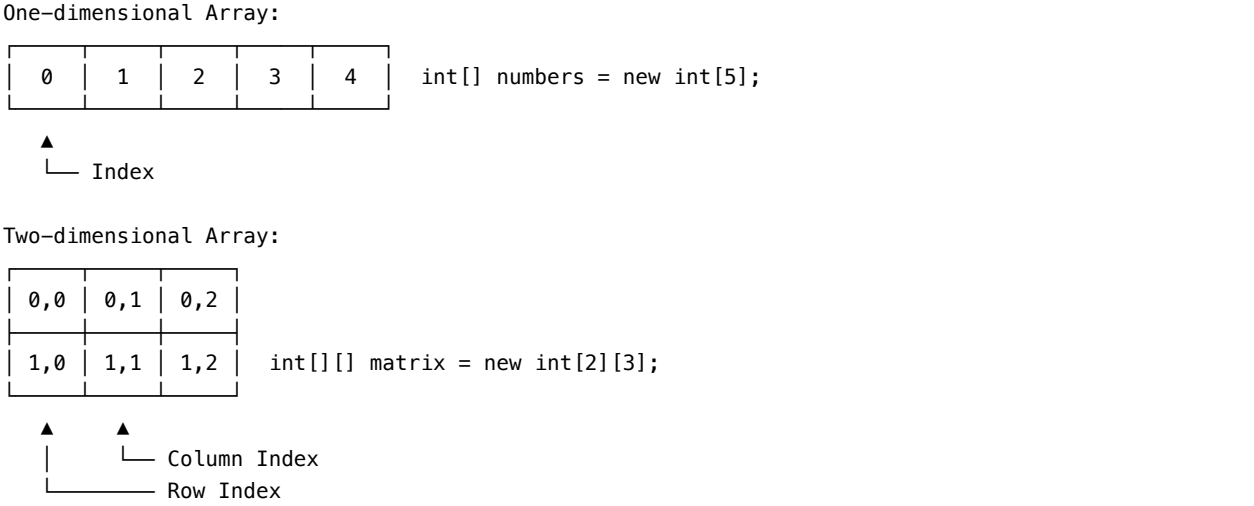
- Arrays
  - Creating and Initializing Arrays
  - Anonymous Arrays
  - Using an Array
  - Multidimensional Arrays
  - The `java.util.Arrays` Class
- Generics
  - Understanding Type Erasure

  - Creating Generic Classes
  - Naming Conventions for Generics
  - Writing Generic Methods and Constructors
  - Returning Generic Types
  - Overloading a Generic Method
  - Implementing Generic Interfaces
  - Creating Generic Records
  - Bounding Generic Types
- The Collections Framework
- The `List` Interface
  - Creating a `List`
  - Working with `List` Methods
- The `Set` Interface
  - Creating a `Set`
  - Working with `Set` Methods
- The `Deque` Interface
  - Creating a `Deque`
  - Working with `Deque` Methods
- The `Map` Interface
  - Creating a `Map`
  - Working with `Map` Methods
  - Overriding `hashCode()`
- Sorting Data
  - The `Comparable` Interface
  - The `Comparator` Interface
  - Comparing `Comparable` and `Comparator`
  - `Collections.sort` and `Collections.binarySearch`
- Summary of Collection Types
- Key Points
- Practice Questions

Arrays

An array is an object that holds a fixed number of values of a single type in contiguous memory locations. These values, or elements, can be of a primitive type or a reference type.

Here's a diagram to help you visualize one and two dimension arrays:



Let's start by reviewing how you create and initialize an array.

Creating and Initializing Arrays

To create an array, you have to declare a variable of the desired array type and then use the `new` keyword to create the array object and assign it to the variable:

```
// Creates an array of integers
int[] myArray;
myArray = new int[5];
```

You can also combine the declaration and the creation of the array in one statement:

```
int[] myArray = new int[5];
```

The number inside the brackets specifies the number of elements the array will hold, in other words, the size of the array. This size must be decided when the array is created and cannot be changed later.

This is an important limitation to keep in mind, you cannot resize an array after it has been created. If you need a data structure that can dynamically grow or shrink, you should consider using one of the collection classes like `ArrayList` instead.

When an array is created, its elements are automatically initialized with default values: - `0` for numeric types - `false` for boolean - `null` for reference types.

However, you can also explicitly initialize an array during creation:

```
int[] myArray = new int[] {10, 20, 30, 40, 50};
```

This creates an array of 5 integers and initializes them with the specified values. The size of the array is determined by the number of values provided.

If you don't need to specify the values at the time of declaration, you can leave some or all elements uninitialized:

```
int[] myArray = new int[5];
myArray[0] = 10;
myArray[1] = 20;
```

This creates an array of 5 integers, initializes the first two, and leaves the rest with their default value of `0`.

It's important to note that all elements of an array must be of the same type. You cannot mix different data types in a single array.

Anonymous Arrays

An anonymous array is an array that is declared and initialized in a single statement without assigning it to a variable:

```
new int[] {10, 20, 30, 40, 50}
```

Anonymous arrays are often used when passing an array as an argument to a method:

```
myMethod(new int[] {10, 20, 30, 40, 50});
```

They provide a convenient way to create and pass an array inline, without the need for a separate variable declaration.

However, anonymous arrays are not limited to method arguments. They can be used anywhere an array is expected, such as in an assignment:

```
int[] myArray = new int[] {10, 20, 30, 40, 50};
```

In this case, the anonymous array is created and immediately assigned to the `myArray` variable.

Using an Array

To access an element of an array, you use the array name followed by the index of the element in square brackets:

```
int[] myArray = new int[] {10, 20, 30, 40, 50};
System.out.println(myArray[0]); // Outputs 10
System.out.println(myArray[2]); // Outputs 30
```

Array indices start at 0, so the first element is at index 0, the second at index 1, and so on.

You can also use a variable for the index:

```
int index = 2;
System.out.println(myArray[index]); // Outputs 30
```

Trying to access an element outside the bounds of the array will result in an `ArrayIndexOutOfBoundsException`.

To find out the number of elements in an array, you can use the length attribute:

```
System.out.println(myArray.length); // Outputs 5
```

Note that this is an attribute, not a method, so you don't use parentheses.

Trying to change the size of an array after it has been created, either by assigning a new array to the variable or by using the length attribute, will result in a compile-time error.

While you can't resize an array, you can copy the contents of one array to another:

```
int[] sourceArray = new int[] {10, 20, 30, 40, 50};
int[] destArray = new int[5];
System.arraycopy(sourceArray, 0, destArray, 0, 5);
```

This copies the elements from `sourceArray` to `destArray`. The arguments specify the source array, the starting position in the source array, the destination array, the starting position in the destination array, and the number of elements to copy.

However, this is not the same as assigning one array to another:

```
int[] sourceArray = new int[] {10, 20, 30, 40, 50};
int[] destArray = sourceArray;
```

This does not create a copy of the array. Instead, it makes `destArray` reference the same array object as `sourceArray`. Changes made through either variable will be reflected in the other, as they both point to the same array in memory.

Multidimensional Arrays

Java also supports multidimensional arrays, which can be thought of as *arrays of arrays*.

The most common type of multidimensional array is the two-dimensional array, often used to represent matrices or tables of data. But Java places no limit on the number of dimensions an array can have.

To declare a multidimensional array, you specify each additional dimension with another set of square brackets. For example, here's how you would declare a two-dimensional array of integers:

```
int[][] matrix;
```

This declares a variable `matrix` that is an array of integer arrays.

You can then create the array with the `new` keyword:

```
matrix = new int[3][4];
```

This creates a two-dimensional array with 3 rows and 4 columns. Essentially, it's an array that contains 3 arrays, each of which contains 4 integers.

Just as with one-dimensional arrays, you can combine the declaration and creation:

```
int[][] matrix = new int[3][4];
```

You can also initialize the array upon creation:

```
int[][] matrix = {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12}
};
```

This creates the same 3x4 array as before, but also initializes it with the specified values.

Accessing elements in a multidimensional array is similar to a one-dimensional array, but now you need to specify an index for each dimension:

```
int[][] matrix = new int[3][4];
matrix[0][0] = 1;
matrix[1][2] = 7;
System.out.println(matrix[1][2]); // Outputs 7
```

Here, `matrix[0][0]` refers to the element in the first row and first column, `matrix[1][2]` refers to the element in the second row and third column, and so on.

You can also use nested loops to iterate over a multidimensional array:

```
int[][] matrix = {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12}
};

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();
}
```

This will be the output:

```
1 2 3 4
5 6 7 8
9 10 11 12
```

The outer loop iterates over the rows, and the inner loop iterates over the columns in each row.

Note that in a multidimensional array, the length attribute gives the number of arrays in the first dimension. To get the length of the arrays in the second dimension, you need to specify an index for the first dimension, like `matrix[i].length`.

Also note that while all the arrays in the second dimension have the same length in this example, this is not a requirement. You can have a *ragged* array where each array in the second dimension has a different length:

```
int[][] ragged = {
    {1, 2, 3, 4},
    {5, 6},
```

```
    {7, 8, 9}
};
```

This flexibility can be useful in certain situations, but it's more common to work with *rectangular* arrays where all the second-dimension arrays have the same length.

The `java.util.Arrays` Class

The `java.util.Arrays` class contains various static methods for manipulating arrays. It provides methods for sorting, searching, comparing, and filling array elements. Let's look at some of the most commonly used methods.

Sorting The `sort()` method sorts the elements of an array into ascending order. It has several overloads for different types of arrays:

```
int[] numbers = {4, 2, 7, 1, 3};
Arrays.sort(numbers);
System.out.println(Arrays.toString(numbers)); // [1, 2, 3, 4, 7]
```

This sorts the `numbers` array in place, modifying the original array.

You can also sort a portion of an array by specifying the start (inclusive) and end (exclusive) indices:

```
int[] numbers = {4, 2, 7, 1, 3};
Arrays.sort(numbers, 1, 4);
System.out.println(Arrays.toString(numbers)); // [4, 1, 2, 7, 3]
```

This sorts only the elements from index 1 to 3, leaving the elements at indices 0 and 4 untouched.

For arrays of objects, the objects must implement the `Comparable` interface for `sort()` to work. Alternatively, you can provide a `Comparator` object to define the sorting order:

```
String[] strings = {"banana", "apple", "cherry"};
Arrays.sort(strings, Comparator.comparingInt(String::length));
System.out.println(Arrays.toString(strings)); // [apple, banana, cherry]
```

This sorts the `strings` array by the length of each string, using a `Comparator` created by the `comparingInt()` method.

Searching The `binarySearch()` method searches for a specific element in a sorted array using the binary search algorithm. If the element is found, it returns its index. If not, it returns a negative value.

```
int[] numbers = {1, 2, 3, 4, 7};
System.out.println(Arrays.binarySearch(numbers, 3)); // 2
System.out.println(Arrays.binarySearch(numbers, 5)); // -5
```

In the first search, the element 3 is found at index 2. In the second search, the element 5 is not found, so the method returns -5. The negative value is calculated as `-(insertion point) - 1`, where the insertion point is the index at which the element would be inserted to maintain the sorted order.

Note that for `binarySearch()` to work correctly, the array must be sorted. If the array is not sorted, the results are undefined.

Using `compare()` The `compare()` method compares two arrays lexicographically (by dictionary order). It returns a negative value if the first array is *less than* the second, a positive value if the first array is *greater than* the second, and zero if they are equal.

```
int[] arr1 = {1, 2, 3};
int[] arr2 = {1, 2, 3};
int[] arr3 = {1, 2, 4};
```

```
System.out.println(Arrays.compare(arr1, arr2)); // 0
System.out.println(Arrays.compare(arr1, arr3)); // -1
System.out.println(Arrays.compare(arr3, arr1)); // 1
```

In comparing `arr1` and `arr2`, the method returns 0 because the arrays are equal. In comparing `arr1` and `arr3`, it returns -1 because `arr1` is lexicographically less than `arr3` (because $3 < 4$). Similarly, in comparing `arr3` and `arr1`, it returns 1.

Using `fill()`

The `fill()` method in the `Arrays` class is used to fill an array or a portion of it with a specific value. It's a convenient way to set all elements to the same value.

The `fill()` method has several overloads: - `fill(array, value)`: Fills the entire array with the specified value. - `fill(array, fromIndex, toIndex, value)`: Fills a portion of the array, from the `fromIndex` (inclusive) to the `toIndex` (exclusive), with the specified value.

Here's an example of using `fill()` to fill an entire array:

```
int[] numbers = new int[5];
Arrays.fill(numbers, 10);
System.out.println(Arrays.toString(numbers)); // [10, 10, 10, 10, 10]
```

This creates an array of 5 integers and fills it entirely with the value 10.

You can also fill just a portion of an array:

```
int[] numbers = {1, 2, 3, 4, 5};
Arrays.fill(numbers, 1, 4, 10);
System.out.println(Arrays.toString(numbers)); // [1, 10, 10, 10, 5]
```

This fills the elements from index 1 to 3 (remember, the `toIndex` is exclusive) with the value 10, leaving the elements at indices 0 and 4 unchanged.

The `fill()` method has overloads for all primitive types and for object references. When used with object references, each element will point to the same object:

```
String[] strings = new String[3];
Arrays.fill(strings, "Hello");
System.out.println(Arrays.toString(strings)); // [Hello, Hello, Hello]
```

This fills the `strings` array with references to the same `"Hello"` string.

It's important to understand that the `Arrays.fill()` method in Java does not create new objects for each element. Instead, it sets each element to reference the same object. If you modify the object through one of these references, all elements in the array will reflect that change. However, this behavior also depends on whether the objects are mutable or immutable.

Here's an example with immutable objects (strings):

```
String[] strings = new String[3];
Arrays.fill(strings, new String("Hello"));
strings[0] = "Hi";
System.out.println(Arrays.toString(strings)); // [Hi, Hello, Hello]
```

`Arrays.fill(strings, new String("Hello"));` sets each element in the array to reference a new `String` object with the value `"Hello"`. Thus, `strings[0]`, `strings[1]`, and `strings[2]` all reference the same `String` object initially. When you update `strings[0] = "Hi";`, you change the reference at `strings[0]` to point to a new `String` object with the value `"Hi"`. Since `String` objects are immutable in Java, this does not affect `strings[1]` and `strings[2]`. The output will be `[Hi, Hello, Hello]`, as modifying one element does not affect the others.

However, when using `Arrays.fill()` with mutable objects, each element will reference the same object. If you modify one instance, all elements in the array will reflect that change:

```
class Point {
    int x, y;

    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public String toString() {
        return "(" + x + ", " + y + ")";
    }
}

public class Main {
    public static void main(String[] args) {
        Point[] points = new Point[3];
        Arrays.fill(points, new Point(0, 0));

        // Modifying one element
        points[0].x = 1;
        points[0].y = 1;

        System.out.println(Arrays.toString(points)); // Output: [(1, 1), (1, 1), (1, 1)]
    }
}
```

Here, `points[0]`, `points[1]`, and `points[2]` all reference the same `Point` object. Changing the `x` and `y` values of `points[0]` affects all three because they all reference the same `Point` instance.

There are many other useful methods in the `Arrays` class, such as `equals()` or `copyOf()`, etc. It's worth exploring the documentation to see what's available.

Generics

If you've been programming in Java for a while, you've probably come across generics at some point. But what exactly are they, and why are they useful?

In simple terms, generics allow you to write code that can work with different types, without losing the benefits of type safety. They provide a way to parameterize types, so that you can create classes, interfaces, and methods that can operate on objects of various types while still maintaining compile-time type checking.

Now, you might be thinking, "Aren't generics just a fancy way to avoid using `Object` everywhere?" It's true that before generics were introduced in Java 5, developers often used the `Object` type to write code that could handle different types. However, this approach has several drawbacks. It requires a lot of explicit casting, which can lead to runtime errors if the wrong type is used. It also doesn't provide any compile-time type safety. Generics, on the other hand, allow you to specify the types you want to work with, providing better type safety and reducing the need for casting.

Understanding Type Erasure

Type erasure is a process where the compiler removes all the generic type information at compile time, replacing it with their bounds or with the `Object` type if no bounds are specified. This means that at

runtime, a generic type like `List<String>` is essentially treated as a plain `List`, without any specific type information.

You might wonder, "If type erasure removes type information, does it mean generics don't provide any type safety at all?" While it's true that the generic type information is not available at runtime due to type erasure, generics still provide significant type safety benefits at compile time. The compiler uses the generic type information to perform type checks and catch potential type-related errors early on. It ensures that you don't accidentally add an object of the wrong type to a generic collection or return the wrong type from a generic method.

However, type erasure does impose some limitations. For instance, you can't use the `instanceof` operator directly with generic types. If you try something like:

```
if (obj instanceof List<String>) {
    // ...
}
```

You'll get a compile error. This is because the generic type information is erased at runtime, so the `instanceof` operator can only check against the raw type (`List` in this case), not the specific parameterized type.

Another limitation is that you can't create arrays of parameterized types. So, you can't do something like:

```
List<String>[] array = new List<String>[10];
```

Again, this is due to type erasure. The compiler doesn't have enough information at runtime to create an array of the specific parameterized type.

You might wonder why Java uses type erasure in the first place. One main reason is to maintain backward compatibility with older versions of Java that didn't have generics. By erasing the generic type information at compile time, generic code can still be used with non-generic legacy code without causing runtime issues.

So, while type erasure can sometimes feel like a limitation, it's a deliberate design choice in Java. It strikes a balance between providing type safety at compile time and maintaining compatibility with earlier versions of the language.

Creating Generic Classes

Now that you have a solid understanding of type erasure, let's explore how to create your own generic classes.

Creating a generic class is quite straightforward. You simply define the class with one or more type parameters in angle brackets after the class name. These type parameters act as placeholders for the actual types that will be used when the class is instantiated.

For example, let's say you want to create a simple generic class called `Pair`, which holds two values of potentially different types:

```
public class Pair<T, U> {
    private T first;
    private U second;

    public Pair(T first, U second) {
        this.first = first;
        this.second = second;
    }

    public T getFirst() {
        return first;
    }

    public U getSecond() {
```

```
        return second;
    }

    public void setFirst(T first) {
        this.first = first;
    }

    public void setSecond(U second) {
        this.second = second;
    }

    public static void main(String[] args) {
        Pair<String, Integer> pair = new Pair<>("Hello", 42);

        // Demonstrate compile-time type safety
        String firstElement = pair.getFirst(); // No casting required
        Integer secondElement = pair.getSecond();

        System.out.println("First: " + firstElement);
        System.out.println("Second: " + secondElement);

        // Compiler will catch type mismatch errors
        // pair.setFirst(100); // Uncommenting this line will cause a compile-time error
    }
}
```

In this example, `T` and `U` are the type parameters. They can be replaced with any valid type when creating an instance of the `Pair` class. For instance, you could create a `Pair<String, Integer>` to hold a pair of a `String` and an `Integer`.

Using `Object` to design a flexible class is not enough. While using `Object` would allow you to store any type of object in your class, it lacks type safety. With generics, you can specify the exact types you want to work with, and the compiler will ensure that only objects of those types are used with your class. This catches potential type-related errors at compile time rather than runtime.

Besides, using generics does not have a significant impact on performance. Remember, the Java compiler performs type erasure, so the generic type information is removed at compile time, and the generated bytecode is essentially the same as if you had used raw types. In most cases, the performance difference is negligible.

Naming Conventions for Generics

When creating generic classes or methods, it's important to follow the established naming conventions for type parameters. While not strictly required by the compiler, adhering to these conventions makes your code more readable and maintainable.

The most common type parameter names are single uppercase letters, such as: - `E` for an element - `K` for a map key - `V` for a map value - `T` for a general type - `S`, `U`, `V`, etc. for additional types

While you could technically use longer names for type parameters, it's generally discouraged. The single-letter names are a widely accepted convention and make the code more concise and easier to read. It's a good idea to stick with the conventional names unless you have a compelling reason to do otherwise.

For example, in the case of maps, the convention is to use `K` for keys and `V` for values, but the compiler won't enforce this. However, following the convention makes your code more consistent and easier for other developers to understand.

These naming conventions provide a consistent vocabulary that developers can rely on when reading and

writing generic code. By following these conventions, you make your code more idiomatic and easier to maintain.

Writing Generic Methods and Constructors

Now that you're familiar with generic classes and type parameters, let's explore another powerful feature of generics: writing generic methods.

Generic methods allow you to write reusable code that can work with different types, providing flexibility and type safety. By defining type parameters at the method level, you can create methods that can accept and return values of varying types.

Here's an example of a generic method:

```
public static <T> void printArray(T[] array) {
    for (T element : array) {
        System.out.println(element);
    }
}
```

In this example, the `printArray` method is defined with a type parameter `T`. The method takes an array of type `T` and prints each element of the array. The type parameter `T` is declared before the return type of the method, enclosed in angle brackets `<>`.

You can invoke this generic method with arrays of different types:

```
String[] strings = { "Hello", "World", "Java" } ;
printArray(strings);

Integer[] integers = { 1, 2, 3, 4, 5 };
printArray(integers);
```

The `printArray` method can be called with an array of strings or an array of integers, demonstrating its flexibility to work with different types.

Here's another example of a generic method that returns a value:

```
public static <T> T getFirst(T[] array) {
    if (array != null && array.length > 0) {
        return array[0];
    }
    return null;
}
```

In this example, the `getFirst` method is defined with a type parameter `T`. It takes an array of type `T` and returns the first element of the array, also of type `T`. If the array is `null` or empty, it returns `null`.

You can invoke this method and assign the result to a variable of the appropriate type:

```
String[] strings = { "Hello", "World", "Java" };
String firstString = getFirst(strings);

Integer[] integers = { 1, 2, 3, 4, 5 };
Integer firstInteger = getFirst(integers);
```

When invoking a generic method, you have the option to explicitly specify the type arguments or let the compiler infer them based on the context.

Here's an example of how to explicitly specify the argument type:

```
String[] strings = { "Hello", "World", "Java" };
String firstString = GenericMethodExample.<String>getFirst(strings);
```

In this example, we explicitly specify the type argument `<String>` when invoking the `getFirst` method. This tells the compiler that the type parameter `T` should be bound to the `String` type.

And here's an example of type inference:

```
Integer[] integers = { 1, 2, 3, 4, 5 };
Integer firstInteger = GenericMethodExample.getFirst(integers);
```

In this case, we omit the explicit type argument and let the compiler infer the type based on the method argument. The compiler infers that the type parameter `T` should be bound to the `Integer` type.

Generic parameters work similarly with constructors:

```
public class GenericBox<T> {
    private T content;

    public GenericBox(T content) {
        this.content = content;
    }

    public T getContent() {
        return content;
    }
}
```

In this example, the `GenericBox` class has a constructor that accepts a generic argument of type `T`. To create an instance of `GenericBox` with a specific type, you can pass the generic argument when calling the constructor:

```
GenericBox<String> stringBox = new GenericBox<>("Hello");
GenericBox<Integer> integerBox = new GenericBox<>(42);
```

By specifying `<String>` or `<Integer>` when creating the `GenericBox` instances, you explicitly define the type of the `content` stored in each box.

You can also pass generic arguments to a static factory method, for example:

```
public class GenericFactory {
    public static <T> List<T> createList(T... elements) {
        return new ArrayList<>(Arrays.asList(elements));
    }
}
```

Here, the `createList` method is a static factory method that creates a new `ArrayList` based on the provided elements. To pass generic arguments when calling this method, you can use the following syntax:

```
List<String> stringList = GenericFactory.<String>createList("Apple", "Banana", "Orange");
List<Integer> integerList = GenericFactory.<Integer>createList(1, 2, 3, 4, 5);
```

By explicitly specifying `<String>` or `<Integer>` before the method name, you indicate the desired type parameter for the created list.

It's important to note that in many cases, the Java compiler can infer the generic type arguments based on the context, such as the types of the method arguments or the variable assignment. In such cases, you can omit the explicit generic argument and let the compiler infer it automatically:

```
List<String> stringList = GenericFactory.createList("Apple", "Banana", "Orange");
```

However, there may be situations where explicitly passing generic arguments is necessary, such as when the compiler cannot infer the type or when you want to enforce a specific type.

Returning Generic Types

In addition to creating generic classes and methods that accept generic type parameters, you can also return generic types from methods. This allows you to write more flexible and reusable code by enabling methods to return values whose types are determined by the type parameters.

Consider this example:

```
public class GenericReturn {
    public static <T> T identity(T value) {
        return value;
    }
}
```

The `identity` method takes a value of type `T` and simply returns it. The method uses the type parameter `T` to specify both the input parameter type and the return type. This is a simple example of returning the same type as the input.

However, you can also return a different type based on the input, for example:

```
public class GenericReturn {
    public static <T, R> R process(T input, Function<T, R> processor) {
        return processor.apply(input);
    }
}
```

In this example, the `process` method takes an input of type `T` and a `Function` that converts `T` to `R`. The method applies the `processor` function to the input and returns the result of type `R`. This demonstrates how you can return a different type based on the input and a provided function.

Or, you can return a generic collection:

```
public class GenericReturn {
    public static <T> List<T> toList(T... elements) {
        return Arrays.asList(elements);
    }
}
```

In this example, the `toList` method takes a varargs parameter of type `T` and returns a `List` of type `T`. This method converts the input elements into a generic list, showcasing how you can return a generic collection.

And here's an example of returning a generic type based on multiple type parameters:

```
public class GenericReturn {
    public static <K, V> Map<K, V> singletonMap(K key, V value) {
        return Collections.singletonMap(key, value);
    }
}
```

The `singletonMap` method takes a key of type `K` and a value of type `V` and returns a `Map` with the key-value pair. This method demonstrates how you can return a generic type that depends on multiple type parameters.

These examples showcase the flexibility and power of returning generic types.

When designing methods with generic return types, consider the following: - Use descriptive and meaningful type parameter names to enhance code readability. - Ensure that the returned type is compatible with the intended usage of the method. - Consider the impact on code complexity and maintainability when using generic return types extensively.

By leveraging generic return types, you can create methods that adapt to different input types and return types, making your code more reusable and applicable to various scenarios.

Overloading a Generic Method

Method overloading is a fundamental feature in Java that allows multiple methods with the same name but different parameter types in the same class. This principle extends to generic methods as well. You can overload a generic method by providing different type parameters or by using different parameter types.

Consider the following example:

```
public class GenericMethodOverloading {
    public static <T> void print(T item) {
        System.out.println("Printing single item: " + item);
    }

    public static <T> void print(T item1, T item2) {
        System.out.println("Printing two items: " + item1 + ", " + item2);
    }

    public static <T, U> void print(T item1, U item2) {
        System.out.println("Printing two items of different types: " + item1 + ", " + item2);
    }
}
```

In this example, we have three overloaded versions of the generic `print` method: 1. The first method takes a single generic parameter `T` and prints it. 2. The second method takes two generic parameters of the same type `T` and prints them. 3. The third method takes two generic parameters of different types `T` and `U` and prints them.

When calling these methods, the compiler will determine which version to invoke based on the number and types of arguments provided.

```
GenericMethodOverloading.print("Hello");
GenericMethodOverloading.print(10, 20);
GenericMethodOverloading.print("Hello", 42);
```

In the above code snippet: - The first method call will invoke the single-parameter version of `print`, with `T` inferred as `String`. - The second method call will invoke the two-parameter version of `print` with the same type, with `T` inferred as `Integer`. - The third method call will invoke the two-parameter version of `print` with different types, with `T` inferred as `String` and `U` inferred as `Integer`.

Overloading generic methods provides flexibility and allows you to define multiple variations of a method that can handle different types or combinations of types.

However, it's important to be cautious when overloading generic methods. The compiler's type inference mechanism may not always be able to determine the intended version of the method to call, especially if the overloaded methods have similar type parameters. In such cases, you may need to explicitly specify the type arguments to disambiguate the method call.

```
GenericMethodOverloading.<String>print("Hello");
```

In this example, we explicitly specify the type argument `<String>` to ensure that the single-parameter version of `print` is called.

Implementing Generic Interfaces

Just as you can define generic classes, you can also define generic interfaces in Java. Generic interfaces provide a way to specify a contract that classes can implement, allowing for greater flexibility and reusability.

Let's review an example to understand how to implement generic interfaces:

```
public interface Processor<T> {
    void process(T data);
```

```
}

public class StringProcessor implements Processor<String> {
    @Override
    public void process(String data) {
        System.out.println("Processing string: " + data);
    }
}

public class IntegerProcessor implements Processor<Integer> {
    @Override
    public void process(Integer data) {
        System.out.println("Processing integer: " + data);
    }
}
```

In this example, we have a generic interface `Processor<T>`. The interface declares a single method `process` that takes an argument of type `T`. The purpose of this interface is to define a contract for processing data of type `T`.

We then have two classes, `StringProcessor` and `IntegerProcessor`, that implement the `Processor` interface with different type parameters.

The `StringProcessor` class implements `Processor<String>`, indicating that it will provide an implementation of the `process` method that handles `String` data. Inside the `process` method, we simply print a message along with the provided string data.

Similarly, the `IntegerProcessor` class implements `Processor<Integer>`, specifying that it will process `Integer` data. The `process` method in this class prints a message along with the provided integer data.

By implementing the generic `Processor` interface, both classes adhere to the contract of processing data, but they can handle different types (`String` and `Integer` in this case).

Here's how you can use the `StringProcessor` and `IntegerProcessor` classes:

```
Processor<String> stringProcessor = new StringProcessor();
stringProcessor.process("Hello, World!");

Processor<Integer> integerProcessor = new IntegerProcessor();
integerProcessor.process(42);
```

In this example, we create instances of `StringProcessor` and `IntegerProcessor` and assign them to variables of type `Processor<String>` and `Processor<Integer>`, respectively. We then invoke the `process` method on each processor, passing the appropriate data type.

The output of this code snippet is:

```
Processing string: Hello, World!
Processing integer: 42
```

Implementing generic interfaces enables code reuse, polymorphism, and the ability to create more general-purpose classes and algorithms.

However, you need to keep in mind the following: - The type parameter specified in the interface declaration must match the type parameter used in the implementing class. - The implementing class must provide an implementation for all the methods declared in the generic interface. - The type parameter can be used within the implementing class to define fields, method parameters, and return types.

Creating Generic Records

Records provide a concise way to define immutable data classes. They can also be generic, allowing you to create flexible and reusable data structures.

Here's an example of creating a generic record:

```
public record Pair<T, U>(T first, U second) {
    public Pair {
        if (first == null || second == null) {
            throw new IllegalArgumentException("Both elements must be non-null");
        }
    }
}
```

In this example, we define a generic record called `Pair`. It has two type parameters, `T` and `U`, representing the types of the first and second elements of the pair.

The `Pair` record has two components: `first` of type `T` and `second` of type `U`. These components are automatically translated into `private final` fields and `public` accessor methods.

We also include a compact constructor in the record definition. The compact constructor allows us to add validation or additional logic during the creation of a `Pair` instance. In this case, we check if either `first` or `second` is `null`, and if so, we throw an `IllegalArgumentException` to enforce that both elements must be non-null.

Creating and using instances of the generic `Pair` record is straightforward:

```
Pair<String, Integer> pair1 = new Pair<>("Hello", 42);
System.out.println(pair1.first() + ", " + pair1.second());

Pair<Double, Boolean> pair2 = new Pair<>(3.14, true);
System.out.println(pair2.first() + ", " + pair2.second());
```

In this example, we create two instances of the `Pair` record with different type arguments. `pair1` is a `Pair<String, Integer>`, representing a pair of a string and an integer. We create it by passing the values "Hello" and 42 to the constructor.

Similarly, `pair2` is a `Pair<Double, Boolean>`, representing a pair of a double and a boolean. We create it by passing the values `3.14` and `true` to the constructor.

We can access the components of the `Pair` instances using the automatically generated accessor methods `first()` and `second()`.

The output of this code snippet would be:

```
Hello, 42
3.14, true
```

Bounding Generic Types

When working with generics, there may be situations where you want to restrict the types that can be used as type arguments. This is where bounding generic types comes into play. Java provides three ways to bound generic types: unbounded wildcards, wildcards with upper bounds, and wildcards with lower bounds.

Unbounded Wildcards Unbounded wildcards, represented by the `?` symbol, provide the most flexibility when working with generic types. They allow any type to be used as the type argument, making them useful in situations where you don't have any specific type constraints.

Consider this example:

```java
public static void printList(List<?> list) {
    for (Object item : list) {
        System.out.println(item);
    }
}
```

Here, we have a generic method `printList` that accepts a `List<?>` as a parameter. The unbounded wildcard `?` means that the method can accept a list of any type. Inside the method, we iterate over the list and print each item.

Here's an example of calling the `printList` method:

```java
List<String> stringList = Arrays.asList("Hello", "World");
printList(stringList);

List<Integer> integerList = Arrays.asList(1, 2, 3);
printList(integerList);
```

In the above code, we create a `List<String>` and a `List<Integer>`, and pass them to the `printList` method. The method can handle lists of any type due to the unbounded wildcard.

One thing to note is that when using an unbounded wildcard, you can only read from the collection and treat the elements as objects of the `Object` class. You cannot add elements to the collection because the compiler doesn't know the specific type of the elements.

Unbounded wildcards are useful when you want to write generic code that can work with any type, without imposing any specific type constraints.

Upper-Bounded Wildcards Upper-bounded wildcards, represented by `? extends type`, restrict the types that can be used as type arguments to subtypes of the specified type. They provide a way to write more specific generic code while still allowing flexibility.

Consider this example:

```java
public static double sumNumbers(List<? extends Number> numbers) {
    double sum = 0;
    for (Number number : numbers) {
        sum += number.doubleValue();
    }
    return sum;
}
```

Here, we have a generic method `sumNumbers` that accepts a `List<? extends Number>` as a parameter. The upper-bounded wildcard `? extends Number` means that the method can accept a list of any type that is a subtype of `Number`, such as `Integer`, `Double`, or `Long`.

Here's an example of calling the `sumNumbers` method:

```java
List<Integer> integerList = Arrays.asList(1, 2, 3);
double integerSum = sumNumbers(integerList);
System.out.println("Sum of integers: " + integerSum);

List<Double> doubleList = Arrays.asList(1.5, 2.7, 3.2);
double doubleSum = sumNumbers(doubleList);
System.out.println("Sum of doubles: " + doubleSum);
```

In the above code, we create a `List<Integer>` and a `List<Double>`, and pass them to the `sumNumbers` method. The method can handle lists of any subtype of `Number` due to the upper-bounded wildcard.

By using an upper-bounded wildcard, we can safely invoke methods defined in the `Number` class, such as `doubleValue()`, on the elements of the list. This allows us to perform specific operations on the elements while maintaining type safety.

However, similar to unbounded wildcards, you cannot add elements to a collection with an upper-bounded wildcard because the compiler doesn't know the specific subtype of the elements.

Upper-bounded wildcards are useful when you want to write generic code that operates on a specific type hierarchy, allowing flexibility within that hierarchy.

Lower-Bounded Wildcards Lower-bounded wildcards, represented by `? super type`, restrict the types that can be used as type arguments to supertypes of the specified type. They provide a way to write generic code that can work with a specific type and its supertypes.

Consider this example:

```
public static void addNumbers(List<? super Integer> numbers) {
    numbers.add(10);
    numbers.add(20);
    numbers.add(30);
}
```

Here, we have a generic method `addNumbers` that accepts a `List<? super Integer>` as a parameter. The lower-bounded wildcard `? super Integer` means that the method can accept a list of any type that is a supertype of `Integer`, such as `Number` or `Object`.

Here's an example of calling the `addNumbers` method:

```
List<Integer> integerList = new ArrayList<>();
addNumbers(integerList);
System.out.println("Integer list: " + integerList);

List<Number> numberList = new ArrayList<>();
addNumbers(numberList);
System.out.println("Number list: " + numberList);
```

In the above code, we create an empty `List<Integer>` and an empty `List<Number>`, and pass them to the `addNumbers` method. The method can add `Integer` objects to both lists because `Integer` is a subtype of `Number` and `Object`.

Unlike unbounded and upper-bounded wildcards, with lower-bounded wildcards, you can safely add elements of the specified type (`Integer` in this case) to the collection. This is because the compiler knows that the collection can hold elements of the specified type or its supertypes.

However, when reading elements from a collection with a lower-bounded wildcard, you can only treat them as objects of the specified type or its supertypes. You cannot assume any more specific type information.

Lower-bounded wildcards are useful when you want to write generic code that can accept a specific type and its supertypes, allowing you to add elements of that type to the collection.

Each type of wildcard serves a specific purpose and provides different capabilities when working with generic types. Remember: - Use unbounded wildcards (`?`) when you don't have any specific type constraints and want to allow any type. - Use upper-bounded wildcards (`? extends type`) when you want to restrict the types to subtypes of a specific type and perform operations specific to that type. - Use lower-bounded wildcards (`? super type`) when you want to restrict the types to supertypes of a specific type and add elements of that type to the collection.

The Collections Framework

One of the most useful parts of the Java standard library is the Collections Framework, which provides a set of reusable components for managing groups of objects. The framework includes several main interfaces that extend from the `java.util.Collection` interface (which in turn, extends from `java.lang.Iterable`) to define the different types of collections:

- The `List` interface represents an ordered collection that allows duplicate elements. Its main implementations are `ArrayList`, which is backed by a resizable array, and `LinkedList`, which uses a doubly-linked list.
- The `Set` interface defines a collection that doesn't allow duplicate elements. The `HashSet` class provides a hash table implementation, while `TreeSet` uses a red-black tree to store its elements, keeping them in ascending order.
- The `Deque` interface, which stands for *double-ended queue*, represents a collection that allows insertion and removal at both ends. Main implementations include `ArrayDeque` and `LinkedList`, with `ArrayDeque` often providing better performance for most operations.
- The `Map` interface maps unique keys to values. The `HashMap` class uses a hash table, providing constant-time performance for basic operations, while `TreeMap` uses a red-black tree and orders its elements by the key's natural ordering or a provided `Comparator`.

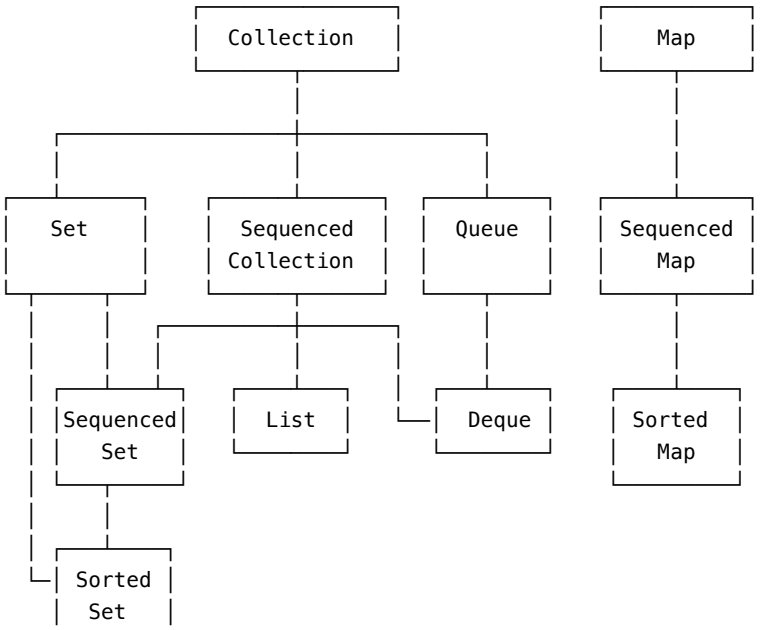
However, it's worth noting that while the `Map` interface is part of the Java Collections Framework, it is not a descendant of the `Collection` interface.

Java 21 introduced three new interfaces to represent collections with a defined encounter order:

- `SequencedCollection`: A collection with a well-defined encounter order, providing uniform APIs for accessing the first and last elements, and processing elements in forward and reverse order.
- `SequencedSet`: A set with a defined encounter order, extending both `Set` and `SequencedCollection`.
- `SequencedMap`: A map with a defined encounter order for its entries, keys, and values.

These new interfaces provide a more consistent way to work with ordered collections across different implementations.

Here's a diagram that shows the hierarchy of these collections, including the new sequenced interfaces:



When declaring a collection, we can take advantage of the *Diamond Operator* (`<>`) to specify the type:

```
List<Integer> numbers = new ArrayList<>();
Map<String, Person> people = new HashMap<>();
```

The compiler will infer the type arguments for the constructor based on the variable declaration.

There are several common operations we can perform on a collection. To add a single element, we use the `add` method:

```
List<String> words = new ArrayList<>();
words.add("hello");
words.add("world");
```

To add all the elements of another collection, use `addAll`:

```
List<String> moreWords = Arrays.asList("goodbye", "cruel", "world");
words.addAll(moreWords);
```

We remove elements with the `remove` method, specifying either the object to remove or its index for ordered collections:

```
words.remove("hello");
words.remove(1); // removes element at index 1
```

The `size` method returns the number of elements currently in the collection:

```
int count = words.size();
```

To remove all elements from a collection, call the `clear` method:

```
words.clear();
```

The `contains` method checks if a collection contains a specified element, returning `true` if found or `false` otherwise:

```
boolean found = words.contains("hello");
```

The `removeIf` method allows removing all elements that satisfy a given predicate:

```
words.removeIf(word -> word.length() < 5);
```

The above code snippet removes all strings with fewer than 5 characters from the `words` list.

The `forEach` method (which actually comes from the `java.lang.Iterable` interface) performs a given action on each element of the collection:

```
words.forEach(word -> System.out.println(word));
```

This prints each word from the list to the console.

The `equals` method checks if another object is equal to the collection. For two collections to be considered equal, they must contain the same elements in the same order (for ordered collections) or the same elements in any order (for unordered collections):

```
List<String> list1 = Arrays.asList("a", "b", "c");
List<String> list2 = Arrays.asList("a", "b", "c");
List<String> list3 = Arrays.asList("c", "b", "a");

System.out.println(list1.equals(list2)); // true
System.out.println(list1.equals(list3)); // false

Set<String> set1 = new HashSet<>(Arrays.asList("a", "b", "c"));
```

```
Set<String> set2 = new HashSet<>(Arrays.asList("c", "b", "a"));

System.out.println(set1.equals(set2)); // true
```

It's important to note that for two collections to be equal, the elements they contain must also implement the `equals` method correctly.

With the introduction of sequenced collections in Java 21, we now have consistent methods for working with the first and last elements of collections with a defined encounter order:

```
SequencedCollection<String> seq = new ArrayList<>(List.of("first", "second", "third"));

String first = seq.getFirst(); // "first"
String last = seq.getLast();   // "third"

seq.addFirst("new first");
seq.addLast("new last");

SequencedCollection<String> reversed =
    seq.reversed(); // [new last, third, second, first, new first]
```

These methods are available on `List`, `Deque`, `LinkedHashSet`, and other collections that implement the new sequenced interfaces.

In the next sections, we'll review in more detail each of these interfaces.

The `List` Interface

As mentioned before, the `List` interface represents an ordered collection that allows duplicate elements. The two main implementations of `List` are `ArrayList` and `LinkedList`. While both classes implement the same interface, they have different performance characteristics.

An `ArrayList` is backed by a dynamic array, which provides amortized constant-time performance for the basic operations (add at the end, get, and set), assuming the index is known. However, inserting or removing elements from the middle of an `ArrayList` can be slow, as it requires shifting all the subsequent elements, resulting in `O(n)` complexity.

On the other hand, a `LinkedList` stores its elements in a doubly-linked list. This provides constant-time performance for insertion and deletion operations at both ends of the list. However, accessing elements by index requires traversing the list from the beginning or the end, which takes linear time. Insertion or deletion in the middle of the list also takes linear time.

Therefore, if your application mainly needs to access elements by index, an `ArrayList` is generally the better choice. If it frequently inserts or deletes elements from the middle of the list, a `LinkedList` may be a better choice.

Creating a `List`

The most common approach to create a `List` instance is to use a constructor:

```
List<String> fruits = new ArrayList<>();
List<String> vegetables = new LinkedList<>();
```

You can also create a `List` from an array using the `Arrays.asList` method:

```
String[] fruitArray = {"apple", "banana", "orange"};
List<String> fruits = Arrays.asList(fruitArray);
```

Note that the `List` returned by `Arrays.asList` is backed by the original array, so any changes made to the array will be reflected in the `List`, and vice versa. Additionally, this `List` has a fixed size, so you cannot add or remove elements.

You can also use the factory methods `List.of` and `List.copyOf` to create unmodifiable lists:

```
List<String> fruits = List.of("apple", "banana", "orange");
List<String> vegetables = List.copyOf(new ArrayList<>(Arrays.asList("carrot", "broccoli", "potato")));
```

The `List.of` method takes a varargs parameter, allowing you to specify the elements individually, while `List.copyOf` creates a new unmodifiable `List` from an existing collection. These unmodifiable lists will throw `UnsupportedOperationException` if you attempt to modify them.

Working with `List` Methods

The `List` interface provides several methods for working with its elements. The `add` method inserts an element at a specified position or appends it to the end of the `List`:

```
List<String> fruits = new ArrayList<>();
fruits.add("apple");
fruits.add(0, "banana");
```

The `get` and `set` methods allow you to access and modify elements by their indices:

```
String fruit = fruits.get(0);
fruits.set(1, "orange");
```

To remove an element, use the `remove` method, specifying either the object to remove or its index:

```
fruits.remove("banana");
fruits.remove(0);
```

The `replaceAll` method applies a given function to each element of the `List`, replacing each element with the result of the function:

```
fruits.replaceAll(String::toUpperCase);
```

To sort the elements of a `List`, use the `sort` method:

```
fruits.sort(Comparator.naturalOrder());
```

The `sort` method uses the natural ordering of the elements, or you can provide a custom `Comparator`.

To convert a `List` to an array, use the `toArray` method:

```
String[] fruitArray = fruits.toArray(new String[0]);
```

The `toArray` method takes an array parameter, which serves as the return type and can also be used to size the resulting array if it's large enough. If the provided array is smaller than the `List`, a new array of the same runtime type will be created with the size of the `List`.

The `Set` Interface

The `Set` interface defines a collection that does not allow duplicate elements. The main implementations of `Set` are `HashSet`, `LinkedHashSet`, and `TreeSet`. Each of these classes have different characteristics and use cases:

- A `HashSet` stores its elements in a hash table, providing constant-time performance for basic operations (add, remove, contains, and size) assuming the hash function disperses the elements properly among the buckets. However, a `HashSet` does not maintain any order of its elements.

- A `LinkedHashSet` is an ordered version of `HashSet` that maintains a doubly-linked list running through all of its entries. This allows `LinkedHashSet` to preserve the insertion order of the elements. The `LinkedHashSet` has slightly higher memory consumption and slightly slower performance for basic operations than a `HashSet`.
- A `TreeSet` stores its elements in a red-black tree, keeping them in ascending order according to their natural ordering or a provided `Comparator`. This provides guaranteed log(n) time cost for basic operations, but it is generally slower than a `HashSet`.

When choosing a `Set` implementation, consider the following: - If you need constant-time performance and don't care about the order of elements, use a `HashSet`. - If you need to maintain the insertion order of elements, use a `LinkedHashSet`. - If you need to keep the elements in a sorted order, use a `TreeSet`.

Creating a `Set`

You can create a `Set` using a constructor, just like with lists:

```
Set<String> fruits = new HashSet<>();
Set<String> vegetables = new LinkedHashSet<>();
Set<String> nuts = new TreeSet<>();
```

Alternatively, you can also use the factory methods `Set.of` and `Set.copyOf` to create unmodifiable sets:

```
Set<String> fruits = Set.of("apple", "banana", "orange");
Set<String> vegetables = Set.copyOf(List.of("carrot", "broccoli", "potato"));
```

Working with `Set` Methods

The `Set` interface provides several methods for working with its elements. The `add` method inserts an element into the `Set` if it's not already present:

```
Set<String> fruits = new HashSet<>();
fruits.add("apple");
fruits.add("banana");
fruits.add("apple"); // This will not be added, as "apple" is already in the Set
```

To check if an element is present in the `Set`, use the `contains` method:

```
boolean containsApple = fruits.contains("apple"); // true
```

To remove an element from the `Set`, use the `remove` method:

```
fruits.remove("banana");
```

The `size` method returns the number of elements in the `Set`:

```
int numberOfFruits = fruits.size();
```

To iterate over the elements of a `Set`, you can use a `for-each` loop or the `forEach` method:

```
for (String fruit : fruits) {
    System.out.println(fruit);
}

fruits.forEach(System.out::println);
```

The `Deque` Interface

The `Deque` interface, which stands for *double-ended queue*, represents a collection that allows insertion and removal at both the head and the tail of the deque. The main implementations of `Deque` are `ArrayDeque` and `LinkedList`:

- An `ArrayDeque` is a resizable array implementation of the `Deque` interface. It provides constant-time performance for insertion and deletion operations at both ends of the deque, making it more efficient than a `LinkedList` for most use cases. An `ArrayDeque` does not have a fixed capacity and will automatically grow as needed.
- A `LinkedList` is a doubly-linked list implementation that also implements the `Deque` interface. It provides constant-time performance for insertion and deletion operations at both ends of the list. `LinkedList` is suitable when you need a `Deque` implementation that can also function as a `List`.

When choosing a `Deque` implementation, consider the following: - If you primarily need a double-ended queue, use an `ArrayDeque` for better performance. - If you need a `Deque` implementation that can also function as a `List`, use a `LinkedList`.

Creating a `Deque`

You can create a `Deque` using a constructor, similar to other collection types:

```
Deque<String> fruits = new ArrayDeque<>();
Deque<String> vegetables = new LinkedList<>();
```

You can also specify an initial capacity for an `ArrayDeque`:

```
Deque<String> fruits = new ArrayDeque<>(20);
```

This creates an `ArrayDeque` with an initial capacity of 20 elements. If the number of elements exceeds the initial capacity, the `ArrayDeque` will automatically grow as needed.

For a `LinkedList`, you can create an empty deque or initialize it with another collection:

```
Deque<String> fruits = new LinkedList<>();
List<String> fruitList = Arrays.asList("apple", "banana", "orange");
Deque<String> fruitDeque = new LinkedList<>(fruitList);
```

Working with `Deque` Methods

The Deque interface provides several methods for working with elements at both ends of the deque. The `addFirst` and `addLast` methods insert elements at the head and the tail of the deque, respectively:

```
Deque<String> fruits = new ArrayDeque<>();
fruits.addFirst("apple");
fruits.addLast("banana");
```

The `getFirst` and `getLast` methods retrieve, but do not remove, the elements at the head and tail of the deque. If the deque is empty, they throw a `NoSuchElementException`:

```
String firstFruit = fruits.getFirst();
String lastFruit = fruits.getLast();
```

To remove and return the elements at the head and tail of the deque, use the `removeFirst` and `removeLast` methods. If the deque is empty, they throw a `NoSuchElementException`:

```
String removedFirstFruit = fruits.removeFirst();
String removedLastFruit = fruits.removeLast();
```

The `Deque` interface also provides methods for using the deque as a stack. The `push` method inserts an element at the head of the deque, the `pop` method removes and returns the element at the head, and the `peek` method retrieves, but does not remove, the element at the head:

```
Deque<String> stack = new ArrayDeque<>();
stack.push("apple");
stack.push("banana");
```

```
String topElement = stack.peek(); // banana
String poppedElement = stack.pop(); // banana
```

These methods are equivalent to using `addFirst`, `removeFirst`, and `getFirst`, respectively, but provide a more intuitive naming convention when using the deque as a stack.

Additionally, the `Deque` interface provides the `offerFirst`, `offerLast`, `peekFirst`, `peekLast`, `pollFirst`, and `pollLast` methods, which are similar to their counterparts without the `offer`, `peek`, or `poll` prefix but behave differently when the deque is empty:

- `offerFirst` and `offerLast`: Insert elements at the head and tail of the deque, respectively. They return a `boolean` value indicating whether the insertion was successful.

```
boolean addedFirst = fruits.offerFirst("apple");
boolean addedLast = fruits.offerLast("banana");
```

- `peekFirst` and `peekLast`: Retrieve, but do not remove, the elements at the head and tail of the deque. They return `null` if the deque is empty.

```
String firstFruit = fruits.peekFirst(); // banana
String lastFruit = fruits.peekLast(); // apple
```

- `pollFirst` and `pollLast`: Remove and return the elements at the head and tail of the deque. They return `null` if the deque is empty.

```
String removedFirstFruit = fruits.pollFirst(); // banana
String removedLastFruit = fruits.pollLast(); // apple
```

These methods are useful when you want to avoid exceptions and handle special cases more gracefully.

The `Map` Interface

The `Map` interface represents a collection that maps unique keys to values. It is not a subtype of the `Collection` interface, but it is still considered part of the Java Collections Framework. The main implementations of `Map` are `HashMap`, `LinkedHashMap`, and `TreeMap`.

- A `HashMap` is an implementation of the `Map` interface that stores key-value pairs in a hash table. It provides constant-time performance for basic operations (put, get, remove) assuming the hash function disperses the elements properly among the buckets. `HashMap` does not guarantee any order of the elements.
- A `LinkedHashMap` is an implementation of the `Map` interface that maintains a doubly-linked list running through all of its entries. This allows it to preserve the insertion order of the key-value pairs. `LinkedHashMap` provides nearly identical performance to `HashMap` for basic operations.
- A `TreeMap` is an implementation of the `Map` interface that stores its entries in a red-black tree, sorted according to the natural ordering of its keys or by a provided `Comparator`. This provides guaranteed `log(n)` time cost for basic operations, but it is generally slower than `HashMap`.

When choosing a `Map` implementation, consider the following: - If you need constant-time performance and don't care about the order of elements, use a `HashMap`. - If you need to maintain the insertion order of key-value pairs, use a `LinkedHashMap`. - If you need to keep the entries sorted by their keys, use a `TreeMap`.

Creating a `Map`

You can create a `Map` using a constructor, similar to other collection types:

```
Map<String, Integer> fruitCounts = new HashMap<>();
Map<String, Integer> vegetableCounts = new LinkedHashMap<>();
Map<String, Integer> nutCounts = new TreeMap<>();
```

You can also create a `Map` with an initial capacity and load factor (for `HashMap` and `LinkedHashMap`):

```
Map<String, Integer> fruitCounts = new HashMap<>(20, 0.8f);
```

This creates a `HashMap` with an initial capacity of 20 and a load factor of 0.8. The load factor determines when the `HashMap` should be resized to maintain performance.

Working with `Map` Methods

The `Map` interface provides several methods for working with its key-value pairs:

- `clear`: Removes all entries from the map.

```
fruitCounts.clear();
```

- `containsKey`: Returns `true` if the map contains the specified key.

```
boolean containsApple = fruitCounts.containsKey("apple");
```

- `containsValue`: Returns `true` if the map contains the specified value.

```
boolean containsCount = fruitCounts.containsValue(5);
```

- `entrySet`: Returns a `Set` view of the entries in the map.

```
Set<Map.Entry<String, Integer>> entries = fruitCounts.entrySet();
```

- `forEach`: Performs the given action for each entry in the map.

```
fruitCounts.forEach((fruit, count) -> System.out.println(fruit + ": " + count));
```

- `get`: Returns the value associated with the specified key, or `null` if the key is not found.

```
Integer appleCount = fruitCounts.get("apple");
```

- `getOrDefault`: Returns the value associated with the specified key, or the given default value if the key is not found.

```
Integer appleCount = fruitCounts.getOrDefault("apple", 0);
```

- `isEmpty`: Returns `true` if the map contains no entries.

```
boolean empty = fruitCounts.isEmpty();
```

- `keySet`: Returns a `Set` view of the keys in the map.

```
Set<String> fruits = fruitCounts.keySet();
```

- `merge`: If the specified key is not already associated with a value or is associated with `null`, associates it with the given non-null value. Otherwise, replaces the associated value with the results of the given remapping function.

```
fruitCounts.merge("apple", 1, Integer::sum);
```

- `put`: Associates the specified value with the specified key in the map.

```
fruitCounts.put("apple", 5);
```

- `putIfAbsent`: If the specified key is not already associated with a value (or is mapped to `null`), associates it with the given value and returns `null`; otherwise, returns the current value.

```
fruitCounts.putIfAbsent("apple", 5);
```

- `remove`: Removes the entry for the specified key from the map if present.

```
fruitCounts.remove("apple");
```

- `replace`: Replaces the entry for the specified key only if it is currently mapped to some value.

```
fruitCounts.replace("apple", 6);
```

- `replaceAll`: Replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.

```
fruitCounts.replaceAll((fruit, count) -> count * 2);
```

- `size`: Returns the number of entries in the map.

```
int numberOfFruits = fruitCounts.size();
```

- `values`: Returns a `Collection` view of the values contained in the map.

```
Collection<Integer> counts = fruitCounts.values();
```

Overriding `hashCode()`

When using a `HashMap` or `LinkedHashMap`, it's essential to ensure that the keys' `hashCode` method is properly overridden. The `hashCode` method should return the same hash code for objects considered equal according to the `equals` method. This is necessary for the map to function correctly and efficiently.

Here's an example of a custom class with properly overridden `hashCode` and `equals` methods:

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

    // Constructor, getters, and setters

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Person person = (Person) o;
        return age == person.age && Objects.equals(name, person.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, age);
    }
}
```

In this example, the `hashCode` method of the `Person` class is implemented by passing the `name` and `age` fields to the `Objects.hash` method. This ensures that the generated hash code is based on the values of these fields.

The `Objects.hash` method is a static utility method provided by the `java.util.Objects` class that generates a hash code for a sequence of input values. Here's the general syntax of the `Objects.hash` method:

```
public static int hash(Object... values)
```

The method accepts any number of arguments of type `Object`, which means you can pass values of different types. It calculates the hash code for each input value using their respective `hashCode` methods and then combines them to produce a single hash code. It also handles `null` values correctly, so you don't need to include `null` checks in your `hashCode` implementation.

It's important to note that when you override the `hashCode` method using `Objects.hash`, you should also override the `equals` method to ensure that objects that are considered equal have the same hash code. This is necessary for the proper functioning of hash-based collections like `HashMap` and `HashSet`.

Sorting Data

Sorting is an operation that allows you to arrange elements in a specific order. In Java, you can sort data using the `Comparable` interface or the `Comparator` interface. The `Comparable` interface defines the natural ordering of elements, while the `Comparator` interface allows you to define custom ordering.

The `Comparable` Interface

To create a class that can be sorted using its natural ordering, you need to implement the `Comparable` interface. It defines a single method, `compareTo`, which compares the current object with another object of the same type.

Here's an example of a `Person` class that implements `Comparable`:

```
class Person implements Comparable<Person> {
    private String name;
    private int age;

    // Constructor, getters, and setters

    @Override
    public int compareTo(Person other) {
        // Compare by age first, then by name if ages are equal
        int ageComparison = Integer.compare(this.age, other.age);
        if (ageComparison != 0) {
            return ageComparison;
        }
        return this.name.compareTo(other.name);
    }
}
```

In this example, the `compareTo` method first compares two `Person` objects based on their age first. If the ages are equal, it compares their names lexicographically. The `compareTo` method returns a negative value, zero, or a positive value if the current object is less than, equal to, or greater than the other object, respectively.

When implementing the `compareTo` method, it's important to handle `null` values correctly to avoid a `NullPointerException`. You can do this by adding a `null` check at the beginning of the method:

```
@Override
public int compareTo(Person other) {
    if (other == null) {
        return 1; // Consider non-null values to be greater than null values
    }
    // Rest of the comparison logic
}
```

In this example, if the `other` object is `null`, the method returns 1, indicating that the current object is greater than the `null` value. You can adjust this behavior based on your specific requirements.

Also, it's important to ensure that the behavior of the `compareTo` method is consistent with the `equals` method. If two objects are considered equal according to the `equals` method, their `compareTo` method should return zero.

Here's an example of an `equals` method that is consistent with the `compareTo` method:

```
class Person implements Comparable<Person> {
    private String name;
    private int age;
```

```
    // Constructor, getters, and setters

    @Override
    public int compareTo(Person other) {
        // Compare by age first, then by name if ages are equal
        int ageComparison = Integer.compare(this.age, other.age);
        if (ageComparison != 0) {
            return ageComparison;
        }
        return this.name.compareTo(other.name);
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Person person = (Person) o;
        return age == person.age && Objects.equals(name, person.name);
    }
}
```

In this example, the `equals` method considers two Person objects equal if they have the same age and name. This is consistent with the `compareTo` method, which compares age first and then name.

The `Comparator` Interface

While the `Comparable` interface defines the natural ordering of elements, the `Comparator` interface allows you to define custom ordering. A `Comparator` is a separate class that contains the comparison logic.

Here's an example of a `Comparator` that compares `Person` objects by their name length:

```
class NameLengthComparator implements Comparator<Person> {
    @Override
    public int compare(Person p1, Person p2) {
        return Integer.compare(p1.getName().length(), p2.getName().length());
    }
}
```

In this example, the `compare` method of the `NameLengthComparator` compares two `Person` objects based on the length of their names. It returns a negative value, zero, or a positive value if the length of the first person's name is less than, equal to, or greater than the length of the second person's name, respectively.

There are several helper methods in the `Comparator` interface that make it easier to build comparators:

- `comparing`: Creates a comparator based on a function that extracts a `Comparable` key from a type.

```
Comparator<Person> nameComparator = Comparator.comparing(Person::getName);
```

- `comparingDouble`, `comparingInt`, `comparingLong`: Create comparators based on functions that extract `double`, `int`, or `long` keys from a type.

```
Comparator<Person> ageComparator = Comparator.comparingInt(Person::getAge);
```

- `naturalOrder`, `reverseOrder`: Create comparators based on the natural ordering or the reverse of the natural ordering of a type.

```
Comparator<String> naturalStringComparator = Comparator.naturalOrder();
Comparator<String> reverseStringComparator = Comparator.reverseOrder();
```

There are also default methods in the `Comparator` interface that allow you to combine and modify comparators:

- `reversed`: Reverses the order of a comparator.

```
Comparator<Person> reversedNameComparator = nameComparator.reversed();
```

- `thenComparing`, `thenComparingDouble`, `thenComparingInt`, `thenComparingLong`: Allow chaining comparators.

```
Comparator<Person> nameAndAgeComparator = nameComparator.thenComparingInt(Person::getAge);
```

In this example, the `nameAndAgeComparator` first compares Person objects by their name, and if the names are equal, it compares them by their age.

Comparing `Comparable` and `Comparator`

Both `Comparable` and `Comparator` are used for sorting elements in Java, but they have some key differences.

In terms of their purpose: - `Comparable` is used to define the natural ordering of elements within a class. It is suitable when the class has an inherent ordering that is appropriate for most use cases. - `Comparator` is used to define custom ordering for elements of a class. It allows for multiple ways to compare elements and is useful when the natural ordering is not appropriate or when you need to sort elements based on different criteria.

In terms of their implementation: - `Comparable` is an interface that is implemented by the class itself. The class must define the `compareTo` method, which compares the current object with another object of the same type and returns a negative value, zero, or a positive value if the current object is less than, equal to, or greater than the other object, respectively. - `Comparator` is an interface that is implemented as a separate class. The class that implements `Comparator` must define the `compare` method, which compares two objects of a specific type and returns a negative value, zero, or a positive value if the first object is less than, equal to, or greater than the second object, respectively.

In terms of their flexibility: - `Comparable` provides a single way to compare elements of a class. Once the `compareTo` method is defined, it becomes the natural ordering for that class. If you need to change the ordering, you have to modify the class itself. - `Comparator` allows for multiple ways to compare elements of a class. You can define multiple `Comparator` classes, each with its own `compare` method, to provide different ordering criteria. This is particularly useful when you need to sort elements based on different attributes or when you want to have alternative sorting options.

Here's an example that demonstrates the flexibility of `Comparator`:

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

    // Constructor, getters, and setters
}

class NameComparator implements Comparator<Person> {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.getName().compareTo(p2.getName());
    }
}

class AgeComparator implements Comparator<Person> {
    @Override
    public int compare(Person p1, Person p2) {
        return Integer.compare(p1.getAge(), p2.getAge());
    }
}
```

```
// Usage
List<Person> people = new ArrayList<>();
// Add elements to the list

// Sort using NameComparator
Collections.sort(people, new NameComparator());

// Sort using AgeComparator
Collections.sort(people, new AgeComparator());
```

In this example, we define two `Comparator` classes: `NameComparator` and `AgeComparator`. `NameComparator` compares `Person` objects based on their names, while `AgeComparator` compares them based on their ages. We can use these `Comparator`s interchangeably to sort the `people` list based on different criteria.

In summary, when choosing between `Comparable` and `Comparator`, consider the following: - If the class has a natural ordering that is appropriate for most use cases and you have control over the class, implement `Comparable`. - If you need multiple ways to compare elements, want to define custom ordering, or need to sort elements of a third-party class, use `Comparator`. - You can use both `Comparable` and `Comparator` together. If a `Comparator` is provided to a sorting method, it takes precedence over the natural ordering defined by `Comparable`.

`Collections.sort` and `Collections.binarySearch`

The `Collections` class provides utility methods for working with collections, including methods for sorting and searching.

The `Collections.sort` method sorts a `List` using its natural ordering (defined by the `Comparable` interface) or a provided `Comparator`:

```
List<Person> people = new ArrayList<>();
// Add elements to the list

// Sort using natural ordering (Comparable)
Collections.sort(people);

// Sort using a custom Comparator
Collections.sort(people, new NameLengthComparator());
```

In this example, the first `Collections.sort` call sorts the `people` list using the natural ordering defined by the `compareTo` method of the `Person` class. The second call sorts the list using the custom `NameLengthComparator`.

The `Collections.binarySearch` method searches for an element in a sorted `List` using the binary search algorithm. The `List` must be sorted in ascending order according to the natural ordering (`Comparable`) or the provided `Comparator`:

```
List<Person> people = new ArrayList<>();
// Add elements to the list and sort it

Person searchKey = new Person("John", 30);
int index = Collections.binarySearch(people, searchKey);
if (index >= 0) {
    System.out.println("Found at index: " + index);
} else {
    System.out.println("Not found");
}
```

In this example, the `Collections.binarySearch` method searches for the `searchKey` object in the sorted `people` list. If the element is found, it returns its index; otherwise, it returns a negative value.

If the `List` is not sorted or is sorted according to a different order than the one used in the binary search, the results are undefined.

It's important to note that when using `Collections.sort` or `Collections.binarySearch` with a custom `Comparator`, the `Comparator` should be consistent with `equals` to ensure proper behavior. If two elements are equal according to the `Comparator`, they should also be equal according to the `equals` method.

Summary of Collection Types

Here are a few tables to help you quickly reference key information about the Java Collections Framework:

Table 1: Collections Interfaces and Implementations

Interface	Description	Main Implementations	Characteristics
`List`	Ordered collection that allows duplicate elements	`ArrayList`, `LinkedList`	`ArrayList` backed by resizable array, `LinkedList` uses doubly-linked list
`Set`	Collection that doesn't allow duplicate elements	`HashSet`, `LinkedHashSet`, `TreeSet`	`HashSet` uses hash table, `LinkedHashSet` maintains insertion order, `TreeSet` uses red-black tree for sorting
`Deque`	Double-ended queue, allows insertion and removal at both ends	`ArrayDeque`, `LinkedList`	`ArrayDeque` resizable array, `LinkedList` doubly-linked list
`Map`	Maps unique keys to values	`HashMap`, `LinkedHashMap`, `TreeMap`	`HashMap` hash table, `LinkedHashMap` maintains insertion order, `TreeMap` red-black tree for sorted keys

Table 2: Core Collections Interfaces Functionality

Interface	Ordering	Duplicates	Null Values
`List`	Ordered	Allowed	Allowed
`Set`	Unordered	Not Allowed	Allowed
`Deque`	Ordered	Allowed	Not Allowed

Table 2.1: Map Interface Functionality

Interface	Ordering	Duplicate Keys	Null Keys	Null Values
`Map`	Unordered	Not Allowed	Allowed	Allowed

Table 3: Common Methods for Collections

Interface	Method	Description
`Collection`	`add(E e)`	Adds an element to the collection
`Collection`	`addAll(Collection<? extends E> c)`	Adds all elements from another collection
`Collection`	`remove(Object o)`	Removes a specified element
`Collection`	`size()`	Returns the number of elements
`Collection`	`clear()`	Removes all elements
`Collection`	`contains(Object o)`	Checks if the collection contains a specified element
`Collection`	`removeIf(Predicate<? super E> filter)`	Removes all elements that satisfy a predicate
`Collection`	`forEach(Consumer<? super E> action)`	Performs an action for each element
`Collection`	`equals(Object o)`	Checks if another object is equal to the collection

Table 4: List-Specific Methods

Method	Description
`add(int index, E element)`	Inserts an element at a specified position
`get(int index)`	Returns the element at a specified position
`set(int index, E element)`	Replaces the element at a specified position
`remove(int index)`	Removes the element at a specified position
`replaceAll(UnaryOperator<E> operator)`	Replaces each element with the result of a function
`sort(Comparator<? super E> c)`	Sorts the list using a comparator
`toArray(T[] a)`	Converts the list to an array

Table 5: Set-Specific Methods

Method	Description
`add(E e)`	Adds an element to the set if not already present
`contains(Object o)`	Checks if the set contains a specified element
`remove(Object o)`	Removes a specified element
`size()`	Returns the number of elements
`forEach(Consumer<? super E> action)`	Performs an action for each element

Table 6: Deque-Specific Methods

Method	Description
`addFirst(E e)`	Inserts an element at the head of the deque
`addLast(E e)`	Inserts an element at the tail of the deque
`getFirst()`	Retrieves, but does not remove, the head of the deque
`getLast()`	Retrieves, but does not remove, the tail of the deque
`removeFirst()`	Removes and returns the head of the deque
`removeLast()`	Removes and returns the tail of the deque
`push(E e)`	Inserts an element at the head of the deque
`pop()`	Removes and returns the element at the head of the deque

Table 7: Map-Specific Methods

Method	Description
`clear()`	Removes all entries from the map
`containsKey(Object key)`	Checks if the map contains a specified key
`containsValue(Object value)`	Checks if the map contains a specified value
`entrySet()`	Returns a set view of the map's entries
`forEach(BiConsumer<? super K,? super V> action)`	Performs an action for each entry
`get(Object key)`	Returns the value associated with a specified key
`getOrDefault(Object key, V defaultValue)`	Returns the value for a key, or a default value if the key is not found
`isEmpty()`	Checks if the map contains no entries
`keySet()`	Returns a set view of the keys in the map
`merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)`	Merges the value with an existing value for the key
`put(K key, V value)`	Associates a value with a key
`putIfAbsent(K key, V value)`	Associates a value with a key if not already associated
`remove(Object key)`	Removes the entry for a key
`replace(K key, V value)`	Replaces the entry for a key
`replaceAll(BiFunction<? super K,? super V,? extends V> function)`	Replaces each value with the result of a function
`size()`	Returns the number of entries
`values()`	Returns a collection view of the values in the map

Key Points

- An array is an object that holds a fixed number of values of a single type in contiguous memory locations.
- To create an array, you declare a variable of the desired array type and use the `new` keyword to create the array object.
- Array elements are automatically initialized with default values (0 for numeric types, `false` for boolean, and `null` for reference types).
- Array indices start at 0. Accessing an element outside the bounds of the array will result in an `ArrayIndexOutOfBoundsException`.
- The `length` attribute gives the number of elements in an array. It is an attribute, not a method, so parentheses are not used.
- Multidimensional arrays are *arrays of arrays*. The most common type is the two-dimensional array, often used to represent matrices or tables of data.
- Anonymous arrays are declared and initialized in a single statement without assigning them to a variable. They are often used when passing an array as an argument to a method.
- The `java.util.Arrays` class contains various static methods for manipulating arrays, including methods for sorting, searching, comparing, and filling array elements.
- The `Arrays.sort()` method sorts the elements of an array into ascending order. It has overloads for different types of arrays and can sort a portion of an array.
- The `Arrays.binarySearch()` method searches for a specific element in a sorted array using the binary search algorithm. The array must be sorted for the method to work correctly.

- The `Arrays.compare()` method compares two arrays lexicographically (by dictionary order).
- The `Arrays.fill()` method fills an array or a portion of it with a specific value. It sets each element to reference the same object for object arrays.
- Generics are a mechanism in Java that allow you to write code that can work with different types while maintaining type safety.
- Type erasure is the process where the compiler removes all generic type information at compile time, replacing it with their bounds or the `Object` type.
- Generic classes are defined with one or more type parameters in angle brackets after the class name. These type parameters act as placeholders for the actual types used when the class is instantiated.
- Generic methods allow you to write reusable code that can work with different types. Type parameters are defined before the method's return type.
- When invoking a generic method, you can explicitly specify the type arguments or let the compiler infer them based on the context.
- Generic constructors and static factory methods can also accept and return generic types.
- When designing methods with generic return types, use descriptive type parameter names, ensure compatibility with intended usage, and consider the impact on code complexity.
- Generic interfaces provide a way to specify a contract that classes can implement, allowing for greater flexibility and reusability.
- Generic records provide a concise way to define immutable data classes that can work with different types.
- Wildcard types (`?`, `? extends T`, `? super T`) allow you to specify unknown types, restrict type parameters to subtypes of a type, or restrict type parameters to supertypes of a type, respectively.
- The Java Collections Framework provides a set of reusable components for managing groups of objects, including the `List`, `Set`, `Deque`, and `Map` interfaces.
- The `Comparable` interface defines the natural ordering of elements within a class, while the `Comparator` interface allows you to define custom ordering for elements of a class.
- The `Collections.sort()` method sorts a `List` using its natural ordering or a provided `Comparator`, while `Collections.binarySearch()` searches for an element in a sorted `List` using the binary search algorithm.

Practice Questions

1. What is the output of the following program?

```java
public class MultiDimArray {
    public static void main(String[] args) {
        int[][] arr = new int[2][3];
        for (int i = 0; i < arr.length; i++) {
            for (int j = 0; j < arr[i].length; j++) {
                arr[i][j] = i + j;
            }
        }
        for (int i = 0; i < arr.length; i++) {
            for (int j = 0; j < arr[i].length; j++) {
                System.out.print(arr[i][j] + " ");
            }
            System.out.println();
        }
```

```
    }
}
```

A)

```
0 0 0
0 0 0
```

B)

```
0 1 2
0 1 2
```

C)

```
0 0 0
1 1 1
```

D)

```
0 1 2
1 2 3
```

2. Which of the following generic method definitions correctly declares a method that returns the first element of a given array?

A)

```
public static T getFirstElement(T[] array) {
    return array[0];
}
```

B)

```
public static <T> T getFirstElement(T[] array) {
    return array[0];
}
```

C)

```
public static <T> getFirstElement(T[] array) {
    return array[0];
}
```

D)

```
public static <T> T[] getFirstElement(T[] array) {
    return array[0];
}
```

3. What is the result of compiling and running the following code?

```
import java.util.*;

public class WildcardTest {
    public static void printList(List<? extends Number> list) {
        for (Number n : list) {
            System.out.print(n + " ");
        }
        System.out.println();
    }

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

```
        List<Integer> ints = Arrays.asList(1, 2, 3);
        List<Double> doubles = Arrays.asList(1.1, 2.2, 3.3);
        List<String> strings = Arrays.asList("one", "two", "three");

        printList(ints);
        printList(doubles);
        printList(strings);
    }
}
```

A) The code compiles and prints: `1 2 3  1.1 2.2 3.3  one two three`

B) The code compiles and prints: `1 2 3  1.1 2.2 3.3`

C) The code does not compile due to an error in the `printList` method.
D) The code does not compile due to an error in the `main` method.
E) The code compiles but throws a runtime exception when executed.

4. What is the output of the following program?

```
import java.util.*;

public class ListExample {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>(Arrays.asList("A", "B", "C", "D"));
        list.add(2, "E");
        System.out.println(list);
    }
}
```

A) `[A, B, E, C, D]`
B) `[A, E, B, C, D]`
C) `[A, B, C, E, D]`
D) `[A, B, C, D, E]`
E) `[A, C, B, E, D]`

5. Which of the following statements about the `Set` interface are true? (Choose all that apply.)

A) A `Set` allows duplicate elements.
B) Elements in a `Set` are maintained in the order they were inserted.
C) The `Set` interface includes methods for adding, removing, and checking the presence of elements.
D) The `Set` interface is implemented by classes like `HashSet`, `LinkedHashSet`, and `TreeSet`.
E) A `Set` guarantees constant-time performance for the basic operations (add, remove, contains).

6. What will be the output of the following program?

```
import java.util.*;

public class DequeExample {
    public static void main(String[] args) {
        Deque<String> deque = new ArrayDeque<>();
        deque.addFirst("A");
        deque.addLast("B");
        deque.addFirst("C");
        deque.addLast("D");

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

A) `[A, B, C, D]`
B) `[C, B, A, D]`
C) `[C, A, B, D]`
D) `[D, B, A, C]`
E) `[A, C, B, D]`

7. What will be the output of the following program?

```
import java.util.*;

public class MapExample {
    public static void main(String[] args) {
        Map<Integer, String> map = new HashMap<>();
        map.put(1, "A");
        map.put(2, "B");
        map.put(3, "C");
        map.put(2, "D");

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

A) `{1=A, 2=B, 3=C, 2=D}`
B) `{1=A, 2=B, 3=C}`
C) `{1=A, 2=D, 3=C, 2=D}`
D) `{1=A, 2=D, 3=C}`
E) `{1=A, 3=C, 2=B}`

8. What is the result of running the following program?

```
import java.util.*;

public class ComparableExample {
    public static void main(String[] args) {
        List<Person> people = new ArrayList<>();
        people.add(new Person("Alice", 30));
        people.add(new Person("Bob", 25));
        people.add(new Person("Charlie", 35));

        Collections.sort(people);

        for (Person p : people) {
            System.out.println(p.getName() + " " + p.getAge());
        }
    }
}

class Person implements Comparable<Person> {
    private String name;
    private int age;

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

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

    public int getAge() {
        return age;
    }

    @Override
    public int compareTo(Person other) {
        return Integer.compare(this.age, other.age);
    }
}
```

A)

```
Alice 30
Bob 25
Charlie 35
```

B)

```
Charlie 35
Alice 30
Bob 25
```

C)

```
Bob 25
Alice 30
Charlie 35
```

D)

```
Bob 25
Charlie 35
Alice 30
```

E)

```
Alice 30
Charlie 35
Bob 25
```

9. What will be the output of the following program when using the provided `Comparator`?

```
import java.util.*;

public class ComparatorExample {
    public static void main(String[] args) {
        List<Person> people = new ArrayList<>();
        people.add(new Person("Alice", 30));
        people.add(new Person("Bob", 25));
        people.add(new Person("Charlie", 35));

        Collections.sort(people, new AgeComparator());

        for (Person p : people) {
            System.out.println(p.getName() + " " + p.getAge());
        }
```

```
    }
}

class Person {
    private String name;
    private int age;

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

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}

class AgeComparator implements Comparator<Person> {
    @Override
    public int compare(Person p1, Person p2) {
        return Integer.compare(p1.getAge(), p2.getAge());
    }
}
```

A)

```
Bob 25
Alice 30
Charlie 35
```

B)

```
Charlie 35
Alice 30
Bob 25
```

C)

```
Alice 30
Bob 25
Charlie 35
```

D)

```
Bob 25
Charlie 35
Alice 30
```

E)

```
Alice 30
Charlie 35
Bob 25
```

Chapter SIX

Arrays, Generics, and Collections

Answers

1. The correct answer is D.

Explanation:

- **A)**

```
0 0 0
0 0 0
```

- This option is incorrect because the array elements are initialized and modified within the loops. The values are not all zeros.

- **B)**

```
0 1 2
0 1 2
```

- This option is incorrect because each row is initialized with incremental values based on the sum of indices, not identical for both rows.

- **C)**

```
0 0 0
1 1 1
```

- This option is incorrect because the values should be the sum of the row index and the column index, not all zeros or all ones for the second row.

- **D)**

```
0 1 2
1 2 3
```

- This is the correct answer. Each element of the array is set to the sum of its indices. So, `arr[0][0] = 0 + 0 = 0`, `arr[0][1] = 0 + 1 = 1`, `arr[0][2] = 0 + 2 = 2`, `arr[1][0] = 1 + 0 = 1`, `arr[1][1] = 1 + 1 = 2`, `arr[1][2] = 1 + 2 = 3`.

2. The correct answer is B.

Explanation:

- **A)**

```
public static T getFirstElement(T[] array) {
    return array[0];
}
```

- This option is incorrect because the generic type `<T>` is missing before the return type `T`.

- **B)**

```
public static <T> T getFirstElement(T[] array) {
    return array[0];
}
```

- This is the correct answer. The generic type `<T>` is correctly declared before the return type `T`.

- **C)**

```
public static <T> getFirstElement(T[] array) {
    return array[0];
}
```

- This option is incorrect because the return type `T` is missing.
- **D)**

```
public static <T> T[] getFirstElement(T[] array) {
    return array[0];
}
```

- This option is incorrect because the return type is `T[]`, which does not match the intended method return type.

3. The correct answer is D.

Explanation:

A) The code compiles and prints:

```
1 2 3
1.1 2.2 3.3
one two three
```

- This option is incorrect. The code does not compile, so it cannot produce any output.

B) The code compiles and prints:

```
1 2 3
1.1 2.2 3.3
```

- This option is incorrect. While this would be the output if the `printList(strings)` line were removed, the code as written does not compile.

C) The code does not compile due to an error in the `printList` method. - This option is incorrect. The `printList` method is correctly defined using an upper bound wildcard `<? extends Number>`.

D) The code does not compile due to an error in the `main` method. - This option is correct. The code fails to compile due to an error in the `main` method. `printList(strings)` causes a compilation error because `String` is not a subclass of `Number`.

E) The code compiles but throws a runtime exception when executed. - This option is incorrect. The code fails to compile so it cannot be executed.

4. The correct answer is A.

Explanation:

- **A)** `[A, B, E, C, D]`
  - This option is correct. The `add` method with an index parameter inserts the specified element at the specified position in the list. All elements after the specified position are shifted to the right. Hence, `"E"` is inserted at index 2, pushing `"C"` and `"D"` to the right.
- **B)** `[A, E, B, C, D]`
  - This option is incorrect. This would be the result if `"E"` were added at index 1, not index 2.
- **C)** `[A, B, C, E, D]`
  - This option is incorrect. This would be the result if `"E"` were added at index 3, not index 2.
- **D)** `[A, B, C, D, E]`
  - This option is incorrect. This would be the result if `"E"` were added at the end of the list, not at index 2.
- **E)** `[A, C, B, E, D]`

- This option is incorrect. This sequence does not follow the proper behavior of the `add` method with index 2. It seems like a random shuffle and doesn't correspond to how elements are shifted when a new element is added.

5. The correct answers are C and D..

Explanation:

- **A)** A `Set` allows duplicate elements.
  - This option is incorrect. One of the primary characteristics of a `Set` is that it does not allow duplicate elements. Each element must be unique.
- **B)** Elements in a `Set` are maintained in the order they were inserted.
  - This option is incorrect. The ordering of elements depends on the specific implementation of the `Set` interface. For example, `HashSet` does not maintain any order, while `LinkedHashSet` maintains insertion order, and `TreeSet` maintains a sorted order.
- **C)** The `Set` interface includes methods for adding, removing, and checking the presence of elements.
  - This option is correct. The `Set` interface provides methods such as `add()`, `remove()`, and `contains()` to manage its elements.
- **D)** The `Set` interface is implemented by classes like `HashSet`, `LinkedHashSet`, and `TreeSet`.
  - This option is correct. `HashSet`, `LinkedHashSet`, and `TreeSet` are all concrete implementations of the `Set` interface, each with different characteristics regarding order and performance.
- **E)** A `Set` guarantees constant-time performance for the basic operations (add, remove, contains).
  - This option is incorrect. This statement is true for `HashSet` specifically, which provides average constant-time performance for these operations. However, it is not true for all `Set` implementations. For example, `TreeSet` provides logarithmic time performance for these operations because it is based on a Red-Black tree.

6. The correct answer is C.

Explanation:

- **A)** `[A, B, C, D]`
  - This option is incorrect.This option ignores the order in which elements are added to the deque. It simply lists elements in the order they appear to be added without considering the `addFirst` and `addLast` methods.
- **B)** `[C, B, A, D]`
  - This option is incorrect. This option incorrectly assumes `"A"` is added after `"B"`, howerver, `addFirst("A")` puts `"A"` at the second position.
- **C)** `[C, A, B, D]`
  - This option is correct. This is indeed the correct output. The method `addFirst("C")` puts "C" at the front, `addFirst("A")` puts `"A"` at the second position, `addLast("B")` adds `"B"` after `"A"`, and `addLast("D")` adds `"D"` at the end. Thus, the final order is `[C, A, B, D]`.
- **D)** `[D, B, A, C]`
  - This option is incorrect. This option shows the reverse order, which does not match how elements are actually added to the deque.
- **E)** `[A, C, B, D]`
  - This option is incorrect. This option incorrectly assumes `"A"` is added before `"C"` despite `addFirst("C")` being called after `addFirst("A")`.

7. The correct answer is D.

Explanation:

- **A)** `{1=A, 2=B, 3=C, 2=D}`
  - This option is incorrect. This option suggests that the map would keep duplicate keys, which is not true for a `Map`. A key can only have one value associated with it at a time.
- **B)** `{1=A, 2=B, 3=C}`

  - This option is incorrect. This option ignores the fact that the value associated with key `2` is updated from `"B"` to `"D"`.
- **C)** `{1=A, 2=D, 3=C, 2=D}`
  - This option is incorrect. This option again suggests that the map can have duplicate keys, which it cannot.
- **D)** `{1=A, 2=D, 3=C}`
  - This option is correct. The `put` method updates the value associated with a key if the key already exists in the map. Therefore, the value associated with key `2` is updated from `"B"` to `"D"`.
- **E)** `{1=A, 3=C, 2=B}`
  - This option is incorrect. This option ignores the update to the value associated with key `2` from `"B"` to `"D"`.

8. The correct answer is C.

Explanation:

- **A)**

```
Alice 30
Bob 25
Charlie 35
```

- This option is incorrect. This option lists the elements in their original order, not the sorted order based on age.

- **B)**

```
Charlie 35
Alice 30
Bob 25
```

- This option is incorrect. This option lists the elements in descending order of age, but the `compareTo` method sorts in ascending order of age.

- **C)**

```
Bob 25
Alice 30
Charlie 35
```

- This option is correct. The `compareTo` method sorts the `Person` objects in ascending order based on their age. Hence, the sorted order is `Bob (25)`, `Alice (30)`, and `Charlie (35)`.

- **D)**

```
Bob 25
Charlie 35
Alice 30
```

- This option is incorrect. This option does not correctly follow the ascending order of age.

- **E)**

```
Alice 30
Charlie 35
Bob 25
```

- This option is incorrect. This option does not correctly follow the ascending order of age.

9 .The correct answer is A.

Explanation:

- **A)**

```
Bob 25
Alice 30
Charlie 35
```

- This option is incorrect. The `AgeComparator` sorts the `Person` objects in ascending order based on their age. Hence, the sorted order is `Bob (25)`, `Alice (30)`, and `Charlie (35)`.

- **B)**

```
Charlie 35
Alice 30
Bob 25
```

- This option is incorrect. This option lists the elements in descending order of age, but the `AgeComparator` sorts in ascending order of age.

- **C)**

```
Alice 30
Bob 25
Charlie 35
```

- This option is incorrect. This option does not correctly follow the ascending order of age.

- **D)**

```
Bob 25
Charlie 35
Alice 30
```

- This option is incorrect. This option does not correctly follow the ascending order of age.

- **E)**

```
Alice 30
Charlie 35
Bob 25
```

- This option is incorrect. This option does not correctly follow the ascending order of age.

Chapter SEVEN

Error Handling and Exceptions

Chapter Content

- Understanding Exceptions
  - Understanding Exception Types
  - Throwing an Exception
  - Custom Exceptions
  - Exceptions and Methods
  - Understanding Stack Traces
  - Recognizing Exception Classes
- Handling Exceptions
  - The `try-catch` Block
  - The `finally` Block
- Automating Resource Management with the `try-with-resources` Block
- Key Points
- Practice Questions

Understanding Exceptions

Being able to run a program is not the same as running the program correctly. Any non-trivial program is susceptible to errors. Errors in user inputs, mathematical operations, and accessing resources, etc.

That's where exceptions come into play. An exception indicates some sort of abnormal or exceptional condition that disrupts the normal flow of the program.

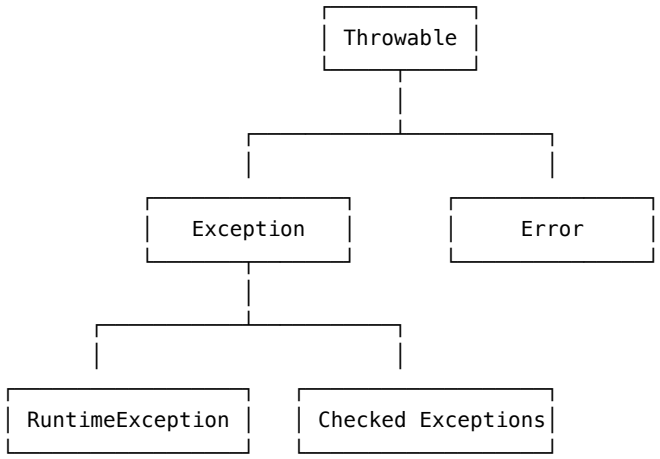
When an exceptional condition arises, an exception is said to be *thrown*. When this happens, the normal flow of the program is disrupted and the execution is transferred to a special block of code called an exception handler, if one exists for the exception.

The purpose of exceptions is to handle these errors gracefully and prevent the program from crashing or behaving unexpectedly. By using exceptions, you can separate the error-handling code from the normal program logic, making your code cleaner and more manageable.

Exceptions don't necessarily stop the program entirely when they happen. When an exception is thrown, it is propagated up the call stack until it is caught and handled by an exception handler. If no suitable handler is found, the program will terminate. But if you have a `try-catch` block in place to handle the exception, your program can deal with the issue and continue running.

Understanding Exception Types

Most common exception classes in Java are subtypes of the `java.lang.Exception` class. But that's not the whole story. To really understand exceptions, we need to look at the exception hierarchy:



At the top of the hierarchy is the `java.lang.Throwable` class. Beneath `Throwable` are two branches: `Exception` and `Error`. While they might sound similar, they actually represent quite different things in Java.

Exceptions are conditions that a reasonable application might want to catch and handle. They typically represent conditions that, while unusual, are not entirely unexpected. For example, trying to open a file that doesn't exist would throw a `FileNotFoundException`.

Errors, on the other hand, are not meant to be caught or handled by your program. They indicate serious problems that a reasonable application should not try to catch. Most such errors are abnormal conditions. For example, if your application runs out of memory, an `OutOfMemoryError` will be thrown.

Beneath the `Exception` class, there are two further categories: checked exceptions and unchecked exceptions (also known as runtime exceptions because they extend from `java.lang.RuntimeException`).

Checked exceptions are exceptional conditions that a well-written application should anticipate and handle. These are typically exceptions that are outside the control of the program, such as a file not being found, a network connection failing, or invalid user input. Checked exceptions are subclasses of `Exception` but not `RuntimeException`.

Unchecked exceptions are exceptional conditions that the application usually cannot anticipate or recover from. These usually indicate programming bugs, such as logic errors or improper use of an API. Unchecked exceptions are subclasses of `RuntimeException`.

While it might seem like unchecked exceptions are sufficient, checked exceptions serve an important purpose. They force the programmer to handle the exception, ensuring that proper error handling code is written. This leads to more robust and reliable code.

On the other hand, unchecked exceptions don't need to be declared in a method's `throws` clause if they can be thrown by the execution of the method. These usually represent defects in the program (bugs) and as such, the API client code cannot reasonably be expected to recover from them or to handle them in any way. Such exceptions most often indicate programming defects, and an unchecked exception is the way the Java programming language allows a developer to indicate a potential defect where the compiler cannot easily detect the problem.

Throwing an Exception

So far, we've talked about what exceptions are and the different types of exceptions. But how do exceptions actually get thrown?

An exception can be thrown in two ways: automatically by the Java runtime system, or explicitly by your code.

Many exceptions are thrown automatically by the Java runtime system. For example, if you try to access an array element with an index that is out of bounds, an `ArrayIndexOutOfBoundsException` will be thrown. If you try to divide a number by zero, an `ArithmeticException` will be thrown.

But you can also throw exceptions explicitly in your code using the `throw` statement. The general form of the `throw` statement is:

```
throw new ExceptionType(messageString);
```

Here, `ExceptionType` is the type of exception you want to throw, and `messageString` is an optional string that provides more information about the exception.

For example, let's say you have a method that accepts an integer parameter age. If the passed-in age is negative, you might want to throw an exception:

```
public void checkAge(int age) {
    if (age < 0) {
        throw new IllegalArgumentException("Age cannot be negative");
    }
    // rest of the method
}
```

In this case, we are throwing an `IllegalArgumentException`, which is a type of unchecked exception.

There are many cases where it's appropriate, and even necessary, for you to throw exceptions in your own code.

By throwing exceptions, you can signal that an error has occurred and provide information about what went wrong. This is especially important when you're writing methods or classes that will be used by other developers. By throwing exceptions, you can communicate to the user of your code that they have used your method or class incorrectly, or that something has gone wrong that they need to handle.

Moreover, by throwing exceptions, you can separate the error-handling code from the normal flow of your program. This makes your code more readable and maintainable.

Custom Exceptions

While Java provides a rich set of built-in exceptions, there are situations where it can be beneficial to create your own custom exceptions.

Custom exceptions allow you to add more context and meaning to the exceptions thrown by your application. They can help to better encapsulate the error conditions specific to your application domain.

For example, if you're writing a library for parsing XML files, you might define a custom `XMLFileParseException` that you throw whenever there's an error parsing an XML file. This communicates to the user of your library exactly what went wrong, as opposed to just throwing a generic `Exception`.

To create a custom checked exception, you simply need to extend the `Exception` class (or one of its subclasses):

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

To create a custom unchecked exception, you extend the `RuntimeException` class (or one of its subclasses):

```
public class InvalidInputException extends RuntimeException {
    public InvalidInputException(String message) {
        super(message);
    }
}
```

Then you can throw your custom exceptions just like any other exception:

```
throw new XMLFileParseException("Error parsing XML file: " + fileName);
throw new InvalidInputException("Input cannot be negative");
```

When deciding whether to make your custom exception checked or unchecked consider the following guidelines:

- Use checked exceptions for exceptional conditions that the caller should recover from. These often represent conditions that are outside the control of the program, such as a file not being found or a network connection failing.
- Use unchecked exceptions (extending `RuntimeException`) for exceptional conditions that the caller usually cannot recover from. These often indicate programming errors, such as trying to access an array element with an out-of-bounds index.

Another thing to consider when creating custom exceptions is serialization. If your exception class is going to be thrown across different JVMs (for example, in a distributed system), it should implement the `java.io.Serializable` interface.

```
public class RemoteServiceException extends Exception implements Serializable {
    // ...
}
```

This ensures that the exception object can be successfully serialized and deserialized when it's transmitted across the network.

Finally, when creating custom exceptions, it's a good practice to provide constructors that accept a message string and a cause exception. The cause is the exception that triggered your exception. This allows you to wrap lower-level exceptions in your higher-level custom exceptions, which can provide more context about the error.

```
public class DataAccessException extends Exception {
    public DataAccessException(String message) {
```

```
        super(message);
    }

    public DataAccessException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

Then you can use it like this:

```
try {
    // some database operation that throws a SQLException
} catch (SQLException ex) {
    throw new DataAccessException("Error accessing database", ex);
}
```

This way, the caller of your code knows that a `DataAccessException` occurred, but can still access the underlying cause (the `SQLException`) if needed for more detailed error handling or logging.

Exceptions and Methods

When a method throws an exception, it must declare this in its method signature. This is done using the `throws` keyword followed by a list of the exceptions that the method might throw.

```
public void readFile(String fileName) throws FileNotFoundException {
    // code that might throw a FileNotFoundException
}
```

In this example, the `readFile` method declares that it might throw a `FileNotFoundException`.

However, not all exceptions need to be declared in the method signature. Only checked exceptions need to be declared. Unchecked exceptions (those that extend `RuntimeException`) do not need to be declared.

This brings us to an important distinction: the difference between `throw` and `throws`.

- `throw` is used to actually throw an exception within a method.
- `throws` is used in a method signature to declare that the method might throw an exception.

A useful analogy for remembering the difference is a baseball game:

- A pitcher throws the ball.
- But before the game, the manager tells the umpire what kind of pitches his pitcher throws (fastball, curveball, etc.).

Similarly in Java:

- A method throws an exception.
- But in the method signature, the method declares the types of exceptions it throws.

Overriding Methods with Exceptions When you override a method in a subclass, you're allowed to declare that the method throws fewer checked exceptions than the method you're overriding.

```
class Parent {
    public void doSomething() throws IOException, SQLException {
        // ...
    }
}

class Child extends Parent {
    @Override
    public void doSomething() throws IOException {
```

```
        // ...
    }
}
```

In this example, the `doSomething` method in the `Parent` class declares that it can throw either an `IOException` or a `SQLException`. But when we override `doSomething` in the `Child` class, we declare that it only throws `IOException`.

This is allowed because it makes the method more usable. A caller of the `Child` class's `doSomething` method only needs to handle `IOException`, not `SQLException`.

However, the reverse is not allowed. If the parent class's method does not declare any exceptions, the overriding method in the child class cannot declare any checked exceptions.

```
class Parent {
    public void doSomething() {
        // ...
    }
}

class Child extends Parent {
    @Override
    public void doSomething() throws IOException {  // Compile-time error
        // ...
    }
}
```

This code will not compile because the overriding method (in `Child`) declares a checked exception (`IOException`) that the original method (in `Parent`) does not declare.

The rule is that an overriding method can declare to throw fewer exceptions or narrower exceptions (subclasses of the declared exceptions) than the original method, but not more or broader exceptions.

This rule exists to ensure that a child class can always be used in place of its parent class without causing any unexpected checked exceptions. This is a fundamental principle of polymorphism and inheritance in Java.

However, notice that this only applies for checked exceptions. Unchecked exceptions can be added freely when overriding methods.

Understanding Stack Traces

When an exception occurs in a Java program, it prints out a stack trace. A stack trace provides information about the exception and the state of the program when the exception occurred.

A stack trace can be incredibly useful when debugging a program. It tells you what went wrong and where in the code it went wrong.

Let's look at an example stack trace:

```
Exception in thread "main" java.lang.NullPointerException
    at com.example.myproject.Book.getTitle(Book.java:16)
    at com.example.myproject.Author.getBookTitles(Author.java:25)
    at com.example.myproject.App.main(Bootstrap.java:14)
```

This stack trace is telling us that a `NullPointerException` occurred in the `getTitle` method of the `Book` class, which was called from line 25 of the `getBookTitles` method of the `Author` class, which in turn was called from line 14 of the `main` method of the `App` class.

Each line in the stack trace represents a method call, with the most recent call at the top. The first line shows the thrown exception, followed by the method calls on the stack at that time.

For each method call, the stack trace shows: - The fully qualified name of the class containing the method - The name of the method - The name of the source code file containing the method - The line number in the source code file where the method call occurred

To read a stack trace, you can start from the top and work your way down. The top line tells you what type of exception was thrown. The lines after that represent the method calls on the stack, with the most recent call at the top.

Each line provides a clue about the program's state when the exception was thrown. You can use these clues to pinpoint the location in your code where the problem occurred.

Recognizing Exception Classes

When you encounter an exception in your Java program, one of the first steps in resolving the issue is to identify what type of exception it is. Previously, you learned about the hierarchy of exception classes, each designed to represent a specific type of problem. Recognizing these classes and understanding when they're thrown can help you diagnose issues more quickly.

Tip 1: Read the Exception Class Name The exception class name often indicates what went wrong. For example, a `NullPointerException` suggests that you're trying to use a `null` reference, an `ArrayIndexOutOfBoundsException` indicates that you're trying to access an array with an invalid index, and an `IOException` signals that something went wrong during an input/output operation.

Familiarizing yourself with the names and meanings of the most common exception classes can help you quickly identify issues in your code.

Tip 2: Understand the Exception Hierarchy Understanding the Java exception hierarchy can also aid in recognizing exceptions. All exceptions in Java inherit from the `Throwable` class, which has two main subclasses: `Exception` and `Error`.

Exceptions that inherit directly from the `Exception` class are checked exceptions. These typically represent issues that should be handled in your code. Common examples include `IOException` and `SQLException`

Exceptions that inherit from the `RuntimeException` class (which is a subclass of `Exception`) are unchecked exceptions. These often indicate programming errors, such as trying to access an array element with an out-of-bounds index (`ArrayIndexOutOfBoundsException`) or trying to use a `null` reference (`NullPointerException`).

Errors, on the other hand, represent serious problems that a reasonable application should not try to catch. These are typically irrecoverable conditions, such as running out of memory (`OutOfMemoryError`) or a stack overflow (`StackOverflowError`).

Tip 3: Read the Exception Message When an exception is thrown, it usually includes a message with more details about what went wrong. This message can be incredibly helpful in diagnosing the issue.

For example, consider this exception message:

```
java.lang.ArrayIndexOutOfBoundsException: Index 10 out of bounds for length 5
```

This message indicates that the code is trying to access the element at index 10 in an array that only has 5 elements.

Tip 4: Look at the Stack Trace The stack trace that accompanies an exception can also provide valuable clues about what went wrong. The stack trace shows the sequence of method invocations that led to the exception.

Each line in the stack trace represents a method call, with the most recent call at the top. The line tells you the name of the method, the class it's in, and the line number in the source code where the call occurred.

By tracing through the stack trace, you can often pinpoint the exact location in your code where the problem occurred.

Tip 5: Consult the Java API Documentation If you encounter an exception that you're not familiar with, the Java API documentation can be a great resource. The documentation for each exception class provides information on when the exception is thrown and often includes examples of how to handle it.

For instance, the documentation for the `ArrayIndexOutOfBoundsException` states:

```
Thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or
```

This provides a clear explanation of when you can expect to encounter this exception.

Finally, here's a list of common unchecked and checked exceptions, as well as error classes:

Common Unchecked Exceptions Unchecked exceptions are those that are not checked at compile-time. They usually represent programming errors, such as logic mistakes or improper use of an API.

1. **ArithmeticException**
   - Thrown when an exceptional arithmetic condition has occurred.
2. **ArrayIndexOutOfBoundsException**
   - Thrown to indicate that an array has been accessed with an illegal index.
3. **ClassCastException**
   - Thrown to indicate that the code has attempted to cast an object to a subclass of which it is not an instance.
4. **IllegalArgumentException**
   - Thrown to indicate that a method has been passed an illegal or inappropriate argument.
5. **IllegalStateException**
   - Signals that a method has been invoked at an illegal or inappropriate time.
6. **NullPointerException**
   - Thrown when an application attempts to use `null` in a case where an object is required.
7. **NumberFormatException**
   - Thrown to indicate that the application has attempted to convert a string to one of the numeric types, but that the string does not have the appropriate format.

Common Checked Exceptions Checked exceptions are those that are checked at compile-time. They usually represent conditions that a reasonable application might want to catch.

1. **ClassNotFoundException**
   - Thrown when an application tries to load a class through its string name but no definition for the class with the specified name could be found.
2. **IOException**
   - Signals that an I/O exception of some sort has occurred.
3. **FileNotFoundException**
   - Signals that an attempt to open the file denoted by a specified pathname has failed.
4. **InterruptedException**
   - Thrown when a thread is waiting, sleeping, or otherwise occupied, and the thread is interrupted.
5. **SQLException**
   - An exception that provides information on a database access error or other errors.
6. **TimeoutException**
   - Thrown when a blocking operation times out.

Common Error Classes Errors are usually thrown by the Java Virtual Machine and indicate serious problems that applications should not try to catch.

1. **OutOfMemoryError**
   - Thrown when the JVM cannot allocate an object because it is out of memory.
2. **StackOverflowError**
   - Thrown when a stack overflow occurs because an application recurses too deeply.
3. **VirtualMachineError**
   - Thrown to indicate that the Java Virtual Machine is broken or has run out of resources necessary for it to continue operating.

4. **UnknownError**
    - Thrown when an unknown but serious exception has occurred.

Handling Exceptions

When an exception occurs in your Java program, you need to handle it to prevent your program from abruptly terminating. This is done using `try-catch` blocks.

The `try-catch` Block

The basic syntax of a try-catch block is as follows:

```
try {
    // code that might throw an exception
} catch (ExceptionType e) {
    // code to handle the exception
}
```

You place the code that might throw an exception in the `try` block. If an exception occurs within the `try` block, it is caught by the `catch` block. The `catch` block specifies the type of exception it can handle (`ExceptionType` in the syntax above) and provides the code to handle that exception.

Here's a concrete example:

```
try {
    File file = new File("example.txt");
    Scanner scanner = new Scanner(file);
    while (scanner.hasNext()) {
        System.out.println(scanner.nextLine());
    }
    scanner.close();
} catch (FileNotFoundException e) {
    System.out.println("File not found: " + e.getMessage());
}
```

In this example, we're trying to read from a file named `example.txt`. If the file doesn't exist, a `FileNotFoundException` will be thrown. The `catch` block catches this exception and prints a message indicating that the file was not found.

You can use multiple `catch` blocks to handle different types of exceptions. If an exception occurs in the `try` block, Java will search for the first `catch` block that can handle the exception, starting from the top.

```
try {
    // code that might throw exceptions
} catch (IOException e) {
    // handle IOException
} catch (SQLException e) {
    // handle SQLException
}
```

In this example, if an `IOException` occurs in the `try` block, it will be handled by the first `catch` block. If a `SQLException` occurs, it will be handled by the second `catch` block.

You can also catch multiple exceptions in a single `catch` block. This is known as a `multi-catch` block.

```
try {
    // code that might throw exceptions
} catch (IOException | SQLException e) {
```

```
    // handle either IOException or SQLException
}
```

In this example, the `catch` block will handle either an `IOException` or a `SQLException`.

This can make your code more concise, but it should only be used when you want to handle the exceptions in the same manner. If you need to handle the exceptions differently, use separate `catch` blocks.

Also, the multi-catch block must catch two or more unrelated exceptions. Unrelated exceptions do not share a parent-child relationship in the exception hierarchy, like `IOException` and `SQLException` in the previous example.

However, it's important to order your `catch` blocks from the most specific to the most general. This means placing catch blocks that catch subclasses of exceptions before those that catch their superclass exceptions. If you placed the `IOException` catch block before the `FileNotFoundException` catch block, the `FileNotFoundException` catch block would never be reached because `FileNotFoundException` is a subclass of `IOException`. As a result, the `IOException` catch block would catch all exceptions of type `IOException`, including `FileNotFoundException`, and the more specific handling code for `FileNotFoundException` would be bypassed.

For example:

```
try {
    // code that might throw exceptions
} catch (FileNotFoundException e) {
    // handle FileNotFoundException
} catch (IOException e) {
    // handle IOException
}
```

In this example, if a `FileNotFoundException` occurs, it will be caught by the first `catch` block. If a different `IOException` occurs, it will be caught by the second `catch` block. This ensures that specific exceptions are handled appropriately before more general exceptions.

The `finally` Block

The `finally` block is used to execute code that should always run, regardless of whether an exception was thrown or not:

```
try {
    // code that might throw an exception
} catch (ExceptionType e) {
    // handle the exception
} finally {
    // code that always runs
}
```

The `finally` block is often used for cleanup tasks, such as closing files or database connections.

If a `return` statement is executed inside the `try` block, the `finally` block will still execute before the method returns:

```
public static int returnTest() {
    try {
        return 1;
    } catch (Exception e) {
        return 2;
    } finally {
        System.out.println("Finally block");
    }
}
```

In this example, even though we're returning from inside the `try` block, the `finally` block will still execute and print `"Finally block"` before the method returns.

The same is true if the `return` statement is in a `catch` block, the `finally` block will still execute before the method returns.

However, `finally` will not run if you call `System.exit()` in the `try` or `catch` block. `System.exit()` causes the Java Virtual Machine to exit, and the `finally` block will not be executed before the program terminates:

```
try {
    System.out.println("Try block");
    System.exit(0);
} catch (Exception e) {
    System.out.println("Catch block");
} finally {
    System.out.println("Finally block");
}
```

In this example, we call `System.exit(0)` in the `try` block, so it will only print `"Try block"` before the program terminates.

You can use a `try` block with a `finally` block and without any `catch` blocks.

```
try {
    // code that might throw an exception
} finally {
    // code that always runs
}
```

This can be useful when you want to ensure that certain code always runs, even if an exception is thrown, but you don't actually want to handle the exception in this method.

Lastly, it's possible for the `finally` block itself to throw an exception. If this happens, and there was also an exception in the `try` block, the exception from the `finally` block will be the one that's actually thrown.

```
try {
    throw new Exception("Exception in try");
} finally {
    throw new Exception("Exception in finally");
}
```

In this example, the exception thrown in the `finally` block will be the one that's actually thrown by the method. The exception from the `try` block will be suppressed.

Automating Resource Management with the `try-with-resources` Block

Introduced in Java 7, the `try-with-resources` statement is a `try` statement that declares one or more resources. A resource is an object that must be closed after the program is finished with it. The `try-with-resources` statement ensures that each resource is closed at the end of the statement.

The basic syntax of a `try-with-resources` statement is:

```
try (Resource declaration) {
    // use the resource
} catch (ExceptionType e1) {
    // catch block
}
```

For a resource to be used in a `try-with-resources` statement, it must implement the `java.lang.AutoCloseable` interface. This interface has a single method, `close()`, which is called automatically at the end of the `try` block:

```
public interface AutoCloseable {
    void close() throws Exception;
}
```

Alternatively, resources can implement the `java.io.Closeable` interface:

```
public interface Closeable  extends AutoCloseable  {
    void close() throws IOException;
}
```

They both declare a `close()` method, and the only practical difference between these two interfaces is that the `close` method of the `Closeable` interface only throws exceptions of type `IOException`, while the `close` method of the `AutoCloseable` interface throws exceptions of type `Exception` (in other words, it can throw any kind of exception):

However, many of Java's standard resources, such as `Scanner`, `FileReader`, and `DatabaseConnection`, already implement `AutoCloseable`.

Resources can be declared inside the parentheses of the `try` statement, separated by semicolons if there are multiple resources.

```
try (Scanner scanner = new Scanner(new File("example.txt"));
     PrintWriter writer = new PrintWriter(new File("output.txt"))) {
    // use the resources
}
```

The resources are declared so they can be closed without doing it explicitly in a `finally` block. Additionally, the resources declared in the `try-with-resources` statement are only in scope inside the `try` block. They are effectively `final`, meaning you cannot assign a new value to them after they have been initialized.

So, if the resources are declared outside the `try-with-resources` statement, they must be final:

```
final Scanner scanner = new Scanner(new File("example.txt"));
final PrintWriter writer = new PrintWriter(new File("output.txt"));

try (scanner; writer) {
    // use the resources
}
```

Or effectively final:

```
Scanner scanner = new Scanner(new File("example.txt"));
PrintWriter writer = new PrintWriter(new File("output.txt"));

// No reassignment after initialization makes them effectively final
try (scanner; writer) {
    // use the resources
}
```

If multiple resources are declared, they have to be separated by a semicolon and they are closed in the reverse order of their declaration. This is important if the resources depend on each other.

```
try (Scanner scanner = new Scanner(new File("example.txt"));
     DatabaseConnection connection = DriverManager.getConnection(DB_URL)) {
    // use the resources
}
```

In this example, connection will be closed before scanner.

As mentioned earlier, the resources declared in a `try-with-resources` statement are effectively `final`. While you don't have to explicitly declare them as `final`, you cannot assign a new value to them after they have

been initialized:

```
try (Scanner scanner = new Scanner(new File("example.txt"))) {
    scanner = new Scanner(new File("other.txt"));  // This will not compile
}
```

However, one thing to be aware of with `try-with-resources` is the possibility of suppressed exceptions.

Suppressed exceptions only occur when both the `try` block and the `close()` method throw exceptions.

If an exception is thrown from the `try` block and another exception is thrown from the automatic `close()` call, the exception from the `close()` call is suppressed. It is added as a suppressed exception to the exception thrown from the `try` block.

```
try (Scanner scanner = new Scanner(new File("example.txt"))) {
    throw new IllegalStateException("Thrown from try");
}
```

If the call to `scanner.close()` also throws an exception, that exception will be added as a suppressed exception to the `IllegalStateException`.

You can retrieve these suppressed exceptions by calling the `Throwable[] java.lang.Throwable.getSuppressed()` method on the exception thrown by the `try` block:

```
try (Scanner scanner = new Scanner(new File("example.txt"))) {
    throw new IllegalStateException("Thrown from try");
} catch (Exception e) {
    System.err.println(e.getMessage());
    Stream.of(e.getSuppressed())
        .forEach(t -> System.err.println(t.getMessage()));
}
```

This is the output (assuming the `close()` method throws an exception):

```
Thrown from try
Close Exception
```

Key Points

- An exception is an abnormal condition that disrupts the normal flow of a program. When an exception occurs, it is said to be *thrown*.
- The purpose of exceptions is to handle errors gracefully and prevent the program from crashing or behaving unexpectedly.
- All exception classes in Java are subtypes of the `java.lang.Exception` class. The two main branches under `Exception` are checked exceptions and unchecked exceptions (also known as runtime exceptions).
- Checked exceptions are exceptional conditions that a well-written application should anticipate and handle, while unchecked exceptions are usually not recoverable.
- An exception can be thrown automatically by the Java runtime or explicitly in code using the `throw` statement.
- Custom exceptions can be created by extending the `Exception` class (for checked exceptions) or `RuntimeException` class (for unchecked exceptions).
- When a method throws an exception, it must declare this in its method signature using the `throws` keyword. Only checked exceptions need to be declared.
- A stack trace provides information about an exception and the state of the program when the exception occurred. It can be used to pinpoint the location in code where a problem occurred.

- Exception classes can be recognized by their name, their place in the exception hierarchy, the exception message, and by consulting the Java API documentation.
- Exceptions are handled using `try-catch` blocks. The `finally` block is used to execute code that should run regardless of whether an exception was thrown.
- The `multi-catch` block allows us to catch two or more unrelated exceptions with a single `catch` block.
- The `finally` block is always executed, even when an exception is caught or when either the `try` or `catch` block contains a `return` statement. However, the `finally` block will not execute if the JVM exits during the `try` or `catch` block, such as by calling `System.exit()`.
- The `try-with-resources` statement, introduced in Java 7, ensures that resources are properly closed after use. Resources used in a `try-with-resources` statement must implement the `AutoCloseable` or `Closeable` interface.

Practice Questions

1. Which of the following statements correctly describes a checked exception in Java?

A. A checked exception is a type of exception that inherits from the `java.lang.RuntimeException` class.
B. A checked exception must be either caught or declared in the method signature using the `throws` keyword.
C. A checked exception is an error that is typically caused by the environment in which the application is running, and it cannot be handled by the application.
D. A checked exception can be thrown by the Java Virtual Machine when a severe error occurs, such as an out-of-memory error.

2. Which of the following code snippets correctly defines and throws a custom checked exception?

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

public class TestCustomException {
    public static void main(String[] args) {
        try {
            methodThatThrowsException();
        } catch (CustomException e) {
            System.out.println(e.getMessage());
        }
    }

    public static void methodThatThrowsException() throws CustomException {
        throw new CustomException("This is a custom checked exception");
    }
}
```

A. This code defines a custom checked exception and correctly throws and handles it.
B. This code defines a custom unchecked exception.
C. This code will not compile because the custom exception is not declared correctly in the method signature.
D. This code will compile but will not throw the custom exception at runtime.

3. Given the following class, what is the result?

```
public class Main {
    protected static int myMethod() {
```

```
        try {
            throw new RuntimeException();
        } catch(RuntimeException e) {
             return 1;
        } finally {
             return 2;
        }
    }
    public static void main(String[] args) {
        System.out.println(myMethod());
    }
}
```

A. `1`
B. `2`
C. Compilation fails
D. An exception occurs at runtime

4. Given the following class, which of the following statement is true?

```
public class Main {
    public static void main(String[] args) {
        try {
            // Do nothing
        } finally {
            // Do nothing
        }
    }
}
```

A. The code doesn't compile correctly.
B. The code would compile correctly if we add a `catch` block.
C. The code would compile correctly if we remove the `finally` block.
D. The code compiles correctly as it is.

5. Which of the following statements are true? (Choose all that apply)

A. In a `try-with-resources`, the `catch` block is required.
B. The `throws` keyword is used to throw an exception.
C. In a `try-with-resources` block, if you declare more than one resource, they have to be separated by a semicolon.
D. If a `catch` block is defined for an exception that couldn't be thrown by the code in the `try` block, a compile-time error is generated.

6. Given the following class, what is the result?:

```
class Connection implements java.io.Closeable {
    public void close() throws IOException {
        throw new IOException("Close Exception");
    }
}

public class Main {
    public static void main(String[] args) {
        try (Connection c = new Connection()) {
            throw new RuntimeException("RuntimeException");
        } catch (IOException e) {
            System.err.println(e.getMessage());
```

```
        }
    }
}
```

A. `Close Exception`
B. `RuntimeException`
C. `RuntimeException` and then `CloseException`
D. Compilation fails
E. The stack trace of an uncaught exception is printed

7. Which of the following exceptions are direct subclasses of `RuntimeException`?

A. `java.io.FileNotFoundException`
B. `java.lang.ArithmeticException`
C. `java.lang.ClassCastException`
D. `java.lang.InterruptedException`

8. Given the following code, what is the result?

```
class MyResource implements AutoCloseable {
    public void close() {
        throw new RuntimeException("Close Exception");
    }
}

public class Main {
    public static void main(String[] args) {
        try (MyResource resource = new MyResource()) {
            throw new RuntimeException("Try Block Exception");
        } catch (RuntimeException e) {
            Throwable[] suppressed = e.getSuppressed();
            if (suppressed.length > 0) {
                for (Throwable t : suppressed) {
                    System.out.println("Suppressed: " + t.getMessage());
                }
            } else {
                System.out.println(e.getMessage());
            }
        }
    }
}
```

A. Only `"Try Block Exception"` is printed.
B. Only `"Close Exception"` is printed.
C. Both `"Try Block Exception"` and `"Close Exception"` are printed.
D. `"Suppressed: Close Exception"` is printed.

Chapter SEVEN

Error Handling and Exceptions

Answers

1. The correct answer is B.

Explanation:

- **A.** A checked exception is a type of exception that inherits from the `java.lang.RuntimeException` class.

  - This option is incorrect. A checked exception does not inherit from `java.lang.RuntimeException`. Checked exceptions are subclasses of `java.lang.Exception` but not of `java.lang.RuntimeException`.
- **B.** A checked exception must be either caught or declared in the method signature using the `throws` keyword.
  - This option is correct. Checked exceptions must be either caught using a `try-catch` block or declared in the method signature with the `throws` keyword. This is to ensure that the exception is properly handled at some point in the code.
- **C.** A checked exception is an error that is typically caused by the environment in which the application is running, and it cannot be handled by the application.
  - This option is incorrect. It describes errors more accurately than checked exceptions. Errors are typically caused by the environment and are not expected to be handled by the application.
- **D.** A checked exception can be thrown by the Java Virtual Machine when a severe error occurs, such as an out-of-memory error.
  - This option is incorrect. It describes errors rather than checked exceptions. Errors like out-of-memory errors are thrown by the JVM and are not meant to be caught or handled by applications in most cases.

2. The correct answer is A.

Explanation:

- **A.** This code defines a custom checked exception and correctly throws and handles it.
  - This option is correct. The code defines a custom checked exception by extending `Exception`. The `methodThatThrowsException` method throws this custom exception, which is then caught and handled in the `main` method.
- **B.** This code defines a custom unchecked exception.
  - This option is incorrect. The code extends `Exception`, not `RuntimeException`, making it a checked exception rather than an unchecked one.
- **C.** This code will not compile because the custom exception is not declared correctly in the method signature.
  - This option is incorrect. The custom exception is correctly declared in the method signature of `methodThatThrowsException`, so it will compile without issues.
- **D.** This code will compile but will not throw the custom exception at runtime.
  - This option is incorrect. The code will throw the custom exception at runtime as expected, and it will be caught and handled in the `catch` block.

3. The correct answer is B.

Explanation:

- **A.** `1`
  - This option is incorrect. Although the `catch` block returns `1`, the `finally` block will override this return value with `2`.
- **B.** `2`
  - This option is correct. The `finally` block always executes and its return value overrides the return value from the `catch` block, resulting in `2` being printed.
- **C.** Compilation fails
  - This option is incorrect. The code compiles without any errors.
- **D.** An exception occurs at runtime
  - This option is incorrect. While a `RuntimeException` is thrown in the `try` block, it is caught by the `catch` block, and no exception propagates to cause a runtime error.

4. The correct answer is D.

Explanation:

- **A.** The code doesn't compile correctly.

  - This option is incorrect. The code does compile correctly. A `try` block can be followed by a `finally` block without a `catch` block.
- **B.** The code would compile correctly if we add a `catch` block.
  - This option is incorrect. While adding a `catch` block is valid, it is not necessary for the code to compile. The `try` block can be used with only a `finally` block.
- **C.** The code would compile correctly if we remove the `finally` block.
  - This option is incorrect. Removing the `finally` block is not necessary for the code to compile. The code is valid with the `finally` block present.
- **D.** The code compiles correctly as it is.
  - This option is correct. The code compiles correctly as it is. A `try` block must be followed by either a `catch` block, a `finally` block, or both.

5. The correct answers are C and D.

Explanation:

- **A.** In a `try-with-resources`, the `catch` block is required.
  - This option is incorrect. In a `try-with-resources` statement, the catch block is optional. The primary purpose of `try-with-resources` is to ensure that each resource is closed at the end of the statement, whether an exception is thrown or not.
- **B.** The `throws` keyword is used to throw an exception.
  - This option is incorrect. The `throws` keyword is used in method declarations to specify that the method can throw an exception, not to throw an exception. The `throw` keyword is used to actually throw an exception.
- **C.** In a `try-with-resources` block, if you declare more than one resource, they have to be separated by a semicolon.
  - This option is correct. In a `try-with-resources` block, if you declare more than one resource, they must be separated by a semicolon.
- **D.** If a `catch` block is defined for an exception that couldn't be thrown by the code in the `try` block, a compile-time error is generated.
  - This option is correct. If a `catch` block is defined for an exception that cannot be thrown by the code in the `try` block, the compiler will generate an error because the `catch` block is unreachable.

6. The correct answer is E.

Explanation:

- **A.** `Close Exception`
  - This option is incorrect. While the `IOException` from `close()` will occur, it will be suppressed by the `RuntimeException`.
- **B.** `RuntimeException`
  - This option is incorrect. The primary exception is `RuntimeException`, but it will not print its message directly because the catch block does not handle it.
- **C.** `RuntimeException` and then `CloseException`
  - This option is incorrect. Although both exceptions occur, the `RuntimeException` is primary, and the `IOException` is suppressed. Both messages are not printed in sequence.
- **D.** Compilation fails
  - This option is incorrect. The code compiles without any errors.
- **E.** The stack trace of an uncaught exception is printed.
  - This option is correct. The `RuntimeException` thrown in the try block is not caught by the `catch (IOException e)` block. Hence, the stack trace of the `RuntimeException` is printed.

7. The correct answers are B and C.

Explanation:

- **A.** `java.io.FileNotFoundException` is incorrect. It is a subclass of `java.io.IOException`, which in turn is a subclass of `java.lang.Exception`, making it a checked exception.

- **B.** `java.lang.ArithmeticException` is correct. It is a direct subclass of `java.lang.RuntimeException` and represents arithmetic errors such as division by zero.
- **C.** `java.lang.ClassCastException` is correct. It is a direct subclass of `java.lang.RuntimeException` and indicates an invalid cast operation.
- **D.** `java.lang.InterruptedException` is incorrect. It is a direct subclass of `java.lang.Exception`, making it a checked exception. It indicates that a thread has been interrupted.

8. The correct answer is D.

Explanation:

- **A.** Only `"Try Block Exception"` is printed.
  - This option is incorrect. The `Try Block Exception` is the main exception and is not directly printed because the `catch` block checks for suppressed exceptions first.
- **B.** Only `"Close Exception"` is printed.
  - This option is incorrect. The `Close Exception` is not directly printed; it is suppressed and accessed via the `getSuppressed` method.
- **C.** Both `"Try Block Exception"` and `"Close Exception"` are printed.
  - This option is incorrect. The code only prints suppressed exceptions, not the main exception message directly.
- **D.** `"Suppressed: Close Exception"` is printed.
  - This option is correct. The `RuntimeException` thrown in the `try` block is the main exception, and the `RuntimeException` from the `close` method is suppressed. The `catch` block prints the suppressed exception message, `"Suppressed: Close Exception"`.

Chapter EIGHT

Functional Interfaces and Lambda Expressions

Chapter Content

- Functional Interfaces
  - The `@FunctionalInterface` Annotation
  - Rules for Defining a Functional Interface
- Lambda Expressions
  - Syntax of a Lambda Expression
  - Lambda Expressions and Anonymous Classes
- Java Built-In Lambda Interfaces
  - `Predicate`
  - `Consumer`
  - `Function`
  - `Supplier`
  - `UnaryOperator`
  - `BiPredicate`
  - `BiConsumer`
  - `BiFunction`
  - `BinaryOperator`
  - Primitive-specific Functional Interfaces
- Method References
  - Static Methods
  - Instance Method of An Object of A Particular Type
  - Instance Method of An Existing Object
  - Constructor
- Key Points

- Practice Questions

Functional Interfaces

Java 8 brought us lambda expressions, a new feature that aims to simplify development by taking a more functional programming approach. But for this to work, Java also introduced the concept of functional interfaces.

A functional interface is an interface that contains only one abstract method. They may contain one or more default methods or static methods, but there can be only one abstract method.

At first glance, you might think that using functional interfaces is not much different than using regular classes and objects. After all, we've been able to define interfaces with a single method for a long time. But the key difference is how they enable the use of lambda expressions.

Lambda expressions let you treat functionality as method arguments, or code as data. Instead of defining a class that implements a single-method interface, you can directly pass a lambda expression as an instance of a functional interface, allowing for cleaner and more concise code.

```
public interface MyInterface {
    public void myMethod();
}

MyInterface ref = () -> System.out.println("Hello World!");
```

In this example, the lambda expression `() -> System.out.println("Hello World!")` is treated as an instance of the `MyInterface` functional interface. We're assigning a block of code to the variable `ref`.

The `@FunctionalInterface` Annotation

Java 8 also introduced the `@FunctionalInterface` annotation, which is used to indicate that an interface is intended to be a functional interface. It's a kind of *hint* to the compiler that you intend for this interface to adhere to the rules of a functional interface:

```
@FunctionalInterface
public interface MyInterface {
    void myMethod();
}
```

However, the `@FunctionalInterface` annotation is not required. If an interface meets the criteria of a functional interface (it has only one abstract method), it's a functional interface whether or not it has the `@FunctionalInterface` annotation.

So why use it?

There are a couple of reasons:

1. It makes your intent clear. By using `@FunctionalInterface`, you're signaling to other developers (and to your future self) that this interface is meant to be used with lambda expressions.
2. It enables compiler checks. If you annotate an interface with `@FunctionalInterface` and then try to add a second abstract method to it, the compiler will throw an error. This can help prevent accidental violations of the functional interface contract.

```
@FunctionalInterface
public interface MyInterface {
    void myMethod();
    void myOtherMethod();  // This will cause a compiler error
}
```

However, the annotation does not become part of the generated bytecode. It's purely for compile-time checks and for developer clarity.

Also, note that if an interface is annotated with `@FunctionalInterface`, but does not actually meet the criteria (for example, if it has no abstract methods at all), the compiler will raise an error:

```
@FunctionalInterface
public interface NonFunctionalInterface {
    // No abstract methods
}  // This will cause a compiler error
```

Rules for Defining a Functional Interface

Functional interfaces do not limit what you can do. You can still define as many default and static methods on the interface as you'd like.

Default methods enable you to add new functionality to the interfaces of your libraries and ensure binary compatibility with code written for older versions of those interfaces. Static methods in interfaces are used for providing utility methods, like `null`-checking for example.

```
interface MyInterface {
    void abstractMethod(int x);
    default void defaultMethod() { }
    static void staticMethod() { }
}
```

Only the `abstractMethod` counts toward the single abstract method test for a functional interface.

It's also important to note that if an interface declares an abstract method overriding one of the public methods of `java.lang.Object`, that also does not count toward the interface's abstract method count since any implementation of the interface will have an implementation from `java.lang.Object` or elsewhere. For example:

```
interface MyInterface {
    boolean equals(Object obj);
    // Other methods
}
```

In this case, `MyInterface` is still a functional interface since `equals` is a public method in `Object`.

Using lambda expressions with functional interfaces is simply a new option in our coding toolbox. You can still use anonymous inner classes or implement the interface the old-fashioned way:

```
MyInterface ref = new MyInterface() {
    @Override
    public void myMethod() {
        System.out.println("Hello World!");
    }
};

// Implementing the interface in a separate class
class MyClass implements MyInterface {
    @Override
    public void myMethod() {
        System.out.println("Hello World!");
    }
}
MyInterface ref = new MyClass();
```

Also, a class or lambda expression can implement multiple functional interfaces if they're compatible. For example, if two interfaces have identical abstract methods, they're effectively the same functional interface:

```
@FunctionalInterface
interface Interface1 {
    void method();
}

@FunctionalInterface
interface Interface2 {
    void method();
}

// Implementing multiple compatible interfaces in a class
class MyClass implements Interface1, Interface2 {
    @Override
    public void method() {
        System.out.println("Hello World!");
    }
}

// Using a lambda expression
Interface1 ref1 = () -> System.out.println("Hello World!");
Interface2 ref2 = () -> System.out.println("Hello World!");
```

And if the built-in functional interfaces like `Runnable` or `Comparator` don't meet your needs, you can easily define your own. Just remember the single abstract method rule.

Lambda Expressions

Lambda expressions allow you to treat functionality as a method argument or code as data, enabling a more functional programming style. For example, they enable you to write code like this:

```
List<Car> compactCars = findCars(cars,
     (Car c) ->
        c.getType().equals(CarTypes.COMPACT)
);
```

Instead of:

```
List<Car> compactCars = findCars(cars,
     new Searchable() {
        public boolean test(Car car) {
           return car.getType().equals(
                     CarTypes.COMPACT);
        }
});
```

In essence, a lambda expression is a concise way to represent a function. The term lambda expression comes from lambda calculus, written as -calculus, where is the Greek letter lambda. This form of calculus deals with defining and applying functions.

Functional interfaces are the foundation upon which lambda expressions are built. For example, consider the following functional interface:

```
@FunctionalInterface
interface MyFunction {
```

```
    int apply(int a);
}
```

You can use a lambda expression wherever an instance of this interface is expected:

```
MyFunction doubler = (int a) -> a * 2;
```

The lambda expression `a -> a * 2` conforms to the signature of the `apply` method in `MyFunction`.

Syntax of a Lambda Expression

A lambda expression has three parts: a list of parameters, an arrow token (->), and a function body.

Here's the basic syntax:

```
(parameters) -> expression
// or
(parameters) -> { statements; }
```

For example, consider this functional interface:

```
@FunctionalInterface
interface MyFunction {
    int apply(int a, int b);
}
```

And this lambda expression that takes two integers and returns their sum:

```
MyFunction f = (int a, int b) -> a + b
```

You can use the `var` keyword in the parameter list of a lambda expression. This allows the type of the parameter to be inferred by the compiler:

```
MyFunction f = (var a, var b) -> a + b
```

You can omit the parameter types, the compiler can also infer them from the context:

```
MyFunction f = (a, b) -> a + b
```

If the lambda expression only takes one parameter, you can even omit the parentheses:

```
@FunctionalInterface
interface MyFunction {
    int apply(int a);
}

//...

MyFunction f = a -> a * 2
```

You can also use the `var` keyword to declare a variable without specifying its type only when the compiler can infer the type from the context.

For example, taking into account the `MyFunction` interface of the previous example and just the following expression:

```
var f = a -> a * 2;
```

You'd get a compile-time error with the following message: "Cannot infer type: lambda expression requires an explicit target type."

You cannot use `var` directly with a lambda expression like `var f = (var a) -> a * 2;` because the lambda needs a target type that `var` cannot provide.

However, in this case:

```
MyInterface f = (a) -> a * 2; // Lambda assigned to functional interface
var fVar = f; // `var` infers type MyInterface
System.out.println(fVar.apply(5)); // Outputs 10
```

You can use `var` because you are assigning a lambda to a previously defined functional interface, where the type can be inferred from the context.

The contexts where the target type (the functional interface) of a lambda expression can be inferred include: - A variable declaration - An assignment - A return statement - An array initializer - Method or constructor arguments - A ternary conditional expression - A cast expression

If you understand the concept, you don't need to memorize this list.

Lambda Expressions and Anonymous Classes

Prior to Java 8, anonymous classes were the primary way to represent a one-off piece of functionality. With the introduction of lambda expressions in Java 8, we now have a more concise way to write certain types of anonymous classes.

Consider this anonymous class:

```
Runnable r1 = new Runnable() {
    public void run() {
        System.out.println("Hello!");
    }
};
```

This can be replaced with a lambda expression:

```
Runnable r2 = () -> System.out.println("Hello!");
```

However, while lambda expressions and anonymous classes share some similarities, they also have significant differences:

Similarities:

- Local variables can only be used if they are declared final or are effectively final.
- You can access instance or static variables of the enclosing class.
- They must not throw more checked exceptions than specified in the throws clause of the functional interface method.

Differences:

- In an anonymous class, `this` refers to the instance of the anonymous class itself. In a lambda expression, `this` refers to the enclosing class instance.
- Default methods of a functional interface cannot be accessed from within lambda expressions, but they can be accessed from anonymous classes.
- Lambda expressions allow you to omit the types of the parameters in the parameter list, which is not possible with anonymous classes.
- If you reference an instance variable inside a lambda expression, you're referencing the variable from the enclosing instance. In an anonymous class, you would be referencing a separate copy of the variable.

Here's an example about using local variables inside the body of a lambda:

```
public class LambdaExample {
    public void testLambda() {
        int localVariable = 10;
        Runnable r = () -> {
            System.out.println("Lambda: " + localVariable);
        };
        r.run();
```

```
    }

    public void testAnonymous() {
        int localVariable = 10;
        Runnable r = new Runnable() {
            public void run() {
                System.out.println("Anonymous: " + localVariable);
            }
        };
        r.run();
    }

    public static void main(String[] args) {
        LambdaExample example = new LambdaExample();
        example.testLambda();
        example.testAnonymous();
    }
}
```

This is the output:

```
Lambda: 10
Anonymous: 10
```

In this example, both the lambda expression and the anonymous class are able to access the `localVariable` defined in their respective methods. However, if we try to modify the `localVariable` after it has been used in the lambda expression or anonymous class, we will get a compilation error:

```
public void testLambda() {
    int localVariable = 10;
    Runnable r = () -> {
        System.out.println("Lambda: " + localVariable); // Compilation error
    };
    localVariable = 20;  // Because of this
    r.run();
}
```

This is because the `localVariable` must be effectively final (its value doesn't change after initialization) in order to be used inside the lambda expression or anonymous class.

Local variables have to be final because of the way they are implemented in Java. Instance variables are stored on the heap, while local variables live on the stack. Variables on the heap are shared across threads, but variables on the stack are confined to the thread they're in.

When you create an instance of an anonymous inner class or a lambda expression, the values of local variables are copied. This prevents thread-related problems and ensures that you are working with a consistent value, as the variable cannot be modified after initialization.

By requiring final (or effectively final) variables, Java ensures thread safety and consistency, as the value cannot be changed, eliminating visibility issues and potential thread problems.

Java Built-In Lambda Interfaces

In the previous section, we used functional interfaces like the following:

```
@FunctionalInterface
interface MyFunction {
    int apply(int a, int b);
}
```

However, you don't have to write an interface like that in each program that uses it (or link a library that contains it). An interface that does the same but accepts any object type already exists in the language.

Java provides functional interfaces for common use cases in the `java.util.function` package.

These are the main five: - `Predicate<T>` - `Consumer<T>` - `Function<T, R>` - `Supplier<T>` - `UnaryOperator<T>`

Where `T` and `R` represent generic types (`T` represents a parameter type and `R` the return type).

They also have specializations for cases where the input parameter is a primitive type (specifically for `int`, `long`, `double`, and `boolean` for `Supplier`), for example: - `IntPredicate` - `LongConsumer` - `BooleanSupplier`

Where the name is preceded by the appropriate primitive type.

Additionally, four of them have binary versions, which means they take two parameters instead of one: - `BiPredicate<L, R>` - `BiConsumer<T, U>` - `BiFunction<T, U, R>` - `BinaryOperator<T>`

Where `T`, `U`, and `R` represent generic types (`T` and `U` represent parameter types and `R` the return type).

The following tables show the complete list of interfaces. You don't have to memorize them, just try to understand them.

Functional Interface	Primitive Versions
`Predicate<T>`	`IntPredicate` `LongPredicate` `DoublePredicate`
`Consumer<T>`	`IntConsumer` `LongConsumer` `DoubleConsumer`
`Function<T, R>`	`IntFunction<R>` `IntToDoubleFunction` `IntToLongFunction` `LongFunction<R>` `LongToDoubleFunction` `LongToIntFunction` `DoubleFunction<R>` `DoubleToIntFunction` `DoubleToLongFunction` `ToIntFunction<T>` `ToDoubleFunction<T>` `ToLongFunction<T>`
`Supplier<T>`	`BooleanSupplier` `IntSupplier` `LongSupplier` `DoubleSupplier`
`UnaryOperator<T>`	`IntUnaryOperator` `LongUnaryOperator` `DoubleUnaryOperator`

Functional Interface	Primitive Versions
`BiPredicate<L, R>`	
`BiConsumer<T, U>`	`ObjIntConsumer<T>` `ObjLongConsumer<T>` `ObjDoubleConsumer<T>`
`BiFunction<T, U, R>`	`ToIntBiFunction<T, U>` `ToLongBiFunction<T, U>` `ToDoubleBiFunction<T, U>`
`BinaryOperator<T>`	`IntBinaryOperator` `LongBinaryOperator` `DoubleBinaryOperator`

Predicate

A predicate is a statement that may be `true` or `false` depending on the values of its variables.

This functional interface can be used anywhere you need to evaluate a `boolean` condition.

This is how the interface is defined:

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
    // Other default and static methods
```

```
    // ...
}
```

The functional descriptor (method signature) is:

```
Predicate<T>
```

Here's an example using an anonymous class:

```
Predicate<String> startsWithA = new Predicate<String>() {
    @Override
    public boolean test(String t) {
        return t.startsWith("A");
    }
};
boolean result = startsWithA.test("Arthur");
```

And with a lambda expression:

```
Predicate<String> startsWithA = t -> t.startsWith("A");
boolean result = startsWithA.test("Arthur");
```

This interface also has the following default methods:

```
default Predicate<T> and(Predicate<? super T> other)
default Predicate<T> or(Predicate<? super T> other)
default Predicate<T> negate()
```

These methods return a composed `Predicate` that represents a short-circuiting logical **AND** and **OR** of this predicate and another and its logical negation.

Short-circuiting means that the other predicate won't be evaluated if the value of the first predicate can predict the result of the operation (if the first predicate returns false in the case of **AND** or if it returns true in the case of **OR**).

These methods are useful to combine predicates and make the code more readable, for example:

```
Predicate<String> startsWithA = t -> t.startsWith("A");
Predicate<String> endsWithA = t -> t.endsWith("A");
boolean result = startsWithA.and(endsWithA).test("Hi");
```

Also, there's a `static` method:

```
static <T> Predicate<T> isEqual(Object targetRef)
```

That returns a `Predicate` that tests if two arguments are equal according to `Objects.equals(Object, Object)`.

There are also primitive versions for `int`, `long`, and `double`. They don't extend from `Predicate`.

For example, here's the definition of `IntPredicate`:

```
@FunctionalInterface
public interface IntPredicate {
    boolean test(int value);
    // And the default methods: and, or, negate
}
```

So instead of using:

```
Predicate<Integer> even = t -> t % 2 == 0;
boolean result = even.test(5);
```

You can use:

```
IntPredicate even = t -> t % 2 == 0;
boolean result = even.test(5);
```

Why?

Just to avoid the conversion from `Integer` to `int` and work directly with primitive types.

Notice that these primitive versions don't have a generic type. Due to the way generics are implemented, parameters of the functional interfaces can be bound only to object types.

Since the conversion from the wrapper type (`Integer`) to the primitive type (`int`) uses more memory and comes with a performance cost, Java provides these versions to avoid autoboxing operations when inputs or outputs are primitives.

Here's the corrected text:

Consumer

`Consumer` represents an operation that accepts a single input argument and returns no result, it just executes some operations on the argument.

This is how the interface is defined:

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
    // And a default method
    // ...
}
```

The functional descriptor (method signature) is:

```
T -> void
```

Here's an example using an anonymous class:

```
Consumer<String> consumeStr = new Consumer<String>() {
    @Override
    public void accept(String t) {
        System.out.println(t);
    }
};
consumeStr.accept("Hi");
```

And with a lambda expression:

```
Consumer<String> consumeStr = t -> System.out.println(t);
consumeStr.accept("Hi");
```

This interface also has the following default method:

```
default Consumer<T> andThen(Consumer<? super T> after)
```

This method returns a composed `Consumer` that performs, in sequence, the operation of the consumer followed by the operation of the parameter.

These methods are useful to combine `Consumer`s and make the code more readable, for example:

```
Consumer<String> first = t ->
    System.out.println("First:" + t);
Consumer<String> second = t ->
    System.out.println("Second:" + t);
first.andThen(second).accept("Hi");
```

The output is:

```
First: Hi
Second: Hi
```

Look how both `Consumers` take the same argument and the order of execution.

There are also primitive versions for `int`, `long`, and `double`. They don't extend from `Consumer`.

For example, here's the definition of `IntConsumer`:

```
@FunctionalInterface
public interface IntConsumer {
    void accept(int value);
    default IntConsumer andThen(IntConsumer after) {
        // ...
    }
}
```

So instead of using:

```
int[] a = { 1,2,3,4,5,6,7,8 };
printList(a, t -> System.out.println(t));
//...
void printList(int[] a, Consumer<Integer> c) {
    for(int i : a) {
        c.accept(i);
    }
}
```

You can use:

```
int[] a = { 1,2,3,4,5,6,7,8 };
printList(a, (IntConsumer) t -> System.out.println(t));
//...
void printList(int[] a, IntConsumer c) {
    for(int i : a) {
        c.accept(i);
    }
}
```

Function

`Function` represents an operation that takes an input argument of a certain type and produces a result of another type.

A common use is to convert or transform from one object to another.

This is how the interface is defined:

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
    // Other default and static methods
    // ...
}
```

The functional descriptor (method signature) is:

```
T -> R
```

Assuming a method:

```
void round(double d, Function<Double, Long> f) {
    long result = f.apply(d);
    System.out.println(result);
}
```

Here's an example using an anonymous class:

```
round(5.4, new Function<Double, Long>() {
    @Override
    public Long apply(Double d) {
        return Math.round(d);
    }
});
```

And with a lambda expression:

```
round(5.4, d -> Math.round(d));
```

This interface also has the following default methods:

```
default <V> Function<V,R> compose(
    Function<? super V,? extends T> before)
default <V> Function<T,V> andThen(
    Function<? super R,? extends V> after)
```

The difference between these methods is that `compose` applies the function represented by the parameter first, and its result serves as the input to the other function. `andThen` first applies the function that calls the method, and its result acts as the input of the function represented by the parameter.

For example:

```
Function<String, String> f1 = s -> s.toUpperCase();
Function<String, String> f2 = s -> s.toLowerCase();
System.out.println(f1.compose(f2).apply("Compose"));
System.out.println(f1.andThen(f2).apply("AndThen"));
```

The output is:

```
COMPOSE
andthen
```

In the first case, `f2` is the first function to be applied. In the second case, `f2` is the last function to be applied.

Also, there's a `static` method:

```
static <T> Function<T, T> identity()
```

That returns a function that always returns its input argument.

In the case of primitive versions, they also apply to `int`, `long`, and `double`, but there are more combinations than the previous interfaces:

- To indicate that the function returns a generic type and takes a primitive argument, the interface is named **XXXFunction**, for example, `IntFunction`: `java @FunctionalInterface public interface IntFunction<R> { R apply(int value); }`
- To indicate that the function returns a primitive type and takes a generic argument, the interface is named **ToXXXFunction**, for example, `ToIntFunction`: `java @FunctionalInterface public interface ToIntFunction<T> { int applyAsInt(T value); }`
- To indicate that the function takes a primitive argument and returns another primitive type, the interface is named **XXXToYYYFunction**, where **XXX** is the argument type and **YYY** is the

return type, for example, `IntToDoubleFunction`: `java @FunctionalInterface public interface IntToDoubleFunction { double applyAsDouble(int value); }`

Remember that these interfaces are for convenience, to work directly with primitives, for example:

`DoubleFunction<R>` instead of `Function<Double, R>`
`ToLongFunction<T>` instead of `Function<T, Long>`
`IntToLongFunction` instead of `Function<Integer, Long>`

Supplier

`Supplier` is the opposite of `Consumer`. It takes no arguments and only returns some value.

This is how the interface is defined:

```
@FunctionalInterface
public interface Supplier<T> {
    T get();
}
```

The functional descriptor (method signature) is:

```
() -> T
```

Here's an example using an anonymous class:

```
String t = "One";
Supplier<String> supplierStr = new Supplier<String>() {
    @Override
    public String get() {
        return t.toUpperCase();
    }
};
System.out.println(supplierStr.get());
```

And with a lambda expression:

```
String t = "One";
Supplier<String> supplierStr = () -> t.toUpperCase();
System.out.println(supplierStr.get());
```

This interface doesn't define default methods.

There are also primitive versions for `int`, `long`, `double`, and `boolean`, but they don't extend from `Supplier`.

For example, here's the definition of `BooleanSupplier`:

```
@FunctionalInterface
public interface BooleanSupplier {
    boolean getAsBoolean();
}
```

These primitive versions are used instead of `Supplier` for their respective types.

UnaryOperator

`UnaryOperator` is just a specialization of the `Function` interface (in fact, this interface extends from it) for when the argument and the result are of the same type.

This is how the interface is defined:

```
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T, T> {
```

```
    // Just the identity
    // method is defined
}
```

The functional descriptor (method signature) is:

```
T -> T
```

Here's an example using an anonymous class:

```
UnaryOperator<String> uOp = new UnaryOperator<String>() {
    @Override
    public String apply(String t) {
        return t.substring(0,2);
    }
};
System.out.println(uOp.apply("Hello"));
```

And with a lambda expression:

```
UnaryOperator<String> uOp = t -> t.substring(0,2);
System.out.println(uOp.apply("Hello"));
```

This interface inherits the default methods of the `Function` interface:

```
default <V> Function<V, T> compose(
    Function<? super V, ? extends T> before)
default <V> Function<T, V> andThen(
    Function<? super T, ? extends V> after)
```

And just defines the `static` method `identity()` for this interface (since `static` methods are not inherited):

```
static <T> UnaryOperator<T> identity()
```

That returns a `UnaryOperator` that always returns its input argument.

There are also primitive versions for `int`, `long`, and `double`. They don't extend from `UnaryOperator`.

For example, here's the definition of `IntUnaryOperator`:

```
@FunctionalInterface
public interface IntUnaryOperator {
    int applyAsInt(int operand);
    // Definitions for compose, andThen, and identity
}
```

So instead of using:

```
int[] a = {1,2,3,4,5,6,7,8};
int sum = sumNumbers(a, t -> t * 2);
//...
int sumNumbers(int[] a, UnaryOperator<Integer> unary) {
    int sum = 0;
    for(int i : a) {
        sum += unary.apply(i);
    }
    return sum;
}
```

You can use:

```
int[] a = {1,2,3,4,5,6,7,8};
int sum = sumNumbers(a, t -> t * 2);
```

```
//...
int sumNumbers(int[] a, IntUnaryOperator unary) {
    int sum = 0;
    for(int i : a) {
        sum += unary.applyAsInt(i);
    }
    return sum;
}
```

BiPredicate

This interface represents a predicate that takes two arguments.

It is defined as follows:

```
@FunctionalInterface
public interface BiPredicate<T, U> {
    boolean test(T t, U u);
    // Default methods are also defined
}
```

The functional descriptor (method signature) is:

```
(T, U) -> boolean
```

Here's an example using an anonymous class:

```
BiPredicate<Integer, Integer> divisible =
    new BiPredicate<Integer, Integer>() {
        @Override
        public boolean test(Integer t, Integer u) {
            return t % u == 0;
        }
    };
boolean result = divisible.test(10, 5);
```

And with a lambda expression:

```
BiPredicate<Integer, Integer> divisible =
    (t, u) -> t % u == 0;
boolean result = divisible.test(10, 5);
```

This interface defines the same default methods as the `Predicate` interface, but with two arguments:

```
default BiPredicate<T, U> and(
    BiPredicate<? super T, ? super U> other) {
    return (t, u) -> test(t, u) && other.test(t, u);
}

default BiPredicate<T, U> or(
    BiPredicate<? super T, ? super U> other) {
    return (t, u) -> test(t, u) || other.test(t, u);
}

default BiPredicate<T, U> negate() {
    return (t, u) -> !test(t, u);
}
```

This interface doesn't have primitive versions.

BiConsumer

This interface represents a consumer that takes two arguments (and doesn't return a result).

This is how it is defined:

```
@FunctionalInterface
public interface BiConsumer<T, U> {
    void accept(T t, U u);
    // andThen default method is defined
}
```

The functional descriptor (method signature) is:

```
(T, U) -> void
```

Here's an example using an anonymous class:

```
BiConsumer<String, String> consumeStr =
    new BiConsumer<String, String>() {
        @Override
        public void accept(String t, String u) {
            System.out.println(t + " " + u);
        }
    };
consumeStr.accept("Hi", "there");
```

And with a lambda expression:

```
BiConsumer<String, String> consumeStr =
    (t, u) -> System.out.println(t + " " + u);
consumeStr.accept("Hi", "there");
```

This interface also has the following default method:

```
default BiConsumer<T, U> andThen(
    BiConsumer<? super T, ? super U> after)
```

This method returns a composed `BiConsumer` that performs, in sequence, the operation of the consumer followed by the operation of the parameter. It will throw `NullPointerException` if the `after` parameter is `null`.

As in the case of a `Consumer`, these methods are useful to combine `BiConsumers` and make the code more readable, for example:

```
BiConsumer<String, String> first = (t, u) -> System.out.println(t.toUpperCase() + u.toUpperCase());
BiConsumer<String, String> second = (t, u) -> System.out.println(t.toLowerCase() + u.toLowerCase());
first.andThen(second).accept("Again", " and again");
```

The output is:

```
AGAIN AND AGAIN
again and again
```

There are also primitive specialization versions for `int`, `long`, and `double`. They don't extend from `BiConsumer`, and instead of taking two `ints`, for example, they take one object and a primitive value as a second argument. So the naming convention changes to **ObjXXXConsumer**, where **XXX** is the primitive type. For example, here's the definition of `ObjIntConsumer`:

```
@FunctionalInterface
public interface ObjIntConsumer<T> {
    void accept(T t, int value);
}
```

So instead of using:

```
int[] a = {1,2,3,4,5,6,7,8};
printList(a, (t, i) -> System.out.println(t + i));
//...
void printList(int[] a, BiConsumer<String, Integer> c) {
    for(int i : a) {
        c.accept("Number:", i);
    }
}
```

You can use:

```
int[] a = {1,2,3,4,5,6,7,8};
printList(a, (t, i) -> System.out.println(t + i));
//...
void printList(int[] a, ObjIntConsumer<String> c) {
    for(int i : a) {
        c.accept("Number:", i);
    }
}
```

BiFunction

This interface represents a function that takes two arguments of different types and produces a result of another type.

This is how it is defined:

```
@FunctionalInterface
public interface BiFunction<T, U, R> {
    R apply(T t, U u);
    // Other default and static methods
    // ...
}
```

The functional descriptor (method signature) is:

```
(T, U) -> R
```

Assuming a method:

```
void round(double d1, double d2, BiFunction<Double, Double, Long> f) {
    long result = f.apply(d1, d2);
    System.out.println(result);
}
```

Here's an example using an anonymous class:

```
round(5.4, 3.8, new BiFunction<Double, Double, Long>() {
    @Override
    public Long apply(Double d1, Double d2) {
        return Math.round(d1 + d2);
    }
});
```

And with a lambda expression:

```
round(5.4, 3.8, (d1, d2) -> Math.round(d1 + d2));
```

This interface, unlike `Function`, has only one default method:

```
default <V> BiFunction<T, U, V> andThen(Function<? super R, ? extends V> after)
```

That returns a composed function that first applies the function that calls `andThen` to its input, and then applies the function represented by the argument to the result.

This interface also has fewer primitive versions than `Function`. It only has the versions that take generic types as arguments and return `int`, `long` and `double` primitive types, with the naming convention **ToXXXBiFunction**, where XXX is the primitive type.

For example, here's the definition of `ToIntBiFunction`:

```
@FunctionalInterface
public interface ToIntBiFunction<T, U> {
    int applyAsInt(T t, U u);
}
```

This replaces `BiFunction`.

`BinaryOperator`

This interface is a specialization of the `BiFunction` interface (in fact, this interface extends it) for when the arguments and the result are of the same type.

This is how the interface is defined:

```
@FunctionalInterface
public interface BinaryOperator<T> extends BiFunction<T, T, T> {
    // Two static methods are defined
}
```

The functional descriptor (method signature) is:

```
(T, T) -> T
```

Here's an example using an anonymous class:

```
BinaryOperator<String> binOp = new BinaryOperator<String>() {
    @Override
    public String apply(String t, String u) {
        return t.concat(u);
    }
};
System.out.println(binOp.apply("Hello", " there"));
```

And with a lambda expression:

```
BinaryOperator<String> binOp = (t, u) -> t.concat(u);
System.out.println(binOp.apply("Hello", " there"));
```

This interface inherits the default method of the `BiFunction` interface:

```
default <V> BiFunction<T, T, V> andThen(Function<? super T, ? extends V> after)
```

And defines two new `static` methods:

```
static <T> BinaryOperator<T> minBy(Comparator<? super T> comparator)
static <T> BinaryOperator<T> maxBy(Comparator<? super T> comparator)
```

That return a `BinaryOperator`, which returns the lesser or greater of two elements according to the specified `Comparator`.

Here's a simple example:

```
BinaryOperator<Integer> biOp = BinaryOperator.maxBy(Comparator.naturalOrder());
System.out.println(biOp.apply(28, 8));
```

As you can see, these methods are just a wrapper to execute a `Comparator`.

`Comparator.naturalOrder()` returns a `Comparator` that compares `Comparable` objects in natural order. To execute it, we just call the `apply()` method with the two arguments needed by the `BinaryOperator`. Unsurprisingly, the output is:

```
28
```

There are also primitive versions for `int`, `long`, and `double`, where the two arguments and the return type are of the same primitive type. They don't extend `BinaryOperator` or `BiFunction`.

For example, here's the definition of `IntBinaryOperator`:

```
@FunctionalInterface
public interface IntBinaryOperator {
    int applyAsInt(int left, int right);
}
```

That you can use instead of `BinaryOperator`.

Primitive-specific Functional Interfaces

There's also a set of functional interfaces that are specifically designed to work with primitive types. These interfaces can provide better performance than their generic counterparts when working with primitives, as they avoid the overhead of boxing and unboxing.

There are several categories of primitive-specific functional interfaces:

1. `ToDoubleFunction<T>`, `ToIntFunction<T>`, `ToLongFunction<T>`: These interfaces represent functions that accept an object of type `T` and return a primitive `double`, `int`, or `long`, respectively. For example, this is how `ToIntFunction<T>` is defined:

```
@FunctionalInterface
public interface ToIntFunction<T> {
    int applyAsInt(T value);
}
```

And here's an example of how to use it:

```
ToIntFunction<String> stringToInt = Integer::parseInt;
int i = stringToInt.applyAsInt("123");  // 123
```

2. `ToDoubleBiFunction<T, U>`, `ToIntBiFunction<T, U>`, `ToLongBiFunction<T, U>`: These interfaces represent functions that accept two objects of types `T` and `U` and return a primitive `double`, `int`, or `long`, respectively. For example, this is how `ToIntBiFunction<T, U>` is defined:

```
@FunctionalInterface
public interface ToIntBiFunction<T, U> {
    int applyAsInt(T t, U u);
}
```

And here's an example of how to use it:

```
ToIntBiFunction<String, String> comparator = String::compareTo;
int result = comparator.applyAsInt("abc", "def");  // a negative value
```

3. `DoubleToIntFunction`, `DoubleToLongFunction`, `IntToDoubleFunction`, `IntToLongFunction`, `LongToDoubleFunction`, `LongToIntFunction`: These interfaces represent functions that accept one primitive type and return another primitive type. For example, this is how `DoubleToIntFunction` is defined:

```
@FunctionalInterface
public interface DoubleToIntFunction {
    int applyAsInt(double value);
}
```

And here's an example of how to use it:

```
DoubleToIntFunction roundDown = d -> (int) d;
int i = roundDown.applyAsInt(9.9);  // 9
```

4. `ObjDoubleConsumer<T>`, `ObjIntConsumer<T>`, `ObjLongConsumer<T>`: These interfaces represent functions that accept an object of type `T` and a primitive `double`, `int`, or `long`, and return `void`. For example, this is how `ObjIntConsumer<T>` is defined:

```
@FunctionalInterface
public interface ObjIntConsumer<T> {

    /**
     * Performs this operation on the given arguments.
     *
     * @param t the first input argument
     * @param value the second input argument
     */
    void accept(T t, int value);
}
```

And here's an example of how to use it:

```
ObjIntConsumer<List<Integer>> listAddInt = List::add;
List<Integer> list = new ArrayList<>();
listAddInt.accept(list, 1);  // [1]
```

These interfaces differ from `DoubleFunction<R>`, `IntFunction<R>`, `LongFunction<R>`, etc., in that the latter accept a primitive and return an object. For example:

```
IntFunction<String> intToString = Integer::toString;
String s = intToString.apply(123);  // "123"
```

The choice of which interface to use depends on your specific needs. If you're working primarily with primitives and want to avoid the overhead of autoboxing and unboxing, the primitive-specific interfaces are a good choice. However, if you need to work with objects, or if the boxing overhead is not a concern, the generic interfaces like `Function<T, R>` and `BiFunction<T, U, R>` are often more convenient.

Method References

As you know, in Java we can use references to objects, either by creating new objects:

```
List list = new ArrayList();
store(new ArrayList());
```

Or by using existing objects:

```
List list2 = list;
isFull(list2);
```

But what about a reference to a *method*?

If we only use a method of an object in another method, we still have to pass the full object as an argument. Wouldn't it be more practical to just pass the method as an argument? Like this for example:

```
isFull(list.size);
```

Thanks to lambda expressions, we can do something like that. We can use methods as if they were objects or primitive values.

And that's because a method reference is the shorthand syntax for a lambda expression that executes just **one** method.

Here's the syntax for a method reference:

```
Object :: methodName
```

You can use lambda expressions instead of using an anonymous class, but sometimes, the lambda expression is really just a call to some method. For example:

```
Consumer<String> c = s -> System.out.println(s);
```

To make the code clearer, you can turn that lambda expression into a method reference:

```
Consumer<String> c = System.out::println;
```

In a method reference, you place the object (or class) that contains the method before the `::` operator and the name of the method after it without arguments.

But you may be thinking:

- How is this clearer?
- What happens to the arguments?
- How can this be a valid expression?
- I don't understand how to construct a valid method reference

First of all, a method reference can't be used for any method. They can be used only to replace a single-method lambda expression.

So to use a method reference you first need a lambda expression with one method. And to use a lambda expression you first need a functional interface, an interface with just one abstract method.

In other words:

Instead of using

AN ANONYMOUS CLASS

you can use

A LAMBDA EXPRESSION

And if this just calls one method, you can use

A METHOD REFERENCE

There are four types of method references:

- A method reference to a *static method*
- A method reference to an *instance method of an object of a particular type*
- A method reference to an *instance method of an existing object*
- A method reference to a *constructor*

Let's begin by explaining the most natural case, a *static method*.

Static Methods

In this case, we have a lambda expression like the following:

```
(args) -> Class.staticMethod(args)
```

That can be turned into the following method reference:

```
Class::staticMethod
```

Notice that between a static method and a static method reference instead of the `.` operator, we use the `::` operator, and that we don't pass arguments to the method reference.

In general, we don't have to pass arguments to method references. However, arguments are treated depending on the type of method reference.

In this case, any arguments (if any) taken by the method are passed automatically behind the curtains.

Wherever we can pass a lambda expression that just calls a static method, we can use a method reference. For example, assuming this class:

```
class Numbers {
    public static boolean isMoreThanFifty(int n1, int n2) {
        return (n1 + n2) > 50;
    }
    public static List<Integer> findNumbers(
        List<Integer> l, BiPredicate<Integer, Integer> p) {
        List<Integer> newList = new ArrayList<>();
        for (Integer i : l) {
            if (p.test(i, i + 10)) {
                newList.add(i);
            }
        }
        return newList;
    }
}
```

We can call the `findNumbers()` method:

```
List<Integer> list = Arrays.asList(12, 5, 45, 18, 33, 24, 40);

// Using an anonymous class
findNumbers(list, new BiPredicate<Integer, Integer>() {
    public boolean test(Integer i1, Integer i2) {
        return Numbers.isMoreThanFifty(i1, i2);
    }
});

// Using a lambda expression
findNumbers(list, (i1, i2) -> Numbers.isMoreThanFifty(i1, i2));

// Using a method reference
findNumbers(list, Numbers::isMoreThanFifty);
```

Instance Method of An Object of A Particular Type

In this case, we have a lambda expression like the following:

```
(obj, args) -> obj.instanceMethod(args)
```

Where an instance of an object is passed, and one of its methods is executed with some optional parameters.

That can be turned into the following method reference:

```
ObjectType::instanceMethod
```

This time, the conversion is not that straightforward. First, in the method reference, we don't use the instance itself. We use its type.

Second, the other argument of the lambda expression, if any, is not used in the method reference, but it's passed behind the curtains like in the static method case.

For example, assuming this class:

```
class Shipment {
    public double calculateWeight() {
        double weight = 0;
        // Calculate weight
        return weight;
    }
}
```

And this method:

```
public List<Double> calculateOnShipments(
    List<Shipment> l, Function<Shipment, Double> f) {
    List<Double> results = new ArrayList<>();
    for (Shipment s : l) {
        results.add(f.apply(s));
    }
    return results;
}
```

We can call that method using:

```
List<Shipment> l = new ArrayList<Shipment>();

// Using an anonymous class
calculateOnShipments(l, new Function<Shipment, Double>() {
    public Double apply(Shipment s) { // The object
        return s.calculateWeight(); // The method
    }
});

// Using a lambda expression
calculateOnShipments(l, s -> s.calculateWeight());

// Using a method reference
calculateOnShipments(l, Shipment::calculateWeight);
```

In this example, we don't pass any arguments to the method. The key point here is that an instance of the object is the parameter of the lambda expression, and we form the reference to the instance method with the type of the instance.

Here's another example where we pass two arguments to the method reference.

Java has a `Function` interface that takes one parameter, a `BiFunction` that takes two parameters, but there's no `TriFunction` that takes three parameters, so let's make one:

```
interface TriFunction<T, U, V, R> {
    R apply(T t, U u, V v);
}
```

Now assume a class with a method that takes two parameters and returns a result, like this:

```
class Sum {
    Integer doSum(String s1, String s2) {
        return Integer.parseInt(s1) + Integer.parseInt(s2);
```

```
    }
}
```

We can wrap the `doSum()` method within a `TriFunction` implementation using an anonymous class:

```
TriFunction<Sum, String, String, Integer> anon =
    new TriFunction<Sum, String, String, Integer>() {
        @Override
        public Integer apply(Sum s, String arg1, String arg2) {
            return s.doSum(arg1, arg2);
        }
    };
System.out.println(anon.apply(new Sum(), "1", "4"));
```

Or using a lambda expression:

```
TriFunction<Sum, String, String, Integer> lambda =
    (Sum s, String arg1, String arg2) -> s.doSum(arg1, arg2);
System.out.println(lambda.apply(new Sum(), "1", "4"));
```

Or just using a method reference:

```
TriFunction<Sum, String, String, Integer> mRef = Sum::doSum;
System.out.println(mRef.apply(new Sum(), "1", "4"));
```

Here:

- The first type parameter of `TriFunction` is the object type that contains the method to execute.
- The second type parameter of `TriFunction` is the type of the first parameter.
- The third type parameter of `TriFunction` is the type of the second parameter.
- The last type parameter of `TriFunction` is the return type of the method to execute. Notice how this is omitted (inferred) in the lambda expression and the method reference.

It may seem odd to just see the interface, the class, and how they are used with a method reference, but this becomes more evident when you see the anonymous class or even the lambda version.

From:

```
(Sum s, String arg1, String arg2) -> s.doSum(arg1, arg2)
```

To

```
Sum::doSum
```

Instance Method of An Existing Object

In this case, we have a lambda expression like the following:

```
(args) -> obj.instanceMethod(args)
```

That can be turned into the following method reference:

```
obj::instanceMethod
```

This time, an instance defined elsewhere is used, and the arguments (if any) are passed behind the scenes like in the static method case.

For example, assuming these classes:

```
class Car {
    private int id;
    private String color;
    // More properties
    // And getters and setters
```

```
}
class Mechanic {
    public void fix(Car c) {
        System.out.println("Fixing car " + c.getId());
    }
}
```

And

this method:

```
public static void execute(Car car, Consumer<Car> c) {
    c.accept(car);
}
```

We can call the above method using:

```
final Mechanic mechanic = new Mechanic();
Car car = new Car();

// Using an anonymous class
execute(car, new Consumer<Car>() {
    public void accept(Car c) {
        mechanic.fix(c);
    }
});

// Using a lambda expression
execute(car, c -> mechanic.fix(c));

// Using a method reference
execute(car, mechanic::fix);
```

The key in this case is to use any object visible by an anonymous class/lambda expression and pass some arguments to an instance method of that object.

Here's another quick example using another `Consumer`:

```
Consumer<String> c = System.out::println;
c.accept("Hello");
```

Constructor

In this case, we have a lambda expression like the following:

```
(args) -> new ClassName(args)
```

That can be turned into the following method reference:

```
ClassName::new
```

The only thing this lambda expression does is to create a new object, so we just reference a constructor of the class with the keyword `new`. As in the other cases, arguments (if any) are not passed in the method reference.

Most of the time, we can use this syntax with two (or three) interfaces from the `java.util.function` package.

If the constructor takes no arguments, a `Supplier` will do the job:

```
// Using an anonymous class
Supplier<List<String>> s = new Supplier<List<String>>() {
    public List<String> get() {
```

```
        return new ArrayList<String>();
    }
};
List<String> l = s.get();

// Using a lambda expression
Supplier<List<String>> s = () -> new ArrayList<String>();
List<String> l = s.get();

// Using a method reference
Supplier<List<String>> s = ArrayList::new;
List<String> l = s.get();
```

If the constructor takes an argument, we can use the `Function` interface. For example:

```
// Using an anonymous class
Function<String, Integer> f =
    new Function<String, Integer>() {
        public Integer apply(String s) {
            return new Integer(s);
        }
    };
Integer i = f.apply("100");

// Using a lambda expression
Function<String, Integer> f = s -> new Integer(s);
Integer i = f.apply("100");

// Using a method reference
Function<String, Integer> f = Integer::new;
Integer i = f.apply("100");
```

If the constructor takes two arguments, we use the `BiFunction` interface:

```
// Using an anonymous class
BiFunction<String, String, Locale> f = new BiFunction<String, String, Locale>() {
    public Locale apply(String lang, String country) {
        return new Locale(lang, country);
    }
};
Locale loc = f.apply("en", "UK");

// Using a lambda expression
BiFunction<String, String, Locale> f = (lang, country) -> new Locale(lang, country);
Locale loc = f.apply("en", "UK");

// Using a method reference
BiFunction<String, String, Locale> f = Locale::new;
Locale loc = f.apply("en", "UK");
```

If you have a constructor with three or more arguments, you would have to create your own functional interface.

You can see that referencing a constructor is very similar to referencing a static method. The difference is that the constructor's *method name* is `new`.

Many of the examples of this chapter are very simple and probably don't justify the use of lambda expressions

or method references.

As mentioned at the beginning of the chapter, use method references if they make your code clearer.

You can avoid the one method restriction by grouping all your code in a static method, for example, and create a reference to that method instead of using a class or a lambda expression with many lines.

But the real power of lambda expressions and method references comes when they are combined with another feature of Java: streams.

That will be the topic of the next chapter.

Key Points

- A functional interface is an interface that contains only one abstract method. It may contain default or static methods.
- The `@FunctionalInterface` annotation is used to indicate an interface is intended to be a functional interface. It enables compiler checks but is not required if the interface meets the functional interface criteria.
- If an interface declares an abstract method overriding a public method in `java.lang.Object`, it doesn't count toward the interface's abstract method count.
- Lambda expressions enable you to treat functionality as a method argument or code as data.
- The syntax of a lambda is: `(parameters) -> expression` or `(parameters) -> { statements; }`.
- You can use `var` in the parameter list of a lambda expression to allow type inference.
- The target type of a lambda expression can be inferred in contexts like variable declarations, assignments, return statements, array initializers, method/constructor arguments, ternary expressions, and cast expressions.
- Lambda expressions are similar to anonymous classes in some ways, like the use of local variables, but differ in their treatment of `this`, default methods, parameter lists, and instance variables.
- Local variables used in a lambda expression or anonymous class must be final or effectively final (not modified after initialization) for thread safety and consistency.
- Java provides built-in functional interfaces in the `java.util.function` package for common use cases. The main ones are `Predicate<T>`, `Consumer<T>`, `Function<T, R>`, `Supplier<T>`, and `UnaryOperator<T>`.
- These interfaces also have primitive specializations (like `IntPredicate`, `LongConsumer`, etc.) to avoid autoboxing overhead when working with primitives.
- There are also binary versions of some of these interfaces that accept two parameters, like `BiPredicate<L, R>`, `BiConsumer<T, U>`, `BiFunction<T, U, R>`, and `BinaryOperator<T>`.
- `Predicate<T>` represents a boolean-valued function that takes an object of type `T`as input. It has `and`, `or`, and `negate` default methods for combining predicates.
- `Consumer<T>` represents an operation that accepts a single input argument and returns no result. It has an `andThen` default method for chaining consumers.
- `Function<T, R>` represents a function that accepts one argument and returns a result. It has `compose` and `andThen` default methods for combining functions.
- `Supplier<T>` represents a supplier of results, it takes no arguments and returns a result.
- `UnaryOperator<T>` represents an operation on a single operand that produces a result of the same type as its operand. It's a specialization of `Function` where the argument and result types are the same.
- Method references provide a way to refer to a method without invoking it, using the `::` operator. They can be used where a lambda expression is expected.

- There are four kinds of method references: to a static method, to an instance method of an object of a particular type, to an instance method of an existing object, and to a constructor.

Practice Questions

1. Which of the following statements are true about functional interfaces in Java? (Choose all that apply.)

A) A functional interface can have multiple `abstract` methods.
B) A functional interface can have default and `static` methods.
C) The `@FunctionalInterface` annotation is mandatory to declare a functional interface.
D) Lambda expressions can be used to instantiate functional interfaces.

2. Which of the following lambda expressions correctly implements the `Comparator<String>` interface?

```
Comparator<String> comparator = /* lambda expression */;
```

A) `(s1, s2) -> s1.compareTo(s2)`
B) `(String s1, s2) -> s1.compareTo(s2)`
C) `s1, s2 -> s1.compareTo(s2)`
D) `(s1, s2) -> return s1.compareTo(s2);`
E) `(s1, s2) -> { s1.compareTo(s2); }`

3. Which of the following Java built-in lambda interfaces represents a function that accepts two arguments and produces a result?

A) `java.util.function.Function`
B) `java.util.function.BiFunction`
C) `java.util.function.Supplier`
D) `java.util.function.Consumer`
E) `java.util.function.Predicate`

4. What is the output of the following code?

```
import java.util.function.Function;

public class Main {
    public static void main(String[] args) {
        Function<Integer, Integer> multiplyByTwo = x -> x * 2;
        Function<Integer, Integer> addThree = x -> x + 3;

        Function<Integer, Integer> combinedFunction = multiplyByTwo.andThen(addThree);

        System.out.println(combinedFunction.apply(5));
    }
}
```

A) `13`
B) `16`
C) `10`
D) `11`
E) `8`

5. Which of the following method references correctly replaces the lambda expression in the code below?

```
import java.util.function.Function;

public class Main {
```

```
    public static void main(String[] args) {
        Function<String, Integer> func = str -> Integer.parseInt(str);
        System.out.println(func.apply("123"));
    }
}
```

A) `String::valueOf`
B) `Integer::valueOf`
C) `Integer::parseInt`
D) `String::parseInt`
E) `Integer::toString`

Chapter EIGHT

Functional Interfaces and Lambda Expressions

Answers

1. The correct answers are B and D.

Explanation:

- **A)** A functional interface can have multiple `abstract` methods.
  - This option is incorrect. A functional interface can have only one abstract method. Having multiple abstract methods would disqualify it from being a functional interface.
- **B)** A functional interface can have default and `static` methods.
  - This option is correct. A functional interface is allowed to have default and static methods, which are not counted as abstract methods.
- **C)** The `@FunctionalInterface` annotation is mandatory to declare a functional interface.
  - This option is incorrect. The `@FunctionalInterface` annotation is not mandatory; it is only a marker to indicate that the interface is intended to be a functional interface. An interface can be a functional interface without this annotation as long as it has exactly one abstract method.
- **D)** Lambda expressions can be used to instantiate functional interfaces.
  - This option is correct. Lambda expressions are used to provide implementations for the single abstract method of a functional interface, making them a key feature for functional programming in Java.

2. The correct answer is A.

Explanation:

- **A)** `(s1, s2) -> s1.compareTo(s2)`
  - This option is correct. This lambda expression correctly implements the `Comparator<String>` interface. It uses the correct syntax for a lambda expression, with parameters enclosed in parentheses and a single expression for the body.
- **B)** `(String s1, s2) -> s1.compareTo(s2)`
  - This option is incorrect. The syntax is invalid because if you specify the type of one parameter, you must specify the type for all parameters. It should be `(String s1, String s2)`.
- **C)** `s1, s2 -> s1.compareTo(s2)`
  - This option is incorrect. Parameters must be enclosed in parentheses. The correct syntax is `(s1, s2)`.
- **D)** `(s1, s2) -> return s1.compareTo(s2);`
  - This option is incorrect. When using a return statement, you must also include curly braces.
- **E)** `(s1, s2) -> { s1.compareTo(s2); }`
  - This option is incorrect. When using curly braces, you must include a return statement for expressions that return a value. The correct syntax would be `(s1, s2) -> { return s1.compareTo(s2); }`.

3. The correct answer is B.

Explanation:

- **A)** `java.util.function.Function`
  - This option is incorrect. `Function` represents a function that takes one argument and produces a result.
- **B)** `java.util.function.BiFunction`
  - This option is correct. `BiFunction` represents a function that takes two arguments and produces a result.
- **C)** `java.util.function.Supplier`
  - This option is incorrect. `Supplier` represents a function that takes no arguments and produces a result.
- **D)** `java.util.function.Consumer`
  - This option is incorrect. `Consumer` represents a function that takes one argument and does not produce a result.
- **E)** `java.util.function.Predicate`
  - This option is incorrect. `Predicate` represents a function that takes one argument and returns a `boolean` value.

4. The correct answer is A.

Explanation:

- **A)** `13`
  - This option is correct. The `combinedFunction` first multiplies 5 by 2 to get 10, then adds 3, resulting in 13.
- **B)** `16`
  - This option is incorrect. It incorrectly assumes that 5 is added after doubling and doubling again.
- **C)** `10`
  - This option is incorrect. It represents only the result of the first function without applying the second function.
- **D)** `11`
  - This option is incorrect. It seems to mistakenly represent 5 plus the first function (double).
- **E)** `8`
  - This option is incorrect. It seems to incorrectly represent the input value doubled without adding 3.

5. The correct answer is C.

Explanation:

- **A)** `String::valueOf`
  - This option is incorrect. `String::valueOf` converts an integer to a string, not a string to an integer.
- **B)** `Integer::valueOf`
  - This option is incorrect. `Integer::valueOf` returns an `Integer` object, while the lambda returns an `int`.
- **C)** `Integer::parseInt`
  - This option is correct. `Integer::parseInt` is a method reference that matches the lambda expression `str -> Integer.parseInt(str)` which converts a string to an integer.
- **D)** `String::parseInt`
  - This option is incorrect. `String` class does not have a `parseInt` method.
- **E)** `Integer::toString`
  - This option is incorrect. `Integer::toString` converts an integer to a string, not a string to an integer.

Chapter NINE

Streams

Chapter Content

The Optional Class

Most programming languages have a data type to represent the absence of a value, and it is known by many names:

`NULL, nil, None, Nothing`

The `null` type was introduced in ALGOL W by Tony Hoare in 1965, and it's considered one of the worst mistakes in computer science. In Tony Hoare's words:

I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object-oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

Still, some may be wondering, what the problem with `null` is?

Well, if you're a little worried by the problems this code might cause, you know the answer already:

```
String summary =
  book.getChapter(10)
      .getSummary().toUpperCase();
```

The problem with that code is that if any of those methods return a `null` reference (for example, if the book doesn't have a tenth chapter), a `NullPointerException` (the most common exception in Java) will be thrown at runtime, stopping the program.

What can we do to avoid this exception?

Perhaps, the easiest way is to check for `null`. Here's one way to do it:

```
String summary = "";
if(book != null) {
    Chapter chapter = book.getChapter(10);
    if(chapter != null) {
        if(chapter.getSummary() != null) {
            summary = chapter.getSummary()
                             .toUpperCase();
        }
    }
}
```

You don't know if any object in this hierarchy can be `null`, so you check every object for it. Obviously, this is not the best solution; it's not very practical and damages readability.

There's another issue. Is checking for `null` really desirable? I mean, what if those objects should never be `null`? By checking for `null`, we hide the error and don't deal with it.

Of course, this is a design issue too. For example, if a chapter has no summary yet, what would be better to use as a default value? An empty string or `null`?

The class `java.util.Optional<T>` addresses this problem.

The job of this class is to encapsulate an optional value, which is an object that can be `null`.

Using the previous example, if we know that not all chapters have a summary, instead of modeling the class like this:

```
class Chapter {
    private String summary;
    // Other attributes and methods
}
```

We can use the `Optional` class:

```
class Chapter {
    private Optional<String> summary;
    // Other attributes and methods
}
```

So if there's a value, the `Optional` class just wraps it. Otherwise, an empty value is represented by the method `Optional.empty()`, which returns a singleton instance of `Optional`.

By using this class instead of `null`, we explicitly declare that the summary attribute is optional. Then, we can avoid `NullPointerExceptions` while having the useful methods of `Optional` at our disposal, which we'll review next.

First, let's see how to create an instance of this class.

To get an empty `Optional` object, use:

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

If you are sure that an object is not `null`, you can wrap it in an `Optional` object this way:

```
Optional<String> summary = Optional.of("A summary");
```

A `NullPointerException` will be thrown if the object is `null`. However, you can use:

```
Optional<String> summary = Optional.ofNullable("A summary");
```

That returns an `Optional` instance with the specified value if it is non-`null`. Otherwise, it returns an empty `Optional`.

If you want to know if an `Optional` contains a value, you can do it like this:

```
if (summary.isPresent()) {
    // Do something
}
```

Or in a more functional style:

```
summary.ifPresent(s -> System.out.println(s));
// Or summary.ifPresent(System.out::println);
```

The `ifPresent()` method takes a `Consumer<T>` as an argument that is executed only if the `Optional` contains a value.

To get the value of an `Optional`, use:

```
String s = summary.get();
```

However, this method will throw a `java.util.NoSuchElementException` if the `Optional` doesn't contain a value, so it's better to use the `ifPresent()` method.

Alternatively, if we want to return something when the `Optional` doesn't contain a value, there are three other methods we can use:

```
String summaryOrDefault = summary.orElse("Default summary");
```

The `orElse()` method returns the argument (which must be of type `T`, in this case a `String`) when the `Optional` is empty. Otherwise, it returns the encapsulated value.

```
String summaryOrDefault =
    summary.orElseGet(() -> "Default summary");
```

The `orElseGet()` method takes a `Supplier<? extends T>` as an argument that returns a value when the `Optional` is empty. Otherwise, it returns the encapsulated value.

```
String summaryOrException =
    summary.orElseThrow(() -> new Exception());
```

The `orElseThrow()` method takes a `Supplier<? extends X>`, where `X` is the type of the exception to throw when the `Optional` is empty. Otherwise, it returns the encapsulated value.

There are versions of the `Optional` class to work with primitives, `OptionalInt`, `OptionalLong`, and `OptionalDouble`, so you can use `OptionalInt` instead of `Optional<Integer>`:

```
OptionalInt optionalInt = OptionalInt.of(1);
int i = optionalInt.getAsInt();
```

However, the use of these primitive versions is not encouraged, especially because they lack three useful methods of `Optional`: `filter()`, `map()`, and `flatMap()`. And since `Optional` just contains one value, the overhead of boxing/unboxing a primitive is not significant.

The `filter()` method returns the `Optional` if a value is present and matches the given predicate. Otherwise, an empty `Optional` is returned.

```
String summaryStr =
    summary.filter(s -> s.length() > 10).orElse("Short summary");
```

The `map()` method is generally used to transform from one type to another. If the value is present, it applies the provided `Function<? super T, ? extends U>` to it. For example:

```
int summaryLength = summary.map(s -> s.length()).orElse(0);
```

The `flatMap()` method is similar to `map()`, but it takes an argument of type `Function<? super T, Optional<U>>` and if the value is present, it returns the `Optional` that results from applying the provided function. Otherwise, it returns an empty `Optional`.

Streams

Suppose you have a list of students and the requirements are to extract the students with a score of `90.0` or greater and sort them by score in ascending order.

One way to do it would be:

```
List<Student> studentsScore = new ArrayList<Student>();
for(Student s : students) {
   if(s.getScore() >= 90.0) {
       studentsScore.add(s);
   }
}
Collections.sort(studentsScore, new Comparator<Student>() {
   public int compare(Student s1, Student s2) {
       return Double.compare(s1.getScore(), s2.getScore());
   }
});
```

Very verbose when we compare it with the implementation that uses streams:

```
List<Student> studentsScore = students
    .stream()
    .filter(s -> s.getScore() >= 90.0)
    .sorted(Comparator.comparing(Student::getScore))
    .collect(Collectors.toList());
```

Don't worry if you don't fully understand the code, we'll see what it means later.

What Are Streams?

First of all, streams are **NOT** collections.

A simple definition is that streams are *wrappers* for collections or arrays. They wrap an existing collection (or another data source) to support operations expressed with lambdas, so you specify what you want to do, not how to do it. You already saw it.

These are the characteristics of a stream:

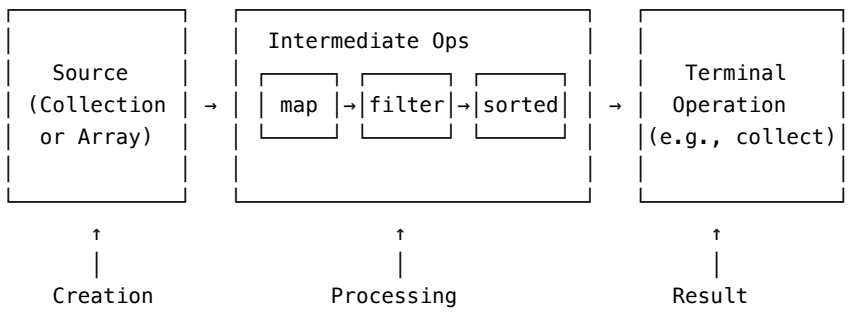
- **Streams work perfectly with lambdas.** All streams operations take functional interfaces as arguments, so you can simplify the code with lambda expressions (and method references).
- **Streams don't store their elements.** The elements are stored in a collection or generated on the fly. They are only carried from the source through a pipeline of operations.
- **Streams are immutable.** Streams don't mutate their underlying source of elements. Instead, they create a new stream reflecting the transformations applied.
- **Streams are not reusable.** Streams can be traversed only once. After a terminal operation is executed (we'll see what this means in a moment), you have to create another stream from the source to further process it.
- **Streams don't support indexed access to their elements.** Again, streams are not collections or arrays. The most you can do is get their first element.
- **Streams are easily parallelizable.** With the call of a method (and following certain rules), you can make a stream execute its operations concurrently, without having to write any multithreading code.
- **Stream operations are lazy when possible.** Streams defer the execution of their operations either until the results are needed or until it's known how much data is needed.

One thing that allows this laziness is the way their operations are designed. Most of them return a new stream, allowing operations to be chained and form a pipeline that enables this kind of optimizations.

To set up this pipeline you:

1. Create the stream.
2. Apply zero or more intermediate operations to transform the initial stream into new streams.
3. Apply a terminal operation to generate a result or a *side-effect*.

Here's a diagram to help you visualize this pipeline:



Creating Streams

A stream is represented by the `java.util.stream.Stream<T>` interface. This works with objects only.

There are also specializations to work with primitive types, such as `IntStream`, `LongStream`, and `DoubleStream`.

There are many ways to create a stream. Let's start with the most popular three.

The first one is creating a stream from a `java.util.Collection` implementation using the `stream()` method:

```
List<String> words = Arrays.asList("hello", "hola", "hallo", "ciao");;
Stream<String> stream = words.stream();
```

The second one is creating a stream from individual values:

```
Stream<String> stream = Stream.of("hello","hola", "hallo", "ciao");
```

The third one is creating a stream from an array:

```
String[] words = {"hello", "hola", "hallo", "ciao"};
Stream<String> stream = Stream.of(words);
```

However, you have to be careful with this last method when working with primitives.

Here's why. Assume an `int` array:

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

When we create a stream from this array like this:

```
Stream.of(nums)
```

We are not creating a stream of `Integers` (`Stream<Integer>`), but a stream of `int` arrays (`Stream<int[]>`). This means that instead of having a stream with five elements we have a stream of one element:

```
System.out.println(Stream.of(nums).count()); // It prints 1!
```

The reason is the signatures of the of method:

```
// returns a stream of one element
static <T> Stream<T> of(T t)
// returns a stream whose elements are the specified values
static <T> Stream<T> of(T... values)
```

Since an int is not an object, but `int[]` is, the method chosen to create the stream is the first (`Stream.of(T t)`), not the one with the vargs, so a stream of `int[]` is created, but since only one array is passed, the result is a stream of one element.

To solve this, we can force Java to choose the varargs version by creating an array of objects (with `Integer`):

```
Integer[] nums = {1, 2, 3, 4, 5};
// It prints 5!
System.out.println(Stream.of(nums).count());
```

Or use a fourth way to create a stream (that it's in fact used inside `Stream.of(T... values)`):

```
int[] nums = {1, 2, 3, 4, 5};
// It also prints 5!
System.out.println(Arrays.stream(nums).count());
```

Or use the primitive version `IntStream`:

```
int[] nums = {1, 2, 3, 4, 5};
// It also prints 5!
System.out.println(IntStream.of(nums).count());
```

So don't use `Stream<T>.of()` when working with primitives.

Here are other ways to create streams:

```
static <T> Stream<T> generate(Supplier<T> s)
```

This method returns an *infinite* stream where each element is generated by the provided `Supplier`, and is generally used with the method:

```
Stream<T> limit(long maxSize)
```

That truncates the stream so that it is no longer than `maxSize` in length.

For example:

```
Stream<Double> s = Stream.generate(new Supplier<Double>() {
   public Double get() {
       return Math.random();
   }
}).limit(5);
```

Or:

```
Stream<Double> s = Stream.generate(() -> Math.random()).limit(5);
```

Or just:

```
Stream<Double> s = Stream.generate(Math::random).limit(5);
```

Which generates a stream of five random `doubles`.

Then we have the `iterate` method:

```
static <T> Stream<T> iterate(T seed, UnaryOperator<T> f)
```

It returns an *infinite* stream produced by the iterative application of a function `f` to an initial element (seed). The first element (`n = 0`) in the stream will be the provided seed. For `n > 0`, the element at position `n` will be the result of applying the function `f` to the element at position `n - 1`. For example:

```
Stream<Integer> s = Stream.iterate(1, new UnaryOperator<Integer>() {
   @Override
   public Integer apply(Integer t) {
```

```
        return t * 2; }
}).limit(5);
```

Or just:

```
Stream<Integer> s = Stream.iterate(1, t -> t * 2).limit(5);
```

That generates the elements `1`, `2`, `4`, `8`, `16`.

There's a `Stream.Builder<T>` class (that follows the builder design pattern) with methods that add an element to the stream being built:

```
void accept(T t)
default Stream.Builder<T> add(T t)
```

For example:

```
Stream.Builder<String> builder = Stream.<String>builder().add("h").add("e").add("l").add("l");
builder.accept("o");
Stream<String> s = builder.build();
```

`IntStream` and `LongStream` define the methods:

```
static IntStream range(int startInclusive, int endExclusive)
static IntStream rangeClosed(int startInclusive, int endInclusive)
static LongStream range(long startInclusive, long endExclusive)
static LongStream rangeClosed(long startInclusive, long endInclusive)
```

That returns a sequential stream for the range of `int` or `long` elements. For example:

```
// stream of 1, 2, 3
IntStream s = IntStream.range(1, 4);
// stream of 1, 2, 3, 4
IntStream s = IntStream.rangeClosed(1, 4);
```

Also, there are methods in the Java API that generate streams. For example:

```
IntStream s1 = new Random().ints(5, 1, 10);
```

Which returns an `IntStream` of five random `int`s from one (inclusive) to ten (exclusive).

Intermediate Operations

You can easily identify intermediate operations; they always return a new stream. This allows the operations to be connected.

For example:

```
Stream<String> s = Stream.of("m", "k", "c", "t")
    .sorted()
    .limit(3)
```

An important feature of intermediate operations is that they don't process the elements until a terminal operation is invoked, meaning they're lazy.

Intermediate operations can be either *stateless* or *stateful*.

Stateless operations retain no state from previous elements when processing a new element so each can be processed independently of operations on other elements.

Stateful operations, such as `distinct` and `sorted`, require processing the entire stream or keeping track of state from previously processed elements to produce a result.

The following table summarizes the methods of the `Stream` interface that represent intermediate operations.

Method	Type	Description
`Stream<T> distinct()`	Stateful	Returns a stream consisting of the distinct elements.
`Stream<T> filter(Predicate<? super T> predicate)`	Stateless	Returns a stream of elements that match the given predicate.
`<R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper)`	Stateless	Returns a stream with the content produced by applying the provided mapping function to each element. There are versions for `int`, `long` and `double` also.
`Stream<T> limit(long maxSize)`	Stateful	Returns a stream truncated to be no longer than `maxSize` in length.
`<R> Stream<R> map(Function<? super T,? extends R> mapper)`	Stateless	Returns a stream consisting of the results of applying the given function to the elements of this stream. There are versions for `int`, `long` and `double` also.
`Stream<T> peek(Consumer<? super T> action)`	Stateless	Returns a stream with the elements of this stream, performing the provided action on each element.
`Stream<T> skip(long n)`	Stateful	Returns a stream with the remaining elements of this stream after discarding the first `n` elements.
`Stream<T> sorted()`	Stateful	Returns a stream sorted according to the natural order of its elements.
`Stream<T> sorted(Comparator<? super T> comparator)`	Stateful	Returns a stream sorted according to the provided `Comparator`.
`Stream<T> parallel()`	N/A	Returns an equivalent stream that is parallel.
`Stream<T> sequential()`	N/A	Returns an equivalent stream that is sequential.
`Stream<T> unordered()`	N/A	Returns an equivalent stream that is unordered.

Terminal Operations

You can also easily identify terminal operations, they always return something other than a stream.

After the terminal operation is performed, the stream pipeline is *consumed*, and can't be used anymore. For example:

```
int[] digits = {0, 1, 2, 3, 4 , 5, 6, 7, 8, 9};
IntStream s = IntStream.of(digits);
long n = s.count();
System.out.println(s.findFirst()); // An exception is thrown
```

If you need to traverse the same stream again, you must return to the data source to get a new one. For example:

```
int[] digits = {0, 1, 2, 3, 4 , 5, 6, 7, 8, 9};
long n = IntStream.of(digits).count();
System.out.println(IntStream.of(digits).findFirst()); // OK
```

The following table summarizes the methods of the `Stream` interface that represent terminal operations.

Method	Description
`boolean allMatch(Predicate<? super T> predicate)`	Returns whether all elements of this stream match the provided predicate. If the stream is empty then `true` is returned and the predicate is not evaluated.
`boolean anyMatch(Predicate<? super T> predicate)`	Returns whether any elements of this stream match the provided predicate. If the stream is empty then `false` is returned and the predicate is not evaluated.

Method	Description
`boolean noneMatch(Predicate<? super T> predicate)`	Returns whether no elements of this stream match the provided predicate. If the stream is empty then `true` is returned and the predicate is not evaluated.
`Optional<T> findAny()`	Returns an `Optional` describing some element of the stream.
`Optional<T> findFirst()`	Returns an `Optional` describing the first element of this stream.
`<R,A> R collect(Collector<? super T,A,R> collector)`	Performs a mutable reduction operation on the elements of this stream using a `Collector`.
`long count()`	Returns the count of elements in this stream.
`void forEach(Consumer<? super T> action)`	Performs an action for each element of this stream.
`void forEachOrdered(Consumer<? super T> action)`	Performs an action for each element of this stream, in the encounter order of the stream if the stream has a defined encounter order.
`Optional<T> max(Comparator<? super T> comparator)`	Returns the maximum element of this stream according to the provided `Comparator`.
`Optional<T> min(Comparator<? super T> comparator)`	Returns the minimum element of this stream according to the provided `Comparator`.
`T reduce(T identity, BinaryOperator<T> accumulator)`	Performs a reduction on the elements of this stream, using the provided identity value and an associative accumulation function, and returns the reduced value.
`Object[] toArray()`	Returns an array containing the elements of this stream.
`<A> A[] toArray(IntFunction<A[]> generator)`	Returns an array containing the elements of this stream, using the provided generator function to allocate the returned array.
`Iterator<T> iterator()`	Returns an iterator for the elements of the stream.
`Spliterator<T> spliterator()`	Returns a spliterator for the elements of the stream.

Lazy Operations

Intermediate operations are deferred until a terminal operation is invoked. The reason is that intermediate operations can usually be merged or optimized by a terminal operation.

Let's take for example this stream pipeline:

```
Stream.of("sun", "pool", "beach", "kid", "island", "sea", "sand")
    .map(str -> str.length())
    .filter(i -> i > 3)
    .limit(2)
    .forEach(System.out::println);
```

Here's what it does:

- It generates a stream of strings,
- Then convert the stream to a stream of `int`s (representing the length of each string)
- Then it filters the lengths greater than three,
- Then it grabs the first two elements of the stream and
- Finally, prints those two elements.

And you may think the map operation is applied to all seven elements, then the `filter` operation again to all seven, then it picks the first two, and finally it prints the values.

But this is not how it works. If we modify the lambda expressions of `map` and `filter` to print a message:

```
Stream.of("sun", "pool", "beach", "kid", "island", "sea", "sand")
    .map(str -> {
```

```
        System.out.println("Mapping: " + str);
        return str.length();
    })
    .filter(i -> {
        System.out.println("Filtering: " + i);
        return i > 3;
    })
    .limit(2)
    .forEach(System.out::println);
```

The order of evaluation will be revealed:

```
Mapping: sun
Filtering: 3
Mapping: pool
Filtering: 4
4
Mapping: beach
Filtering: 5
5
```

From this example, we can see that the stream applied operations only until it found enough elements to return a result (due to the `limit(2)` operation). This is called *short-circuiting*.

Short-circuit operations cause intermediate operations to be processed until a result can be produced.

In such a way, because of lazy and short-circuit operations, streams don't execute all operations on all their elements. Instead, the elements of the stream go through a pipeline of operations until the point a result can be deduced or generated.

You can see short-circuiting as a subclassification. There's only one short-circuit intermediate operation:

```
Stream<T> limit(long maxSize)
```

Because it doesn't need to process all the elements of the stream to create a stream of a given size.

The rest are terminal:

```
boolean anyMatch(Predicate<? super T> predicate)
boolean allMatch(Predicate<? super T> predicate)
boolean noneMatch(Predicate<? super T> predicate)
Optional<T> findFirst()
Optional<T> findAny()
```

Because as soon as you find a matching element, there's no need to continuing processing the stream.

Primitive Streams

Most of the time, we'll use a `Stream<T>` that contains objects as elements. However, there are also specialized streams for handling primitive types like `int`, `long`, and `double` that allow you to avoid the overhead of auto-boxing and auto-unboxing elements into their wrapper classes. These primitive streams are `IntStream`, `LongStream`, and `DoubleStream`.

Each primitive stream has methods analogous to the ones in the regular `Stream` class, like `map()` (transforms elements), `filter()` (selects elements based on a predicate), `reduce()` (aggregates elements), etc. But because they can only deal with their corresponding primitive types, there are also specialized methods to handle them. Let's review the most important ones.

The `average()` method returns an `OptionalDouble` with the arithmetic mean of elements, or an empty `OptionalDouble` if the primitive stream is empty:

```
IntStream stream = IntStream.range(1, 10);
OptionalDouble ave = stream.average();
System.out.println(ave.getAsDouble());
```

This code prints the average of the numbers from 1 to 9 (not including 10):

```
5.0
```

If you need to convert a primitive stream to a regular object stream, use the `boxed()` method:

```
Stream<Double> boxed = DoubleStream.of(1.2, 2.4).boxed();
```

To find the maximum value in the primitive stream, use `max()`:

```
IntStream stream = IntStream.of(1, 10, 2, 20);
OptionalInt max = stream.max();
System.out.println(max.getAsInt());
```

This prints:

```
20
```

Each primitive stream has its own `max()` method that returns its corresponding `Optional` type (`OptionalInt`, `OptionalLong`, `OptionalDouble`). The same goes for `min()`.

One peculiarity of the `IntStream` and `LongStream` is that they have special methods `range()` and `rangeClosed()` to generate a sequence of numbers in a range.

`range(int a, int b)` creates an `IntStream` of values from `a` (inclusive) to `b` (exclusive). `rangeClosed(int a, int b)` does the same including `b`:

```
IntStream stream = IntStream.range(1, 5);
stream.forEach(System.out::println);
```

This prints:

```
1
2
3
4
```

While:

```
LongStream stream = LongStream.rangeClosed(1, 5);
stream.forEach(System.out::println);
```

Prints:

```
1
2
3
4
5
```

Note that there are no `range()` or `rangeClosed()` methods in `DoubleStream`.

The `sum()` method returns the sum of all the elements:

```
IntStream stream = IntStream.of(1, 10, 2, 20);
int sum = stream.sum();
System.out.println(sum);
```

This prints:

```
33
```

Again, each primitive stream has its own dedicated `sum()` method that returns the primitive type result (`int`, `long`, `double`).

Finally, each primitive stream has a `summaryStatistics()` method that returns a summary of stats of the elements. Let's see an example using `IntStream`:

```
IntStream stream = IntStream.of(1, 10, 2, 20);
IntSummaryStatistics stats = stream.summaryStatistics();
System.out.println(stats);
```

This prints:

```
IntSummaryStatistics{count=4, sum=33, min=1, average=8.250000, max=20}
```

`LongStream` and `DoubleStream` have analogous `LongSummaryStatistics` and `DoubleSummaryStatistics` classes.

These summary statistics objects provide methods to obtain each stat separately (`getCount()`, `getSum()`, `getMin()`, `getAverage()`, `getMax()`).

If you need more advanced stats, you can use a `Collector` and the `summarizingInt()`, `summarizingLong()`, or `summarizingDouble()` methods as arguments:

```
List<Integer> list = List.of(1, 10, 2, 20);
IntSummaryStatistics stats = list.stream()
        .collect(Collectors.summarizingInt(i -> i));
System.out.println(stats);
```

This prints:

```
IntSummaryStatistics{count=4, sum=33, min=1, average=8.250000, max=20}
```

Now let's talk in more detail about some of the most common stream operations.

Filtering Streams

Filtering is one of the most common operations when when you work with streams in Java. It allows you to select only the elements that satisfy a given predicate, discarding the rest. The `filter()` method is used for this purpose:

```
Stream<T> filter(Predicate<? super T> predicate);
```

The `filter()` method takes a `Predicate` functional interface as argument. A `Predicate` is a function that takes an element and returns a `boolean`. Only the elements for which the predicate returns `true` will be included in the resulting stream.

Let's review a simple example:

```
List<Integer> list = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
List<Integer> evenList = list.stream()
        .filter(i -> i % 2 == 0)
        .collect(Collectors.toList());
System.out.println(evenList);
```

This code filters the original list, keeping only the even numbers. It prints:

```
[2, 4, 6, 8, 10]
```

You can chain multiple `filter()` calls to apply several conditions:

```
List<Integer> list = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
List<Integer> filteredList = list.stream()
        .filter(i -> i > 3)
        .filter(i -> i < 8)
```

```
        .collect(Collectors.toList());
System.out.println(filteredList);
```

This selects the numbers greater than 3 and less than 8:

```
[4, 5, 6, 7]
```

The `Predicate` interface also has default methods that allow you to combine predicates using logical operations: - `default Predicate<T> and(Predicate<? super T> other)` - `default Predicate<T> or(Predicate<? super T> other)` - `default Predicate<T> negate()`

For example, to get the numbers greater than 3 and less than 8 you can also do:

```
List<Integer> list = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
List<Integer> filteredList = list.stream()
        .filter(i -> i > 3 && i < 8)
        .collect(Collectors.toList());
```

Or using `Predicate` methods:

```
Predicate<Integer> greaterThan3 = i -> i > 3;
Predicate<Integer> lessThan8 = i -> i < 8;
List<Integer> filteredList = list.stream()
        .filter(greaterThan3.and(lessThan8))
        .collect(Collectors.toList());
```

The `filter()` method is stateless, which means that the execution of the predicate for one element doesn't affect the execution for another.

A very useful method related to `filter()` is `distinct()`:

```
Stream<T> distinct();
```

This method returns a stream of unique elements, discarding the duplicates:

```
List<Integer> list = List.of(1, 2, 2, 3, 4, 4, 5);
List<Integer> distinctList = list.stream()
        .distinct()
        .collect(Collectors.toList());
System.out.println(distinctList);
```

This prints:

```
[1, 2, 3, 4, 5]
```

You can think of `distinct()` as a special filtering operation.

Finally, there are two other methods similar to `filter()` but with a different purpose: - `default Stream<T> takeWhile(Predicate<? super T> predicate)` - `default Stream<T> dropWhile(Predicate<? super T> predicate)`

`takeWhile()` returns a stream that contains the longest prefix of elements taken from the original stream that match the given predicate.

```
List<Integer> list = List.of(2, 4, 6, 7, 8, 10, 11);
List<Integer> prefixList = list.stream()
        .takeWhile(i -> i % 2 == 0)
        .collect(Collectors.toList());
System.out.println(prefixList);
```

This selects the even numbers from the beginning of the stream until it finds the first odd number (7):

```
[2, 4, 6]
```

The opposite is `dropWhile()`, which discards the longest prefix of elements that satisfy the predicate and returns a stream of the remaining elements:

```
List<Integer> list = List.of(2, 4, 6, 7, 8, 10, 11);
List<Integer> postfixList = list.stream()
        .dropWhile(i -> i % 2 == 0)
        .collect(Collectors.toList());
System.out.println(postfixList);
```

This discards the initial even numbers and returns the rest of the stream:

```
[7, 8, 10, 11]
```

It is important to note that the predicates used in `takeWhile()` and `dropWhile()` must be stateless. The execution for one element shouldn't affect the execution for another, otherwise the results will be unpredictable.

Mapping Streams

When working with streams, we often need to transform the elements from one type to another or extract certain data from them. This is where the `map()` and `flatMap()` operations come into play.

The `map()` method applies a function to each element of the stream, transforming it into a new element. It's like having a machine that takes in raw materials (the original elements) and outputs refined products (the transformed elements).

```
<R> Stream<R> map(Function<? super T, ? extends R> mapper);
```

The `map()` method takes a `Function` as argument, which is an interface that represents a function that accepts one argument and returns a result. In this case, it takes an element of type `T` and returns an element of type `R`.

Here's an example:

```
List<String> list = List.of("1", "2", "3", "4", "5");
List<Integer> intList = list.stream()
        .map(Integer::parseInt)
        .collect(Collectors.toList());
System.out.println(intList);
```

This code converts a list of strings into a list of integers using the `parseInt` method of the `Integer` class. It prints:

```
[1, 2, 3, 4, 5]
```

You can chain multiple `map()` operations to perform successive transformations:

```
List<String> list = List.of("1", "2", "3", "4", "5");
List<Integer> doubledList = list.stream()
        .map(Integer::parseInt)
        .map(i -> i * 2)
        .collect(Collectors.toList());
System.out.println(doubledList);
```

This first converts the strings to integers and then multiplies each number by 2:

```
[2, 4, 6, 8, 10]
```

Now, what if instead of transforming each element, you want to extract multiple elements from each one? This is where `flatMap()` comes in.

`flatMap()` is like having a machine that takes in containers filled with raw materials, unpacks each container, processes the materials, and then outputs the refined products in a single stream.

```
<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper);
```

The `flatMap()` method takes a function that returns a stream for each element. Then, it flattens all these streams into a single one.

A common use case is when you have a stream of lists and you want to process the elements of all the lists as a single stream:

```
List<List<Integer>> listOfLists = List.of(
        List.of(1, 2, 3),
        List.of(4, 5, 6),
        List.of(7, 8, 9)
);
List<Integer> flattenedList = listOfLists.stream()
        .flatMap(List::stream)
        .collect(Collectors.toList());
System.out.println(flattenedList);
```

This code flattens the list of lists into a single list:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

When working with primitive streams, there are specialized mapping operations to avoid boxing and unboxing costs: - `IntStream mapToInt(ToIntFunction<? super T> mapper)` - `LongStream mapToLong(ToLongFunction<? super T> mapper)` - `DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper)`

These methods take a `ToIntFunction`, `ToLongFunction`, and `ToDoubleFunction` respectively, and return an `IntStream`, `LongStream`, and `DoubleStream`.

```
List<String> list = List.of("1", "2", "3", "4", "5");
IntStream intStream = list.stream()
        .mapToInt(Integer::parseInt);
intStream.forEach(System.out::println);
```

This code converts the stream of strings to an `IntStream` and prints each element:

```
1
2
3
4
5
```

You can also map one primitive type to another:

```
IntStream intStream = IntStream.range(1, 6);
DoubleStream doubleStream = intStream.mapToDouble(i -> i / 2.0);
doubleStream.forEach(System.out::println);
```

This converts an `IntStream` to a `DoubleStream`, dividing each number by 2:

```
0.5
1.0
1.5
2.0
2.5
```

Decomposing Streams

When working with streams, sometimes we need to break them down into smaller parts, analyze their elements, or combine them in certain ways. This is what we call decomposing streams and there are several operations that allow us to do this.

First, let's talk about `skip()` and `limit()`. These methods allow us to cut a stream into parts, discarding some elements and keeping others.

`skip(long n)` returns a stream that discards the first `n` elements of the original stream. It's like cutting off the top part of a log.

Here's an example:

```
List<Integer> list = List.of(1, 2, 3, 4, 5);
List<Integer> skippedList = list.stream()
        .skip(2)
        .collect(Collectors.toList());
System.out.println(skippedList);
```

This skips the first two elements and collects the rest into a new list:

```
[3, 4, 5]
```

On the other hand, `limit(long maxSize)` returns a stream that truncates the original stream to be no longer than `maxSize`. It's like cutting off the bottom part of a log.

Here's an example:

```
List<Integer> list = List.of(1, 2, 3, 4, 5);
List<Integer> limitedList = list.stream()
        .limit(3)
        .collect(Collectors.toList());
System.out.println(limitedList);
```

This keeps only the first three elements and discards the rest:

```
[1, 2, 3]
```

You can combine `skip()` and `limit()` to extract a substream:

```
List<Integer> list = List.of(1, 2, 3, 4, 5);
List<Integer> subList = list.stream()
        .skip(1)
        .limit(3)
        .collect(Collectors.toList());
System.out.println(subList);
```

This skips the first element and then takes the next three:

```
[2, 3, 4]
```

Now, let's talk about `forEach()` and `forEachOrdered()`. These methods allow us to perform an action on each element of the stream.

`forEach(Consumer action)` performs the given action on each element. The order of processing is not guaranteed to be the encounter order if the stream is parallel.

```
List<Integer> list = List.of(1, 2, 3, 4, 5);
list.stream()
    .forEach(System.out::println);
```

This prints each element of the stream:

```
1
2
3
4
5
```

`forEachOrdered(Consumer action)` is similar, but it guarantees that the action is performed on the elements in the encounter order of the stream if it is a parallel stream:

```
List<Integer> list = List.of(1, 2, 3, 4, 5);
list.stream()
    .forEachOrdered(System.out::println);
```

This also prints each element, but ensuring the order:

```
1
2
3
4
5
```

The `allMatch()`, `anyMatch()`, and `noneMatch()` methods allow us to check if certain conditions hold for the elements of the stream.

`allMatch(Predicate predicate)` returns `true` if all elements satisfy the predicate, `false` otherwise:

```
List<Integer> list = List.of(2, 4, 6, 8, 10);
boolean allEven = list.stream()
        .allMatch(i -> i % 2 == 0);
System.out.println(allEven);
```

The above example checks if all elements are even:

```
true
```

`anyMatch(Predicate predicate)` returns `true` if any element satisfies the predicate, `false` otherwise:

```
List<Integer> list = List.of(1, 2, 3, 4, 5);
boolean anyEven = list.stream()
        .anyMatch(i -> i % 2 == 0);
System.out.println(anyEven);
```

This checks if any element is even:

```
true
```

`noneMatch(Predicate predicate)` returns `true` if no element satisfies the predicate, `false` otherwise:

```
List<Integer> list = List.of(1, 3, 5, 7, 9);
boolean noneEven = list.stream()
        .noneMatch(i -> i % 2 == 0);
System.out.println(noneEven);
```

This checks if no element is even:

```
true
```

The `findFirst()` and `findAny()` methods return an element of the stream, if one exists.

`findFirst()` returns an `Optional` describing the first element of the stream, or an empty `Optional` if the stream is empty:

```
List<Integer> list = List.of(1, 2, 3, 4, 5);
Optional<Integer> firstElem = list.stream()
        .findFirst();
System.out.println(firstElem.get());
```

This finds and prints the first element:

```
1
```

`findAny()` returns an `Optional` describing some element of the stream, or an empty `Optional` if the stream is empty. In parallel streams, it's useful when you don't care about the specific element, just that one exists:

```
List<Integer> list = List.of(1, 2, 3, 4, 5);
Optional<Integer> anyElem = list.parallelStream()
        .findAny();
System.out.println(anyElem.get());
```

The above example finds and prints any element (the specific element is not guaranteed due to the parallel processing):

```
3
```

Concatenating Streams

Sometimes, when working with streams, we need to combine them, merging their elements into a single stream. This is what we call concatenating streams, and there are several ways to achieve this in Java.

The most straightforward way to concatenate streams is by using the `concat()` method. This static method takes two streams as input and returns a new stream that is the concatenation of the two input streams:

```
static <T> Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b)
```

It's like joining two pipes, letting the water (elements) flow from one to the other.

Let's see an example:

```
Stream<Integer> stream1 = Stream.of(1, 2, 3);
Stream<Integer> stream2 = Stream.of(4, 5, 6);
Stream<Integer> concatenated = Stream.concat(stream1, stream2);
concatenated.forEach(System.out::println);
```

This concatenates `stream1` and `stream2`, and prints the elements of the resulting stream:

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

It's important to note that `concat()` is a static method and doesn't modify the original streams. Instead, it creates a new stream that lazily pulls elements from the first stream and then the second stream when requested.

Also, keep in mind that you can only concatenate streams of the same type. If you try to concatenate streams of different types, you'll get a compilation error.

Another way to concatenate streams is by using the `flatMap()` method in conjunction with `Stream.of()`.

`Stream.of()` creates a stream from a variable number of arguments. You can pass the streams you want to concatenate as arguments to `Stream.of()`, and then use `flatMap()` to flatten the resulting stream of streams into a single stream:

```
Stream<Integer> stream1 = Stream.of(1, 2, 3);
Stream<Integer> stream2 = Stream.of(4, 5, 6);
Stream<Integer> concatenated = Stream.of(stream1, stream2)
        .flatMap(stream -> stream);
concatenated.forEach(System.out::println);
```

This code does the same as the previous example, but using `flatMap()` and `Stream.of()`.

This approach is more verbose than using `concat()` directly, but it can be handy when you have a collection of streams that you want to concatenate.

For example, let's say you have a list of streams:

```
List<Stream<Integer>> listOfStreams = List.of(
        Stream.of(1, 2, 3),
        Stream.of(4, 5, 6),
        Stream.of(7, 8, 9)
);
```

You can concatenate all these streams into one using `flatMap()` and `Stream.of()`:

```
Stream<Integer> concatenated = listOfStreams.stream()
        .flatMap(stream -> stream);
concatenated.forEach(System.out::println);
```

This prints:

```
1
2
3
4
5
6
7
8
9
```

Here, we first create a stream from the `List` of streams using the `stream()` method. Then, we use `flatMap()` to flatten this stream of streams into a single stream.

It's like having a bunch of pipes and joining them all into one big pipe.

However, one thing to keep in mind when concatenating streams is the encounter order. The resulting stream will have the elements of the first stream followed by the elements of the second stream, and so on, in the order they were concatenated.

Reducing Streams

When working with streams, we often need to combine the elements in some way to produce a single result. This is what we call reducing a stream, and it's one of the most powerful operations in the Java Streams API.

The `reduce()` operation allows us to perform a reduction on the elements of the stream, using an associative accumulation function. It's like cooking a dish:

1. You start with a bunch of raw ingredients (the elements of the stream).
2. You apply a recipe (the accumulation function) to combine them.
3. You end up with a single cooked dish (the result of the reduction).

The `reduce()` method has three forms:

```
Optional<T> reduce(BinaryOperator<T> accumulator)

T reduce(T identity, BinaryOperator<T> accumulator)

<U> U reduce(U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner)
```

Let's start with the first one. This form of `reduce()` takes a single parameter: the accumulation function. This is a `BinaryOperator`, which means it's a function that takes two elements of the stream and combines them into one.

For example, let's say we have a stream of integers, and we want to find their sum:

```
Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5);
Optional<Integer> sum = stream.reduce((a, b) -> a + b);
System.out.println(sum.get());
```

This prints:

```
15
```

Here, the accumulation function `(a, b) -> a + b` takes two integers and returns their sum. The `reduce()` operation applies this function to the elements of the stream, two at a time, until all elements have been processed and a single result is obtained.

It's important to note that this form of `reduce()` returns an `Optional`. This is because the stream might be empty, in which case there would be no elements to reduce, and therefore no result to return. The `Optional` allows us to handle this case gracefully.

The second form of `reduce()` takes two parameters: an identity value and the accumulation function.

The identity value is the starting point of the reduction, and it's also the value that will be returned if the stream is empty. It's like the base ingredient in our cooking analogy.

```
Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5);
Integer sum = stream.reduce(0, (a, b) -> a + b);
System.out.println(sum);
```

This also prints:

```
15
```

But in this case, we start the reduction with 0, and we get a plain `Integer` as a result. This is not an `Optional` because, even if the stream is empty, we can still return the identity value.

The third form of `reduce()` is a bit more complex. It takes three parameters: an identity value, an accumulation function, and a combiner function.

The identity value and the accumulation function serve the same purposes as they do in the second form. The combiner function is used to combine the results of the reduction when the stream is processed in parallel.

This form of `reduce()` is useful for parallel processing, ensuring that the reduction operation is performed correctly across multiple threads.

For example, let's say we want to concatenate a stream of strings:

```
Stream<String> stream = Stream.of("a", "b", "c", "d", "e");
String concatenated = stream.reduce("", (a, b) -> a + b, (a, b) -> a + b);
System.out.println(concatenated);
```

This prints:

```
abcde
```

Here, the identity value is an empty string, the accumulation function concatenates two strings, and the combiner function also concatenates two strings.

In this case, the combiner function is necessary to ensure correctness in parallel processing, even though string concatenation is associative.

For example, suppose we want to calculate the sum of the lengths of a list of strings, but we want to give extra weight to strings that start with a vowel by doubling their length:

```
boolean startsWithVowel(String str) {
    return str.matches("^[AEIOUaeiou].*");
}

// ...

Stream<String> stream = Stream.of("apple", "banana", "orange", "grape", "pear");

int sumOfLengths = stream.reduce(0,
    (sum, str) -> sum + (startsWithVowel(str) ? str.length() * 2 : str.length()),
    Integer::sum);

System.out.println(sumOfLengths);
```

This code prints:

```
37
```

Here, the identity value is 0, the accumulation function adds either the doubled length of a string (if it starts with a vowel) or its normal length to the running sum, and the combiner function sums two intermediate results.

In this example, the combiner function `Integer::sum` is important for correctly combining partial sums when the stream is processed in parallel, ensuring that the final result is accurate regardless of the order of processing.

Collecting Results

After processing a stream, we often need to collect the results into a data structure for further use. This is where the `collect()` operation and the `Collectors` class come into play.

Using Basic Collectors

The `collect()` method is a terminal operation that allows us to accumulate the elements of a stream into a collection or other data structure. It takes a `Collector`, which specifies how the elements should be collected.

The `Collectors` class provides a wide variety of pre-defined collectors for common use cases. We've used `Collectors.toList()` in some of the previous examples, but let's look at some of these collectors in more detail.

The most straightforward collectors are `toList()` and `toSet()`, which collect the elements of the stream into a `List` or `Set`, respectively:

```
Stream<String> stream = Stream.of("cat", "dog", "elephant", "fox", "giraffe");
List<String> list = stream.collect(Collectors.toList());
System.out.println(list);
```

The above example prints:

```
[cat, dog, elephant, fox, giraffe]
```

If you need to collect into a specific type of collection, you can use `toCollection()` and provide a supplier for the collection:

```
Stream<String> stream = Stream.of("cat", "dog", "elephant", "fox", "giraffe");
LinkedList<String> linkedList = stream.collect(Collectors.toCollection(LinkedList::new));
System.out.println(linkedList);
```

This collects the elements into a `LinkedList`.

The `joining()` collector allows you to concatenate the elements of a stream into a single string, optionally with a delimiter, prefix, and suffix:

```
Stream<String> stream = Stream.of("cat", "dog", "elephant", "fox", "giraffe");
String joined = stream.collect(Collectors.joining(", "));
System.out.println(joined);
```

This prints:

```
cat, dog, elephant, fox, giraffe
```

There are also collectors for computing simple statistics about numeric streams, such as `counting()`, `summing()`, `averaging()`, and `summarizing()`:

```
Stream<Integer> stream1 = Stream.of(1, 2, 3, 4, 5);
long count = stream1.collect(Collectors.counting());
System.out.println(count);

Stream<Integer> stream2 = Stream.of(1, 2, 3, 4, 5);
double average = stream2.collect(Collectors.averagingInt(i -> i));
System.out.println(average);

Stream<Integer> stream3 = Stream.of(1, 2, 3, 4, 5);
int sum = stream3.collect(Collectors.summingInt(i -> i));
System.out.println(sum);

Stream<Integer> stream4 = Stream.of(1, 2, 3, 4, 5);
IntSummaryStatistics stats = stream4.collect(Collectors.summarizingInt(i -> i));
System.out.println(stats);
```

This is the output:

```
5
3.0
15
IntSummaryStatistics{count=5, sum=15, min=1, average=3.000000, max=5}
```

These collectors come in three flavors for the three primitive types: `int`, `long`, and `double`.

The `maxBy()` and `minBy()` collectors allow you to find the maximum and minimum elements according to a given `Comparator`:

```
Stream<String> stream = Stream.of("cat", "dog", "elephant", "fox", "giraffe");
Optional<String> max = stream.collect(Collectors.maxBy(Comparator.comparingInt(String::length)));
max.ifPresent(System.out::println);
```

This prints `"elephant"`, the longest string in the stream.

Collecting into Maps

One of the most powerful features of the `Collectors` class is the ability to collect elements into a `Map`.

The simplest way to do this is with the `toMap()` collector, which takes two functions: one to extract the key from each element, and one to extract the value:

```
Stream<String> stream = Stream.of("elephant", "fox", "giraffe");
Map<Integer, String> map = stream.collect(Collectors.toMap(String::length, s -> s));
System.out.println(map);
```

This collects the strings into a map, using their length as the key:

```
{3=fox, 7=giraffe, 8=elephant}
```

If there are duplicate keys, the `toMap()` collector will throw an exception. To handle this, you can provide a merge function as a third argument:

```
Stream<String> stream = Stream.of("cat", "elephant", "fox", "giraffe");
Map<Integer, String> map = stream.collect(Collectors.toMap(String::length, s -> s, (s1, s2) -> s1 + "," + s2));
System.out.println(map);
```

Now, if multiple strings have the same length, they will be joined with a comma. This is the output of the above example:

```
{3=cat,fox, 7=giraffe, 8=elephant}
```

Grouping, Partitioning, Mapping, and Teeing

The `groupingBy()` collector allows you to group the elements of a stream according to a classification function:

```
Stream<String> stream = Stream.of("cat", "dog", "elephant", "fox", "giraffe");
Map<Integer, List<String>> map = stream.collect(Collectors.groupingBy(String::length));
System.out.println(map);
```

This groups the strings by their length:

```
{3=[cat, dog, fox], 7=[giraffe], 8=[elephant]}
```

You can also provide a downstream collector to specify how the groups should be collected:

```
Stream<String> stream = Stream.of("cat", "dog", "elephant", "fox", "giraffe");
Map<Integer, Set<String>> map = stream.collect(Collectors.groupingBy(String::length, Collectors.toSet()));
```

This collects the groups into `Sets` instead of `Lists`.

The `partitioningBy()` collector is a special case of `groupingBy()` that partitions the stream into two groups according to a predicate:

```
Stream<String> stream = Stream.of("cat", "dog", "elephant", "fox", "giraffe");
Map<Boolean, List<String>> map = stream.collect(Collectors.partitioningBy(s -> s.length() > 5));
System.out.println(map);
```

This partitions the strings into those longer than 5 characters and those not longer than 5 characters. This is the result of the example above:

```
{false=[cat, dog, fox], true=[elephant, giraffe]}
```

The `mapping()` collector allows you to apply a function to each element before collecting the results:

```
Stream<String> stream = Stream.of("cat", "dog", "elephant", "fox", "giraffe");
List<Integer> list = stream.collect(Collectors.mapping(String::length, Collectors.toList()));
System.out.println(list);
```

This collects the lengths of the strings into a list. This is the result:

```
[3, 3, 8, 3, 7]
```

Finally, a powerful and less commonly known collector is the `Collectors.teeing()` collector. This collector allows you to perform two separate collection operations on a single stream and then combine their results using a merger function. This can be particularly useful when you need to perform two different operations on the same data set and then combine the outcomes in a meaningful way.

The general form of the `teeing()` method is as follows:

```
public static <T, R1, R2, R> Collector<T, ?, R> teeing(
    Collector<? super T, A1, R1> downstream1,
    Collector<? super T, A2, R2> downstream2,
```

```
    BiFunction<? super R1, ? super R2, R> merger
)
```

It takes three arguments: 1. **downstream1**: The first collector to apply. 2. **downstream2**: The second collector to apply. 3. **merger**: A function that merges the results of the two collectors.

For example, let's say you have a list of integers and we want to calculate both the sum and the count of the integers in one pass through the stream, and then combine these results into a single result.

Here's how you can achieve this:

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5);
var result = numbers.stream().collect(Collectors.teeing(
    Collectors.summingInt(Integer::intValue),  // First collector: Sum of the integers
    Collectors.counting(),                      // Second collector: Count of the integers
    (sum, count) -> String.format("Sum: %d, Count: %d", sum, count)  // Merger function
));

System.out.println(result);
```

This is the output:

```
Sum: 15, Count: 5
```

As you can see, this collector simplifies the code for complex aggregation tasks by eliminating the need for multiple passes over the stream. You can use any combination of collectors, and the merger function allows for flexible combination of the results.

Key Points

- The `Optional` class is used to encapsulate an optional value and avoid `null` references. It provides methods like `isPresent()`, `ifPresent()`, `get()`, `orElse()`, `orElseGet()`, and `orElseThrow()` to work with the contained value.
- Streams are wrappers for collections or arrays that allow operations to be expressed with lambdas. They don't store elements, are immutable and not reusable, don't support indexed access, are easily parallelizable, and defer execution until needed.
- Streams can be created from collections using `stream()`, from individual values using `Stream.of()`, from arrays using `Arrays.stream()`, and in other ways like `generate()`, `iterate()`, and `range()`.
- Intermediate stream operations always return a new stream and are lazy, only processing elements when a terminal operation is invoked. They can be stateless (like `filter()` and `map()`) or stateful (like `distinct()` and `sorted()`).
- Terminal operations return something other than a stream and consume the stream pipeline. They include `forEach()`, `count()`, `collect()`, `findFirst()`, `findAny()`, `anyMatch()`, `allMatch()`, and `noneMatch()`.
- Primitive streams `IntStream`, `LongStream` and `DoubleStream` avoid boxing/unboxing overhead. They have methods like `average()`, `max()`, `min()`, `sum()`, `range()`, and `summaryStatistics()`.
- Short-circuit operations like `limit()`, `findFirst()` and `anyMatch()` allow streams to avoid processing all elements by producing a result as soon as enough elements have been processed.
- The `filter()` method is used to select only the elements of a stream that satisfy a given predicate. It returns a new stream containing only the filtered elements.
- The `distinct()` method returns a stream of unique elements, discarding duplicates. It can be thought of as a special filtering operation.

- The `takeWhile()` method returns a stream that contains the longest prefix of elements that match a given predicate, while `dropWhile()` discards this prefix and returns the remaining elements.
- The `map()` method transforms each element of a stream into a new element by applying a function. It returns a new stream of the transformed elements.
- The `flatMap()` method is used to flatten a stream of collections into a single stream of elements. It applies a function that returns a stream to each element, and then flattens all these streams into one.
- Primitive streams (`IntStream`, `LongStream`, `DoubleStream`) have specialized mapping operations to avoid boxing and unboxing costs.
- The `skip()` method discards the first n elements of a stream, while `limit()` truncates a stream to be no longer than a specified size.
- The `forEach()` method performs an action on each element of a stream, while `forEachOrdered()` does the same but guarantees the order of processing for parallel streams.
- The `allMatch()`, `anyMatch()`, and `noneMatch()` methods check if certain conditions hold for the elements of a stream.
- The `findFirst()` method returns the first element of a stream, while `findAny()` returns any element (useful for parallel streams).
- The `concat()` method concatenates two streams into a single stream. Alternatively, `flatMap()` can be used with `Stream.of()` to concatenate multiple streams.
- The `reduce()` method performs a reduction on the elements of a stream using an associative accumulation function. It can return an `Optional` result, or accept an identity value to return a non-optional result.
- The `collect()` method is used to accumulate the elements of a stream into a collection or other data structure, using a `Collector` to specify how the elements should be collected.
- The `Collectors` class provides a variety of predefined collectors, including `toList()`, `toSet()`, `toMap()`, `joining()`, `counting()`, `summing()`, `averaging()`, `maxBy()`, `minBy()`, `groupingBy()`, `partitioningBy()`, `mapping()`, and `teeing()`.

Practice Questions

1. Which of the following lines of code demonstrates the use of the `Optional` class to handle a potentially `null` value to avoid an exception?

```
import java.util.Optional;

public class Main {
    public static void main(String[] args) {
        String value = getValue();
        // Insert code here
    }

    public static String getValue() {
        return null; // This method may return null
    }
}
```

A) `Optional<String> optional = new Optional<>(value);`
B) `Optional<String> optional = Optional.of(value);`
C) `Optional<String> optional = Optional.ofNullable(value);`
D) `Optional<String> optional = Optional.empty(value);`
E) `Optional<String> optional = Optional.nullable(value);`

2. Which of the following lines of code correctly demonstrates the use of a terminal operation?

```
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class Main {
    public static void main(String[] args) {
        List<String> list = List.of("apple", "banana", "cherry", "date");

        Stream<String> stream = list.stream()
                                    .filter(s -> s.length() > 5)
                                    .peek(System.out::println)
                                    .map(String::toUpperCase);

        // Insert terminal operation here
    }
}
```

A) `stream.filter(s -> s.contains("A"));`
B) `stream.map(String::toLowerCase);`
C) `stream.distinct();`
D) `stream.limit(2);`
E) `stream.collect(Collectors.toList());`

3. Which of the following lines of code correctly uses a primitive stream to calculate the sum of an array of integers?

```
import java.util.stream.IntStream;

public class Main {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3, 4, 5};

        // Insert code here to calculate sum
    }
}
```

A) `int sum = numbers.stream().sum();`
B) `int sum = IntStream.range(0, numbers.length).sum();`
C) `int sum = IntStream.from(numbers).sum();`
D) `int sum = IntStream.of(numbers).sum();`
E) `int sum = IntStream.range(numbers).sum();`

4. Which of the following lines of code correctly filters a stream to include only strings with a length greater than 3?

```
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class Main {
    public static void main(String[] args) {
        List<String> list = List.of("one", "two", "three", "four");

        Stream<String> stream = list.stream();
```

```
        // Insert code here to filter the stream
    }
}
```

A) `Stream<String> filteredStream = stream.filter(s -> s.length() > 3);`
B) `Stream<String> filteredStream = stream.map(s -> s.length() > 3);`
C) `Stream<String> filteredStream = stream.collect(Collectors.filtering(s -> s.length() > 3));`
D) `Stream<String> filteredStream = stream.filtering(s -> s.length() > 3);`
E) `Stream<String> filteredStream = stream.filterByLength(3);`

5. Which of the following lines of code correctly maps a stream of strings to their lengths?

```
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class Main {
    public static void main(String[] args) {
        List<String> list = List.of("apple", "banana", "cherry", "date");

        Stream<String> stream = list.stream();

        // Insert code here to map the stream
    }
}
```

A) `Stream<String> lengthStream = stream.map(s -> s.length());`
B) `Stream<String> lengthStream = stream.mapToInt(s -> s.length());`
C) `Stream<Integer> lengthStream = stream.map(s -> s.length());`
D) `IntStream lengthStream = stream.map(s -> s.length());`
E) `Stream<String> lengthStream = stream.flatMap(s -> Stream.of(s.length()));`

6. Which of the following lines of code correctly limits the stream to the first 3 elements after skipping the first 2 elements?

```
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class Main {
    public static void main(String[] args) {
        List<String> list = List.of("one", "two", "three", "four", "five", "six");

        Stream<String> stream = list.stream();

        // Insert code here to skip and limit the stream
    }
}
```

A) `Stream<String> resultStream = stream.skip(2).limit(3);`
B) `Stream<String> resultStream = stream.limit(3).skip(2);`
C) `Stream<String> resultStream = stream.skip(3).limit(2);`
D) `Stream<String> resultStream = stream.limit(2).skip(3);`
E) `Stream<String> resultStream = stream.slice(2, 5);`

7. Which of the following lines of code correctly concatenates two streams?

```
import java.util.List;
```

```
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class Main {
    public static void main(String[] args) {
        List<String> list1 = List.of("one", "two", "three");
        List<String> list2 = List.of("four", "five", "six");

        Stream<String> stream1 = list1.stream();
        Stream<String> stream2 = list2.stream();

        // Insert code here to concatenate the streams
    }
}
```

A) `Stream<String> resultStream = Stream.concat(stream1, stream2.collect(Collectors.toList()));`
B) `Stream<String> resultStream = Stream.concat(stream1, stream2);`
C) `Stream<String> resultStream = stream1.concat(stream2);`
D) `Stream<String> resultStream = stream1.merge(stream2);`
E) `Stream<String> resultStream = Stream.of(stream1, stream2);`

8. Which of the following lines of code uses the reduce method to correctly calculate the product of all elements in a stream of integers?

```
import java.util.List;
import java.util.stream.Stream;

public class Main {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(1, 2, 3, 4, 5);

        Stream<Integer> stream = numbers.stream();

        // Insert code here to calculate the product
    }
}
```

A) `int product = stream.reduce(1, (a, b) -> a + b);`
B) `int product = stream.reduce((a, b) -> a * b);`
C) `int product = stream.reduce(0, (a, b) -> a * b);`
D) `Optional<Integer> product = stream.reduce(1, (a, b) -> a * b);`
E) `int product = stream.reduce(1, (a, b) -> a * b, (a, b) -> a * b);`

9. Which of the following lines of code correctly collects the elements of a stream into a `Set` and also ensures that the original order of the elements is maintained?

```
import java.util.List;
import java.util.Set;
import java.util.stream.Collectors;
import java.util.stream.Stream;
import java.util.LinkedHashSet;

public class Main {
    public static void main(String[] args) {
        List<String> list = List.of("apple", "banana", "cherry", "date");

        Stream<String> stream = list.stream();
```

```
        // Insert code here to collect the elements into a Set while maintaining order
    }
}
```

A) `Set<String> resultSet = stream.collect(Collectors.toSet());`
B) `Set<String> resultSet = stream.collect(Collectors.toCollection(LinkedHashSet::new));`
C) `Set<String> resultSet = stream.collect(Collectors.toCollection(TreeSet::new));`
D) `Set<String> resultSet = stream.collect(Collectors.toList());`
E) `Set<String> resultSet = stream.collect(Collectors.toMap());`

Chapter NINE

Streams

Answers

1. The correct answer is C.

Explanation:

- **A)** `Optional<String> optional = new Optional<>(value);`
  - This option is incorrect because `Optional` does not have a public constructor. Instead, static factory methods like `of` and `ofNullable` should be used.
- **B)** `Optional<String> optional = Optional.of(value);`
  - This option is incorrect because `Optional.of(value)` throws a `NullPointerException` if `value` is `null`. In this scenario, since `getValue()` can return `null`, this line could lead to an exception.
- **C)** `Optional<String> optional = Optional.ofNullable(value);`
  - This option is correct because `Optional.ofNullable(value)` will return an `Optional` describing the specified value if non-null, or an empty `Optional` if the value is `null`. This is the appropriate way to handle a potentially `null` value.
- **D)** `Optional<String> optional = Optional.empty(value);`
  - This option is incorrect because `Optional.empty()` does not accept any arguments. It simply returns an empty `Optional`.
- **E)** `Optional<String> optional = Optional.nullable(value);`
  - This option is incorrect because there is no method `nullable` in the `Optional` class. The correct method for this purpose is `ofNullable`.

2. The correct answer is E.

Explanation:

- **A)** `stream.filter(s -> s.contains("A"));`
  - This option is incorrect because `filter` is an intermediate operation. It returns a new stream with elements that match the given predicate.
- **B)** `stream.map(String::toLowerCase);`
  - This option is incorrect because `map` is an intermediate operation. It returns a new stream with elements that are the results of applying the given function.
- **C)** `stream.distinct();`
  - This option is incorrect because `distinct` is an intermediate operation. It returns a new stream with distinct elements.
- **D)** `stream.limit(2);`
  - This option is incorrect because `limit` is an intermediate operation. It returns a new stream that is truncated to be no longer than the given size.
- **E)** `stream.collect(Collectors.toList());`
  - This option is correct because `collect` is a terminal operation. It triggers the processing of the stream and collects the elements into a `List`.

3. The correct answer is D.

Explanation:

- **A)** `int sum = numbers.stream().sum();`
  - This option is incorrect because arrays do not have a `stream` method directly on them. You need to use a method from a utility class like `IntStream` to create a stream.
- **B)** `int sum = IntStream.range(0, numbers.length).sum();`
  - This option is incorrect because `IntStream.range(0, numbers.length)` generates a stream of integers from 0 to the length of the array, not the elements of the array itself.
- **C)** `int sum = IntStream.from(numbers).sum();`
  - This option is incorrect because `IntStream` does not have a `from` method. The correct method is `of`.
- **D)** `int sum = IntStream.of(numbers).sum();`
  - This option is correct because `IntStream.of(numbers).sum()` correctly creates an `IntStream` from the array and calculates the sum of its elements.
- **E)** `int sum = IntStream.range(numbers).sum();`
  - This option is incorrect because `IntStream.range` requires two arguments (a start and an end index) and is used to generate a stream of numbers within a range, not to sum an array.

4. The correct answer is A.

Explanation:

- **A)** `Stream<String> filteredStream = stream.filter(s -> s.length() > 3);`
  - This option is correct because `filter` is the correct intermediate operation to apply a predicate to each element of the stream and return a new stream containing only elements that match the predicate.
- **B)** `Stream<String> filteredStream = stream.map(s -> s.length() > 3);`
  - This option is incorrect because `map` is used to transform elements of the stream and does not filter them. The result would be a stream of `Boolean` values instead of the original strings.
- **C)** `Stream<String> filteredStream = stream.collect(Collectors.filtering(s -> s.length() > 3));`
  - This option is incorrect because `Collectors.filtering` is not a valid method. Filtering is done through the `filter` method on the stream itself, not via collectors.
- **D)** `Stream<String> filteredStream = stream.filtering(s -> s.length() > 3);`
  - This option is incorrect because there is no `filtering` method on the stream. The correct method is `filter`.
- **E)** `Stream<String> filteredStream = stream.filterByLength(3);`
  - This option is incorrect because there is no `filterByLength` method on the stream. The correct method to use is `filter`.

5. The correct answer is C.

Explanation:

- **A)** `Stream<String> lengthStream = stream.map(s -> s.length());`
  - This option is incorrect because the `map` method will transform the elements to `Integer`, not `String`. The correct type for the resulting stream should be `Stream<Integer>`.
- **B)** `Stream<String> lengthStream = stream.mapToInt(s -> s.length());`
  - This option is incorrect because `mapToInt` produces an `IntStream`, not a `Stream<String>`. Additionally, the resulting stream type would not be `Stream<String>`.
- **C)** `Stream<Integer> lengthStream = stream.map(s -> s.length());`
  - This option is correct because `map` transforms each string in the stream to its length, resulting in a `Stream<Integer>`.
- **D)** `IntStream lengthStream = stream.map(s -> s.length());`
  - This option is incorrect because `map` produces a `Stream<R>`, not an `IntStream`. The correct method for producing an `IntStream` would be `mapToInt`.
- **E)** `Stream<String> lengthStream = stream.flatMap(s -> Stream.of(s.length()));`

- This option is incorrect because `flatMap` is used to flatten nested streams and not simply map to another type. Additionally, the resulting stream type would not be `Stream<String>`.

6. The correct answer is A.

Explanation:

- **A)** `Stream<String> resultStream = stream.skip(2).limit(3);`
  - This option is correct because `skip(2)` skips the first 2 elements of the stream, and `limit(3)` limits the stream to the next 3 elements. Therefore, the resulting stream will contain the 3rd, 4th, and 5th elements of the original list.
- **B)** `Stream<String> resultStream = stream.limit(3).skip(2);`
  - This option is incorrect because `limit(3)` first limits the stream to the first 3 elements, and then `skip(2)` skips 2 of those elements, resulting in a stream with only the 3rd element.
- **C)** `Stream<String> resultStream = stream.skip(3).limit(2);`
  - This option is incorrect because `skip(3)` skips the first 3 elements, and `limit(2)` then limits the stream to the next 2 elements, resulting in a stream with the 4th and 5th elements.
- **D)** `Stream<String> resultStream = stream.limit(2).skip(3);`
  - This option is incorrect because `limit(2)` first limits the stream to the first 2 elements, and then `skip(3)` would attempt to skip more elements than are available, resulting in an empty stream.
- **E)** `Stream<String> resultStream = stream.slice(2, 5);`
  - This option is incorrect because there is no `slice` method in the Stream API. The correct methods to achieve the desired result are `skip` and `limit`.

7. The correct answer is B.

Explanation:

- **A)** `Stream<String> resultStream = Stream.concat(stream1, stream2.collect(Collectors.toList()));`
  - This option is incorrect because `Stream.concat` expects two streams as arguments. `stream2.collect(Collectors.toL` converts `stream2` into a `List`, not a `Stream`.
- **B)** `Stream<String> resultStream = Stream.concat(stream1, stream2);`
  - This option is correct because `Stream.concat(stream1, stream2)` correctly concatenates the two streams into a single stream containing all elements from both streams.
- **C)** `Stream<String> resultStream = stream1.concat(stream2);`
  - This option is incorrect because `Stream` does not have an instance method `concat`. The `concat` method is a static method of the `Stream` class.
- **D)** `Stream<String> resultStream = stream1.merge(stream2);`
  - This option is incorrect because there is no `merge` method in the `Stream` API. The correct method for concatenating streams is `Stream.concat`.
- **E)** `Stream<String> resultStream = Stream.of(stream1, stream2);`
  - This option is incorrect because `Stream.of(stream1, stream2)` creates a stream of streams, resulting in `Stream<Stream<String>>` rather than a single concatenated `Stream<String>`.

8. The correct answer is E.

Explanation:

- **A)** `int product = stream.reduce(1, (a, b) -> a + b);`
  - This option is incorrect because the reduction operation is using addition instead of multiplication. The correct operation for calculating the product should be `(a, b) -> a * b`.
- **B)** `int product = stream.reduce((a, b) -> a * b);`
  - This option is incorrect because it does not provide an identity value, which is necessary for the reduction operation when dealing with an empty stream. Without an identity value, the result is an `Optional<Integer>` rather than an `int`.
- **C)** `int product = stream.reduce(0, (a, b) -> a * b);`
  - This option is incorrect because the identity value for multiplication should be `1`, not `0`. Using `0` as the identity value would result in a product of `0` regardless of the stream elements.

- **D)** `Optional<Integer> product = stream.reduce(1, (a, b) -> a * b);`
  - This option is incorrect because the correct use of the `reduce` method with an identity value does not return an `Optional`. It should return the result directly as `int`.
- **E)** `int product = stream.reduce(1, (a, b) -> a * b, (a, b) -> a * b);`
  - This option is correct because it correctly uses the `reduce` method with an identity value of `1` and a combiner function that multiplies the results. This form of `reduce` is suitable for parallel processing as well, ensuring the product is correctly calculated across multiple segments of the stream.

9. The correct answer is B.

Explanation:

- **A)** `Set<String> resultSet = stream.collect(Collectors.toSet());`
  - This option is incorrect because `Collectors.toSet()` does not guarantee the order of the elements. The implementation returned by this collector does not preserve the order of insertion.
- **B)** `Set<String> resultSet = stream.collect(Collectors.toCollection(LinkedHashSet::new));`
  - This option is correct because `Collectors.toCollection(LinkedHashSet::new)` collects the elements into a `LinkedHashSet`, which maintains the order of insertion.
- **C)** `Set<String> resultSet = stream.collect(Collectors.toCollection(TreeSet::new));`
  - This option is incorrect because `TreeSet` sorts the elements according to their natural ordering (or by a comparator, if provided). This does not necessarily preserve the original order of the stream elements.
- **D)** `Set<String> resultSet = stream.collect(Collectors.toList());`
  - This option is incorrect because `Collectors.toList()` collects the elements into a `List`, not a `Set`.
- **E)** `Set<String> resultSet = stream.collect(Collectors.toMap());`
  - This option is incorrect because `Collectors.toMap()` is used to collect the elements into a `Map`, not a `Set`.

Chapter TEN

Concurrency and Multithreading

Chapter Content

- Introducing Threads
- Virtual Threads
  - Characteristics of Virtual Threads
  - Creating Virtual Threads
  - Virtual Threads vs Platform Threads
- Threading Problems
  - Deadlock
  - Starvation
  - Livelock
  - Race Conditions
- Writing Thread-Safe Code
  - Accessing Data with `volatile`
  - Protecting Data with Atomic Classes
  - Synchronized Blocks
  - Synchronizing on Methods
  - The `Lock` Interface
  - The `CyclicBarrier` Class
- The Concurrency API
  - The `ExecutorService` Interface
  - Submitting Tasks

  - The `Callable` interface
  - Scheduling Tasks
  - Executors Factory Methods
  - Virtual Thread-Aware Executor
- Concurrent Collections
- Parallel Streams
  - Creating Parallel Streams
  - Parallel Decomposition
  - Methods of Stream that Perform Order-Based Tasks
  - Reducing Parallel Streams
  - Combining Results in Parallel Streams
- Key Points
- Practice Questions

Introducing Threads

Threads allow multiple paths of execution to occur concurrently within a single program. Each thread represents a separate path of execution, allowing different parts of the code to run simultaneously. You can think of threads like lanes on a highway. Just as multiple lanes allow many cars to drive down the road simultaneously, multiple threads allow different segments of code to execute concurrently within the same application. However, just as cars in different lanes need to coordinate when merging or exiting, threads must coordinate carefully when accessing shared resources to avoid conflicts.

In Java 21, there are two types of threads: 1. **Platform threads:** These are the traditional threads that are mapped directly to operating system threads. 2. **Virtual threads:** These are lightweight threads managed by the Java Virtual Machine (JVM).

Let's start by looking at platform threads, which have been around since the early days of Java.

To create a new platform thread, you can extend the `Thread` class or implement the `Runnable` interface. When extending `Thread`, you override the `run()` method to define the code that will execute in the new thread:

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

To launch the new thread, create an instance of the class and call its `start()` method:

```
MyThread myThread = new MyThread();
myThread.start();
```

The `start()` method initiates a new thread that executes the code defined in `run()`. Alternatively, you can create a new thread by implementing `Runnable`:

```
@FunctionalInterface
public interface Runnable {
    public abstract void run();
}
```

And passing an instance to the `Thread` constructor:

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

```
MyRunnable myRunnable = new MyRunnable();
Thread myThread = new Thread(myRunnable);
myThread.start();
```

However, in Java 21 you can use the `Thread.Builder.OfPlatform` class to create platform threads. Here's a simple example:

```
// Create a Runnable task
Runnable task = () -> {
   System.out.println("Platform thread is running");
};

// Create a platform thread using Thread.Builder.OfPlatform
Thread platformThread = Thread.ofPlatform().start(task);

// Wait for the thread to finish
try {
   platformThread.join();
} catch (InterruptedException e) {
   e.printStackTrace();
}
```

In this example: - A `Runnable` task is defined, which simply prints a message to the console. - The `Thread.ofPlatform().start(task)` method is used to create and start a new platform thread with the given task. - The `join()` method is called on the thread to wait for it to finish execution.

You can customize the platform thread by setting its name, priority, and other properties using the builder pattern. Here's an example:

```
// Create a Runnable task
Runnable task = () -> {
   System.out.println("Custom platform thread is running");
};

// Create and customize a platform thread
Thread platformThread = Thread.ofPlatform()
                             .name("CustomThread", 0)
                             .priority(Thread.MAX_PRIORITY)
                             .unstarted(task);

// Start the thread
platformThread.start();

// Wait for the thread to finish
try {
   platformThread.join();
} catch (InterruptedException e) {
   e.printStackTrace();
}
```

In this example: - The `name("CustomThread", 0)` method sets the name of the thread to `"CustomThread"` with a unique number appended if necessary. - The `priority(Thread.MAX_PRIORITY)` method sets the priority of the thread to the maximum value. - The `unstarted(task)` method creates the thread but does not start it immediately. You need to call `start()` to begin execution.

On the other hand, you can use the `Thread.Builder.OfVirtual` class to create a virtual thread:

```
Thread vThread = Thread.ofVirtual().start(() -> {
    System.out.println("Hello from a virtual thread!");
});
vThread.join();
```

For platform threads, Java distinguishes between daemon and non-daemon threads. Daemon threads are those that do not prevent the JVM from exiting when the program finishes. They run in the background and are typically used for tasks like garbage collection, background cleanup, etc. The JVM will continue running as long as there is at least one active non-daemon thread. Daemon threads are terminated when all non-daemon threads complete. To make a thread a daemon, call its `setDaemon(true)` method before starting it:

```
public class DaemonThreadExample {
    public static void main(String[] args) {
        Thread daemonThread = new Thread(() -> {
            while (true) {
                System.out.println("Daemon thread is running");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });

        daemonThread.setDaemon(true);
        daemonThread.start();

        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Main thread exiting");
    }
}
```

In this example, we create a daemon thread using a lambda expression. The daemon thread runs in an infinite loop, printing a message every second. We set the thread to be a daemon by calling `setDaemon(true)` before starting it.

The main thread sleeps for 5 seconds and then continues its execution. When the main thread (which is a non-daemon thread) terminates, the JVM will automatically terminate the daemon thread.

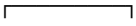
You can also use `Thread.ofPlatform()` to set the daemon status:

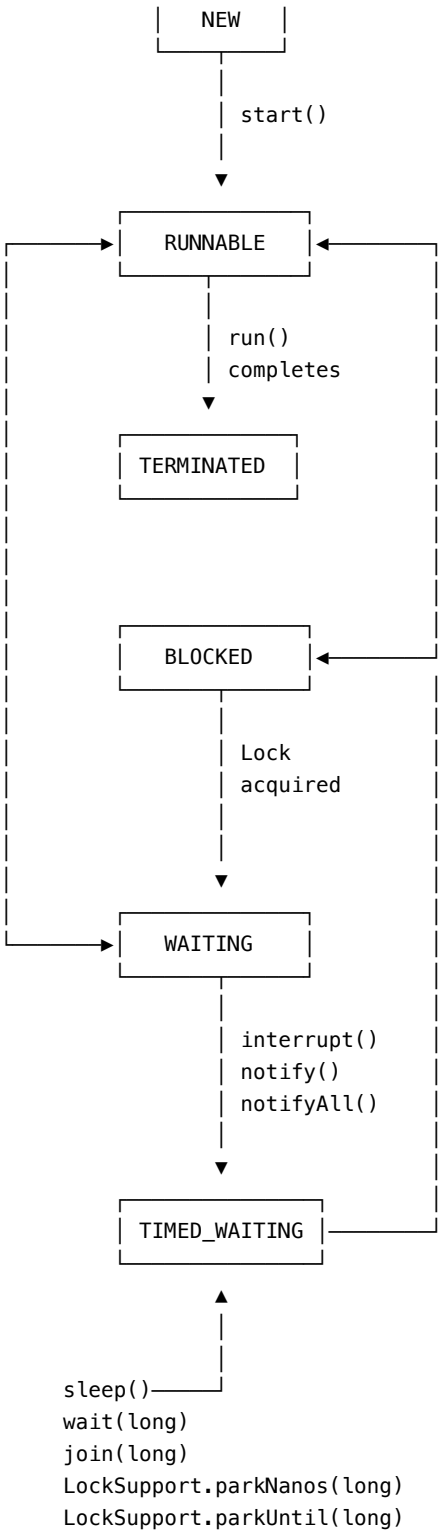
```
Thread platformThread = Thread.ofPlatform()
                                    .name("CustomThread", 0)
                                    .daemon(false)
```

The default implementation invokes `daemon(boolean)` with a value of `true`.

On the other hand, virtual threads are always daemon threads, so they do not prevent the JVM from exiting when the program finishes.

A thread progresses through several states during its life cycle:





This lifecycle applies to both platform and virtual threads, although the internal management of these states differs between the two types.

The static `Thread.sleep(long millis)` method causes the current thread to suspend execution for the specified number of milliseconds:

```
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    // Handle interruption
}
```

To prematurely wake a sleeping or waiting thread, you can call its `interrupt()` method. This will throw an `InterruptedException` in the target thread, which must be handled:

```
public class InterruptExample {
    public static void main(String[] args) {
        Thread thread = Thread.ofVirtual().start(() -> {
            try {
                System.out.println("Thread is going to sleep");
                Thread.sleep(5000);
                System.out.println("Thread woke up");
            } catch (InterruptedException e) {
                System.out.println("Thread was interrupted");
            }
        });

        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        thread.interrupt();

        try {
            // Ensures the virtual thread completes before the main thread exits
            thread.join();
        }  catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

In this example, we create a virtual thread that goes to sleep for 5 seconds. The main thread sleeps for 2 seconds and then calls `interrupt()` on the other thread.

When `interrupt()` is called, it sets the interrupted status of the target thread. If the target thread is sleeping or waiting, it will immediately throw an `InterruptedException`. The thread can then handle the interruption appropriately. Also, after calling `thread.interrupt()`, we call `thread.join()` to ensure the main thread waits for the virtual thread to complete.

In the example, the output will be:

```
Thread is going to sleep
Thread was interrupted
```

The thread's sleep is prematurely interrupted after 2 seconds, and it catches the `InterruptedException` and prints a message.

Virtual Threads

Characteristics of Virtual Threads

The key characteristic of virtual threads is their lightweight nature. Unlike platform threads, which are mapped directly to OS threads, virtual threads are managed by the JVM, which maps a large number of virtual threads to a small number of OS threads.

When a virtual thread needs to run, the Java runtime attaches it to a regular thread (called a *carrier thread*). This is like giving the virtual thread a temporary vehicle to move around in.

The operating system then schedules this platform thread as it normally would. While attached to this carrier, the virtual thread can execute its code.

If the virtual thread needs to perform an operation that might take a while, like reading from a file or waiting for a network response, it can detach from its carrier. This is called unmounting. When this happens, the carrier becomes available, and the JVM can use it to run a different virtual thread.

However, there are some situations where a virtual thread can't detach from its carrier. This is called being *pinned* to the carrier. Two common scenarios where this happens are: 1. When the virtual thread is using a `synchronized` block or method. These are used to control access to shared resources, and they need to stay on the same carrier to work properly.

2. When the virtual thread is running code that interacts directly with the operating system or other low-level functions (native methods or foreign functions). These also need to stay on the same carrier because they're tightly connected to the underlying system.

In these pinned situations, the virtual thread has to stay attached to its carrier until it's done with these special operations. This means the carrier can't be used for other virtual threads during this time, which can potentially reduce some of the efficiency benefits of virtual threads.

For this reason, virtual threads are particularly effective for I/O-bound tasks. The JVM can suspend the virtual thread and free up the platform thread to do other work. This is efficient because I/O operations often involve waiting for external resources, during which time the CPU isn't actively engaged.

However, virtual threads are not intended for long-running CPU-intensive operations. These operations actively use the CPU for extended periods. When a virtual thread is performing a CPU-intensive task, it's continuously using the platform thread it's running on. In this case, there's no opportunity for the runtime to suspend the virtual thread and reassign the platform thread elsewhere, because the CPU is constantly busy with the intensive computation.

Additionally, virtual threads have a fixed priority that cannot be changed. This design choice simplifies the scheduling of virtual threads, as they are intended to be used for parallelism in a more straightforward manner compared to platform threads.

Virtual threads do not have a thread name by default. The `getName` method returns the empty string if a thread name is not set:

```
// Create a virtual thread without setting its name
Thread virtualThread = Thread.ofVirtual().start(() -> {
    System.out.println("Running in a virtual thread");
});

// Check and print the name of the virtual thread
String threadName = virtualThread.getName();
if (threadName.isEmpty()) {
    System.out.println("The virtual thread has no name.");
} else {
    System.out.println("The virtual thread name is: " + threadName);
}
```

In this example, the `virtualThread` is created without setting its name, so calling `getName` on it returns an empty string. The output will be:

```
Running in a virtual thread
The virtual thread has no name.
```

You can set the name of a virtual thread by using the `name` method of the `Thread.Builder.OfVirtual` class:

```
// Create a virtual thread with a specific name
Thread virtualThread = Thread.ofVirtual()
    .name("MyVirtualThread")
    .start(() -> {
        System.out.println("Running in a virtual thread");
    });

// Check and print the name of the virtual thread
String threadName = virtualThread.getName();
if (threadName.isEmpty()) {
    System.out.println("The virtual thread has no name.");
} else {
    System.out.println("The virtual thread name is: " + threadName);
}
```

In this example, the `virtualThread` is created with the name `"MyVirtualThread"` using the `name` method. The output will be:

```
Running in a virtual thread
The virtual thread name is: MyVirtualThread
```

Another important feature of virtual threads is their ability to simplify concurrent programming. With virtual threads, you can write straightforward, sequential-looking code that actually runs concurrently. This can make your code easier to read and maintain, as you don't need to explicitly manage thread pools or use complex asynchronous programming models.

Creating Virtual Threads

Java 21 introduces several ways to create and use virtual threads. Let's explore these methods in detail.

The primary way to create a virtual thread is by using the `Thread.ofVirtual()` method. This method returns a `Thread.Builder` that can be used to configure and start a virtual thread. Here's an example:

```
Thread vThread = Thread.ofVirtual().start(() -> {
    System.out.println("Virtual thread is running");
});

try {
    vThread.join();
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

In this example, we create and start a virtual thread in one line. The `start()` method takes a `Runnable` and immediately starts the thread.

If you want to create a virtual thread without starting it immediately, you can use the `unstarted()` method instead:

```
Thread vThread = Thread.ofVirtual().unstarted(() -> {
    System.out.println("Virtual thread is running");
});
```

```
vThread.start();
try {
    vThread.join();
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

Another convenient method for creating and starting a virtual thread is `Thread.startVirtualThread(Runnable task)`. This static method creates a virtual thread, starts it, and returns the `Thread` object:

```
Thread vThread = Thread.startVirtualThread(() -> {
    System.out.println("Virtual thread created with startVirtualThread");
});

try {
    vThread.join();
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

For cases where you need to create multiple virtual threads with the same configuration, you can use a `java.util.concurrent.ThreadFactory`. The `Thread.ofVirtual().factory()` method returns a `ThreadFactory` that creates virtual threads:

```
ThreadFactory virtualThreadFactory = Thread.ofVirtual().factory();

for (int i = 0; i < 10; i++) {
    Thread vThread = virtualThreadFactory.newThread(() -> {
        System.out.println("Virtual thread " + Thread.currentThread().threadId() + " is running");
    });
    vThread.start();
}
```

This example creates a `ThreadFactory` for virtual threads and uses it to create and start 10 virtual threads.

You can also customize the `ThreadFactory` to set names for the virtual threads it creates:

```
ThreadFactory namedVirtualThreadFactory = Thread.ofVirtual().name("worker-", 0).factory();

for (int i = 0; i < 5; i++) {
    Thread vThread = namedVirtualThreadFactory.newThread(() -> {
        System.out.println(Thread.currentThread().getName() + " is running");
    });
    vThread.start();
}
```

This will create virtual threads with names like `worker-0`, `worker-1`, and so on.

It's worth noting that while virtual threads are very lightweight, they're not free. Creating millions of virtual threads that do nothing but immediately terminate is still a non-trivial operation. In practice, you should create virtual threads as needed for actual concurrent tasks, rather than creating a huge number upfront.

Virtual Threads vs Platform Threads

Here's quick comparison of virtual threads and platform threads:

Characteristic	Virtual Threads	Platform Threads
Management	Managed by JVM	Managed by OS
Resource usage	Very lightweight	Heavier, limited by OS
Scalability	Can create millions	Typically limited to thousands
Blocking behavior	Automatically yields carrier thread	Blocks OS thread
Use case	Ideal for I/O-bound tasks	Better for CPU-bound tasks
Stack size	Grows and shrinks as needed	Fixed size
Thread-local variables	Should be used cautiously	Can be used freely

Virtual threads excel in scenarios with many concurrent, mostly-idle tasks. For example, in a web server handling many simultaneous connections, each connection could be handled by its own virtual thread. This allows for simple, synchronous-style code that scales extremely well.

Here's an example that demonstrates the scalability difference:

```
long start = System.currentTimeMillis();

List<FutureTask<Integer>> tasks = new ArrayList<>();
ThreadFactory threadFactory = Thread.ofVirtual().factory();

for (int i = 0; i < 100_000; i++) {
    int taskId = i;
    FutureTask<Integer> task = new FutureTask<>(() -> {
        Thread.sleep(Duration.ofSeconds(1));
        return taskId;
    });
    tasks.add(task);
    threadFactory.newThread(task).start();
}

for (FutureTask<Integer> task : tasks) {
    task.get();
}

long end = System.currentTimeMillis();
System.out.println("Time taken: " + (end - start) + "ms");
```

This code snippet will execute 100,000 virtual threads, each sleeping for one second and then returning their task ID. When I executed the program, the total execution time was less than 2 seconds:

```
Time taken: 1713ms
```

But by changing this line:

```
ThreadFactory threadFactory = Thread.ofVirtual().factory();
```

To the following to use platform threads:

```
ThreadFactory threadFactory = Thread.ofPlatform().factory();
```

The program ran out of resources. This was the output:

```
[0.307s][warning][os,thread] Failed to start thread "Unknown thread" - pthread_create failed (EAGAIN) for attributes:
[0.307s][warning][os,thread] Failed to start the native thread for java.lang.Thread "Thread-2021"
Exception in thread "main" java.lang.OutOfMemoryError: unable to create native thread: possibly out of memory or proces
    at java.base/java.lang.Thread.start0(Native Method)
```

```
    at java.base/java.lang.Thread.start(Thread.java:1526)
    at App.main(App.java:42)
```

However, it's worth mentioning that while virtual threads bring many benefits, they don't change the fundamental principles of concurrent programming. You still need to properly synchronize access to shared mutable state, regardless of whether you're using virtual or platform threads.

Let's review some common threading problems next.

Threading Problems

When working with threads, it's important to be aware of potential problems that can arise due to the complex nature of concurrent programming. These issues can lead to unexpected behavior, reduced performance, or even complete program failure.

In the context of multi-threaded programming, problems start to occur when threads get stuck in a state where they cannot proceed, preventing the program from moving forward. Let's talk about some of the most common problems.

Deadlock

A deadlock occurs when two or more threads are unable to proceed because each thread is waiting for a resource that another thread holds, resulting in a circular dependency. It's a situation where threads are permanently blocked, waiting for each other to release the resources they need.

Imagine two friends, Anne and Joe, who are each trying to cross a narrow bridge from opposite ends. The bridge is so narrow that only one person can cross at a time. Anne starts walking from one end, and Joe starts walking from the other end. When they meet in the middle, neither can continue forward, and neither can go back because there's no space to turn around. They're stuck in a situation where neither can proceed, and neither can retreat. This deadlock situation halts their progress, similar to how a deadlock in Java halts the execution of threads waiting on each other to release resources.

In the context of multi-threaded programming, resources are typically locks or other synchronization mechanisms used to control access to shared data. Deadlocks occur when the following four conditions are simultaneously met:

1. **Mutual Exclusion:** At least one resource must be held in a non-sharable mode, meaning only one thread can use the resource at a time.
2. **Hold and Wait:** A thread must be holding at least one resource while waiting to acquire additional resources held by other threads.
3. **No Preemption:** Resources cannot be forcibly taken away from a thread; they must be released voluntarily by the thread holding them.
4. **Circular Wait:** There must be a circular chain of two or more threads, each waiting for a resource held by the next thread in the chain.

Consider this class that illustrates the deadlock analogy:

```java
public class DeadlockExample {
    private static final Object narrowBridgePart1 = new Object();
    private static final Object narrowBridgePart2 = new Object();

    public static void main(String[] args) {
        Runnable anneTask = () -> {
            synchronized (narrowBridgePart1) {
                System.out.println("Anne: Holding part 1 of the bridge...");
                try {
                    Thread.sleep(1000);
```

```
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("Anne: Waiting for part 2 of the bridge...");
                synchronized (narrowBridgePart2) {
                    System.out.println("Anne: Holding part 1 and part 2 of the bridge...");
                }
            }
        };

        Runnable joeTask = () -> {
            synchronized (narrowBridgePart2) {
                System.out.println("Joe: Holding part 2 of the bridge...");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("Joe: Waiting for part 1 of the bridge...");
                synchronized (narrowBridgePart1) {
                    System.out.println("Joe: Holding part 1 and part 2 of the bridge...");
                }
            }
        };

        // Create and start Anne's thread
        Thread.ofPlatform().start(anneTask);

        // Create and start Joe's thread
        Thread.ofPlatform().start(joeTask);
    }
}
```

In this example, we have two threads, `Anne` and `Joe`, and two locks, `narrowBridgePart1` and `narrowBridgePart2`. The program encounters a deadlock when the following sequence of events occurs:

1. `Anne` acquires `narrowBridgePart1` and enters the first synchronized block.
2. `Joe` acquires `narrowBridgePart2` and enters the first synchronized block.
3. `Anne` attempts to acquire `narrowBridgePart2` in the second synchronized block but is blocked because `narrowBridgePart2` is held by `Joe`.
4. `Joe` attempts to acquire `narrowBridgePart1` in the second synchronized block but is blocked because `narrowBridgePart1` is held by `Anne`.

At this point, both threads are waiting for each other to release the parts of the bridge they hold, resulting in a deadlock. Anne and Joe are stuck in the middle of the bridge, unable to proceed or retreat, just like the threads in a deadlock situation in Java. The program will hang indefinitely, with no thread being able to proceed.

To avoid deadlocks, it's important to follow best practices such as:

- **Acquiring locks in a consistent order:** If multiple locks need to be acquired, they should be acquired in the same order by all threads to avoid circular wait conditions.
- **Timeout mechanisms:** Use timeout mechanisms when attempting to acquire locks, so that threads don't wait indefinitely if they are unable to acquire a lock.

- **Resource ordering:** Assign a numerical order to resources and ensure that threads acquire resources in ascending order to prevent circular wait conditions.
- **Lock granularity:** Use fine-grained locks when possible, locking only the necessary sections of code to reduce the likelihood of contention and deadlocks.

Starvation

Starvation occurs when a thread is perpetually denied access to a shared resource, preventing it from making progress. In other words, a thread is *starved* of the resources it needs to complete its task. Starvation can happen when other threads continuously acquire the shared resource, causing the starved thread to wait indefinitely.

Think of a scenario where a group of people is waiting in line to buy tickets for a popular concert. If someone cuts in line repeatedly or if the ticket seller keeps serving only a certain group of people, some individuals may never get a chance to buy tickets. They are essentially starved of the opportunity to make their purchase.

In multi-threaded programs, starvation often arises when threads are assigned different priorities. Java assigns priorities to threads ranging from 1 (lowest) to 10 (highest), with 5 being the default priority. When threads with higher priorities are continuously given preference over threads with lower priorities, the lower-priority threads may suffer from starvation.

Let's review this class that illustrates the concert ticket analogy:

```
public class ConcertTicketStarvationExample {
    private static final Object ticketSeller = new Object();

    public static void main(String[] args) {
        Runnable impatientFanTask = () -> {
            while (true) {
                synchronized (ticketSeller) {
                    System.out.println("Impatient Fan: Bought a ticket");
                    // Simulate buying a ticket
                }
            }
        };

        Runnable patientFanTask = () -> {
            while (true) {
                synchronized (ticketSeller) {
                    System.out.println("Patient Fan: Bought a ticket");
                    // Simulate buying a ticket
                }
            }
        };

        // Create and start the impatient fan thread with MAX_PRIORITY
        Thread impatientFan = Thread.ofPlatform()
                                .priority(Thread.MAX_PRIORITY)
                                .start(impatientFanTask);

        // Create and start the patient fan thread with MIN_PRIORITY
        Thread patientFan = Thread.ofPlatform()
                                .priority(Thread.MIN_PRIORITY)
                                .start(patientFanTask);
    }
}
```

In this example, we have two threads, `impatientFan` and `patientFan`, competing for the same lock object, `ticketSeller`. The threads are assigned different priorities: `impatientFan` has the maximum priority (10), while `patientFan` has the minimum priority (1).

When the program runs, `impatientFan`, having a higher priority, is likely to acquire the lock more frequently than `patientFan`. As a result, `patientFan` may starve, waiting for its turn to access the shared resource. The output of the program might show that `impatientFan` acquires the lock repeatedly, while `patientFan` doesn't get a chance to execute as often as `impatientFan`.

It's important to note that thread priorities are not guaranteed to be strictly followed by the Java Virtual Machine (JVM). The JVM's thread scheduler uses priorities as a hint for making scheduling decisions but may not always adhere to them. Nevertheless, assigning proper priorities to threads can help reduce the risk of starvation.

To mitigate starvation, consider the following approaches:

- **Fair scheduling:** Use fair scheduling mechanisms, such as fair locks or semaphores, which ensure that threads are granted access to shared resources in the order they requested them.
- **Avoid long-running tasks:** Break down long-running tasks into smaller units of work, allowing other threads to have a chance to execute in between.
- **Adjust thread priorities:** Assign appropriate priorities to threads based on their importance and resource requirements. However, be cautious when manipulating thread priorities, as it can lead to complex and hard-to-predict behavior.
- **Timeout mechanisms:** Implement timeout mechanisms that allow threads to abandon waiting for a resource if they have been waiting for too long.

Livelock

Livelock occurs when two or more threads are actively responding to each other's actions but are unable to make progress. Unlike deadlock, where threads are stuck in a waiting state, threads in a livelock are constantly changing their state in response to the actions of other threads. However, despite the continuous activity, no real progress is made towards completing the intended task.

Imagine a scenario where a husband and wife are sitting at a table with only one spoon to share for their meal. Both are extremely polite and insist that the other should eat first. The husband, holding the spoon, offers it to the wife, but she refuses and insists that he eats first. This back-and-forth continues indefinitely, with neither of them ever eating because they keep offering the spoon to each other.

In the context of multi-threaded programming, livelock often occurs when threads are repeatedly yielding to each other without making any meaningful progress. Livelock can also happen when threads keep retrying an operation that persistently fails due to the actions of other threads.

Consider this program:

```
public class LivelockExample {

    static class Spoon {
        private Diner owner;

        public Spoon(Diner d) {
            owner = d;
        }

        public Diner getOwner() {
            return owner;
        }
```

```
    public synchronized void setOwner(Diner d) {
        owner = d;
    }

    public synchronized void use() {
        System.out.println(owner.name + " is eating.");
    }
}

static class Diner {
    private String name;
    private boolean isHungry;

    public Diner(String n) {
        name = n;
        isHungry = true;
    }

    public String getName() {
        return name;
    }

    public boolean isHungry() {
        return isHungry;
    }

    public void eatWith(Spoon spoon, Diner spouse) {
        while (isHungry) {
            if (spoon.getOwner() != this) {
                try {
                    Thread.sleep(1); // wait for the spoon to be free
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                continue;
            }

            if (spouse.isHungry()) {
                System.out.println(name + ": " + spouse.getName() + " you eat first.");
                spoon.setOwner(spouse);
                continue;
            }

            spoon.use();
            isHungry = false;
            System.out.println(name + ": I am done eating.");
            spoon.setOwner(spouse);
        }
    }
}

public static void main(String[] args) {
    Diner husband = new Diner("Husband");
    Diner wife = new Diner("Wife");
```

```
        Spoon spoon = new Spoon(husband);

        Thread husbandThread =
            Thread.ofPlatform().start(() -> husband.eatWith(spoon, wife));
        Thread wifeThread =
            Thread.ofPlatform().start(() -> wife.eatWith(spoon, husband));
    }
}
```

This program is kind of complex, so let me walk you through it step by step.

First, we have a `Spoon` class:

```
static class Spoon {
    private Diner owner;

    public Spoon(Diner d) {
        owner = d;
    }

    public Diner getOwner() {
        return owner;
    }

    public synchronized void setOwner(Diner d) {
        owner = d;
    }

    public synchronized void use() {
        System.out.println(owner.name + " is eating.");
    }
}
```

Think of the spoon as a shared resource. This class keeps track of who currently has the spoon. It has a few methods:

- A constructor to set the initial owner of the spoon.
- `getOwner()` to find out who currently has the spoon.
- `setOwner(Diner d)` to change the owner of the spoon.
- `use()` to simulate the action of using the spoon to eat, which just prints a message.

Next, we have the `Diner` class:

```
static class Diner {
    private String name;
    private boolean isHungry;

    public Diner(String n) {
        name = n;
        isHungry = true;
    }

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

```
    public boolean isHungry() {
        return isHungry;
    }

    public void eatWith(Spoon spoon, Diner spouse) {
        while (isHungry) {
            if (spoon.getOwner() != this) {
                try {
                    Thread.sleep(1); // wait for the spoon to be free
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                continue;
            }

            if (spouse.isHungry()) {
                System.out.println(name + ": " + spouse.getName() + " you eat first.");
                spoon.setOwner(spouse);
                continue;
            }

            spoon.use();
            isHungry = false;
            System.out.println(name + ": I am done eating.");
            spoon.setOwner(spouse);
        }
    }
}
```

It represents each person who wants to eat. Each diner has a name and a flag to indicate if they are hungry. The key part of this class is the `eatWith` method, which is where the livelock happens. This method does the following:

1. It checks if the diner owns the spoon.
2. If they don't, they wait a little and check again.
3. If they do own the spoon, they check if their spouse is hungry.
4. If the spouse is hungry, they offer the spoon to the spouse and wait.
5. If the spouse is not hungry, they use the spoon to eat and then stop being hungry.

In the `main` method, we create two `Diner` objects: husband and wife. We also create one `Spoon` object and give it to the husband initially. Then we start two threads, one for each diner. Each thread runs the `eatWith` method for their respective diner, trying to use the spoon:

```
public static void main(String[] args) {
    Diner husband = new Diner("Husband");
    Diner wife = new Diner("Wife");

    Spoon spoon = new Spoon(husband);

    Thread husbandThread =
        Thread.ofPlatform().start(() -> husband.eatWith(spoon, wife));
    Thread wifeThread =
```

```
        Thread.ofPlatform().start(() -> wife.eatWith(spoon, husband));
}
```

When the program runs, both the husband and wife are trying to eat using the spoon. Here's what happens step by step:

1. The husband starts with the spoon.
2. The husband checks if the wife is hungry (she is), so he offers the spoon to her.
3. The wife now has the spoon. She checks if the husband is hungry (he is), so she offers the spoon back to him.
4. This process repeats endlessly, with both the husband and wife constantly offering the spoon to each other without either of them ever eating.

This continuous back-and-forth without making any progress is a livelock. Both threads (the husband and wife) are active and continuously changing their state, but they are not able to proceed with eating because they keep deferring to each other.

To resolve livelocks, consider the following approaches:

- **Randomized backoff:** Introduce randomness in the yielding mechanism. Instead of immediately yielding, threads can wait for a random amount of time before retrying. This reduces the likelihood of threads continuously yielding to each other in a synchronized manner.
- **Resource ordering:** Assign a specific order to the resources or conditions that threads are waiting for. Ensure that threads acquire resources or check conditions in a consistent order to avoid circular dependencies.
- **Timeout mechanisms:** Implement timeout mechanisms that allow threads to abandon waiting and take alternative actions if they have been waiting for too long. This prevents threads from indefinitely yielding to each other.
- **Locking strategies:** Use appropriate locking strategies, such as read-write locks or fine-grained locks, to minimize contention and reduce the chances of livelock.

Race Conditions

Race conditions occur when multiple threads access shared data concurrently, and the final outcome depends on the relative timing of their executions. In other words, the behavior of the program becomes unpredictable and inconsistent because the threads *race* each other to perform operations on the shared data. Race conditions can lead to incorrect results, data corruption, and unexpected program behavior.

Imagine a scenario where two people, Anne and Joe, have a joint bank account. They both independently decide to withdraw money from an ATM at the same time. Suppose the account initially has a balance of $100. Anne tries to withdraw $50, while Joe tries to withdraw $70. If the ATM processes their requests concurrently without proper synchronization, the outcomes become unpredictable. The final balance could be $50, $30, or even negative $20, depending on the order in which the withdrawals are processed.

In multi-threaded programs, race conditions typically arise when multiple threads access shared variables or resources without appropriate synchronization mechanisms. The threads may read and write the shared data simultaneously, leading to inconsistent or unexpected results.

The following class simulates the race condition described in the analogy:

```
public class BankAccount {
    private int balance;

    public BankAccount(int initialBalance) {
        this.balance = initialBalance;
    }
```

```
    public void withdraw(String name, int amount) {
        if (balance >= amount) {
            System.out.println(name + " is going to withdraw " + amount);
            try {
                // Simulate the time taken to process withdrawal
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            balance -= amount;
            System.out.println(name + " completed the withdrawal of " + amount);
        } else {
            System.out.println(name + " tried to withdraw " + amount + " but insufficient balance.");
        }
        System.out.println("Current balance: " + balance);
    }

    public static void main(String[] args) {
        BankAccount account = new BankAccount(100);

        Runnable anneWithdrawal = () -> {
            account.withdraw("Anne", 50);
        };

        Runnable joeWithdrawal = () -> {
            account.withdraw("Joe", 70);
        };

        Thread anneThread = Thread.ofPlatform().unstarted(anneWithdrawal);
        Thread joeThread = Thread.ofPlatform().unstarted(joeWithdrawal);

        anneThread.start();
        joeThread.start();

        try {
            anneThread.join();
            joeThread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Final balance: " + account.balance);
    }
}
```

The class uses threads to represent Anne and Joe withdrawing money from a shared bank account: 1. The `BankAccount` class has a balance that both Anne and Joe will try to withdraw from.

2. The `withdraw` method checks if there is enough balance, simulates the processing time with `Thread.sleep(100)`, and then deducts the amount from the balance.
3. The `main` method creates a `BankAccount` instance with an initial balance of $100.
4. It defines two `Runnable` tasks for Anne and Joe, each trying to withdraw money.

5. Two threads are created and started to simulate concurrent withdrawals.
6. The `join` method ensures the main thread waits for both withdrawal operations to complete before printing the final balance.

Running this code multiple times can produce negative final balances, illustrating the race condition caused by unsynchronized access to the shared `balance` variable.

To prevent race conditions, it's essential to use synchronization mechanisms that ensure exclusive access to shared resources. Some common techniques include:

- **Locks**: Use lock objects, such as `ReentrantLock` (from `java.util.concurrent.locks`) or `synchronized` blocks, to ensure that only one thread can access the shared resource at a time.
- **Atomic variables**: Use atomic variables, such as `AtomicInteger`, which provide thread-safe operations for reading and writing shared variables.
- **Concurrent data structures**: Utilize thread-safe data structures from the `java.util.concurrent` package, such as `ConcurrentHashMap` or `CopyOnWriteArrayList`, which are designed to handle concurrent access.
- **Synchronization primitives**: Employ synchronization primitives like semaphores, barriers, or latches to coordinate thread execution and access to shared resources.

It's important to note that while synchronization is necessary to prevent race conditions, excessive synchronization can lead to performance overhead and potential liveness issues like deadlocks. Therefore, it's important to strike a balance and synchronize only when necessary, using granular locks and minimizing the scope of synchronized regions.

In general, identifying and resolving threading problems requires careful analysis and understanding of the program's behavior. By being aware of issues like deadlocks, starvation, livelocks, and race conditions, you can design and implement thread-safe code in concurrent programs.

In the next section, we'll explore techniques for synchronizing access to shared resources and coordinating thread execution to prevent these common problems.

Writing Thread-Safe Code

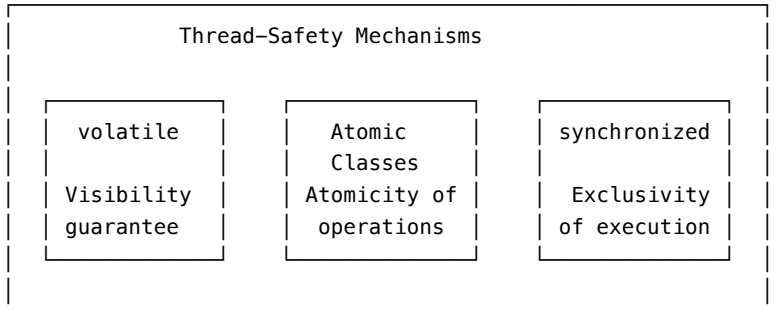
When developing multi-threaded applications, it's important to ensure that the code is thread-safe.

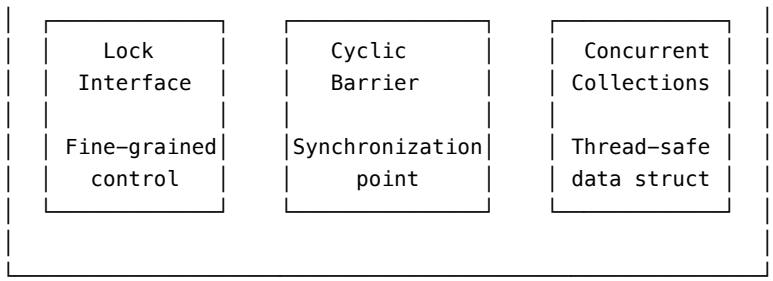
Thread-safety is the property of a program or a piece of code that guarantees its correct execution in a multi-threaded environment. A thread-safe code ensures that the shared data remains consistent and the program produces the expected output, regardless of the interleaving or timing of thread execution.

To achieve thread-safety, we need to address two main concerns: 1. **Data Visibility:** Ensuring that changes made by one thread are visible to other threads.

2. **Data Consistency:** Maintaining the integrity and correctness of shared data when multiple threads access and modify it concurrently.

Java provides several mechanisms to tackle these concerns and facilitate thread-safe programming:





Let's explore them in more detail.

Accessing Data with `volatile`

The `volatile` keyword in Java is used to indicate that a variable may be modified by multiple threads concurrently. When a variable is declared as `volatile`, it guarantees that any write to that variable will be immediately visible to other threads, and any subsequent read will always see the most up-to-date value.

Here's an example:

```
public class VolatileExample {
    private static volatile boolean flag = false;

    public static void main(String[] args) {
        Thread thread1 = Thread.ofVirtual().unstarted(() -> {
            while (!flag) {
                // Wait for the flag to become true
            }
            System.out.println("Thread 1 finished");
        });

        Thread thread2 = Thread.ofVirtual().unstarted(() -> {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            flag = true;
            System.out.println("Thread 2 set the flag");
        });

        thread1.start();
        thread2.start();

        try {
            thread1.join();
            thread2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

In this example, we have a `volatile` variable named `flag`. `thread1` continuously checks the value of `flag` and waits for it to become `true`. `thread2` sleeps for a second and then sets `flag` to `true`.

By declaring `flag` as `volatile`, we ensure that when `thread2` modifies its value, the change is immediately

visible to `thread1`. This guarantees that `thread1` will see the updated value and exit the waiting loop.

However, it's important to note that `volatile` only ensures visibility and does not provide atomicity or mutual exclusion. If multiple threads perform compound operations (such as read-modify-write) on a `volatile` variable concurrently, it can still lead to race conditions. In such cases, additional synchronization mechanisms are required.

Protecting Data with Atomic Classes

Java provides a set of atomic classes in the `java.util.concurrent.atomic` package that offer thread-safe operations on single variables. These classes ensure that the operations performed on the variables are atomic, meaning they are executed as a single, indivisible unit of work.

Some commonly used atomic classes are: - `AtomicBoolean`: Provides atomic operations on a boolean value.

- `AtomicInteger`: Provides atomic operations on an integer value.
- `AtomicLong`: Provides atomic operations on a long value.
- `AtomicReference<V>`: Provides atomic operations on an object reference of type `V`.

Here's an example using `AtomicInteger`:

```
import java.util.concurrent.atomic.AtomicInteger;

public class AtomicExample {
    private static AtomicInteger count = new AtomicInteger(0);

    public static void main(String[] args) throws InterruptedException {
        Thread thread1 = Thread.ofPlatform().start(() -> {
            for (int i = 0; i < 1000; i++) {
                count.incrementAndGet();
            }
        });

        Thread thread2 = Thread.ofPlatform().start(() -> {
            for (int i = 0; i < 1000; i++) {
                count.incrementAndGet();
            }
        });

        thread1.join();
        thread2.join();

        System.out.println("Final count: " + count.get());
    }
}
```

In this example, we have two threads that increment the `count` variable 1000 times each. The `count` variable is an instance of `AtomicInteger`, which provides thread-safe operations for incrementing and retrieving its value.

By using `AtomicInteger`, we ensure that the increment operation is performed atomically, avoiding race conditions. The `incrementAndGet()` method atomically increments the value and returns the updated value. The `get()` method retrieves the current value of the `AtomicInteger`.

Atomic classes provide various methods for performing thread-safe operations on variables. Some common methods include: - `get()`: Returns the current value.

- `set(type newValue)`: Sets the value to `newValue`.

- `getAndSet(type newValue)`: Sets the value to `newValue` and returns the previous value.
- `incrementAndGet()`: Atomically increments the value by one and returns the updated value.
- `getAndIncrement()`: Atomically increments the value by one and returns the previous value.
- `decrementAndGet()`: Atomically decrements the value by one and returns the updated value.
- `getAndDecrement()`: Atomically decrements the value by one and returns the previous value.

Here's an example that demonstrates the usage of these methods:

```
import java.util.concurrent.atomic.AtomicInteger;

public class AtomicMethodsExample {
    private static AtomicInteger value = new AtomicInteger(0);

    public static void main(String[] args) {
        // get(): Returns the current value
        int currentValue = value.get();
        System.out.println("Current value: " + currentValue);

        // set(type newValue): Sets the value to newValue
        value.set(10);
        System.out.println("Value after set(10): " + value.get());

        // getAndSet(type newValue): Sets the value to newValue and returns the previous value
        int previousValue = value.getAndSet(20);
        System.out.println("Previous value: " + previousValue);
        System.out.println("Value after getAndSet(20): " + value.get());

        // incrementAndGet(): Atomically increments the value by one and returns the updated value
        int incrementedValue = value.incrementAndGet();
        System.out.println("Value after incrementAndGet(): " + incrementedValue);

        // getAndIncrement(): Atomically increments the value by one and returns the previous value
        previousValue = value.getAndIncrement();
        System.out.println("Previous value: " + previousValue);
        System.out.println("Value after getAndIncrement(): " + value.get());

        // decrementAndGet(): Atomically decrements the value by one and returns the updated value
        int decrementedValue = value.decrementAndGet();
        System.out.println("Value after decrementAndGet(): " + decrementedValue);

        // getAndDecrement(): Atomically decrements the value by one and returns the previous value
        previousValue = value.getAndDecrement();
        System.out.println("Previous value: " + previousValue);
        System.out.println("Value after getAndDecrement(): " + value.get());
    }
}
```

This is the output of the program:

```
Current value: 0
Value after set(10): 10
Previous value: 10
Value after getAndSet(20): 20
Value after incrementAndGet(): 21
```

```
Previous value: 21
Value after getAndIncrement(): 22
Value after decrementAndGet(): 21
Previous value: 21
Value after getAndDecrement(): 20
```

This example demonstrates the usage of the various methods provided by the `AtomicInteger` class:

1. `get()`: Retrieves the current value of the `AtomicInteger` using `get()` and print it.
2. `set(type newValue)`: Sets the value of the `AtomicInteger` to 10 using `set(10)` and then print the updated value.
3. `getAndSet(type newValue)`: Sets the value of the `AtomicInteger` to 20 using `getAndSet(20)`. This method returns the previous value, which we store in the `previousValue` variable and print. We also print the updated value after the operation.
4. `incrementAndGet()`: Atomically increments the value of the `AtomicInteger` by one using `incrementAndGet()`. This method returns the updated value after the increment, which we store in the `incrementedValue` variable and print.
5. `getAndIncrement()`: Atomically increments the value of the `AtomicInteger` by one using `getAndIncrement()`. This method returns the previous value before the increment, which we store in the `previousValue` variable and print. We also print the updated value after the operation.
6. `decrementAndGet()`: Atomically decrements the value of the `AtomicInteger` by one using `decrementAndGet()`. This method returns the updated value after the decrement, which we store in the `decrementedValue` variable and print.
7. `getAndDecrement()`: Atomically decrements the value of the `AtomicInteger` by one using `getAndDecrement()`. This method returns the previous value before the decrement, which we store in the `previousValue` variable and print. We also print the updated value after the operation.

You can use similar methods for other atomic classes like `AtomicLong`, `AtomicBoolean`, etc., depending on the type of variable you need to work with.

Synchronized Blocks

In Java, the `synchronized` keyword is used to achieve mutual exclusion and synchronize access to shared resources. When a block of code is marked as `synchronized`, only one thread can execute that block at a time, while other threads attempting to enter the `synchronized` block will be blocked until the lock is released.

The general syntax for using a `synchronized` block is as follows:

```
synchronized (lockObject) {
    // Code block that requires synchronization
}
```

Here, `lockObject` is an object that serves as the lock. The thread that enters the `synchronized` block must acquire the lock on `lockObject` before executing the code inside the block. Once the thread exits the `synchronized` block, it automatically releases the lock, allowing other threads to acquire it and enter the block.

Here's an example that demonstrates the usage of a `synchronized` block:

```
public class SynchronizedExample {
    private static int count = 0;
    private static final Object lock = new Object();

    public static void increment() {
```

```
        synchronized (lock) {
            count++;
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Thread thread1 = Thread.ofPlatform().start(() -> {
            for (int i = 0; i < 1000; i++) {
                increment();
            }
        });

        Thread thread2 = Thread.ofPlatform().start(() -> {
            for (int i = 0; i < 1000; i++) {
                increment();
            }
        });

        thread1.join();
        thread2.join();

        System.out.println("Final count: " + count);
    }
}
```

In this example, we have a shared variable `count` that needs to be incremented by multiple threads. To ensure thread-safety, we use a `synchronized` block inside the `increment()` method. The `lock` object serves as the lock for synchronization.

When a thread enters the `increment()` method, it acquires the lock on `lock` before entering the `synchronized` block. Once inside the block, the thread increments the `count` variable. After exiting the block, the lock is automatically released, allowing other threads to acquire it and enter the block.

By synchronizing access to the `count` variable using a `synchronized` block, we ensure that only one thread can increment the variable at a time, preventing race conditions and maintaining data consistency.

Synchronizing on Methods

In addition to using `synchronized` blocks, Java allows you to synchronize entire methods using the `synchronized` keyword. The lock associated with the method depends on whether the method is an instance method or a static method.

For instance methods, the lock is associated with the object on which the method is invoked. Each instance of the class has its own lock, so multiple threads can simultaneously execute synchronized instance methods on different instances of the class.

On the other hand, for static methods, the lock is associated with the class itself, rather than any specific instance. Since there is only one class object per JVM, only one thread can execute a synchronized static method in the class at a time, regardless of the number of instances of that class.

Here's an example of synchronizing a method:

```
public class SynchronizedMethodExample {
    private static int count = 0;

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

```
    public static void main(String[] args) throws InterruptedException {
        Thread thread1 = Thread.ofPlatform().start(() -> {
            for (int i = 0; i < 1000; i++) {
                increment();
            }
        });

        Thread thread2 = Thread.ofPlatform().start(() -> {
            for (int i = 0; i < 1000; i++) {
                increment();
            }
        });

        thread1.join();
        thread2.join();

        System.out.println("Final count: " + count);
    }
}
```

In this example, the `increment()` method is declared as `synchronized`. When a thread invokes the `increment()` method, it automatically acquires the lock associated with the object on which the method is called (in this case, the class itself since the method is static).

Only one thread can execute the `increment()` method at a time, while other threads attempting to invoke the method will be blocked until the lock is released. This ensures that the `count` variable is incremented atomically and avoids race conditions.

It's important to note that synchronizing static methods can potentially lead to reduced concurrency since there is only one lock associated with the entire class. If multiple threads need to access different shared resources within the class, synchronizing at the method level may be too coarse-grained, and using `synchronized` blocks or more fine-grained locking mechanisms might be more appropriate.

Synchronizing methods provides a cleaner and more concise way of achieving thread-safety compared to using `synchronized` blocks. However, it's important to note that synchronizing an entire method can potentially lead to reduced concurrency if the method contains code that doesn't require synchronization.

In general, it's recommended to synchronize only the critical sections of code that access shared resources, using `synchronized` blocks or methods judiciously to strike a balance between thread-safety and performance.

This is particularly important with virtual threads. Remember, there's a limitation when it comes to using `synchronized` blocks or methods with virtual threads.

When a virtual thread performs a blocking operation (like I/O) inside a `synchronized` block or method, it causes the virtual thread scheduler to block an operating system thread. This situation is called *pinning*. Normally, outside of a synchronized context, the virtual thread would not block an OS thread during such operations.

Pinning can negatively impact server throughput if the blocking operation takes a long time and occurs frequently. However, using `synchronized` for short-lived or infrequent operations shouldn't cause problems.

If you detect frequent and long-lived pinning the recommendation is to replace `synchronized` blocks with a lock (like `ReentrantLock`) in those specific areas.

The `Lock` Interface

Java provides the `Lock` interface in the `java.util.concurrent.locks` package as an alternative to the `synchronized` keyword. The `Lock` interface offers more flexibility and control over lock acquisition and release

compared to the implicit locking mechanism of `synchronized`.

The main methods provided by the `Lock` interface are:

1. `void lock()`: Acquires the lock, blocking until the lock is available.
2. `void unlock()`: Releases the lock. Always call `unlock()` in a `finally` block to ensure proper lock release.
3. `boolean tryLock()`: Attempts to acquire the lock without blocking. Returns `true` if the lock is acquired, `false` otherwise.
4. `boolean tryLock(long time, TimeUnit unit)`: Attempts to acquire the lock while blocking for a specified amount of time. Returns `true` if the lock is acquired within the specified time, `false` otherwise.
5. `Condition newCondition()`: Creates a new `Condition` instance associated with the lock for coordinating thread execution based on conditions.

The `java.util.concurrent.locks` package provides a few implementations of the `Lock` interface, including:

- `ReentrantLock`: The most commonly used implementation, providing the same basic functionality as `synchronized` but with additional features such as fairness control and lock status queries.
- `ReentrantReadWriteLock.ReadLock` and `ReentrantReadWriteLock.WriteLock`: Provide a pair of associated locks for read and write access. Multiple threads can acquire the read lock simultaneously, while only one thread can acquire the write lock at a time.

To use a `Lock`, follow these steps:

1. Create an instance of the desired `Lock` implementation.
2. Acquire the lock using `lock()`, `tryLock()`, or `tryLock(long time, TimeUnit unit)`.
3. Perform the critical section operations while holding the lock.
4. Release the lock using `unlock()` in a `finally` block.

Here's an example of using a `ReentrantLock`:

```
Lock lock = new ReentrantLock();
try {
    lock.lock();
    // Critical section
} finally {
    lock.unlock();
}
```

The `Lock` interface provides additional features compared to `synchronized`, such as:

- Non-blocking lock attempts with `tryLock()`:

```
Lock lock = new ReentrantLock();
if (lock.tryLock()) {
    try {
        // Critical section
    } finally {
        lock.unlock();
    }
} else {
    // Lock not acquired, perform alternative actions
}
```

- Timed lock attempts with `tryLock(long time, TimeUnit unit)`:

```
Lock lock = new ReentrantLock();
try {
    if (lock.tryLock(1, TimeUnit.SECONDS)) {
        try {
            // Critical section
        } finally {
            lock.unlock();
        }
    } else {
        // Lock not acquired within the specified time
    }
} catch (InterruptedException e) {
    // Handle interruption
}
```

- Fairness control:

```
Lock lock = new ReentrantLock(true); // Creating a fair lock
try {
    lock.lock();
    // Critical section
} finally {
    lock.unlock();
}
```

The `ReentrantLock` class is the most common implementation of the `Lock` interface. It provides explicit lock acquisition and release, exception handling for incorrect lock usage, and lock reentrancy.

Here's an example comparing `synchronized` and `ReentrantLock`:

```
// Using synchronized
synchronized (lock) {
    // Critical section
}

// Using ReentrantLock
Lock lock = new ReentrantLock();
try {
    lock.lock();
    // Critical section
} finally {
    lock.unlock();
}
```

The `ReentrantReadWriteLock.ReadLock` and `ReentrantReadWriteLock.WriteLock` classes provide a way to handle concurrent read and write access to a shared resource. Here's a simplified example:

```
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class ReadWriteLockExample {
    private final ReentrantReadWriteLock rwLock = new ReentrantReadWriteLock();
    private final ReentrantReadWriteLock.ReadLock readLock = rwLock.readLock();
    private final ReentrantReadWriteLock.WriteLock writeLock = rwLock.writeLock();
    private int sharedResource = 0;

    public void write(int value) {
        writeLock.lock();
```

```
        try {
            sharedResource = value;
            System.out.println("Written: " + value);
        } finally {
            writeLock.unlock();
        }
    }

    public void read() {
        readLock.lock();
        try {
            System.out.println("Read: " + sharedResource);
        } finally {
            readLock.unlock();
        }
    }

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

        Thread writer = Thread.ofPlatform().start(() -> {
            example.write(42);
        });

        Thread reader = Thread.ofPlatform().start(() -> {
            example.read();
        });

        try {
            writer.join();
            reader.join();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
```

As you can see, the `Lock` interface provides more advanced features and control compared to the `synchronized` keyword, allowing for fine-grained locking, non-blocking lock attempts, and fairness control. However, it also requires explicit management of lock acquisition and release, which can be error-prone if not handled properly.

The `CyclicBarrier` Class

In concurrent programming, there are scenarios where multiple threads need to work together and synchronize their progress at certain points. The `java.util.concurrent.CyclicBarrier` class provides a synchronization aid that allows a set of threads to wait for each other to reach a common barrier point before proceeding further.

The `CyclicBarrier` class is designed to facilitate coordination between a fixed number of threads. It is particularly useful when you have a group of threads that need to perform tasks in parallel and then wait for each other to finish before moving on to the next stage.

Here's how the `CyclicBarrier` works:

1. When creating a `CyclicBarrier`, you specify the number of threads that need to reach the barrier before they can all proceed.

2. Each thread performs its task and then calls the `await()` method on the `CyclicBarrier` to indicate that it has reached the barrier.
3. The thread calling `await()` is blocked until all the specified number of threads have reached the barrier.
4. Once all threads have reached the barrier, the barrier is released, and all threads can proceed.
5. If desired, you can specify a barrier action, which is a `Runnable` task that is executed by one of the threads after all threads have reached the barrier but before they are released.

Here's a simple example that demonstrates the usage of `CyclicBarrier`:

```
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class CyclicBarrierExample {
    private static final int NUM_THREADS = 3;

    public static void main(String[] args) {
        CyclicBarrier barrier = new CyclicBarrier(NUM_THREADS, () -> {
            System.out.println("All threads reached the barrier");
        });

        for (int i = 0; i < NUM_THREADS; i++) {
            final int threadId = i;
            Thread.ofPlatform().start(() -> {
                try {
                    System.out.println("Thread " + threadId + " is performing task");
                    Thread.sleep(1000); // Simulating task execution
                    System.out.println("Thread " + threadId + " reached the barrier");
                    barrier.await();
                    System.out.println("Thread " + threadId + " continued after the barrier");
                } catch (InterruptedException | BrokenBarrierException e) {
                    e.printStackTrace();
                }
            });
        }
    }
}
```

In this example, we create a `CyclicBarrier` with a count of `NUM_THREADS` (3 in this case). We also specify a barrier action that will be executed once all threads have reached the barrier.

We then start three threads, each performing a task (simulated by sleeping for a short duration). After completing its task, each thread calls `await()` on the barrier to indicate that it has reached the synchronization point.

The output of the program will be similar to the following:

```
Thread 0 is performing task
Thread 1 is performing task
Thread 2 is performing task
Thread 1 reached the barrier
Thread 0 reached the barrier
Thread 2 reached the barrier
All threads reached the barrier
Thread 2 continued after the barrier
Thread 1 continued after the barrier
```

```
Thread 0 continued after the barrier
```

As you can see, all threads perform their tasks concurrently. Once all threads have reached the barrier, the barrier action is executed, and then all threads proceed further.

The `CyclicBarrier` is called *cyclic* because it can be reused after all threads have passed the barrier. You can call `await()` again on the same barrier object, and it will wait for the specified number of threads to reach the barrier again.

It's important to note that if any thread leaves the barrier prematurely by interrupting itself or throwing an exception, all other threads waiting on the barrier will receive a `BrokenBarrierException`. In such cases, you need to handle the exception appropriately and decide whether to continue or terminate the execution.

The Concurrency API

Java provides a powerful and flexible Concurrency API in the `java.util.concurrent` package, which offers a wide range of classes and interfaces for managing concurrent operations. This API simplifies the development of concurrent applications by providing high-level abstractions and utilities for managing threads, coordinating tasks, and synchronizing access to shared resources.

The Concurrency API was introduced in Java 5 and has been continuously enhanced in subsequent versions. It includes several key components, such as:

1. **Executors:** The `Executor` and `ExecutorService` interfaces provide a way to manage the execution of tasks in a thread pool, allowing you to focus on defining the tasks rather than managing the threads directly.
2. **Concurrent Collections:** The `java.util.concurrent` package offers thread-safe collections, such as `ConcurrentHashMap`, `CopyOnWriteArrayList`, and `BlockingQueue`, which provide better performance and scalability compared to using synchronized collections.
3. **Synchronizers:** Classes like `CountDownLatch`, `CyclicBarrier`, `Semaphore`, and `Phaser` help coordinate the actions of multiple threads, allowing them to wait for each other or control access to shared resources.
4. **Locks:** The `Lock` interface and its implementations, provide more advanced locking mechanisms compared to the `synchronized` keyword.
5. **Atomic Variables:** The `java.util.concurrent.atomic` package provides atomic variables, such as `AtomicInteger` and `AtomicReference`, which offer thread-safe operations on single variables without the need for explicit synchronization.

These components work together to provide a comprehensive framework for building concurrent and parallel applications in Java.

In previous sections, we have covered atomic variables, locks, and `CyclicBarrier`. In this section, we are going to focus on executors.

The `ExecutorService` Interface

The `ExecutorService` interface is a central part of the Concurrency API and extends the `Executor` interface. It provides methods for submitting tasks for execution and managing the lifecycle of the underlying thread pool.

A thread pool is a collection of pre-created and reusable threads that are ready to perform tasks. It acts as a pool of worker threads that can be used to execute tasks concurrently.

Imagine you have a big task that needs to be done, like painting a house. You could do it all by yourself, but it would take a long time. Instead, you decide to hire a group of workers to help paint the house. These workers are like a thread pool. When you have a task that needs to be executed, such as painting a room,

you assign it to one of the workers in the pool. The worker takes the task, performs it, and when finished, returns to the pool, ready to take on another task.

The advantage of using a thread pool is that you don't have to create a new worker (thread) every time you have a task to execute. Creating a new thread for each task can be expensive in terms of time and resources. Instead, you have a pre-created pool of workers (threads) that are ready to take on tasks as they come in. The thread pool manages the lifecycle of the threads, meaning it creates the threads when the pool is initialized and destroys them when the pool is shut down. It also handles the allocation of tasks to the available threads in the pool.

Here's an example of creating an `ExecutorService` using the `Executors` factory class:

```
ExecutorService executorService = Executors.newFixedThreadPool(5);
```

In this case, we create a fixed thread pool with 5 threads using the `Executors.newFixedThreadPool()` method.

The primary methods of the `ExecutorService` interface include:

- `void execute(Runnable command)`: Submits a `Runnable` task for execution without returning a result.
- `<T> Future<T> submit(Callable<T> task)`: Submits a `Callable` task for execution and returns a `Future` representing the pending result of the task.
- `<T> Future<T> submit(Runnable task, T result)`: Submits a `Runnable` task for execution and returns a `Future` representing the given result upon completion.
- `Future<?> submit(Runnable task)`: Submits a `Runnable` task for execution and returns a `Future` representing the task's completion.
- `<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)`: Submits a collection of `Callable` tasks for execution and returns a list of `Future` objects representing the results of each task.
- `<T> T invokeAny(Collection<? extends Callable<T>> tasks)`: Submits a collection of `Callable` tasks for execution and returns the result of one of the successfully completed tasks.

The `ExecutorService` interface also provides methods for managing the lifecycle of the thread pool:

- `void shutdown()`: Initiates an orderly shutdown of the `ExecutorService`, in which previously submitted tasks are executed, but no new tasks are accepted. This method does not wait for the running tasks to complete.
- `List<Runnable> shutdownNow()`: Attempts to stop all actively executing tasks and halts the processing of waiting tasks. It returns a list of the tasks that were awaiting execution.
- `boolean isShutdown()`: Returns `true` if the `ExecutorService` has been shut down, either by calling `shutdown()` or `shutdownNow()`.
- `boolean isTerminated()`: Returns `true` if all tasks have completed following a shutdown request.

It's important to properly shut down an `ExecutorService` when it's no longer needed to allow graceful termination of the threads and to release any resources held by the thread pool.

Here's an example showing the use of these methods:

```
ExecutorService executorService = Executors.newFixedThreadPool(5);

// Submit tasks for execution
executorService.execute(() -> {
    System.out.println("Task 1 executed by " + Thread.currentThread().getName());
});
executorService.execute(() -> {
    System.out.println("Task 2 executed by " + Thread.currentThread().getName());
});
```

```java
// Initiate orderly shutdown
executorService.shutdown();

// Check if the ExecutorService has been shut down
boolean isShutdown = executorService.isShutdown();
System.out.println("ExecutorService is shut down: " + isShutdown);

// Wait for all tasks to complete and check if the ExecutorService has terminated
try {
    boolean isTerminated = executorService.awaitTermination(1, TimeUnit.MINUTES);
    System.out.println("ExecutorService is terminated: " + isTerminated);
} catch (InterruptedException e) {
    throw new RuntimeException(e);
}
```

In this example, we create an `ExecutorService`, submit tasks for execution using the `execute()` method, initiate an orderly shutdown using `shutdown()`, and check the status of the `ExecutorService` using `isShutdown()` and `isTerminated()` methods.

However, it's worth noting that since Java 19, `ExecutorService` extends the `AutoCloseable` interface. This allows us to use a `try-with-resources` block, which automatically calls the `close()` method at the end of the `try` block. So the above example can be rewritten as follows:

```java
try (ExecutorService executorService = Executors.newFixedThreadPool(5)) {
    // Submit tasks for execution
    executorService.execute(() -> {
        System.out.println("Task 1 executed by " + Thread.currentThread().getName());
    });
    executorService.execute(() -> {
        System.out.println("Task 2 executed by " + Thread.currentThread().getName());
    });
} // ExecutorService.close() is called automatically here, which calls shutdown()
```

This change simplifies the use of `ExecutorService` instances and helps prevent resource leaks by ensuring that the executor is properly shut down, even if an exception occurs.

Submitting Tasks

The `ExecutorService` interface provides several methods for submitting tasks for execution:

- `void execute(Runnable command)`: Submits a `Runnable` task for execution without returning a result. The `execute()` method is inherited from the `Executor` interface.
- `<T> Future<T> submit(Callable<T> task)`: Submits a `Callable` task for execution and returns a `Future` representing the pending result of the task. The `Future` allows you to retrieve the result once the task completes.
- `<T> Future<T> submit(Runnable task, T result)`: Submits a `Runnable` task for execution and returns a `Future` representing the given result upon completion. This is useful when you want to return a specific result from a `Runnable` task.
- `Future<?> submit(Runnable task)`: Submits a `Runnable` task for execution and returns a `Future` representing the task's completion. The `Future`'s `get()` method will return `null` upon completion.

In addition to submitting individual tasks, the `ExecutorService` interface also provides methods for submitting multiple tasks at once:

- `<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)`: Submits a collection of `Callable` tasks for execution and returns a list of `Future` objects representing the results of each task. This method blocks until all tasks have completed.
- `<T> T invokeAny(Collection<? extends Callable<T>> tasks)`: Submits a collection of `Callable` tasks for execution and returns the result of one of the completed tasks. This method blocks until at least one task has completed successfully.

These methods allow you to submit multiple tasks concurrently and retrieve their results using the `Future` interface.

Consider the following example:

```
try (ExecutorService executorService = Executors.newFixedThreadPool(5)) {
    // Submit a Runnable task using execute()
    executorService.execute(() -> {
        System.out.println("Task executed by " + Thread.currentThread().getName());
    });

    // Submit a Callable task using submit()
    Future<String> future = executorService.submit(() -> {
        // Perform some computation
        return "Result of the task";
    });

    // Submit multiple Callable tasks using invokeAll()
    List<Callable<Integer>> tasks = Arrays.asList(
            () -> 1,
            () -> 2,
            () -> 3
    );
    List<Future<Integer>> futures = executorService.invokeAll(tasks);

    // Submit multiple Callable tasks using invokeAny()
    Integer result = executorService.invokeAny(tasks);
} catch (Exception e) {
    e.printStackTrace();
}
```

This example demonstrates submitting tasks using `execute()` for a `Runnable` task, `submit()` for a `Callable` task, `invokeAll()` for submitting multiple `Callable` tasks and retrieving their results as a list of `Future` objects, and `invokeAny()` for submitting multiple `Callable` tasks and retrieving the result of one of the completed tasks.

When submitting tasks using the `submit()` or `invokeAll()` methods, you receive `Future` objects representing the pending results of the tasks. The `Future` interface provides methods to check the status of a task and retrieve its result:

- `boolean isDone()`: Returns `true` if the task has completed, either normally or through an exception.
- `boolean isCancelled()`: Returns `true` if the task was cancelled before it completed normally.
- `boolean cancel(boolean mayInterruptIfRunning)`: Attempts to cancel the execution of the task. If the task has already completed or been cancelled, this method has no effect.
- `V get()`: Waits if necessary for the task to complete and retrieves its result. If the task throws an exception, it is wrapped in an `ExecutionException`.
- `V get(long timeout, TimeUnit unit)`: Waits if necessary for at most the given time for the task to

complete and retrieves its result. If the timeout expires before the task completes, a `TimeoutException` is thrown.

In the case of the `invokeAny()` method, the actual result (`T`) of the first completed task is returned.

These methods allow you to synchronize the main thread with the completion of the submitted tasks and retrieve their results when needed.

Consider the following example:

```
try (ExecutorService executorService = Executors.newSingleThreadExecutor()) {
    // Submit a Callable task using submit()
    Future<String> future = executorService.submit(() -> {
        // Simulate a long-running task
        Thread.sleep(2000);
        return "Result of the task";
    });

    // Check if the task is done
    boolean isDone = future.isDone();
    System.out.println("Task is done: " + isDone);

    // Cancel the task
    //boolean isCancelled = future.cancel(true);
    //System.out.println("Task is cancelled: " + isCancelled);

    // Retrieve the result of the task
    // Calling get() on an already cancelled task will throw
    // a CancellationException, regardless of the timeout value
    String result = null;
    try {
        result = future.get(1, TimeUnit.SECONDS);
    } catch (InterruptedException | ExecutionException | TimeoutException e) {
        e.printStackTrace();
    }
    System.out.println("Result: " + result);
}
```

In this example, we submit a `Callable` task using `submit()`, check if the task is done using `isDone()`, attempt to cancel the task using `cancel()`, and retrieve the task's result using `get()` with a timeout. If the task completes within the specified timeout, the result is obtained. Otherwise, a `TimeoutException` is thrown.

The `Callable` Interface

As you have seen from the previous examples, the `Callable` interface is similar to the `Runnable` interface but with a few key differences. While `Runnable` represents a task that can be executed concurrently, `Callable` represents a task that returns a result and that may throw an exception.

Here's the declaration of the `Callable` interface:

```
public interface Callable<V> {
    V call() throws Exception;
}
```

The `Callable` interface has a single method, `call()`, which returns a value of type `V` and may throw an exception. This is in contrast to the `Runnable` interface, which has a `void run()` method that does not return a value or throw checked exceptions.

The main differences between `Callable` and `Runnable` are:

1. **Return Value:** `Callable` tasks can return a result, whereas `Runnable` tasks cannot. The `call()` method of `Callable` returns a value of the specified type `V`, while the `run()` method of `Runnable` is `void` and does not return a value.
2. **Exception Handling:** `Callable` tasks can throw checked exceptions, whereas `Runnable` tasks cannot. The `call()` method of `Callable` declares that it may throw an `Exception`, while the `run()` method of `Runnable` does not declare any checked exceptions.

Here's an example that demonstrates the usage of `Callable`:

```
try (ExecutorService executorService = Executors.newSingleThreadExecutor()) {
    // Create a Callable task
    Callable<Integer> task = () -> {
        // Perform some computation
        int result = 0;
        for (int i = 1; i <= 10; i++) {
            result += i;
        }
        return result;
    };

    // Submit the Callable task to the ExecutorService
    Future<Integer> future = executorService.submit(task);

    // Retrieve the result of the task
    try {
        Integer result = future.get();
        System.out.println("Result: " + result); // Prints 55
    } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
    }
}
```

In this example, we create a `Callable` task that performs a simple computation and returns the result. We submit the task to the `ExecutorService` using the `submit()` method, which returns a `Future` object representing the pending result. We then use the `get()` method of `Future` to retrieve the result of the task. If the task throws an exception, the exception is wrapped in an `ExecutionException`, but an `InterruptedException` can also be thrown if the current thread was interrupted while waiting.

The choice between using `Callable` and `Runnable` depends on whether you need to return a result from the task and handle checked exceptions. If your task does not need to return a value and does not throw checked exceptions, you can use `Runnable`. However, if your task needs to return a result or throws checked exceptions, you should use `Callable`.

Scheduling Tasks

In addition to executing tasks immediately, the Concurrency API provides the ability to schedule tasks for execution at a later time or to execute tasks repeatedly with a fixed delay or at a fixed rate. This functionality is provided by the `ScheduledExecutorService` interface, which extends the `ExecutorService` interface.

The `ScheduledExecutorService` interface provides the following methods for scheduling tasks:

1. `schedule(Runnable command, long delay, TimeUnit unit)`: Schedules a `Runnable` task to be executed after the specified `delay`, expressed in the given `TimeUnit`.
2. `schedule(Callable<V> callable, long delay, TimeUnit unit)`: Schedules a `Callable` task to be executed after the specified `delay`, expressed in the given `TimeUnit`, and returns a `ScheduledFuture` representing the pending result.

3. `scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)`: Schedules a `Runnable` task to be executed periodically, with a fixed time interval between the start of one execution and the start of the next. The `initialDelay` parameter specifies the delay before the first execution, and the `period` parameter specifies the fixed time interval between executions.
4. `scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit)`: Schedules a `Runnable` task to be executed repeatedly, with a fixed delay between the end of one execution and the start of the next. The `initialDelay` parameter specifies the delay before the first execution, and the `delay` parameter specifies the fixed delay between executions.

Here's an example that demonstrates the usage of `ScheduledExecutorService` methods:

```
try (ScheduledExecutorService scheduledExecutorService =
            Executors.newSingleThreadScheduledExecutor()) {
    // Schedule a task to run after a delay of 2 seconds
    scheduledExecutorService.schedule(() -> {
        System.out.println("Task executed after 2 seconds delay");
    }, 2, TimeUnit.SECONDS);

    // Schedule a task to run repeatedly at a fixed rate of 1 second
    scheduledExecutorService.scheduleAtFixedRate(() -> {
        System.out.println("Task executed at fixed rate");
    }, 0, 1, TimeUnit.SECONDS);

    // Schedule a task to run repeatedly with a fixed delay of 500 milliseconds
    scheduledExecutorService.scheduleWithFixedDelay(() -> {
        System.out.println("Task executed with fixed delay");
    }, 0, 500, TimeUnit.MILLISECONDS);

    // Keep the main thread alive for 5 seconds
    try {
        Thread.sleep(5000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

In this example, we create a `ScheduledExecutorService` using the `Executors.newSingleThreadScheduledExecutor()` method. We then demonstrate the usage of the `schedule()`, `scheduleAtFixedRate()`, and `scheduleWithFixedDelay()` methods.

The `schedule()` method is used to schedule a task to run after a delay of 2 seconds. The `scheduleAtFixedRate()` method is used to schedule a task to run repeatedly at a fixed rate of 1 second, meaning that the next execution will start exactly 1 second after the previous execution starts, regardless of how long the task takes to complete. The `scheduleWithFixedDelay()` method is used to schedule a task to run repeatedly with a fixed delay of 500 milliseconds between the end of one execution and the start of the next.

It's important to note that the `ScheduledExecutorService` does not automatically terminate after the scheduled tasks are executed. You need to explicitly shut it down using the `shutdown()` method once you no longer need it.

Remember that the `ScheduledExecutorService` uses a limited number of threads to execute the scheduled tasks, so it's essential to choose the appropriate execution method based on your requirements and ensure the scheduled tasks do not overwhelm the available resources.

Executors Factory Methods

Throughout this section, we have used various factory methods provided by the `Executors` class to create instances of `ExecutorService` and `ScheduledExecutorService`. The `Executors` class is a utility class that offers several static factory methods for creating different types of thread pools and executor services.

Here's an overview of the commonly used factory methods provided by the `Executors` class:

1. `ExecutorService newSingleThreadExecutor()`: Creates an `ExecutorService` that uses a single worker thread to execute tasks. Tasks are guaranteed to be executed sequentially, and no more than one task will be active at any given time.
2. `ScheduledExecutorService newSingleThreadScheduledExecutor()`: Creates a single-threaded `ScheduledExecutorService` that can schedule tasks to run after a given delay or to execute periodically.
3. `ExecutorService newCachedThreadPool()`: Creates a thread pool that creates new threads as needed but will reuse previously constructed threads when they are available. Idle threads are kept in the pool for 60 seconds before being terminated and removed from the pool.
4. `ExecutorService newFixedThreadPool(int nThreads)`: Creates a thread pool with a fixed number of threads. The `nThreads` parameter specifies the number of threads in the pool. If additional tasks are submitted when all threads are active, they will wait in a queue until a thread becomes available.
5. `ScheduledExecutorService newScheduledThreadPool(int corePoolSize)`: Creates a thread pool that can schedule tasks to run after a given delay or to execute periodically. The `corePoolSize` parameter specifies the number of threads to keep in the pool, even if they are idle.

Here are code examples demonstrating the usage of each factory method:

```
// newSingleThreadExecutor()
try (ExecutorService singleThreadExecutor = Executors.newSingleThreadExecutor()) {
    singleThreadExecutor.submit(() -> {
        System.out.println("Task executed by single thread");
    });
}

// newSingleThreadScheduledExecutor()
try (ScheduledExecutorService singleThreadScheduledExecutor = Executors.newSingleThreadScheduledExecutor()) {
    singleThreadScheduledExecutor.schedule(() -> {
        System.out.println("Task scheduled by single thread scheduled executor");
    }, 2, TimeUnit.SECONDS);
}

// newCachedThreadPool()
try (ExecutorService cachedThreadPool = Executors.newCachedThreadPool()) {
    for (int i = 0; i < 5; i++) {
        cachedThreadPool.submit(() -> {
            System.out.println("Task executed by cached thread pool");
        });
    }
}

// newFixedThreadPool(int nThreads)
try (ExecutorService fixedThreadPool = Executors.newFixedThreadPool(3)) {
    for (int i = 0; i < 10; i++) {
        fixedThreadPool.submit(() -> {
            System.out.println("Task executed by fixed thread pool");
        });
```

```
    }
}

// newScheduledThreadPool(int corePoolSize)
try (ScheduledExecutorService scheduledThreadPool = Executors.newScheduledThreadPool(2)) {
    scheduledThreadPool.scheduleAtFixedRate(() -> {
        System.out.println("Task scheduled by scheduled thread pool");
    }, 0, 1, TimeUnit.SECONDS);

    // Keep the main thread alive for 3 seconds
    try {
        Thread.sleep(3000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

In these examples, we create different types of executor services using the respective factory methods provided by the `Executors` class.

The `newSingleThreadExecutor()` method creates an `ExecutorService` with a single worker thread, ensuring that tasks are executed sequentially. The `newSingleThreadScheduledExecutor()` method creates a single-threaded `ScheduledExecutorService` for scheduling tasks with delays or periodic execution.

The `newCachedThreadPool()` method creates a thread pool that creates new threads as needed and reuses idle threads. The `newFixedThreadPool(int nThreads)` method creates a thread pool with a fixed number of threads specified by the `nThreads` parameter.

The `newScheduledThreadPool(int corePoolSize)` method creates a `ScheduledExecutorService` with a fixed number of threads specified by the `corePoolSize` parameter. It allows scheduling tasks with delays or periodic execution.

These factory methods provide convenient ways to create different types of executor services based on specific requirements. They encapsulate the complexities of thread creation, management, and termination, allowing developers to focus on defining and submitting tasks.

It's important to choose the appropriate factory method based on your application's needs. Consider factors such as the number of tasks, concurrency requirements, scheduling needs, and resource constraints when selecting a suitable executor service.

In any case, remember to properly shut down the executor services using the `shutdown()` method when they are no longer needed to ensure graceful termination and resource cleanup.

Virtual Thread-Aware Executor

With the introduction of virtual threads, Java introduced a new factory method in the `Executors` class to create `ExecutorService` instances that work seamlessly with virtual threads: `newVirtualThreadPerTaskExecutor()`.

This method is equivalent to invoking `newThreadPerTaskExecutor(ThreadFactory)` with a thread factory that creates virtual threads. `newThreadPerTaskExecutor(ThreadFactory)` creates an `Executor` that starts a new `Thread` for each task. The number of threads created by the `Executor` is unbounded.

The key to understanding this is by recognizing that while they behave like platform threads, virtual threads represent a different concept. Platform threads, being scarce resources, need careful management, often through thread pools. The question "How many threads should we have in the pool?" is a common consideration with platform threads.

On the other hand, there can be millions of virtual threads. Instead of representing a shared, pooled resource, each virtual thread should represent a task in your application. The number of virtual threads you should use should be equal to the number of concurrent tasks in your application.

So, instead of using a shared thread pool executor like this:

```
try (var sharedThreadPoolExecutor = Executors.newFixedThreadPool(4)) {
    Future<TaskA> f1 = sharedThreadPoolExecutor.submit(task1);
    Future<TaskB> f2 = sharedThreadPoolExecutor.submit(task2);
    // ... use futures
}
```

You can use a virtual thread executor:

```
try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
    Future<TaskA> f1 = executor.submit(task1);
    Future<TaskB> f2 = executor.submit(task2);
    // ... use futures
}
```

The `Executors.newVirtualThreadPerTaskExecutor()` method returns an `ExecutorService` that doesn't use a thread pool. Instead, it creates a new virtual thread for each submitted task. This executor is lightweight, allowing you to create a new one as easily as you would any simple object.

Since the `ExecutorService` extends the `AutoCloseable` interface, at the end of the `try-with-resources` block, the `close()` method waits for all tasks submitted to the `ExecutorService` (all virtual threads spawned by the `ExecutorService`) to terminate.

This pattern is particularly useful when you need to concurrently perform multiple outgoing calls to different services. Here's an example:

```
void handle(Request request, Response response) {
    var url1 = ...
    var url2 = ...
    try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
        var future1 = executor.submit(() -> fetchURL(url1));
        var future2 = executor.submit(() -> fetchURL(url2));
        response.send(future1.get() + future2.get());
    } catch (ExecutionException | InterruptedException e) {
        response.fail(e);
    }
}

String fetchURL(URL url) throws IOException {
    try (var in = url.openStream()) {
        return new String(in.readAllBytes(), StandardCharsets.UTF_8);
    }
}
```

This example creates a new virtual thread for each URL fetch operation, even though these are small, short-lived tasks. This is precisely the kind of scenario for virtual threads.

It's important to understand that converting `n` platform threads to `n` virtual threads won't yield significant benefits. The real power comes from converting tasks to virtual threads. As a rule of thumb, if your application never reaches 10,000 virtual threads or more, it's unlikely to fully benefit from virtual threads. This could mean either your application's load is too light to need improved throughput, or you haven't represented enough tasks as virtual threads.

Concurrent Collections

When working with Java collections like `ArrayList`, `HashMap`, etc., in a multi-threaded environment, you may have encountered a `ConcurrentModificationException`. This exception is thrown when one thread is iterating over a collection while another thread tries to modify it structurally, for example, by adding or removing elements.

The solution is to use thread-safe, concurrent collections instead. Java provides several concurrent collection classes that allow multiple threads to access and modify them safely, without the risk of `ConcurrentModificationException`.

Some key concurrent collection classes include:

- `java.util.concurrent.ConcurrentHashMap<K,V>`
  A thread-safe version of `HashMap` that achieves high concurrency using advanced techniques like CAS (Compare-And-Swap) operations. This allows multiple threads to read and write to the map simultaneously:

  ```
  Map<String, Integer> map = new ConcurrentHashMap<>();
  map.put("apple", 1);
  map.put("banana", 2);
  ```

- `java.util.concurrent.ConcurrentLinkedQueue<E>`
  A thread-safe queue based on linked nodes. It allows multiple threads to add elements at the tail and remove elements from the head concurrently:

  ```
  Queue<String> queue = new ConcurrentLinkedQueue<>();
  queue.add("task1");
  queue.add("task2");
  String task = queue.poll();
  ```

- `java.util.concurrent.ConcurrentSkipListMap<K,V>`
  A concurrent, sorted map that provides similar functionality to `TreeMap` using a skip list, maintaining elements in sorted order based on their natural ordering or a provided `Comparator`. It allows concurrent access by multiple threads:

  ```
  ConcurrentNavigableMap<Integer, String> map = new ConcurrentSkipListMap<>();
  map.put(1, "one");
  map.put(2, "two");
  String value = map.get(1);
  ```

- `java.util.concurrent.ConcurrentSkipListSet<E>`
  A concurrent, sorted set that provides similar functionality to `TreeSet` using a skip list, maintaining elements in sorted order according to their natural ordering, or by a `Comparator` provided at set creation time, depending on which constructor is used. It offers `log (n)` time cost for add, remove, and contains operations:

  ```
  Set<String> set = new ConcurrentSkipListSet<>();
  set.add("apple");
  set.add("banana");
  set.add("orange");
  System.out.println(set); // [apple, banana, orange]
  ```

- `java.util.concurrent.CopyOnWriteArrayList<E>` and `java.util.concurrent.CopyOnWriteArraySet<E>`
  These classes are thread-safe variants of `ArrayList` and `HashSet` respectively. They achieve thread-safety by creating a fresh copy of the underlying array every time a write operation (add, set, remove, etc.) is performed. This means that multiple threads can safely iterate over the collection without the need for synchronization. However, the copy-on-write behavior can consume significant memory if the collection is large and write operations are frequent:

```
List<Integer> list = new CopyOnWriteArrayList<>();
list.add(1);
list.add(2);
list.add(3);
System.out.println(list); // [1, 2, 3]

Set<String> set = new CopyOnWriteArraySet<>();
set.add("apple");
set.add("banana");
set.add("apple");
System.out.println(set); // [apple, banana]
```

- `java.util.concurrent.LinkedBlockingQueue<E>`
  A thread-safe variant of LinkedList that implements the `BlockingQueue` interface. It's useful for implementing the producer-consumer pattern, where one or more threads produce items and put them into the queue, and one or more consumer threads take items out of the queue and process them. If the queue is empty, consumers will block until an item becomes available. If the queue reaches its maximum capacity, producers will block until space becomes available:

```
BlockingQueue<String> queue = new LinkedBlockingQueue<>(10);

// Producer thread
new Thread(() -> {
    try {
        for (int i = 0; i < 20; i++) {
            queue.put("item-" + i);
            System.out.println("Produced: " + "item-" + i);
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}).start();

// Consumer thread
new Thread(() -> {
    try {
        for (int i = 0; i < 20; i++) {
            String item = queue.take();
            System.out.println("Consumed: " + item);
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}).start();
```

In this example, the producer thread tries to put 20 items into the queue, but the queue has a maximum capacity of 10. When the limit is reached, the producer will block until the consumer has taken some items out. The consumer thread continually takes items from the queue and processes them. If the queue becomes empty, the consumer will block until the producer puts more items in.

When running the example above, the exact output may vary due to the concurrent execution of threads.

In addition to these purpose-built concurrent classes, in the `java.util.Collections` class, Java also provides methods to obtain synchronized versions of regular collections. These synchronization wrappers add a layer of thread-safety around an existing non-concurrent collection.

Some examples of these methods are: - `synchronizedCollection(Collection<T> c)`

- `synchronizedList(List<T> list)`
- `synchronizedMap(Map<K,V> m)`
- `synchronizedNavigableMap(NavigableMap<K,V> m)`
- `synchronizedNavigableSet(NavigableSet<T> s)`
- `synchronizedSet(Set<T> s)`
- `synchronizedSortedMap(SortedMap<K,V> m)`
- `synchronizedSortedSet(SortedSet<T> s)`

For example, to create a synchronized version of an `ArrayList`:

```
List<String> list = new ArrayList<>();
List<String> syncList = Collections.synchronizedList(list);
```

Now `syncList` is a thread-safe collection that can be safely accessed and modified by multiple threads. However, the synchronization is done at the method level, meaning each method of the collection is synchronized. This can limit concurrency compared to the purpose-built concurrent collections that often use more sophisticated techniques like CAS operations and non-blocking algorithms.

In general, it's preferable to use the concurrent collection classes directly, as they are designed from the ground up for high concurrency. The synchronization wrappers are useful when you need to add thread-safety to an existing collection or when using a less common collection type that doesn't have a direct concurrent equivalent.

Parallel Streams

In the world of Java streams, there's a feature that can greatly enhance performance when working with large datasets: parallel streams.

A parallel stream is a stream that splits its elements into multiple chunks, processing each chunk with a different thread in parallel. This can significantly speed up operations on large datasets by leveraging the power of multi-core processors.

However, there's an important concern to keep in mind when using parallel streams: the order of elements. Unlike regular sequential streams, the order of elements in a parallel stream is not guaranteed unless specifically enforced. This means that operations like `forEach`, which rely on encounter order, may produce unexpected results.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
numbers.parallelStream().forEach(System.out::println);
```

When running the above example, the output will show an unpredictable order. For example:

```
7
6
8
9
10
1
3
5
4
2
```

Another key consideration when using parallel streams is to avoid stateful lambda expressions. A stateful lambda is one that modifies shared state across invocations. In a parallel stream, multiple threads may be

executing the same lambda concurrently, which can lead to race conditions and unpredictable behavior if the lambda is stateful:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
int[] state = {0}; // Shared state

numbers.parallelStream().forEach(n -> {
    // Simulate some processing time
    try {
        Thread.sleep(100);
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
    state[0] += n; // Stateful lambda, unsafe
});

System.out.println(state[0]); // Unpredictable result due to race conditions
```

In this example, we use an array to hold the shared state, which allows us to modify it inside the lambda expression. The `Thread.sleep(100)` call introduces a small delay, increasing the likelihood of race conditions. Sometimes, the output will be 15. Other times, it will be 13 or something else.

To avoid these issues, it's important to use stateless lambda expressions when working with parallel streams.

Creating Parallel Streams

There are a few ways to create a parallel stream in Java:

1. Using the `parallelStream()` method on a collection:

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

2. Using the `parallel()` method on an existing stream:

   ```
   List<String> list = Arrays.asList("a", "b", "c");
   Stream<String> stream = list.stream();
   Stream<String> parallelStream = stream.parallel();
   ```

3. Using `StreamSupport.stream()` with a specified parallelism flag:

   ```
   List<String> list = Arrays.asList("a", "b", "c");
   boolean isParallel = true;
   Stream<String> parallelStream = StreamSupport.stream(list.spliterator(), isParallel);
   ```

Parallel Decomposition

Parallel decomposition is the process of breaking a task into smaller, independent subtasks that can be processed concurrently, and then combining the results to produce the final output. This is a fundamental concept in parallel computing, and it's key to understanding how parallel streams work under the hood.

When you invoke a terminal operation on a parallel stream, the Java runtime performs a parallel decomposition of the stream behind the scenes. This involves several steps:

1. **Splitting the Stream into Substreams**: The original stream is divided into multiple smaller substreams. The division is typically recursive, and does not necessarily match the number of processor cores directly. Each substream represents a portion of the original stream that can be processed independently, allowing for optimal utilization of computing resources.

2. **Processing Each Substream Independently**: Each substream is processed by a separate thread from `ForkJoinPool`, Java's built-in thread pool for parallel execution. `ForkJoinPool` uses a work-stealing

algorithm to balance the load, and dynamically allocates tasks among threads. This allows multiple substreams to be processed concurrently, leveraging the power of multi-core processors. Each thread applies the stream operations to its assigned substream independently of the others.

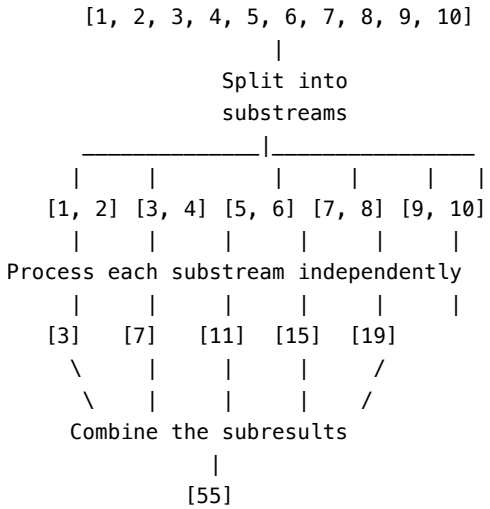
3. **Combining the Results**: Once all the substreams have been processed, their individual results need to be combined to produce the final result. The combining process also leverages `ForkJoinPool`'s capabilities to parallelize this step, especially for associative operations. The specific way in which the results are combined depends on the terminal operation. For example, with a `reduce` operation, the results of each substream's reduction are combined using the provided accumulator function. For a `collect` operation, the results are combined using the provided combiner function.

Consider the following example:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
int sum = numbers.parallelStream().reduce(0, Integer::sum);
System.out.println(sum); // Output: 55
```

In this example, the reduce operation is performed in parallel. The stream is split into substreams, each substream is summed independently, and then the results are combined to produce the final sum.

Here's a visual representation of this process:



The power of parallel decomposition lies in its ability to break down a large task into smaller, more manageable pieces that can be processed concurrently. This can lead to significant performance improvements, especially for computationally intensive tasks operating on large datasets.

However, it's important to note that not all operations can be parallelized effectively. For parallel decomposition to work, the subtasks must be independent - that is, the processing of one subtask should not depend on the results of another. This is why stateful lambda expressions can cause problems in parallel streams, as they introduce dependencies between subtasks.

Additionally, the cost of splitting the stream and combining the results should be taken into account. For small streams or simple operations, the overhead of parallel decomposition may outweigh the benefits of concurrent processing. The Java runtime attempts to make intelligent decisions about when to parallelize a stream based on factors like the stream size and the complexity of the operations, but it's still important to understand the implications of using parallel streams in your particular use case.

Methods of Stream that Perform Order-Based Tasks

There are certain operations that rely on the encounter order of elements. These operations are known as order-based tasks, and they can behave differently when used with parallel streams compared to sequential streams. Let's take a closer look at some of these methods and their implications.

`forEach` and `forEachOrdered` It's important to understand the difference between the `forEach` and `forEachOrdered` terminal operations.

The `forEach` operation, as we've seen earlier, is used to perform an action on each element of a stream. When used with a parallel stream, `forEach` does not guarantee the order in which the elements will be processed. Each substream is processed independently by a different thread, and the order in which the threads are scheduled is non-deterministic.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
numbers.parallelStream().forEach(System.out::println);
// Possible output: 3, 1, 4, 2, 5
```

In this example, the numbers may be printed in any order, depending on how the parallel stream is split and how the threads are scheduled.

This non-deterministic ordering can be beneficial in certain scenarios. For example, if you're performing an operation where the order doesn't matter, such as adding elements to a thread-safe collection or updating counters in a thread-safe manner, `forEach` can significantly boost performance by allowing operations to be performed in parallel without the overhead of maintaining order.

On the other hand, `forEachOrdered` guarantees that the action will be performed on the elements in the encounter order, even when used with a parallel stream. This means that the elements will be processed in the same order as they would be in a sequential stream, even though the processing is happening in parallel.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
numbers.parallelStream().forEachOrdered(System.out::println);
// Output: 1, 2, 3, 4, 5
```

In this case, the numbers are always printed in their original order, regardless of how the parallel stream is split and processed.

However, this ordering guarantee comes at a cost. To maintain the encounter order, `forEachOrdered` introduces a degree of synchronization and communication between the threads processing the substreams. This can reduce the performance benefits of parallelism, especially for large streams or complex operations.

So, when should you use `forEach`, and when should you use `forEachOrdered`? The answer depends on your specific use case.

Use `forEach` when: - The order of processing doesn't matter. - You're performing thread-safe operations (like adding to a `ConcurrentHashMap`). - You want to maximize performance and parallelism.

Use `forEachOrdered` when: - The order of processing is important. - You're performing order-dependent operations (like printing, adding to a `List`). - You're willing to sacrifice some performance for deterministic ordering.

It's worth noting that in many cases, if you need deterministic ordering, it may be more efficient to use a sequential stream instead of a parallel stream with `forEachOrdered`. The sequential stream will maintain the encounter order naturally, without the overhead of parallel decomposition and synchronization.

`findFirst()` The `findFirst()` method returns an `Optional` describing the first element of the stream, or an empty `Optional` if the stream is empty. In a sequential stream, this is straightforward, it simply returns the first element encountered in the stream.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
Optional<Integer> first = numbers.stream().findFirst();
System.out.println(first.get()); // Output: 1
```

However, when used with a parallel stream, `findFirst()` returns the first element from the first substream that produces a result. Since the order in which substreams are processed is non-deterministic, the element returned by `findFirst()` on a parallel stream may not always be the same.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
Optional<Integer> first = numbers.parallelStream().findFirst();
System.out.println(first.get()); // Output: non-deterministic (could be 1, 2, 3, 4, or 5)
```

`limit()` The `limit()` method returns a stream consisting of the first `n` elements of the original stream. In a sequential stream, this is again straightforward - it simply returns the first `n` elements in the encounter order.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
numbers.stream().limit(3).forEach(System.out::println);
// Output: 1, 2, 3
```

When used with a parallel stream, `limit()` returns the first `n` elements from the stream, but the order in which they are returned may not match the encounter order. This is because each substream is processed independently, and the first `n` elements from the combined results of the substreams are returned.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
numbers.parallelStream().limit(3).forEach(System.out::println);
// Possible output: 1, 3, 2
```

`skip()` The `skip()` method is the complement of `limit()`. It returns a stream consisting of the remaining elements of the original stream after discarding the first `n` elements. In a sequential stream, this skips the first `n` elements in the encounter order.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
numbers.stream().skip(3).forEach(System.out::println);
// Output: 4, 5
```

In a parallel stream, `skip()` discards the first `n` elements from the combined results of the substreams. However, since the substreams are processed independently, the elements that are skipped may not be the first `n` elements in the encounter order.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
numbers.parallelStream().skip(3).forEach(System.out::println);
// Possible output: 5, 4 (but could also be 1, 5 or 2, 4 or other combinations)
```

The non-deterministic behavior of these order-based methods when used with parallel streams can lead to surprising and potentially incorrect results if not handled correctly. If your operation relies on the encounter order of elements, it's generally safer to use a sequential stream.

However, there are situations where the non-deterministic ordering may be acceptable, or even desirable. For example, if you're using `findFirst()` to find any element matching a certain predicate, and you don't care which matching element is returned, using a parallel stream can provide a performance boost.

As with all aspects of parallel programming, the key is to understand the behavior and implications of the methods you're using, and to carefully consider whether the potential performance benefits outweigh the risks of non-deterministic results.

Reducing Parallel Streams

Reduction operations, such as `reduce()`, `collect()`, and `sum()`, are powerful tools for combining the elements of a stream into a single result. When used with parallel streams, these operations can provide significant performance benefits by allowing the reduction to be performed concurrently on multiple substreams. However, there are certain pitfalls to be aware of, particularly when it comes to the choice of accumulator function.

The accumulator function combines elements during a reduction operation. For example, in the `reduce()` method, the accumulator function takes two parameters: the partial result of the reduction so far, and the next element to be incorporated.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
int sum = numbers.parallelStream().reduce(0, Integer::sum);
System.out.println(sum); // Output: 15
```

In this example, the accumulator function is `Integer::sum`, which simply adds two integers together.

For a reduction in a parallel stream to produce correct results, the accumulator function must be associative and stateless. An associative function is one in which the order of application doesn't matter. That is, `(a op b) op c` is equal to `a op (b op c)`, where `op` is the accumulator function.

However, certain accumulator functions can lead to issues in parallel streams. For example, using a mutable accumulator:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
ArrayList<Integer> list = numbers.parallelStream().reduce(
        new ArrayList<>(),
        (l, i) -> { l.add(i); return l; },
        (l1, l2) -> { l1.addAll(l2); return l1; });
System.out.println(list); // Output: non-deterministic (could be [1, 2, 3, 4, 5], [1, 3, 5, 2, 4], etc.)
```

The output is non-deterministic because we're using a mutable `ArrayList` as the accumulator. The lambda expressions are modifying the same `ArrayList` concurrently from multiple threads, leading to race conditions and non-deterministic results.

To avoid these issues and ensure deterministic, correct results from parallel reductions, follow these best practices:

1. **Use associative and stateless accumulator functions**. If your accumulator function is not associative, consider using a sequential stream instead.
2. **Avoid using mutable accumulators**. If you need to collect results into a mutable container, use the `collect()` method with a concurrent collector, such as `toConcurrentMap()`, instead of `reduce()`. Concurrent collectors are designed to handle parallel modifications safely.
3. **Be cautious with floating-point arithmetic**. Due to the limitations of floating-point representation, floating-point addition and multiplication are not strictly associative. If absolute precision is required, consider using a sequential stream or a different numerical representation.
4. **Test your reductions thoroughly**. Do it with different stream sizes and different levels of parallelism to ensure that they produce consistent, correct results.

Combining Results in Parallel Streams

The `collect()` method is a terminal operation that allows you to accumulate elements of a stream into a collection or other data structure. When used with parallel streams, `collect()` can provide significant performance benefits by allowing the accumulation to be performed concurrently on multiple substreams. However, to ensure correct and efficient operation, there are certain considerations to keep in mind.

Remember, the `collect()` method takes a `Collector`, which specifies how the elements of the stream should be accumulated. A `Collector` is defined by four components:

1. A supplier function that creates a new result container.
2. An accumulator function that adds an element to the result container.
3. A combiner function that merges two result containers into one.
4. A finisher function that performs an optional final transformation on the result container.

The Java `Collectors` class provides a wide variety of predefined collectors, such as `toList()`, `toSet()`, `toMap()`, `groupingBy()`, and more.

When using `collect()` with a parallel stream, there are several key considerations to ensure correct and efficient operation:

1. **The collector should be concurrent.** This means that the accumulator and combiner functions must be thread-safe and should not depend on the order in which elements are processed. The `Collectors` class provides several concurrent collectors, such as `toConcurrentMap()`, `groupingByConcurrent()`, etc:

```
List<String> strings = Arrays.asList("a", "b", "c", "d", "e");
ConcurrentMap<String, Integer> map = strings.parallelStream()
        .collect(Collectors.toConcurrentMap(s -> s, s -> 1, Integer::sum));
System.out.println(map); // Output: {a=1, b=1, c=1, d=1, e=1}
```

2. **If the collector is not concurrent, consider using a concurrent result container.** For example, you can collect into a `ConcurrentHashMap` or a `CopyOnWriteArrayList`:

```
List<String> strings = Arrays.asList("a", "b", "c", "d", "e");
ConcurrentHashMap<String, Integer> map = strings.parallelStream()
        .collect(ConcurrentHashMap::new,
                 (m, s) -> m.put(s, 1),
                 ConcurrentHashMap::putAll);
System.out.println(map); // Output: {a=1, b=1, c=1, d=1, e=1}
```

3. **Be careful with order-dependent collectors.** Collectors like `Collectors.toList()` and `Collectors.toCollection(ArrayList::new)` preserve the encounter order of elements in a sequential stream, but not necessarily in a parallel stream. If the order of elements in the result is important, consider using `Collectors.toCollection(LinkedHashSet::new)` or collecting to a concurrent container and then copying to an ordered container:

```
List<String> strings = Arrays.asList("a", "b", "c", "d", "e");
List<String> list = strings.parallelStream()
        .collect(Collectors.toCollection(CopyOnWriteArrayList::new))
        .stream()
        .sorted()
        .collect(Collectors.toList());
System.out.println(list); // Output: [a, b, c, d, e]
```

4. **Consider the characteristics of the collector.** The `Collector` interface defines three characteristics (`java.util.stream.Collector.Characteristics`):

    - `CONCURRENT`: Indicates that this collector is concurrent, meaning that the result container can support the accumulator function being called concurrently with the same result container from multiple threads.
    - `UNORDERED`: Indicates that the collection operation does not commit to preserving the encounter order of input elements.
    - `IDENTITY_FINISH`: Indicates that the finisher function is the identity function and can be left out.

These characteristics provide hints to the stream framework about how the collector can be optimized. For example, if a collector is `UNORDERED`, the stream framework can freely rearrange the elements, which can enable certain optimizations:

```
List<String> strings = Arrays.asList("a", "b", "c", "d", "e");
Set<String> set = strings.parallelStream()
        .collect(Collectors.toUnmodifiableSet()); // UNORDERED collector
System.out.println(set); // Output: [a, b, c, d, e] (possibly in a different order)
```

By understanding these considerations and choosing the appropriate collector for your use case, you can effectively harness the power of `collect()` with parallel streams to achieve significant performance improvements in your stream-based operations.

In addition to the predefined collectors provided by the `Collectors` class, you can also create your own custom collectors using the `Collector.of()` method. This allows you to define your own supplier, accumulator, combiner, and finisher functions to collect elements into a custom data structure or perform a custom accumulation operation.

Here's an example using a sequential stream for string concatenation, which ensures deterministic output and better performance:

```
List<String> strings = Arrays.asList("a", "b", "c", "d", "e");
String concatenated

String concatenated = strings.stream() // Using a sequential stream
    .collect(Collector.of(
        StringBuilder::new,               // Supplier
        StringBuilder::append,            // Accumulator
        (sb1, sb2) -> {
            sb1.append(sb2);
            return sb1;
        },                                // Combiner
        StringBuilder::toString           // Finisher
    ));

System.out.println(concatenated); // Output: abcde
```

In this example, we use a sequential stream to concatenate a list of strings into a single string. A custom collector is created using `Collector.of()`, with `StringBuilder` as the container for accumulating the strings. The `StringBuilder::append` method is used as the accumulator, ensuring that strings are appended in the correct order. The combiner is defined to merge `StringBuilder` instances during parallel processing, but since we are using a sequential stream, it ensures the concatenation is performed efficiently and deterministically. Finally, the `StringBuilder::toString` method is used as the finisher to produce the final concatenated string. This approach guarantees the correct order of elements and optimal performance for string concatenation.

However, string concatenation is inherently sequential, and using a parallel stream here is likely to be less efficient than using a sequential stream. In fact, the output of this operation is non-deterministic for a parallel stream, because the order in which the substreams are combined is not guaranteed.

A better example for parallel streams using a custom collector might involve a task that can benefit from parallel processing and has a well-defined order of elements. Consider this example that adds up integers, where parallel processing can provide performance benefits:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
Integer sum = numbers.parallelStream() // Using a parallel stream
    .collect(Collector.of(
        () -> new int[1],              // Supplier
        (a, t) -> a[0] += t,           // Accumulator
        (a1, a2) -> {
            a1[0] += a2[0];
            return a1;
        },                             // Combiner
        a -> a[0]                      // Finisher
    ));

System.out.println(sum); // Output: 15
```

In this example, a custom collector is defined using `Collector.of()`, with an integer array as the container to hold the sum. The accumulator function adds each integer to the array's single element, and the combiner function merges two arrays by summing their elements. The finisher function extracts the summed value

from the array.

However, creating custom collectors that are both correct and efficient for parallel streams can be challenging. It requires a deep understanding of concurrency, thread safety, and the characteristics of the stream and collector. If possible, it's generally recommended to use the predefined collectors or compose them to achieve your desired operation.

Key Points

- Threads allow multiple paths of execution to occur concurrently within a single program.
- In Java 21, there are two types of threads: platform threads and virtual threads.
- Platform threads are traditional threads mapped directly to operating system threads.
- Virtual threads are lightweight threads managed by the Java Virtual Machine (JVM).
- To create a platform thread, you can extend the `Thread` class, implement the `Runnable` interface, or use `Thread.Builder.OfPlatform`.
- Virtual threads can be created using `Thread.ofVirtual()`, `Thread.startVirtualThread()`, or a `ThreadFactory` from `Thread.ofVirtual().factory()`.
- Virtual threads are particularly effective for I/O-bound tasks but not for long-running CPU-intensive operations.
- Virtual threads are always daemon threads and have a fixed priority that cannot be changed.
- The `start()` method initiates a new thread that executes the code defined in the `run()` method.
- The `sleep()` method causes the current thread to suspend execution for a specified number of milliseconds.
- The `interrupt()` method can be used to prematurely wake a sleeping or waiting thread.
- Thread lifecycle states include `NEW`, `RUNNABLE`, `BLOCKED`, `WAITING`, `TIMED_WAITING`, and `TERMINATED`.
- Common threading problems include:
  - Deadlock: When two or more threads are unable to proceed because each is waiting for resources held by the others.
  - Starvation: When a thread is perpetually denied access to shared resources it needs to progress.
  - Livelock: When threads are actively responding to each other but unable to make progress.
  - Race conditions: When the program's behavior depends on the relative timing of thread executions.
- To prevent race conditions, use synchronization mechanisms like locks, atomic variables, or concurrent data structures.
- Proper thread management and synchronization are crucial for writing efficient and bug-free concurrent programs.
- Thread-safety ensures correct execution of code in a multi-threaded environment, maintaining data consistency and visibility.
- The `volatile` keyword guarantees immediate visibility of variable changes across threads but doesn't provide atomicity for compound operations.
- Atomic classes (e.g., `AtomicInteger`, `AtomicBoolean`) offer thread-safe operations on single variables without explicit synchronization.
- The `synchronized` keyword can be used on methods or blocks to ensure exclusive access to a shared resource, preventing race conditions.

- The `Lock` interface provides more flexible and fine-grained control over locking compared to `synchronized`, including features like timed lock attempts and fairness control.
- `CyclicBarrier` allows a fixed number of threads to wait for each other at a common synchronization point before proceeding.
- The `ExecutorService` interface manages thread pools and task execution, providing methods for submitting and managing concurrent tasks.
- The `Callable` interface represents a task that returns a result and can throw exceptions, unlike `Runnable` which doesn't return a value.
- `ScheduledExecutorService` allows scheduling tasks for future or periodic execution.
- The `Executors` class provides factory methods for creating different types of thread pools and executor services (e.g., `newFixedThreadPool`, `newCachedThreadPool`).
- Virtual threads, introduced in recent Java versions, offer a lightweight alternative to platform threads for concurrent programming.
- The `Executors.newVirtualThreadPerTaskExecutor()` method creates an `ExecutorService` that spawns a new virtual thread for each submitted task, ideal for I/O-bound operations.
- When using virtual threads, focus on representing tasks as threads rather than managing a fixed thread pool size.
- Virtual threads show significant benefits when your application can utilize thousands of concurrent threads, particularly for I/O-bound tasks.
- Concurrent collections are thread-safe alternatives to regular collections for use in multi-threaded environments.
- Key concurrent collections include:
  - `ConcurrentHashMap<K,V>`: A thread-safe version of `HashMap`.
  - `ConcurrentLinkedQueue<E>`: A thread-safe queue based on linked nodes.
  - `ConcurrentSkipListMap<K,V>`: A concurrent version of `TreeMap`.
  - `ConcurrentSkipListSet<E>`: A scalable concurrent version of `TreeSet`.
  - `CopyOnWriteArrayList<E>` and `CopyOnWriteArraySet<E>`: Thread-safe variants of `ArrayList` and `HashSet`.
  - `LinkedBlockingQueue<E>`: A thread-safe variant of `LinkedList` implementing the `BlockingQueue` interface.
- The `Collections` class provides methods to obtain synchronized versions of regular collections (e.g., `synchronizedList()`, `synchronizedMap()`).
- Parallel streams split elements into multiple chunks for concurrent processing.
- Parallel streams are created using `parallelStream()` on collections or `parallel()` on existing streams.
- The order of elements in parallel streams is not guaranteed unless specifically enforced.
- Avoid stateful lambda expressions in parallel streams to prevent race conditions.
- Parallel decomposition involves splitting the stream, processing substreams independently, and combining results. It's effective for large datasets and computationally intensive tasks.
- For order-based stream operations:
  - `forEach()`: Does not guarantee order in parallel streams.
  - `forEachOrdered()`: Maintains encounter order even in parallel streams, but may reduce performance benefits.
  - `findFirst()`: Returns the first element from the first processed substream in parallel streams.
  - `limit()` and `skip()`: May not maintain encounter order in parallel streams.

- Reduction operations (`reduce()`, `collect()`, `sum()`) can be performed concurrently on substreams.
- Accumulator functions must be associative and stateless for correct parallel reduction.
- Avoid mutable accumulators in parallel streams.
- Be careful with floating-point arithmetic in parallel reductions.
- To combining results in parallel streams use concurrent collectors (e.g., `toConcurrentMap()`, `groupingByConcurrent()`) for thread-safety.
- Consider concurrent result containers (e.g., `ConcurrentHashMap`, `CopyOnWriteArrayList`) for non-concurrent collectors.
- Be aware of order-dependent collectors and their behavior in parallel streams.
- The `Collector` interface defines three characteristics (`java.util.stream.Collector.Characteristics`):
  - `CONCURRENT`: Indicates that this collector is concurrent, meaning that the result container can support the accumulator function being called concurrently with the same result container from multiple threads.
  - `UNORDERED`: Indicates that the collection operation does not commit to preserving the encounter order of input elements.
  - `IDENTITY_FINISH`: Indicates that the finisher function is the identity function and can be left out.
- Custom collectors can be created using `Collector.of()` method. They require careful consideration of thread-safety and efficiency for parallel streams. Generally, prefer predefined collectors or their compositions when possible.

Practice Questions

1. Which of the following lines of code correctly creates and starts a new virtual thread?

```
public class Main {
    public static void main(String[] args) {
        Runnable task = () -> {
            for (int i = 0; i < 5; i++) {
                System.out.println("Task is running");
            }
        };
        // Insert code here to create and start a new virtual thread
    }
}
```

A) `Thread thread = Thread.ofVirtual(); thread.start(task);`
B) `Thread thread = Thread.ofVirtual().unstarted(task).run();`
C) `Thread thread = Thread.ofVirtual().start(task);`
D) `Thread thread = Thread.ofVirtual(); task.run();`
E) `Thread thread = Thread.start(task);`

2. Which of the options correctly uses a `synchronized` block to ensure that only one thread at a time can execute a critical section that increments a shared counter?

```
public class Main {
    private static int counter = 0;
    public static void main(String[] args) {
        Runnable task = () -> {
            for (int i = 0; i < 1000; i++) {
                // Insert synchronized block here
            }
        };
```

```
        Thread thread1 = Thread.ofPlatform().start(task);
        Thread thread2 = Thread.ofPlatform().start(task);
        try {
            thread1.join();
            thread2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Final counter value: " + counter);
    }
}
```

A) `synchronized (this) { counter++; }`
B) `synchronized (Main.class) { counter++; }`
C) `synchronized (task) { counter++; }`
D) `synchronized (counter) { counter++; }`
E) `synchronized (System.out) { counter++; }`

3. Which of the following statements about atomic classes is correct? (Choose all that apply)

A) `AtomicInteger` is part of the `java.util.concurrent.atomic` package, but it does not provide atomic operations for increment and decrement.
B) `AtomicReference` can only be used with reference types, not primitive types.
C) `AtomicLong` supports atomic operations on `long` values, including `getAndIncrement()` and `compareAndSet()` methods.
D) `AtomicBoolean` can be used to perform atomic arithmetic operations on `boolean` values.

4. Which of the following code snippets correctly uses the `Lock` interface to ensure thread-safe access to a shared resource?

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Counter {
    private int count = 0;
    private Lock lock = new ReentrantLock();

    public void increment() {
        // Insert code here
    }

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

A)

```
lock.lock();
try {
    count++;
} finally {
    lock.unlock();
}
```

B)

```
lock.lock();
```

```
count++;
lock.unlock();
```

C)

```
try {
    lock.lock(() -> {
        count++;
    });
} finally {
    lock.unlock();
}
```

D)

```
synchronized(lock) {
    count++;
}
```

5. Which of the following code snippets correctly demonstrates the usage of an `ExecutorService` with a `try-with-resources` block?

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

public class ExecutorExample {
    public static void main(String[] args) {
        // Insert code here to create and use an ExecutorService with try-with-resources
    }
}
```

A)

```
try (ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor()) {
    executor.submit(() -> System.out.println("Task executed"));
}
```

B)

```
try (ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor()) {
    executor.submit(() -> System.out.println("Task executed"));
} finally {
    executor.shutdown();
}
```

C)

```
ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor();
try {
    executor.submit(() -> System.out.println("Task executed"));
} finally {
    executor.close();
}
```

D)

```
try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
    executor.submit(() -> System.out.println("Task executed"));
    executor.awaitTermination(1, TimeUnit.SECONDS);
}
```

6. Which of the following code snippets correctly demonstrates how to get a result from a `Callable` task using an `ExecutorService` with `try-with-resources`?

```
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;

public class CallableExample {
    public static void main(String[] args) {
        Callable<Integer> task = () -> {
            return 123;
        };
        // Insert code here
    }
}
```

A)

```
try (ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor()) {
    Future<Integer> future = executor.submit(task);
    System.out.println(future.get());
}
```

B)

```
try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
    Future<Integer> future = executor.submit(task);
    Integer result = future.get(1, TimeUnit.SECONDS);
    System.out.println(result);
}
```

C)

```
try (ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor()) {
    Future<Integer> future = executor.submit(task);
    executor.shutdown();
    System.out.println(future.get());
}
```

D)

```
try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
    Future<Integer> future = executor.submit(task);
    try {
        Integer result = future.get();
        System.out.println(result);
    } catch (InterruptedException | ExecutionException e) {
        // Handle exceptions
    }
}
```

7. Which of the following statements about Java's concurrent collections is correct?

A) `ConcurrentHashMap` allows concurrent read and write operations, and retrieval operations do not block even when updates are being made.
B) `CopyOnWriteArrayList` is optimized for scenarios with a high number of write operations compared to

read operations.
C) `ConcurrentSkipListSet` does not kept elements sorted.
D) `BlockingQueue` implementations like `LinkedBlockingQueue` allow elements to be added and removed concurrently without any internal locking mechanisms.

8. Which of the following statements about parallel streams is correct?

A) Parallel streams always improve the performance of a program by utilizing multiple threads.
B) Parallel streams can lead to incorrect results if the operations performed are not thread-safe.
C) The order of elements in a parallel stream is always preserved compared to the original stream.
D) Using parallel streams guarantees that the operations on elements will execute in a fixed order.

9. Which of the following code snippets correctly demonstrates how to reduce a parallel stream to compute the sum of its elements?

```
import java.util.Arrays;
import java.util.List;

public class ParallelStreamExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
        // Insert code here
    }
}
```

A)

```
int sum = numbers.parallelStream().reduce(1, Integer::sum);
System.out.println(sum);
```

B)

```
int sum = numbers.parallelStream().reduce(0, Integer::sum);
System.out.println(sum);
```

C)

```
int sum = numbers.stream().reduce(0, Integer::sum);
System.out.println(sum);
```

D)

```
int sum = numbers.parallelStream().collect(reduce(0, Integer::sum));
System.out.println(sum);
```

Chapter TEN

Concurrency and Multithreading

Answers

1. The correct answer is C.

Explanation:

- **A)** `Thread thread = Thread.ofVirtual(); thread.start(task);`
  - This option is incorrect because `Thread.ofVirtual()` returns a `Thread.Builder`, not a `Thread`. The `start()` method on `Thread.Builder` takes a `Runnable`, but this syntax is incorrect.
- **B)** `Thread thread = Thread.ofVirtual().unstarted(task).run();`
  - This option is incorrect because calling `run()` directly does not start a new thread. It executes the task in the current thread.

- **C)** `Thread thread = Thread.ofVirtual().start(task);`
  - This option is correct. It uses the new thread builder API in Java 21 to create and start a virtual thread in one step.
- **D)** `Thread thread = Thread.ofVirtual(); task.run();`
  - This option is incorrect because it doesn't actually start a new thread. It creates a thread builder but doesn't use it, and then runs the task in the current thread.
- **E)** `Thread thread = Thread.start(task);`
  - This option is incorrect because `Thread.start(task)` is not a valid static method in Java 21.

2. The correct answer is B.

Explanation: - **A)** `synchronized (this) { counter++; }` - This option is incorrect because `this` cannot be used in a static context. In the `main` method, `this` is not available. For a static field like `counter`, you need to synchronize on a static object or class.

- **B)** `synchronized (Main.class) { counter++; }`
  - This option is correct because synchronizing on `Main.class` ensures that only one thread can enter the synchronized block at a time for all instances of `Main`, which is appropriate for protecting static fields like `counter`.
- **C)** `synchronized (task) { counter++; }`
  - This option is incorrect because `task` is a `Runnable` object, and synchronizing on it does not effectively control access to the shared static field `counter`.
- **D)** `synchronized (counter) { counter++; }`
  - This option is incorrect because `counter` is a primitive type (`int`), and you cannot synchronize on a primitive type. Synchronization requires an object.
- **E)** `synchronized (System.out) { counter++; }`
  - This option is incorrect because synchronizing on `System.out` is not related to controlling access to `counter`. It would also interfere with other potential uses of `System.out`.

3. The correct answers are B and C.

Explanation:

- **A)** `AtomicInteger` is part of the `java.util.concurrent.atomic` package, but it does not provide atomic operations for increment and decrement.
  - This statement is incorrect. `AtomicInteger` provides atomic operations for increment and decrement, such as `incrementAndGet()` and `decrementAndGet()`.
- **B)** `AtomicReference` can only be used with reference types, not primitive types.
  - This statement is correct. `AtomicReference` is designed to work with reference types and cannot be used with primitive types directly.
- **C)** `AtomicLong` supports atomic operations on `long` values, including `getAndIncrement()` and `compareAndSet()` methods.
  - This statement is correct. `AtomicLong` provides atomic operations on `long` values, including `getAndIncrement()` and `compareAndSet()` methods.
- **D)** `AtomicBoolean` can be used to perform atomic arithmetic operations on `boolean` values.
  - This statement is incorrect. `AtomicBoolean` is used for atomic updates to `boolean` values, but it does not support atomic arithmetic operations.

4. The correct answer is A.

Explanation:

- **A)**

```
lock.lock();
try {
    count++;
} finally {
```

```
    lock.unlock();
}
```

- This option correctly acquires the lock before modifying the shared resource and ensures the lock is released in the `finally` block, which is the proper use of the `Lock` interface.
- **B)**

```
lock.lock();
count++;
lock.unlock();
```

- This option is incorrect because if an exception occurs between `lock.lock()` and `lock.unlock()`, the lock will not be released, potentially causing a deadlock.
- **C)**

```
try {
    lock.lock(() -> {
        count++;
    });
} finally {
    lock.unlock();
}
```

- This option is incorrect because that's not a valid `lock.lock()` call.
- **D)**

```
synchronized(lock) {
    count++;
}
```

- This option is incorrect because the `synchronized` block is used with the `lock` object itself, which is not the correct usage of the `Lock` interface and does not provide the intended functionality.

5. The correct answer is A.

Explanation:

- **A)**

```
try (ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor()) {
    executor.submit(() -> System.out.println("Task executed"));
}
```

- This option is correct. It uses the `try-with-resources` block with the new `Executors.newVirtualThreadPerTaskExecutor()` method introduced in Java 21. The `ExecutorService` will be automatically closed when the `try` block exits, eliminating the need for explicit shutdown calls.
- **B)**

```
try (ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor()) {
    executor.submit(() -> System.out.println("Task executed"));
} finally {
    executor.shutdown();
}
```

- This option is incorrect because it unnecessarily calls `shutdown()` in the finally block. With `try-with-resources`, the `ExecutorService` is automatically closed, making the explicit `shutdown()` call redundant and potentially harmful.
- **C)**

```
ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor();
try {
    executor.submit(() -> System.out.println("Task executed"));
} finally {
    executor.close();
}
```

- This option is incorrect because it doesn't use the `try-with-resources` syntax. While it does correctly close the `ExecutorService`, it doesn't take advantage of the automatic resource management provided by `try-with-resources`.
- **D)**

```
try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
    executor.submit(() -> System.out.println("Task executed"));
    executor.awaitTermination(1, TimeUnit.SECONDS);
}
```

- This option is incorrect because it unnecessarily calls `awaitTermination()`. In a `try-with-resources` block, the `ExecutorService` is automatically closed when the block exits, making the explicit wait for termination unnecessary and potentially causing the thread to block for 1 second.

6. The correct answer is D.

Explanation:

- **A)**

```
try (ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor()) {
    Future<Integer> future = executor.submit(task);
    System.out.println(future.get());
}
```

- This option is incorrect because it doesn't handle the potential `InterruptedException` and `ExecutionException` that `future.get()` can throw.
- **B)**

```
try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
    Future<Integer> future = executor.submit(task);
    Integer result = future.get(1, TimeUnit.SECONDS);
    System.out.println(result);
}
```

- This option is incorrect because it doesn't handle the potential exceptions (`InterruptedException`, `ExecutionException`, and `TimeoutException`) that `future.get(long, TimeUnit)` can throw.
- **C)**

```
try (ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor()) {
    Future<Integer> future = executor.submit(task);
    executor.shutdown();
    System.out.println(future.get());
}
```

- This option is incorrect because it unnecessarily calls `executor.shutdown()`. In a `try-with-resources` block, the `ExecutorService` is automatically closed when the block exits. Also, it doesn't handle the potential exceptions from `future.get()`.
- **D)**

```
try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
    Future<Integer> future = executor.submit(task);
    try {
        Integer result = future.get();
        System.out.println(result);
    } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
    }
}
```

- This option is correct. It uses `try-with-resources` to automatically close the `ExecutorService`, properly submits the `Callable` task, retrieves the result using `Future.get()`, and handles the potential `InterruptedException` and `ExecutionException` that might be thrown.

7. The correct answer is A.

Explanation:

- **A)** `ConcurrentHashMap` allows concurrent read and write operations, and retrieval operations do not block even when updates are being made.
  - This statement is correct. `ConcurrentHashMap` is designed to handle concurrent access, allowing multiple threads to read and write simultaneously without blocking read operations during updates.
- **B)** `CopyOnWriteArrayList` is optimized for scenarios with a high number of write operations compared to read operations.
  - This statement is incorrect. `CopyOnWriteArrayList` is optimized for scenarios where read operations are far more frequent than write operations because it creates a new copy of the array on each write, which can be costly if writes are frequent.
- **C)** `ConcurrentSkipListSet` does not kept elements sorted.
  - This statement is incorrect. `ConcurrentSkipListSet` keep elements according to their natural ordering, or by a `Comparator` provided at set creation time.
- **D)** `BlockingQueue` implementations like `LinkedBlockingQueue` allow elements to be added and removed concurrently without any internal locking mechanisms.
  - This statement is incorrect. `BlockingQueue` implementations like `LinkedBlockingQueue` do use internal locking mechanisms to handle concurrent access safely.

8. The correct answer is B.

Explanation:

- **A)** Parallel streams always improve the performance of a program by utilizing multiple threads.
  - This statement is incorrect because parallel streams do not always improve performance. The overhead of managing multiple threads can sometimes outweigh the benefits, especially for small datasets or simple operations.
- **B)** Parallel streams can lead to incorrect results if the operations performed are not thread-safe.
  - This statement is correct. When using parallel streams, care must be taken to ensure that the operations performed on the elements are thread-safe. Failure to do so can lead to race conditions and incorrect results.
- **C)** The order of elements in a parallel stream is always preserved compared to the original stream.
  - This statement is incorrect. The order of elements in a parallel stream is not guaranteed to be the same as in the original stream unless special care is taken to preserve the order, such as using ordered stream operations.
- **D)** Using parallel streams guarantees that the operations on elements will execute in a fixed order.
  - This statement is incorrect because parallel streams do not guarantee the order of execution of operations on elements. The operations may execute in a non-deterministic order due to the concurrent nature of parallel processing.

9. The correct answer is B.

Explanation:

- **A)**

```
int sum = numbers.parallelStream().reduce(1, Integer::sum);
System.out.println(sum);
```

- This option is incorrect because it uses `1` as the identity value. The identity value for sum should be `0`, as it is the neutral element for addition. Starting the reduction with `1` will result in an incorrect sum that is incremented by `1`.

- **B)**

```
int sum = numbers.parallelStream().reduce(0, Integer::sum).collect();
System.out.println(sum);
```

- This option is correct. It correctly uses `parallelStream()` to create a parallel stream and the `reduce` method with the identity value `0` and the method reference `Integer::sum` to sum the elements.

- **C)**

```
int sum = numbers.stream().reduce(0, Integer::sum);
System.out.println(sum);
```

- This option is incorrect because it uses a sequential stream (`stream()`) instead of a parallel stream. While it correctly sums the elements, it does not demonstrate the use of a parallel stream as specified in the question.

- **D)**

```
int sum = numbers.parallelStream().collect(reduce(0, Integer::sum));
System.out.println(sum);
```

- This option is incorrect because it attempts to use the `collect()` method in combination with `reduce()`, which is not the correct syntax. The `collect()` method is used for mutable reduction and is typically used with collectors, not with the `reduce()` operation directly.

Chapter ELEVEN

The Date/Time API

Chapter Content

- Core Date/Time Classes
  - The `LocalDate` Class
  - The `LocalTime` Class
  - The `LocalDateTime` Class
  - The `Instant` Class
  - The `Period` Class
  - The `Duration` Class
- Time Zones and Daylight Savings
  - The `ZoneId` and `ZoneOffset` Classes
  - The `ZonedDateTime` Class
  - Daylight Savings
  - The `OffsetDateTime` and `OffsetTime` Classes
  - Parsing and Formatting
- Key Points
- Practice Questions

Core Date/Time Classes

Java 8 introduced a new Date/Time API in the `java.time` package. The classes that belong to this API are immutable and thread-safe.

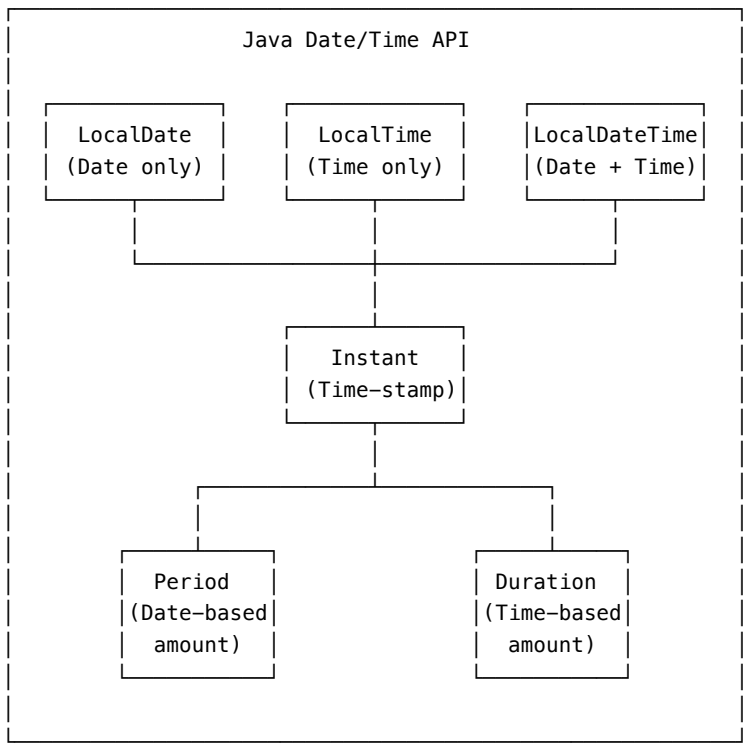
In this section, we'll review the following classes:

- **LocalDate:** Represents a date with the year, month, and day of the month information. For example, `2025-08-25`.
- **LocalTime:** Represents a time with hour, minute, second, and nanosecond information. For example, `13:21:05.123456789`.
- **LocalDateTime:** A combination of the above. For example, `2025-08-25T13:21:05.123456789`.
- **Instant:** Represents a single point in time with seconds and nanoseconds. For example, `1970-01-01T00:00:00Z (seconds since epoch: 923456789, nanoseconds: 186,054,812)`.
- **Period:** Represents an amount of time in terms of years, months, and days. For example, `5 years, 2 months, and 9 days`.
- **Duration:** Represents an amount of time in terms of seconds and nanoseconds. For example, `12.87656 seconds`.

With the exception of `Instant`, these classes don't store or represent a time zone.

Also, `LocalDate`, `LocalTime`, `LocalDateTime`, and `Instant` implement the interface `java.time.temporal.Temporal`, so they all have similar methods. `Period` and `Duration` implement the interface `java.time.temporal.TemporalAmount`, which also makes them very similar.

Here's a diagram to help you visualize the classes of the Date/Time API:



The `LocalDate` Class

The key to learning how to use this class is to understand that it holds the year, month, day, and derived information of a date. All of its methods use this information or have a version to work with each of these components.

The following are the most important methods of this class.

To create an instance, we can use the static method `of`:

```
// With year (-999999999 to 999999999), month (1 to 12), day of the month (1 - 31)
LocalDate newYear2001 = LocalDate.of(2001, 1, 1);
// This version uses the enum java.time.Month
LocalDate newYear2002 = LocalDate.of(2002, Month.JANUARY, 1);
```

Notice that, unlike `java.util.Date`, months start from one. If you try to create a date with invalid values (like February 29 on a non-leap year), an exception will be thrown. For today's date, use `now()`:

```
LocalDate today = LocalDate.now();
```

Once we have an instance of `LocalDate`, we can get the year, the month, and the day with methods like the following:

```
int year = today.getYear();
int month = today.getMonthValue();
Month monthAsEnum = today.getMonth(); // As an enum: JANUARY, FEBRUARY, etc.
int dayYear = today.getDayOfYear();
int dayMonth = today.getDayOfMonth();
DayOfWeek dayWeekEnum = today.getDayOfWeek(); // As an enum: MONDAY, TUESDAY, etc.
```

We can also use the `get` method:

```
int get(java.time.temporal.TemporalField field); // value as int
long getLong(java.time.temporal.TemporalField field); // value as long
```

This method takes an implementation of the interface `java.time.temporal.TemporalField` to access a specific field of a date. `java.time.temporal.ChronoField` is an enumeration that implements this interface, so we can have, for example:

```
int year2 = today.get(ChronoField.YEAR);
int month2 = today.get(ChronoField.MONTH_OF_YEAR);
int dayYear2 = today.get(ChronoField.DAY_OF_YEAR);
int dayMonth2 = today.get(ChronoField.DAY_OF_MONTH);
int dayWeek = today.get(ChronoField.DAY_OF_WEEK);
long dayEpoch = today.getLong(ChronoField.EPOCH_DAY);
```

The supported values for `ChronoField` are:

- `DAY_OF_WEEK`
- `ALIGNED_DAY_OF_WEEK_IN_MONTH`
- `ALIGNED_DAY_OF_WEEK_IN_YEAR`
- `DAY_OF_MONTH`
- `DAY_OF_YEAR`
- `EPOCH_DAY`
- `ALIGNED_WEEK_OF_MONTH`
- `ALIGNED_WEEK_OF_YEAR`
- `MONTH_OF_YEAR`
- `PROLEPTIC_MONTH`
- `YEAR_OF_ERA`
- `YEAR`
- `ERA`

Using an unsupported value will throw an exception. The same is true when getting a value that doesn't fit into an `int` with `get(TemporalField)`.

To compare a `LocalDate` against another instance, we have three methods and another one for leap years:

```
boolean after = newYear2001.isAfter(newYear2002); // false
boolean before = newYear2001.isBefore(newYear2002); // true
boolean equal = newYear2001.equals(newYear2002); // false
boolean leapYear = newYear2001.isLeapYear(); // false
```

Once an instance of this class is created, it cannot be modified, but we can create another instance from an existing one.

One way is by using the `with()` method and its variations:

```
LocalDate newYear2003 = newYear2001.with(ChronoField.YEAR, 2003);
LocalDate newYear2004 = newYear2001.withYear(2004);
LocalDate december2001 = newYear2001.withMonth(12);
LocalDate february2001 = newYear2001.withDayOfYear(32);
// Since these methods return a new instance, we can chain them!
LocalDate xmas2001 = newYear2001.withMonth(12).withDayOfMonth(25);
```

Another way is by adding or subtracting years, months, days, or even weeks:

```
// Adding
LocalDate newYear2005 = newYear2001.plusYears(4);
LocalDate march2001 = newYear2001.plusMonths(2);
LocalDate january15_2001 = newYear2001.plusDays(14);
LocalDate lastWeekJanuary2001 = newYear2001.plusWeeks(3);
LocalDate newYear2006 = newYear2001.plus(5, ChronoUnit.YEARS);

// Subtracting
LocalDate newYear2000 = newYear2001.minusYears(1);
LocalDate nov2000 = newYear2001.minusMonths(2);
LocalDate dec30_2000 = newYear2001.minusDays(2);
LocalDate lastWeekDec2000 = newYear2001.minusWeeks(1);
LocalDate newYear1999 = newYear2001.minus(2, ChronoUnit.YEARS);
```

Notice that the plus and minus methods take a `java.time.temporal.ChronoUnit` enumeration, which is different from `java.time.temporal.ChronoField`. The supported values are:

- `DAYS`
- `WEEKS`
- `MONTHS`
- `YEARS`
- `DECADES`
- `CENTURIES`
- `MILLENNIA`
- `ERAS`

Finally, the method `toString()` returns the date in the format `uuuu-MM-dd`:

```
System.out.println(newYear2001.toString()); // Prints 2001-01-01
```

The `LocalTime` Class

The key to learning how to use this class is to keep in mind that it holds the hour, minutes, seconds, and nanoseconds. All of its methods use this information or have a version to work with each of them.

The following are the most important methods of this class. As you can see, they are the same (or very similar) methods as `LocalDate`, adapted to work with time (hours, minutes, seconds) instead of date (days, months, years).

To create an instance, we can use the `static` method `of`:

```
// With hour (0–23) and minutes (0–59)
LocalTime fiveThirty = LocalTime.of(5, 30);
// With hour, minutes, and seconds (0–59)
LocalTime noon = LocalTime.of(12, 0, 0);
// With hour, minutes, seconds, and nanoseconds (0–999_999_999)
LocalTime almostMidnight = LocalTime.of(23, 59, 59, 999_999_999);
```

If you try to create a time with an invalid value (like `LocalTime.of(24, 0)`), an exception will be thrown. To get the current time, use `now()`:

```
LocalTime now = LocalTime.now();
```

Once we have an instance of `LocalTime`, we can get the hour, the minutes, and other information with methods like the following:

```
int hour = now.getHour();
int minute = now.getMinute();
int second = now.getSecond();
int nanosecond = now.getNano();
```

We can also use the `get()` method:

```
int value = now.get(java.time.temporal.TemporalField field); // value as int
long valueLong = now.getLong(java.time.temporal.TemporalField field); // value as long
```

Just like in the case of `LocalDate`, we can have, for example:

```
int hourAMPM = now.get(ChronoField.HOUR_OF_AMPM); // 0 – 11
int hourDay = now.get(ChronoField.HOUR_OF_DAY); // 0 – 23
int minuteDay = now.get(ChronoField.MINUTE_OF_DAY); // 0 – 1,439
int minuteHour = now.get(ChronoField.MINUTE_OF_HOUR); // 0 – 59
int secondDay = now.get(ChronoField.SECOND_OF_DAY); // 0 – 86,399
int secondMinute = now.get(ChronoField.SECOND_OF_MINUTE); // 0 – 59
long nanoDay = now.getLong(ChronoField.NANO_OF_DAY); // 0–86_399_999_999
int nanoSecond = now.get(ChronoField.NANO_OF_SECOND); // 0–999_999_999
```

The supported values for `ChronoField` are:

- `NANO_OF_SECOND`
- `NANO_OF_DAY`
- `MICRO_OF_SECOND`
- `MICRO_OF_DAY`
- `MILLI_OF_SECOND`
- `MILLI_OF_DAY`
- `SECOND_OF_MINUTE`
- `SECOND_OF_DAY`
- `MINUTE_OF_HOUR`
- `MINUTE_OF_DAY`
- `HOUR_OF_AMPM`
- `CLOCK_HOUR_OF_AMPM`
- `HOUR_OF_DAY`
- `CLOCK_HOUR_OF_DAY`
- `AMPM_OF_DAY`

Using a different value will throw an exception. The same is true when getting a value that doesn't fit into an `int` using `get(TemporalField)`.

To check a time object against another one, we have three methods:

```
boolean after = fiveThirty.isAfter(noon); // false
boolean before = fiveThirty.isBefore(noon); // true
boolean equal = noon.equals(almostMidnight); // false
```

Like `LocalDate`, once an instance of `LocalTime` is created we cannot modify it, but we can create another instance from an existing one.

One way is through the `with` method and its versions:

```
LocalTime ten = noon.with(ChronoField.HOUR_OF_DAY, 10);
LocalTime eight = noon.withHour(8);
LocalTime twelveThirty = noon.withMinute(30);
LocalTime thirtyTwoSeconds = noon.withSecond(32);
// Since these methods return a new instance, we can chain them!
LocalTime secondsNano = noon.withSecond(20).withNano(999_999);
```

Of course, another way is by adding or subtracting hours, minutes, seconds, or nanoseconds:

```
// Adding
LocalTime sixThirty = fiveThirty.plusHours(1);
LocalTime fiveForty = fiveThirty.plusMinutes(10);
LocalTime plusSeconds = fiveThirty.plusSeconds(14);
LocalTime plusNanos = fiveThirty.plusNanos(99_999_999);
LocalTime sevenThirty = fiveThirty.plus(2, ChronoUnit.HOURS);

// Subtracting
LocalTime fourThirty = fiveThirty.minusHours(1);
LocalTime fiveTen = fiveThirty.minusMinutes(20);
LocalTime minusSeconds = fiveThirty.minusSeconds(2);
LocalTime minusNanos = fiveThirty.minusNanos(1);
LocalTime fiveTwenty = fiveThirty.minus(10, ChronoUnit.MINUTES);
```

Notice that the `plus` and `minus` versions take a `java.time.temporal.ChronoUnit` enumeration, which is different from `java.time.temporal.ChronoField`. The supported values are:

- `NANOS`
- `MICROS`
- `MILLIS`
- `SECONDS`
- `MINUTES`
- `HOURS`
- `HALF_DAYS`

Finally, the method `toString()` returns the time in the format `HH:mm:ss.SSSSSSSSS`, omitting the parts with value zero (for example, just returning `HH:mm` if it has zero seconds/nanoseconds):

```
System.out.println(fiveThirty.toString()); // Prints 05:30
```

The `LocalDateTime` Class

The key to learning how to use this class is to remember that it combines `LocalDate` and `LocalTime` classes.

It represents both a date and a time, with information like year, month, day, hours, minutes, seconds, and nanoseconds. Other fields, such as day of the year, day of the week, and week of year can also be accessed.

To create an instance, we can use either the static method `of()` or from a `LocalDate` or `LocalTime` instance:

```
// Setting seconds and nanoseconds to zero
LocalDateTime dt1 = LocalDateTime.of(2024, 9, 19, 14, 5);
// Setting nanoseconds to zero
LocalDateTime dt2 = LocalDateTime.of(2024, 9, 19, 14, 5, 20);
// Setting all fields
LocalDateTime dt3 = LocalDateTime.of(2024, 9, 19, 14, 5, 20, 9);
// Assuming this date
LocalDate date = LocalDate.now();
// And this time
LocalTime time = LocalTime.now();
// Combine the above date with the given time like this
LocalDateTime dt4 = date.atTime(14, 30, 59, 999999);
// Or this
LocalDateTime dt5 = date.atTime(time);
// Combine this time with the given date. Notice that LocalTime
// only has this method to be combined with a LocalDate
LocalDateTime dt6 = time.atDate(date);
```

If you try to create an instance with an invalid value or date, an exception will be thrown. To get the current date/time use `now()`:

```
LocalDateTime now = LocalDateTime.now();
```

Once we have an instance of `LocalDateTime`, we can get the information with the methods we know from `LocalDate` and `LocalTime`, such as:

```
int year = now.getYear();
int dayYear = now.getDayOfYear();
int hour = now.getHour();
int minute = now.getMinute();
```

We can also use the `get()` method:

```
int get(java.time.temporal.TemporalField field)
long getLong(java.time.temporal.TemporalField field)
```

For example:

```
int month = now.get(ChronoField.MONTH_OF_YEAR);
int minuteHour = now.get(ChronoField.MINUTE_OF_HOUR);
```

The supported values for `ChronoField` are:

- `NANO_OF_SECOND`
- `NANO_OF_DAY`
- `MICRO_OF_SECOND`
- `MICRO_OF_DAY`
- `MILLI_OF_SECOND`
- `MILLI_OF_DAY`
- `SECOND_OF_MINUTE`
- `SECOND_OF_DAY`
- `MINUTE_OF_HOUR`
- `MINUTE_OF_DAY`
- `HOUR_OF_AMPM`
- `CLOCK_HOUR_OF_AMPM`
- `HOUR_OF_DAY`
- `CLOCK_HOUR_OF_DAY`
- `AMPM_OF_DAY`
- `DAY_OF_WEEK`

- ALIGNED_DAY_OF_WEEK_IN_MONTH
- ALIGNED_DAY_OF_WEEK_IN_YEAR
- DAY_OF_MONTH
- DAY_OF_YEAR
- EPOCH_DAY
- ALIGNED_WEEK_OF_MONTH
- ALIGNED_WEEK_OF_YEAR
- MONTH_OF_YEAR
- PROLEPTIC_MONTH
- YEAR_OF_ERA
- YEAR
- ERA

Using a different value will throw an exception. The same is true when getting a value that doesn't fit into an `int` with `get(TemporalField)`.

To check a `LocalDateTime` object against another one, we have three methods:

```
boolean after = now.isAfter(dt1); // true
boolean before = now.isBefore(dt1); // false
boolean equal = now.equals(dt1); // false
```

Once an instance of `LocalTime` is created, we cannot modify it, but we can create another instance from an existing one.

One way is through the with method and its versions:

```
LocalDateTime dt7 = now.with(ChronoField.HOUR_OF_DAY, 10);
LocalDateTime dt8 = now.withMonth(8);
// Since these methods return a new instance, we can chain them!
LocalDateTime dt9 = now.withYear(2013).withMinute(0);
```

Another way is by adding or subtracting years, months, days, weeks, hours, minutes, seconds, or nanoseconds:

```
// Adding
LocalDateTime dt10 = now.plusYears(4);
LocalDateTime dt11 = now.plusWeeks(3);
LocalDateTime dt12 = now.plus(2, ChronoUnit.HOURS);

// Subtracting
LocalDateTime dt13 = now.minusMonths(2);
LocalDateTime dt14 = now.minusNanos(1);
LocalDateTime dt15 = now.minus(10, ChronoUnit.SECONDS);
```

In this case, the supported values for `ChronoUnit` are:

- NANOS
- MICROS
- MILLIS
- SECONDS
- MINUTES
- HOURS
- HALF_DAYS
- DAYS
- WEEKS
- MONTHS
- YEARS
- DECADES
- CENTURIES

- MILLENNIA
- ERAS

Finally, the method `toString()` returns the date-time in the format `uuuu-MM-dd'T'HH:mm:ss.SSSSSSSSS`, omitting the parts with value zero, for example:

```
System.out.println(dt1.toString()); // Prints 2024-09-19T14:05
```

The `Instant` Class

Although in practical terms, a `LocalDateTime` instance represents an instant in the timeline, there is another class that may be more appropriate.

The `java.time.Instant` class represents an instant in the number of seconds that have passed since the epoch, a convention used in UNIX/POSIX systems and set at midnight of January 1, 1970 UTC time.

From that date, time is measured in 86,400 seconds per day. This information is stored as a `long`. The class also supports nanosecond precision, stored as an `int`.

You can create an instance of this class with the following methods:

```
// Setting seconds
Instant fiveSecondsAfterEpoch = Instant.ofEpochSecond(5);
// Setting seconds and nanoseconds (can be negative)
Instant sixSecTwoNanBeforeEpoch = Instant.ofEpochSecond(-6, -2);
// Setting milliseconds after (can be before also) epoch
Instant fiftyMilliSecondsAfterEpoch = Instant.ofEpochMilli(50);
```

For the current instance of the system clock use:

```
Instant now = Instant.now();
```

Once we have an instance of `Instant`, we can get the information with the following methods:

```
long seconds = now.getEpochSecond(); // Gets the seconds
int nanos1 = now.getNano(); // Gets the nanoseconds
// Gets the value as an int
int millis = now.get(ChronoField.MILLI_OF_SECOND);
// Gets the value as a long
long nanos2 = now.getLong(ChronoField.NANO_OF_SECOND);
```

The supported `ChronoField` values are:

- NANO_OF_SECOND
- MICRO_OF_SECOND
- MILLI_OF_SECOND
- INSTANT_SECONDS

Using any other value will throw an exception. The same is true when getting a value that doesn't fit into an `int` using `get(TemporalField)`.

To check an `Instant` object against another one, we have three methods:

```
boolean after = now.isAfter(fiveSecondsAfterEpoch); // true
boolean before = now.isBefore(fiveSecondsAfterEpoch); // false
boolean equal = now.equals(fiveSecondsAfterEpoch); // false
```

Once an instance of this object is created, we cannot modify it, but we can create another instance from an existing one.

One way is by using the `with` method:

```
Instant i1 = now.with(ChronoField.NANO_OF_SECOND, 10);
```

Another way is by adding or subtracting seconds, milliseconds, or nanoseconds:

```
// Adding
Instant i10 = now.plusSeconds(400);
Instant i11 = now.plusMillis(98622200);
Instant i12 = now.plusNanos(300013890);
Instant i13 = now.plus(2, ChronoUnit.MINUTES);

// Subtracting
Instant i14 = now.minusSeconds(2);
Instant i15 = now.minusMillis(1);
Instant i16 = now.minusNanos(1);
Instant i17 = now.minus(10, ChronoUnit.SECONDS);
```

The supported `ChronoUnit` values are:

- `NANOS`
- `MICROS`
- `MILLIS`
- `SECONDS`
- `MINUTES`
- `HOURS`
- `HALF_DAYS`
- `DAYS`

Finally, the method `toString()` returns a representation of the `Instant` in the format `uuuu-MM-dd'T'HH:mm:ss.SSSSSSSSS`, for example:

```
// Prints 1970-01-01T00:00:00.050Z
System.out.println(fiftyMilliSecondsAfterEpoch.toString());
```

Notice that it contains zone time information (`Z`). This is because `Instant` represents a point in time from the epoch of `1970-01-01Z` in the UTC zone time.

The `Period` Class

The `java.time.Period` class represents an amount of time in terms of years, months, and days.

You can create an instance of this class with the following methods:

```
// Setting years, months, days (can be negative)
Period period5y4m3d = Period.of(5, 4, 3);
// Setting days (can be negative), years and months will be zero
Period period2d = Period.ofDays(2);
// Setting months (can be negative), years and days will be zero
Period period2m = Period.ofMonths(2);
// Setting weeks (can be negative). The resulting period will
// be in days (1 week = 7 days). Years and months will be zero
Period period14d = Period.ofWeeks(2);
// Setting years (can be negative), days and months will be zero
Period period2y = Period.ofYears(2);
```

A `Period` can also be thought of as the difference between two `LocalDates`. Luckily, there's a method that supports this concept:

```
LocalDate march2003 = LocalDate.of(2003, 3, 1);
LocalDate may2003 = LocalDate.of(2003, 5, 1);
Period dif = Period.between(march2003, may2003); // 2 months
```

The start date is included, but not the end date.

Be careful about how the date is calculated.

First, complete months are counted, and then the remaining number of days is calculated. The number of months is then split into years (1 year equals 12 months). A month is considered if the end day of the month is greater than or equal to the start day of the month.

The result of this method can be a negative period if the end is before the start (year, month, and day will have a negative sign).

Here are some examples:

```
// dif1 will be 1 year 2 months 2 days
Period dif1 = Period.between(LocalDate.of(2000, 2, 10), LocalDate.of(2001, 4, 12));
// dif2 will be 25 days
Period dif2 = Period.between(LocalDate.of(2013, 5, 9), LocalDate.of(2013, 6, 3));
// dif3 will be -2 years -3 days
Period dif3 = Period.between(LocalDate.of(2014, 11, 3), LocalDate.of(2012, 10, 31));
```

Once we have an instance of `Period`, we can get the information with the following methods:

```
int days = period5y4m3d.getDays();
int months = period5y4m3d.getMonths();
int year = period5y4m3d.getYears();
long days2 = period5y4m3d.get(ChronoUnit.DAYS);
```

Notice that the `get` method returns a `long` type.

Also, the supported `ChronoUnit` values are:

- `DAYS`
- `MONTHS`
- `YEARS`

Using any other value will throw an exception.

Once an instance of `Period` is created, we cannot modify it, but we can create another instance based on an existing one.

One way is by using the `with` method and its versions:

```
Period period8d = period2d.withDays(8);
// Since these methods return a new instance, we can chain them!
Period period2y1m2d = period2d.withYears(2).withMonths(1);
```

Another way is by adding or subtracting years, months, or days:

```
// Adding
Period period9y4m3d = period5y4m3d.plusYears(4);
Period period5y7m3d = period5y4m3d.plusMonths(3);
Period period5y4m6d = period5y4m3d.plusDays(3);
Period period7y4m3d = period5y4m3d.plus(period2y);

// Subtracting
Period period5y4m1d = period5y4m3d.minusYears(2);
Period period5y3m3d = period5y4m3d.minusMonths(1);
Period period5y4m2d = period5y4m3d.minusDays(1);
Period period3y4m3d = period5y4m3d.minus(period2y);
```

The methods `plus` and `minus` take an implementation of the interface `java.time.temporal.TemporalAmount` (another instance of `Period` or an instance of `Duration`).

Finally, the `toString()` method returns the period in the format `P<years>Y<months>M<days>D`, for example:

```
System.out.println(period5y4m3d.toString()); // Prints P5Y4M3D
```

A zero period will be represented as zero days, `P0D`.

The `Duration` Class

The `java.time.Duration` class is similar to the `Period` class, the only thing is that it represents an amount of time in terms of seconds and nanoseconds.

You can create an instance of this class with the following methods:

```
Duration oneDay = Duration.ofDays(1); // 1 day = 86400 seconds
Duration oneHour = Duration.ofHours(1); // 1 hour = 3600 seconds
Duration oneMin = Duration.ofMinutes(1); // 1 minute = 60 seconds
Duration tenSeconds = Duration.ofSeconds(10);
// Set seconds and nanoseconds (if they are outside the range
// 0 to 999,999,999, the seconds will be altered, like below)
Duration twoSeconds = Duration.ofSeconds(1, 1000000000);
// Seconds and nanoseconds are extracted from the passed millisecs
Duration oneSecondFromMillis = Duration.ofMillis(2);
// Seconds and nanoseconds are extracted from the passed nanos
Duration oneSecondFromNanos = Duration.ofNanos(1000000000);
Duration oneSecond = Duration.of(1, ChronoUnit.SECONDS);
```

Valid values of `ChronoUnit` for the method `Duration.of(long amount, TemporalUnit unit)` are:

- `NANOS`
- `MICROS`
- `MILLIS`
- `SECONDS`
- `MINUTES`
- `HOURS`
- `HALF_DAYS`
- `DAYS`

A `Duration` can also be created as the difference between two implementations of the interface `java.time.temporal.Temporal`, as long as they support seconds (and for more accuracy, nanoseconds), like `LocalTime`, `LocalDateTime`, and `Instant`. So we can have something like this:

```
Duration diff = Duration.between(Instant.ofEpochSecond(123456789), Instant.ofEpochSecond(99999));
```

The result can be negative if the end is before the start. A negative duration is indicated by a negative sign in the seconds part. For example, a duration of -100 nanoseconds is stored as -1 second plus 999,999,900 nanoseconds.

If the objects are of different types, then the duration is calculated based on the type of the first object. This only works if the first argument is a `LocalTime` and the second is a `LocalDateTime` (because it can be converted to `LocalTime`). Otherwise, an exception is thrown.

Once we have an instance of `Duration`, we can get the information with the following methods:

```
// The nanoseconds part the duration, from 0 to 999,999,999
int nanos = oneSecond.getNano();
// The seconds part of the duration, positive or negative
long seconds = oneSecond.getSeconds();
// It supports SECONDS and NANOS. Other units throw an exception
long oneSec = oneSecond.get(ChronoUnit.SECONDS);
```

Note that the methods `getSeconds()` and `get(TemporalUnit)` return a `long` type. Additionally, the latter only supports `SECONDS` and `NANOS` as arguments.

Once an instance of `Duration` is created, we cannot modify it, but we can create another instance from an existing one. One way is to use the `withNanos` and `withSeconds` methods:

```
Duration duration1sec8nan = oneSecond.withNanos(8);
Duration duration2sec1nan = oneSecond.withSeconds(2).withNanos(1);
```

Another way is by adding or subtracting days, hours, minutes, seconds, milliseconds, or nanoseconds:

```
// Adding
Duration plus4Days = oneSecond.plusDays(4);
Duration plus3Hours = oneSecond.plusHours(3);
Duration plus3Minutes = oneSecond.plusMinutes(3);
Duration plus3Seconds = oneSecond.plusSeconds(3);
Duration plus3Millis = oneSecond.plusMillis(3);
Duration plus3Nanos = oneSecond.plusNanos(3);
Duration plusAnotherDuration = oneSecond.plus(twoSeconds);
Duration plusChronoUnits = oneSecond.plus(1, ChronoUnit.DAYS);

// Subtracting
Duration minus4Days = oneSecond.minusDays(4);
Duration minus3Hours = oneSecond.minusHours(3);
Duration minus3Minutes = oneSecond.minusMinutes(3);
Duration minus3Seconds = oneSecond.minusSeconds(3);
Duration minus3Millis = oneSecond.minusMillis(3);
Duration minus3Nanos = oneSecond.minusNanos(3);
Duration minusAnotherDuration = oneSecond.minus(twoSeconds);
Duration minusChronoUnits = oneSecond.minus(1, ChronoUnit.DAYS);
```

Methods `plus` and `minus` take either another `Duration` or a valid `ChronoUnit` value (the same values used to create an instance).

Finally, the method `toString()` returns the duration with the format `PTnHnMnS`. Any fractional seconds are placed after a decimal point in the seconds section. If a section has a zero value, it's omitted. For example:

```
2 days 4 minutes PT48H4M
45 seconds 99 milliseconds PT45.099S
```

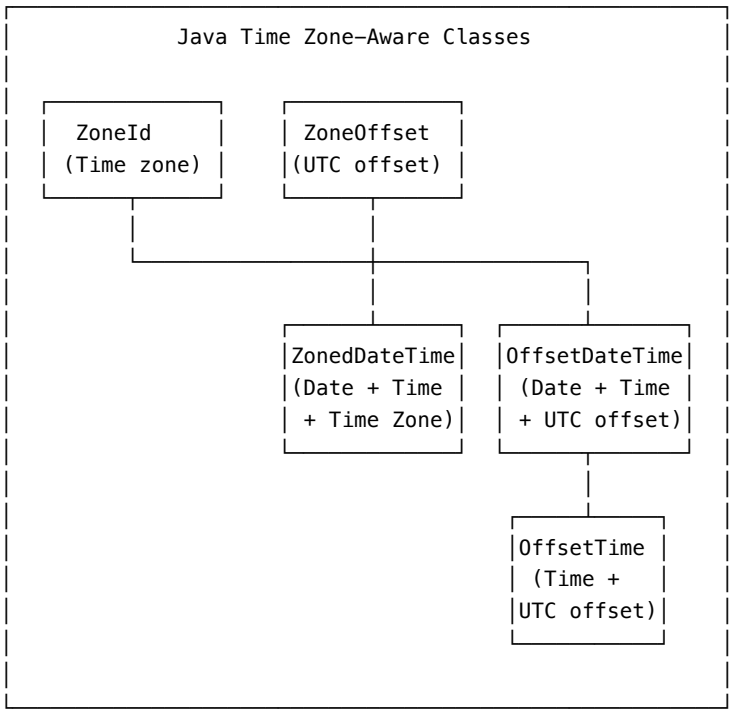
Time Zones and Daylight Savings

If you want to work with time zone information, the Date/Time API has the following classes:

- **ZoneId:** Represents the ID of time zone. For example, `Asia/Tokyo`.
- **ZoneOffset:** Represents a time zone offset. It's a subclass of `ZoneId`. For example, `-06:00`.
- **ZonedDateTime:** Represents a date/time with time zone information. For example, `2025-08-30T20:05:12.463-05:00[America/Mexico_City]`.
- **OffsetDateTime:** Represents a date/time with an offset from UTC/Greenwich. For example, `2025-08-30T20:05:12.463-05:00`.
- **OffsetTime:** Represents a time with an offset from UTC/Greenwich. For example, `20:05:12.463-05:00`.

Just like the classes in the previous section, these are located in the `java.time` package and are immutable.

Here's a diagram to visualize the zone-aware classes:




```
Key Points:
- ZoneId represents a time zone (like "America/New_York")
- ZoneOffset represents a fixed offset from UTC (like "+05:00")
- ZonedDateTime combines LocalDateTime with a ZoneId
- OffsetDateTime combines LocalDateTime with a ZoneOffset
- OffsetTime combines LocalTime with a ZoneOffset
```

The `ZoneId` and `ZoneOffset` Classes

The world is divided into time zones in which the same standard time is kept. By convention, a time zone is expressed as the number of hours different from the Coordinated Universal Time (*UTC*). Since the Greenwich Mean Time (*GMT*) and the Zulu time (*Z*), used in the military, have no offset from *UTC*, they're often used as synonyms.

Java uses the Internet Assigned Numbers Authority (IANA) database of time zones, which keeps a record of all known time zones around the world and is updated many times per year.

Each time zone has an ID, represented by the class `java.time.ZoneId`. There are three types of ID:

The first type just states the offset from UTC/GMT time. They are represented by the class `ZoneOffset` and they consist of digits starting with `+` or `-`, for example, `+02:00`.

The second type also states the offset from UTC/GMT time, but with one of the following prefixes: *UTC*, *GMT*, and *UT*, for example, `UTC+11:00`. They are also represented by the class `ZoneOffset`.

The third type is region-based. These IDs have the format *area/city*, for example, `Europe/London`.

You can get all the available zone IDs with the `static` method:

```
Set<String> getAvailableZoneIds()
```

For example, to print them to the console:

```
ZoneId.getAvailableZoneIds().stream().forEach(System.out::println);
```

To get the zone ID of your system, use the `static` method:

```
ZoneId.systemDefault()
```

Under the hood, it uses `java.util.TimeZone.getDefault()` to find the default time zone and converts it to a `ZoneId`.

If you want to create a specific `ZoneId` object use the method `of()`:

```
ZoneId singaporeZoneId = ZoneId.of("Asia/Singapore");
```

This method parses the ID producing a `ZoneId` or a `ZoneOffset` (which extends from `ZoneId`). A `ZoneOffset` is returned if, for example, ID is `Z`, or starts with `+` or `-`. For example:

```
ZoneId zoneId = ZoneId.of("Z"); // Z represents the zone ID for UTC
ZoneId zoneId2 = ZoneId.of("-2"); // -02:00
```

The rules for this method are:

- If the zone ID equals `Z`, the result is `ZoneOffset.UTC`. Any other letter will throw an exception.
- If the zone ID starts with `+` or `-`, the ID is parsed as a `ZoneOffset` using `ZoneOffset.of(String)`.
- If the zone ID equals `GMT`, `UTC` or `UT` then the result is a `ZoneId` with the same ID and rules equivalent to `ZoneOffset.UTC`.
- If the zone ID starts with `UTC+`, `UTC-`, `GMT+`, `GMT-`, `UT+` or `UT-` then the ID is split in two, with a two or three letter prefix and a suffix starting with the sign. The suffix is parsed as a `ZoneOffset`. The result will be a `ZoneId` with the specified prefix and the normalized offset ID.
- All other IDs are parsed as region-based zone IDs. If the format is invalid (it has to match the expression `[A-Za-z][A-Za-z0-9~/._+-]+`) or is not found, an exception is thrown.

Remember that a `ZoneOffset` represents an offset, generally from UTC. This class has a lot more constructors than `ZoneId`:

```
// The offset must be in the range of -18 to +18
ZoneOffset offsetHours = ZoneOffset.ofHours(1);
// The range is -18 to +18 for hours and 0 to ± 59 for minutes
// If the hours are negative, the minutes must be negative or zero
ZoneOffset offsetHrMin = ZoneOffset.ofHoursMinutes(1, 30);
// The range is -18 to +18 for hours and 0 to ± 59 for mins and secs
// If the hours are negative, mins and secs must be negative or zero
ZoneOffset offsetHrMinSe = ZoneOffset.ofHoursMinutesSeconds(1, 30, 0);
// The offset must be in the range -18:00 to +18:00
// Which corresponds to -64800 to +64800
ZoneOffset offsetTotalSeconds = ZoneOffset.ofTotalSeconds(3600);
// The range must be from +18:00 to -18:00
ZoneOffset offset = ZoneOffset.of("+01:30:00");
```

The formats accepted by the `of()` method are:

- *Z (for UTC)*
- *+h*
- *+hh*
- *+hh:mm*
- *-hh:mm*
- *+hhmm*
- *-hhmm*
- *+hh:mm:ss*
- *-hh:mm:ss*
- *+hhmmss*
- *-hhmmss*

If you pass an invalid format or an out-of-range value to any of these methods, an exception is thrown.

To get the value of the offset, you can use:

```
// Gets the offset as int
int offsetInt = offset.get(ChronoField.OFFSET_SECONDS);
// Gets the offset as long
long offsetLong= offset.getLong(ChronoField.OFFSET_SECONDS);
// Gets the offset in seconds
int offsetSeconds = offset.getTotalSeconds();
```

`ChronoField.OFFSET_SECONDS` is the only accepted value of `ChronoField`, so the three statements above return the same result. Other values throw an exception.

Anyway, once you have a `ZoneId` object, you can use it to create a `ZonedDateTime` instance.

The `ZonedDateTime` Class

A `java.time.ZonedDateTime` object represents a point in time relative to a time zone.

A `ZonedDateTime` object has three parts:

- A date
- A time
- A time zone

This means that it stores all date and time fields to a precision of nanoseconds, and a time zone with a zone offset.

Here's an example:

```
2025-08-31 T08:45:20.000 +02:00[Africa/Cairo]
```

Where the parts are as follows:

Date	Time	Offset	Time zone
`2025-08-31`	`T08:45:20.000`	`+02:00`	`[Africa/Cairo]`

Once you have a `ZoneId` object, you can combine it with a `LocalDate`, a `LocalDateTime`, or an `Instant` to transform it into `ZonedDateTime`:

```
ZoneId australiaZone = ZoneId.of("Australia/Victoria");

LocalDate date = LocalDate.of(2020, 7, 3);
ZonedDateTime zonedDate = date.atStartOfDay(australiaZone);

LocalDateTime dateTime = LocalDateTime.of(2020, 7, 3, 9, 0);
ZonedDateTime zonedDateTime = dateTime.atZone(australiaZone);

Instant instant = Instant.now();
ZonedDateTime zonedInstant = instant.atZone(australiaZone);
```

Or using the `of` method:

```
ZonedDateTime zonedDateTime2 =
    ZonedDateTime.of(LocalDate.now(), LocalTime.now(), australiaZone);
ZonedDateTime zonedDateTime3 =
    ZonedDateTime.of(LocalDateTime.now(), australiaZone);
ZonedDateTime zonedDateTime4 =
```

```
    ZonedDateTime.ofInstant(Instant.now(), australiaZone);
// year, month, day, hours, minutes, seconds, nanoseconds, zoneId
ZonedDateTime zonedDateTime5 =
    ZonedDateTime.of(2025, 1, 30, 13, 59, 59, 999, australiaZone);
```

You can also get the current date/time from the system clock in the default time zone with:

```
ZonedDateTime now = ZonedDateTime.now();
```

From a `ZonedDateTime` you can get `LocalDate`, `LocalTime`, or a `LocalDateTime` (without the time zone part) with:

```
LocalDate currentDate = now.toLocalDate();
LocalTime currentTime = now.toLocalTime();
LocalDateTime currentDateTime = now.toLocalDateTime();
```

`ZonedDateTime` also has most of the methods of `LocalDateTime` that we reviewed in the previous section:

```
// To get the value of a specified field
int day = now.getDayOfMonth();
int dayYear = now.getDayOfYear();
int nanos = now.getNano();
Month monthEnum = now.getMonth();
int year = now.get(ChronoField.YEAR);
long micro = now.getLong(ChronoField.MICRO_OF_DAY);
// This is new, gets the zone offset such as "-03:00"
ZoneOffset offset = now.getOffset();
// To create another instance
ZonedDateTime zdt1 = now.with(ChronoField.HOUR_OF_DAY, 10);
ZonedDateTime zdt2 = now.withSecond(49);
// Since these methods return a new instance, we can chain them!
ZonedDateTime zdt3 = now.withYear(2023).withMonth(12);

// The following two methods are specific to ZonedDateTime
// Returns a copy of the date/time with a different zone, retaining the instant
ZonedDateTime zdt4 = now.withZoneSameInstant(australiaZone);
// Returns a copy of this date/time with a different time zone,
// retaining the local date/time if it's valid for the new time zone
ZonedDateTime zdt5 = now.withZoneSameLocal(australiaZone);

// Adding
ZonedDateTime zdt6 = now.plusDays(4);
ZonedDateTime zdt7 = now.plusWeeks(3);
ZonedDateTime zdt8 = now.plus(2, ChronoUnit.HOURS);

// Subtracting
ZonedDateTime zdt9 = now.minusMinutes(20);
ZonedDateTime zdt10 = now.minusNanos(99999);
ZonedDateTime zdt11 = now.minus(10, ChronoUnit.SECONDS);
```

The `toString()` method returns the date/time in the format of a `LocalDateTime` followed by a `ZoneOffset`, optionally, a `ZoneId` if it is not the same as the offset, and omitting the parts with value zero:

```
// Prints 2024-09-19T00:30Z
System.out.println(
    ZonedDateTime.of(2024, 9, 19, 0, 30, 0, 0, ZoneId.of("Z")));
// Prints, for example, 2024-06-17T19:48:39.113332-04:00[America/Montreal]
```

```
System.out.println(
    ZonedDateTime.now(ZoneId.of("America/Montreal")));
```

Daylight Savings

Many countries in the world adopt what is called Daylight Saving Time (DST), the practice of advancing the clock by an hour when daylight saving time starts, typically in spring but sometimes in late winter or early autumn, depending on the region.

When daylight saving time ends, clocks are set back by an hour. This is done to make better use of natural daylight.

`ZonedDateTime` is fully aware of DST.

For example, let's take a country where DST is fully observed, like Italy (UTC+1 in standard time, UTC+2 in daylight saving time).

In 2023, DST started in Italy on March 26th and ended on October 29th. This means that on:

*March 26, 2023 at **2:00:00** A.M. clocks were turned **forward** 1 hour to*
*March 26, 2023 at **3:00:00** A.M. local daylight time instead*
(So a time like March 26, 2023 2:30:00 A.M. didn't actually exist!)

*October 29, 2023 at **3:00:00** A.M. clocks were turned **backward** 1 hour to*
*October 29, 2023 at **2:00:00** A.M. local daylight time instead*
(So a time like October 29, 2023 2:30:00 A.M. actually existed twice!)

If we create an instance of `LocalDateTime` with this date/time and print it:

```
LocalDateTime ldt = LocalDateTime.of(2023, 3, 26, 2, 30);
System.out.println(ldt);
```

The result will be:

```
2023-03-26T02:30 // Wrong
```

But if we create an instance of `ZonedDateTime` for Italy (notice that the format uses a city, not a country) and print it:

```
ZonedDateTime zdt = ZonedDateTime.of(
   2023, 3, 26, 2, 30, 0, 0, ZoneId.of("Europe/Rome"));
System.out.println(zdt);
```

The result will be just like in the real world when using DST:

```
2023-03-26T03:30+02:00[Europe/Rome] // Correct
```

But be careful. We have to use a regional `ZoneId`, a `ZoneOffset` won't do the trick because this class doesn't have the zone rules information to account for DST:

```
ZonedDateTime zdt1 = ZonedDateTime.of(
   2023, 3, 26, 2, 30, 0, 0, ZoneOffset.ofHours(2));
System.out.println(zdt1);

ZonedDateTime zdt2 = ZonedDateTime.of(
   2023, 3, 26, 2, 30, 0, 0, ZoneId.of("UTC+2"));
System.out.println(zdt2);
```

The result will be:

```
2023-03-26T02:30+02:00              // Wrong
2023-03-26T02:30+02:00[UTC+02:00]   // Wrong
```

When we create an instance of `ZonedDateTime` for Italy, we have to add an hour to see the effect:

```
ZonedDateTime zdt3 = ZonedDateTime.of(
   2023, 10, 29, 2, 30, 0, 0, ZoneId.of("Europe/Rome"));
System.out.println(zdt3);

ZonedDateTime zdt4 = zdt3.plusHours(1);
System.out.println(zdt4);
```

The result will be:

```
2023-10-29T02:30+02:00[Europe/Rome]
2023-10-29T02:30+01:00[Europe/Rome]
```

Otherwise we will be creating the `ZonedDateTime` at 3:00 of the new time:

```
ZonedDateTime zdt5 = ZonedDateTime.of(
        2023, 10, 29, 3, 30, 0, 0, ZoneId.of("Europe/Rome"));
System.out.println(zdt5); // Prints 2023-10-29T03:30+01:00[Europe/Rome]
```

We also need to be careful when adjusting the time across the DST boundary using the `plus()` and `minus()` methods that take a `TemporalAmount` implementation, in other words, a `Period` or a `Duration`. This is because both differ in their treatment of daylight savings time.

Consider one hour before the start of DST in Italy:

```
ZonedDateTime zdt6 = ZonedDateTime.of(
   2023, 3, 26, 1, 0, 0, 0, ZoneId.of("Europe/Rome"));
```

When we add a `Duration` of one day:

```
System.out.println(zdt6.plus(Duration.ofDays(1)));
```

The result is:

```
2023-03-27T02:00+02:00[Europe/Rome]
```

When we add a `Period` of one day:

```
System.out.println(zdt6.plus(Period.ofDays(1)));
```

The result is:

```
2023-03-27T01:00+02:00[Europe/Rome]
```

The reason is that `Period` adds a *conceptual* date, while `Duration` adds *exactly* one day (24 hours or 86,400 seconds) and when it crosses the DST boundary, one hour is added, and the final time is *02:00* instead of *01:00*.

The `OffsetDateTime` and `OffsetTime` Classes

`OffsetDateTime` represents an object with date/time information and an offset from UTC, for example, `2025-01-01T11:30-06:00`.

You may think `Instant`, `OffsetDateTime`, and `ZonedDateTime` are very much alike, after all, they all store the date and time to a nanosecond precision. However, there are subtle but important differences:

- `Instant` represents a point in time in the UTC time zone.
- `OffsetDateTime` represents a point in time with an offset (any offset).
- `ZonedDateTime` represents a point in time in a time zone (any time zone), adding full time zone rules like daylight saving time adjustments.

`OffsetTime` represents a time with an offset from UTC, for example, `11:30-06:00`. The common way to create an instance of these classes is:

```
OffsetDateTime odt = OffsetDateTime.of(
    LocalDateTime.now(), ZoneOffset.of("+03:00"));
OffsetTime ot = OffsetTime.of(
    LocalTime.now(), ZoneOffset.of("-08:00"));

System.out.println(odt);
System.out.println(ot);
```

If you run the above example, the output would be similar to this:

```
2024-06-17T19:19:32.645941+03:00
19:19:32.648413-08:00
```

Both classes have practically the same methods as their `LocalDateTime`, `ZonedDateTime`, and `LocalTime` counterparts. With an offset from UTC and without time zone variations, they always represent an exact instant in time, which may be more suitable for certain types of applications (the Java documentation recommends `OffsetDateTime` when communicating with a database or in a network protocol).

Parsing and Formatting

`java.time.format.DateTimeFormatter` is the class used for parsing and formatting dates. It can be used in two ways:

- The date/time classes (represented by `T`) `LocalDate`, `LocalTime`, `LocalDateTime`, `ZonedDateTime`, and `OffsetDateTime` all have the following three methods:

  ```
  // Formats the date/time object using the specified formatter
  String format(DateTimeFormatter formatter)

  // Obtains an instance of a date/time object (of type T) from a string with a default format
  static T parse(CharSequence text)

  // Obtains an instance of a date/time object (of type T) from a string using a specific formatter
  static T parse(CharSequence text, DateTimeFormatter formatter)
  ```

- `DateTimeFormatter` has the following two methods:

  ```
  // Formats a date/time object using the formatter instance
  String format(TemporalAccessor temporal)

  // Parses the text producing a temporal object
  TemporalAccessor parse(CharSequence text)
  ```

All format methods throw the runtime exception `java.time.DateTimeException`.

All parse methods throw the runtime exception `java.time.format.DateTimeParseException`.

`DateTimeFormatter` provides three ways to format date/time objects:

- Predefined formatters
- Locale-specific formatters
- Formatters with custom patterns

Predefined Formatters

Formatter	Description	Example
`BASIC_ISO_DATE`	Date fields without separators	*20250803*

`ISO_LOCAL_DATE`
`ISO_LOCAL_TIME`
`ISO_LOCAL_DATE_TIME`
	Date fields with separators	*2025-08-03*
		13:40:10
		2025-08-03T13:40:10

Formatter	Description	Example
`ISO_OFFSET_DATEISO_OFFSET_TIMEISO_DATE_TIME`	Date with time separators and zone offset	*2025-08-03+07:0013:40:10+07:002025-08-03T13:40:10+07:00*
`ISO_ZONED_DATE_TIME`	A zoned date and time	*2025-08-03T13:40:10+07:00[Asia/Bangkok]*
`ISO_DATEISO_TIMEISO_DATE_TIME`	Date or Time with or without offsetDateTime with ZoneId	*2025-08-03+07:0013:40:102025-08-03T13:40:10+07:00[Asia/Bangkok]*
`ISO_INSTANT`	Date and Time of an Instant	*2025-08-03T13:40:10Z*
`ISO_ORDINAL_DATE`	Year and day of the year	*2025-200*
`ISO_WEEK_DATE`	Year, week and day of the week	*2025-W34-2*
`RFC_1123_DATE_TIME`	RFC 1123 / RFC 822 date format	*Sun, 3 Aug 2025 13:40:10 GMT*

Locale-specific Formatters

Style	Date	Time
`SHORT`	8/3/15	*1:40 PM*
`MEDIUM`	Aug 03, 2025	*1:40:00 PM*
`LONG`	August 03, 2025	*1:40:00 PM PDT*
`FULL`	Monday, August 03, 2025	*1:40:00 PM PDT*

Custom Patterns

Symbol	Meaning	Examples
`G`	Era	*AD; Anno Domini; A*
`u`	Year	*2025; 15*
`y`	Year of Era	*2025; 15*
`D`	Day of Year	*150*
`M / L`	Month of Year	*7; 07; Jul; July; J*
`d`	Day of Month	*20*
`Q / q`	Quarter of year	*2; 02; Q2; 2nd quarter*
`Y`	Week-based Year	*2025; 15*
`w`	Week of Week-based Year	*30*
`W`	Week of Month	*2*
`E`	Day of Week	*Tue; Tuesday; T*
`e / c`	Localized Day of Week	*2; 02; Tue; Tuesday; T*
`F`	Week of Month	*2*
`a`	AM/PM of Day	*AM*
`h`	Hour (1-12)	*10*
`K`	Hour (0-11)	*1*
`k`	Hour (1-24)	*20*
`H`	Hour (0-23)	*23*
`m`	Minute	*10*
`s`	Second	*11*
`S`	Fraction of Second	*999*
`A`	Milli of Day	*2345*
`n`	Nano of Second	*865437987*
`N`	Nano of Day	*12986497300*

Symbol	Meaning	Examples
`V`	Time Zone ID	*Asia/Manila; Z; -06:00*
`z`	Time Zone Name	*Pacific Standard Time; PST*
`O`	Localized Zone Offset	*GMT+4; GMT+04:00; UTC-04:00;*
`X`	Zone Offset ('Z' for zero)	*Z; -08; -0830; -08:30*
`x`	Zone Offset	*+0000; -08; -0830; -08:30*
`Z`	Zone Offset	*+0000; -0800; -08:00*
`'`	Escape for Text	
`''`	Single Quote	
`[ ]`	Optional Section Start / End	
`# { }`	Reserved for future use	

Assuming:

```
LocalDate ldt = LocalDate.of(2025, 1, 20);
```

These are examples of using a predefined formatter:

```
System.out.println(DateTimeFormatter.ISO_DATE.format(ldt));
System.out.println(ldt.format(DateTimeFormatter.ISO_DATE));
```

The output will be:

```
2025-01-20
2025-01-20
```

These are examples of using a localized style:

```
DateTimeFormatter formatter = DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);
// With the current locale
System.out.println(formatter.format(ldt));
System.out.println(ldt.format(formatter));
// With another locale
System.out.println(formatter.withLocale(Locale.GERMAN).format(ldt));
```

One output can be:

```
1/20/25
1/20/25
20.01.25
```

And these are examples of using a custom pattern:

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("QQQQ Y");
// With the current locale
System.out.println(formatter.format(ldt));
System.out.println(ldt.format(formatter));
// With another locale
System.out.println(formatter.withLocale(Locale.GERMAN).format(ldt));
```

One output can be:

```
1st quarter 2025
1st quarter 2025
1. Quartal 2025
```

If the formatter uses information that is not available, a `DateTimeException` will be thrown. For example, using a `DateTimeFormatter.ISO_OFFSET_DATE` with a `LocalDate` instance (it doesn't have offset information).

To parse a date and/or time value from a string, use one of the `parse` methods. For example:

```java
// Format according to ISO-8601
String dateTimeStr1 = "2025-06-29T14:45:30";
// Custom format
String dateTimeStr2 = "2025/06/29 14:45:30";
LocalDateTime ldt = LocalDateTime.parse(dateTimeStr1);
// Using DateTimeFormatter
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy/MM/dd HH:mm:ss");
// DateTimeFormatter returns a TemporalAccessor instance
TemporalAccessor ta = formatter.parse(dateTimeStr2);
// LocalDateTime returns an instance of the same type
ldt = LocalDateTime.parse(dateTimeStr2, formatter);
```

The version of `parse()` of the date/time objects takes a string in a format according to ISO-8601, which is:

Class	Format	Example
`LocalDate`	`uuuu-MM-dd`	*2024-12-03*
`LocalTime`	`HH:mm:ss`	*10:15*
`LocalDateTime`	`uuuu-MM-dd'T'HH:mm:ss`	*2024-12-03T10:15:30*
`ZonedDateTime`	`uuuu-MM-dd'T'HH:mm:ssXXXXX[VV]`	*2023-12-03T10:15:30+01:00[Europe/Paris]*
`OffsetDateTime`	`uuuu-MM-dd'T'HH:mm:ssXXXXX`	*2023-12-03T10:15:30+01:00*
`OffsetTime`	`HH:mm:ssXXXXX`	*10:15:30+01:00*

If the formatter uses information that is not available or if the pattern of the format is invalid, a `DateTimeParseException` will be thrown.

In the Localization chapter, we'll revisit the `DateTimeFormatter` class, focusing on its application in localization.

Key Points

- `LocalDate`, `LocalTime`, `LocalDateTime`, `Instant`, `Period`, and `Duration` are the core classes of the new Java Date/Time API, located in the package `java.time`. They are immutable, thread-safe, and with the exception of `Instant`, they don't store or represent a time-zone.
- `LocalDate`, `LocalTime`, `LocalDateTime`, and `Instant` implement interface `java.time.temporal.Temporal`, so they all have similar methods. While `Period` and `Duration` implement interface `java.time.temporal.TemporalAmount`, which also makes them very similar.
- `LocalDate` represents a date with the year, month, and day of the month information. You can create an instance using:

  ```java
  LocalDate.of(2025, 8, 1);
  ```

- Valid `ChronoField` values to use with the `get()` method are: `DAY_OF_WEEK`, `ALIGNED_DAY_OF_WEEK_IN_MONTH`, `ALIGNED_DAY_OF_WEEK_IN_YEAR`, `DAY_OF_MONTH`, `DAY_OF_YEAR`, `EPOCH_DAY`, `ALIGNED_WEEK_OF_MONTH`, `ALIGNED_WEEK_OF_YEAR`, `MONTH_OF_YEAR`, `PROLEPTIC_MONTH`, `YEAR_OF_ERA`, `YEAR`, and `ERA`.
- Valid `ChronoUnits` values to use with the `plus()` and `minus()` methods are: `DAYS`, `WEEKS`, `MONTHS`, `YEARS`, `DECADES`, `CENTURIES`, `MILLENNIA`, and `ERAS`.
- `LocalTime` represents a time with hour, minutes, seconds, and nanoseconds information. You can create an instance using:

  ```java
  LocalTime.of(14, 20, 50, 99999);
  ```

- Valid `ChronoField` values to use with the `get()` method are: `NANO_OF_SECOND`, `NANO_OF_DAY`, `MICRO_OF_SECOND`, `MICRO_OF_DAY`, `MILLI_OF_SECOND`, `MILLI_OF_DAY`, `SECOND_OF_MINUTE`, `SECOND_OF_DAY`, `MINUTE_OF_HOUR`, `MINUTE_OF_DAY`, `HOUR_OF_AMPM`, `CLOCK_HOUR_OF_AMPM`, `HOUR_OF_DAY`, `CLOCK_HOUR_OF_DAY`, and `AMPM_OF_DAY`.
- Valid `ChronoUnits` values to use with the `plus()` and `minus()` methods are: `NANOS`, `MICROS`, `MILLIS`, `SECONDS`, `MINUTES`, `HOURS`, and `HALF_DAYS`.
- `LocalDateTime` is a combination of `LocalDate` and `LocalTime`. You can create an instance using:

  ```
  LocalDateTime.of(2025, 8, 1, 14, 20, 50, 99999);
  ```

- Valid `ChronoField` and `ChronoUnits` values are a combination of the ones used for `LocalDate` and `LocalTime`.
- `Instant` represents a single point in time in seconds and nanoseconds. You can create an instance using:

  ```
  Instant.ofEpochSecond(134556767, 999999999);
  ```

- Valid `ChronoField` values to use with the `get()` method are: `NANO_OF_SECOND`, `MICRO_OF_SECOND`, `MILLI_OF_SECOND`, and `INSTANT_SECONDS`.
- Valid `ChronoUnit` values to use with the `plus()` and `minus()` methods are: `NANOS`, `MICROS`, `MILLIS`, `SECONDS`, `MINUTES`, `HOURS`, `HALF_DAYS`, and `DAYS`.
- `Period` represents an amount of time in terms of years, months and days. You can create an instance using:

  ```
  Period.of(3, 12, 30);
  ```

- Valid `ChronoUnits` values to use with the `get()` method are: `DAYS`, `MONTHS`, `YEARS`.
- `Duration` represents an amount of time in terms of seconds and nanoseconds. You can create an instance using:

  ```
  Duration.ofSeconds(50, 999999);
  ```

- Valid `ChronoUnits` values to use with the `of()` method are: `NANOS`, `MICROS`, `MILLIS`, `SECONDS`, `MINUTES`, `HOURS`, `HALF_DAYS`, and `DAYS`. With the `get()` method, only `NANOS` and `SECONDS` are valid.
- `ZoneId`, `ZoneOffset`, `ZonedDateTime`, `OffsetDateTime`, and `OffsetTime` are the classes of the Java Date/Time API that store information about time zones and time offsets. They are located in the `java.time` package and are immutable.
- Each time zone has an ID, represented by the class `ZoneId`. There are three types of ID.
- The first type just states the offset from UTC/GMT time. They are represented by the class `ZoneOffset` and they consist of digits starting with `+` or `-`, for example `+02:00`.
- The second type also states the offset from UTC/GMT time, but with one of the following prefixes: UTC, GMT and UT, for example, `UTC+11:00`. They are also represented by the class `ZoneOffset`.
- The third type is region based. These IDs have the format *area/city*, for example, *Europe/London*.
- If you want to create a specific `ZoneId` object, use the method `of`:

  ```
  ZoneId.of("Asia/Singapore");
  ZoneId.of("+3");
  ZoneId.of("Z");
  ```

- The first method above methods produces an object of type `ZoneId`. The other two produce an object of type `ZoneOffset`.
- A `java.time.ZonedDateTime` object represents a point in time relative to a time zone.

- A `ZonedDateTime` object has three parts: a date, a time, and a time zone. It can be created with either:

```
ZoneId australiaZone = ZoneId.of("Australia/Victoria");
ZonedDateTime zonedDateTime5 =
    ZonedDateTime.of(2025, 1, 30, 13, 59, 59, 999, australiaZone);
```

- Or by using `LocalDate`, `LocalTime`, `LocalDateTime`, or `Instant` plus `ZoneId`.
- If we create an instance of `ZonedDateTime` for a region where a Daylight Saving Time (DST) is observed, the instance will support it, advancing the clock one hour when DST starts, and setting it back when DST ends.
- `Period` and `Duration` differ in their treatment of DST.
- `Period` adds a conceptual day to a date, while `Duration` adds an exact day to a date, without taking into account DST.
- `OffsetDateTime` represents an object with date/time information and an offset from UTC, for example, `2025-01-01T11:30-06:00`.
- `OffsetTime` represents a time with an offset from UTC, for example, `11:30-06:00`.
- `java.time.format.DateTimeFormatter` is a class for parsing and formatting dates. It can be used in two ways:
  - The date/time classes `LocalDate`, `LocalTime`, `LocalDateTime`, `ZonedDateTime`, `OffsetDateTime` all have the methods:

```
String format(DateTimeFormatter formatter)
static T parse(CharSequence text)
static T parse(CharSequence text, DateTimeFormatter formatter)
```

  - `DateTimeFormatter` has the following two methods:

```
String format(TemporalAccessor temporal)
TemporalAccessor parse(CharSequence text)
```

- All `format` methods throw the runtime exception `java.time.DateTimeException`, while all `parse` methods throw the runtime exception `java.time.format.DateTimeParseException`.

Practice Questions

1. Which of the following are valid ways to create a `LocalDate` object?

A. `LocalDate.of(2014);`
B. `LocalDate.with(2014, 1, 30);`
C. `LocalDate.of(2014, 0, 30);`
D. `LocalDate.now().plusDays(5);`

2. Which of the following options is the result of executing this line?

```
LocalDate.of(2014, 1, 2).atTime(14, 30, 59, 999999)
```

A. A `LocalDate` instance representing `2014-01-02`
B. A `LocalTime` instance representing `14:30:59:999999`
C. A `LocalDateTime` instance representing `2014-01-02 14:30:59:999999`
D. An exception is thrown

3. Which of the following are valid `ChronoUnit` values for `LocalTime`? (Choose all that apply)

A. `YEAR`
B. `NANOS`
C. `DAY`
D. `HALF_DAYS`

4. Which of the following statements are true? (Choose all that apply)

A. `java.time.Period` implements `java.time.temporal.Temporal`
B. `java.time.Instant` implements `java.time.temporal.Temporal`
C. `LocalDate` and `LocalTime` are thread-safe
D. `LocalDateTime.now()` will return the current time in UTC zone

5. Which of the following options is a valid way to get the nanoseconds part of an `Instant` object referenced by `i`?

A. `int nanos = i.getNano();`
B. `long nanos = i.get(ChronoField.NANOS);`
C. `long nanos = i.get(ChronoUnit.NANOS);`
D. `int nanos = i.getEpochNano();`

6. Which of the following options is the result of executing this line?

```
System.out.println(
   Period.between(
       LocalDate.of(2025, 3, 20),
       LocalDate.of(2025, 2, 20))
);
```

A. `P29D`
B. `P-29D`
C. `P1M`
D. `P-1M`

7. Which of the following options is the result of executing this line?

```
System.out.println(
   Duration.between(
       LocalDateTime.of(2025, 3, 20, 18, 0),
       LocalTime.of(18, 5) )
);
```

A. `PT5M`
B. `PT-5M`
C. `PT300S`
D. An exception is thrown

8. Which of the following are valid `ChronoField` values for `LocalDate`?

A. `DAY_OF_WEEK`
B. `HOUR_OF_DAY`
C. `DAY_OF_MONTH`
D. `MILLI_OF_SECOND`

9. Which of the following are valid ways to create a `ZoneId` object?

A. `ZoneId.ofHours(2);`
B. `ZoneId.of("2");`
C. `ZoneId.of("-1");`
D. `ZoneId.of("America/Canada");`

10. Which of the following options is the result of executing these lines?

```
ZoneOffset offset = ZoneOffset.of("Z");
System.out.println(
   offset.get(ChronoField.HOUR_OF_DAY)
);
```

A. `0`
B. `1`
C. `12:00`
D. An exception is thrown

11. Assuming a local time zone of `+2:00`, which of the following options is the result of executing these lines?

```
ZonedDateTime zdt =
   ZonedDateTime.of(2025, 02, 28, 5, 0, 0, 0,
       ZoneId.of("+05:00"));
System.out.println(zdt.toLocalTime());
```

A. `05:00`
B. `17:00`
C. `02:00`
D. `03:00`

12. Assuming that DST starts on October, 4, 2025 at 0:00:00, which of the following is the result of executing the above lines?

```
ZonedDateTime zdt =
    ZonedDateTime.of(2025, 10, 4, 0, 0, 0, 0,
        ZoneId.of("America/Asuncion"))
    .plus(Duration.ofHours(1));
System.out.println(zdt);
```

A. `2025-10-04T00:00-03:00[America/Asuncion]`
B. `2025-10-04T01:00-03:00[America/Asuncion]`
C. `2025-10-04T02:00-03:00[America/Asuncion]`
D. `2025-10-03T23:00-03:00[America/Asuncion]`

13. Which of the following statements are true? (Choose all that apply)

A. `java.time.ZoneOffset` is a subclass of `java.time.ZoneId`.
B. `java.time.Instant` can be obtained from `java.time.ZonedDateTime`.
C. `java.time.ZoneOffset` can manage DST.
D. `java.time.OffsetDateTime` represents a point in time in the UTC time zone.

14. Which of the following options is the result of executing these lines?

```
DateTimeFormatter formatter =
   DateTimeFormatter.ofLocalizedTime(FormatStyle.SHORT);
System.out.println(
   formatter
   .withLocale(Locale.ENGLISH)
   .format(LocalDateTime.of(2025, 5, 7, 16, 0))
);
```

A. `5/7/15 4:00 PM`
B. `5/7/15`
C. `4:00 PM`
D. `4:00:00 PM`

15. Which of the following statements is true about these lines?

```
DateTimeFormatter formatter =
   DateTimeFormatter.ofPattern("HH:mm:ss X");
OffsetDateTime odt =
   OffsetDateTime.parse("11:50:20 Z", formatter);
```

A. The pattern `HH:mm:ss X` is invalid.
B. An `OffsetDateTime` is created successfully.
C. `Z` is an invalid offset.
D. An exception is thrown at runtime.

Chapter ELEVEN

The Date API

Answers

1. The correct answer is D.

Explanation:

- **A)** `LocalDate.of(2014);`
  - This option is incorrect. The `LocalDate.of()` method requires a year, month, and day to be specified. Providing only a year will result in a compilation error.
- **B)** `LocalDate.with(2014, 1, 30);`
  - This option is incorrect. The `LocalDate` class does not have a `with()` method that takes three int arguments for year, month, and day. The correct method to use is `LocalDate.of(int year, int month, int dayOfMonth)`.
- **C)** `LocalDate.of(2014, 0, 30);`
  - This option is incorrect. The month value is 0, but months in the `LocalDate` class are indexed starting from 1. Valid month values are from 1 to 12, so using 0 will throw a `DateTimeException`.
- **D)** `LocalDate.now().plusDays(5);`
  - This option is correct. It accurately obtains the current date using `LocalDate.now()` and then adds 5 days to it using the `plusDays()` method. This will create a new `LocalDate` object representing the date 5 days from now.

2. The correct answer is C.

Explanation:

- **A)** A `LocalDate` instance representing `2014-01-02`
  - This option is incorrect. The `atTime` method does not return a `LocalDate`, but rather combines the `LocalDate` with the provided time parameters to create a `LocalDateTime` object.
- **B)** A `LocalTime` instance representing `14:30:59:999999`
  - This option is incorrect. The `atTime` method does not return a `LocalTime`, but rather combines the `LocalDate` with the provided time parameters to create a `LocalDateTime` object. Additionally, `LocalTime` does not have nanosecond precision, so `999999` nanoseconds would be an invalid `LocalTime`.
- **C)** A `LocalDateTime` instance representing `2014-01-02 14:30:59:999999`
  - This option is correct. The `atTime` method takes a `LocalDate` and combines it with the provided hour, minute, second, and nanosecond parameters to create a `LocalDateTime` object representing that date and time. The resulting `LocalDateTime` will be `2014-01-02 14:30:59:999999`.
- **D)** An exception is thrown
  - This option is incorrect. The provided parameters of 14 for hour, 30 for minute, 59 for second, and 999999 for nanosecond are all valid values for their respective fields, so combining them with the `LocalDate` will not throw an exception.

3. The correct answers are B and D.

Explanation:

- **A)** `YEAR`
  - This option is incorrect. `YEAR` is not a valid `ChronoUnit` for `LocalTime`. `LocalTime` represents a time of day without any date information, so units of `YEAR` do not apply.

- **B)** `NANOS`
  - This option is correct. `NANOS` is a valid `ChronoUnit` for `LocalTime`. `LocalTime` has nanosecond precision, so you can perform operations on `LocalTime` using the `NANOS` unit.
- **C)** `DAY`
  - This option is incorrect. `DAY` is not a valid `ChronoUnit` for `LocalTime`. Similar to `YEAR`, `LocalTime` has no concept of days since it only represents a time, not a date.
- **D)** `HALF_DAYS`
  - This option is correct. `HALF_DAYS` is a valid `ChronoUnit` for `LocalTime`. A day can be divided into two 12-hour periods (AM and PM), so `HALF_DAYS` can be used with `LocalTime` to represent a difference or addition of 12 hour chunks of time.

4. The correct answers are B and C.

Explanation:

- **A)** `java.time.Period` implements `java.time.temporal.Temporal`
  - This option is incorrect. `java.time.Period` does not implement the `java.time.temporal.Temporal` interface. `Period` represents a span of time between two dates and is not itself a temporal object.
- **B)** `java.time.Instant` implements `java.time.temporal.Temporal`
  - This option is correct. `java.time.Instant` does implement the `java.time.temporal.Temporal` interface. `Instant` represents a point in time on the timeline and can be thought of as a temporal object.
- **C)** `LocalDate` and `LocalTime` are thread-safe.
  - This option is correct. `LocalDate` and `LocalTime` are indeed thread-safe. All the core Java Time classes, including `LocalDate`, `LocalTime`, `LocalDateTime`, `Instant`, etc., are designed to be immutable and thread-safe.
- **D)** `LocalDateTime.now()` will return the current time in UTC zone
  - This option is incorrect. `LocalDateTime.now()` returns the current date and time using the system clock in the default time zone, not necessarily in the UTC zone. To get the current time in UTC, you would use `LocalDateTime.now(ZoneOffset.UTC)` or `Instant.now()`.

5. The correct answer is A.

Explanation:

- **A)** `int nanos = i.getNano();`
  - This option is correct. The `Instant` class does have a `getNano()` method that returns the nanosecond part of the `Instant` as an `int`. This is a valid way to get the nanoseconds.
- **B)** `long nanos = i.get(ChronoField.NANOS);`
  - This option is incorrect. You can use the `get(TemporalField)` method of `Instant` to get the value of a specific `ChronoField`. Passing `ChronoField.NANO_OF_SECOND` (not `ChronoField.NANO`) will return the nanosecond part of the `Instant` as a `long`.
- **C)** `long nanos = i.get(ChronoUnit.NANOS);`
  - This option is incorrect. While `Instant` does have a `get(TemporalUnit)` method, `ChronoUnit.NANOS` is not a valid argument for it. `ChronoUnit` values are used for durations and periods, not for fields of a temporal object.
- **D)** `int nanos = i.getEpochNano();`
  - This option is incorrect. The `Instant` class does have a `getEpochSecond()` method that returns the number of seconds since the Unix epoch, but there is no corresponding `getEpochNano()` method.

6. The correct answer is D.

Explanation:

- **A)** `P29D`
  - This option is incorrect. The `Period.between` method calculates the period between the second date and the first date, in that order. Since the first date (2025-03-20) is later than the second date (2025-02-20), the resulting period will be negative, not positive.

- **B)** `P-29D`
  - This option is incorrect. While the resulting period will be negative, it will not be represented as `-29D`. A `Period` first counts the number of complete months, then the remaining days.
- **C)** `P1M`
  - This option is incorrect. The resulting period will be negative because the first date is later than the second date.
- **D)** `P-1M`
  - This option is correct. The `Period.between` method subtracts the second date from the first date. In this case, `2025-03-20` minus `2025-02-20` results in a period of -1 month, which is represented as `P-1M`. The `Period` class first calculates the difference in complete months, and then any remaining days. Since the difference is exactly one month, the result is `P-1M`.

7. The correct answer is D.

Explanation:

- **A)** `PT5M`
  - This option is incorrect. `PT5M` represents a duration of 5 minutes, which would be the result if the second time point was 5 minutes after the first. However, since `LocalTime.of(18, 5)` is being compared to a `LocalDateTime`, this causes an issue because they are not of the same type.
- **B)** `PT-5M`
  - This option is incorrect. `PT-5M` represents a duration of negative 5 minutes. Similar to option A, this would only be the case if the second time point was before the first. The main issue is that there is a type mismatch between `LocalDateTime` and `LocalTime`.
- **C)** `PT300S`
  - This option is incorrect. `PT300S` represents a duration of 300 seconds (or 5 minutes), which again would be the result if the second time point was 5 minutes after the first. However, this still doesn't resolve the type mismatch issue between `LocalDateTime` and `LocalTime`.
- **D)** An exception is thrown
  - This option is correct. An exception is thrown because there is a type mismatch between `LocalDateTime.of(2025, 3, 20, 18, 0)` and `LocalTime.of(18, 5)`. The `Duration.between` method requires two temporal objects of the same type.

8. The correct answers are A and C.

Explanation:

- **A)** `DAY_OF_WEEK`
  - This option is correct. `DAY_OF_WEEK` is a valid `ChronoField` value for `LocalDate`. It represents the day of the week, an integer from 1 (Monday) to 7 (Sunday), which can be extracted from a `LocalDate`.
- **B)** `HOUR_OF_DAY`
  - This option is incorrect. `HOUR_OF_DAY` is not a valid `ChronoField` value for `LocalDate`. `HOUR_OF_DAY` pertains to `LocalTime` or `LocalDateTime`, which include time components, whereas `LocalDate` only deals with date components.
- **C)** `DAY_OF_MONTH`
  - This option is correct. `DAY_OF_MONTH` is a valid `ChronoField` value for `LocalDate`. It represents the day of the month, which can be extracted from a `LocalDate`.
- **D)** `MILLI_OF_SECOND`
  - This option is incorrect. `MILLI_OF_SECOND` is not a valid `ChronoField` value for `LocalDate`. `MILLI_OF_SECOND` pertains to time components, specifically for `LocalTime` or `LocalDateTime`, and `LocalDate` only deals with date components.

9. The correct answer is C.

Explanation:

- **A)** `ZoneId.ofHours(2);`

  - This option is incorrect. The method `ofHours(int)` belongs to the `ZoneOffset` class, not `ZoneId`.
- **B)** `ZoneId.of("2");`
  - This option is incorrect. The format of the offset is incorrect for `ZoneId`. It should be a proper time-zone ID or start with a sign (`+` or `-`).
- **C)** `ZoneId.of("-1");`
  - This option is correct. `ZoneId.of("-1")` is valid since it follows the correct format for time-zone offsets.
- **D)** `ZoneId.of("America/Canada");`
  - This option is incorrect. The format for zone regions should be in the `"Area/City"` format, not `"Area/Country"`. A valid example would be `"America/Montreal"`.

10. The correct answer is D.

Explanation:

- **A)** `0`
  - This option is incorrect. The method `offset.get(ChronoField.HOUR_OF_DAY)` does not return the hour value of the `ZoneOffset`. `ZoneOffset` represents a time-zone offset from UTC/Greenwich, and calling `get(ChronoField.HOUR_OF_DAY)` on it is not appropriate.
- **B)** `1`
  - This option is incorrect. Similar to option A, the `get` method of `ZoneOffset` with `ChronoField.HOUR_OF_DAY` does not produce this result. The `ZoneOffset` class is not meant to provide such a field directly.
- **C)** `12:00`
  - This option is incorrect. `12:00` is not a valid response for the method call as it implies a time representation, while `ZoneOffset` is dealing with offset values rather than specific time of day values.
- **D)** An exception is thrown
  - This option is correct. An exception is thrown because `ZoneOffset` does not support the field `ChronoField.HOUR_OF_DAY`. The `ZoneOffset` class provides offset values in terms of seconds rather than specific chrono fields like hour of day.

11. The correct answer is A.

Explanation:

- **A)** `05:00`
  - This option is correct. `ZonedDateTime.of(2025, 02, 28, 5, 0, 0, 0, ZoneId.of("+05:00"))` creates a `ZonedDateTime` instance with the specified date, time, and time zone offset of +05:00. Calling `toLocalTime()` on this instance returns the local time, which is `05:00`, as no conversion to the local time zone of +2:00 is done in this code snippet.
- **B)** `17:00`
  - This option is incorrect. `17:00` would be the time if the code converted the given time (05:00) from the +05:00 time zone to the local time zone of +02:00, which it does not.
- **C)** `02:00`
  - This option is incorrect. `02:00` does not correspond to any logical result based on the given time and time zone offset.
- **D)** `03:00`
  - This option is incorrect. `03:00` also does not correspond to any logical result based on the given time and time zone offset.

12. The correct answer is C.

Explanation:

- **A)** `2025-10-04T00:00-03:00[America/Asuncion]`
  - This option is incorrect. The initial time is `2025-10-04T00:00-03:00[America/Asuncion]` before DST starts. When 1 hour is added, the time will shift forward by 1 hour, but since DST starts

at this moment, the offset will change.

- **B)** `2025-10-04T01:00-03:00[America/Asuncion]`
  - This option is incorrect. Adding 1 hour to the initial time `2025-10-04T00:00-03:00[America/Asuncion]` while considering the start of DST (which typically adds 1 hour to the local time) means that the effective time would be adjusted by the DST transition.
- **C)** `2025-10-04T02:00-03:00[America/Asuncion]`
  - This option is correct. Initially, the time is `2025-10-04T00:00-03:00[America/Asuncion]`. With the addition of 1 hour and considering the DST start at `2025-10-04T00:00`, the time advances to `2025-10-04T02:00-03:00[America/Asuncion]`, as it effectively skips the 01:00 hour.
- **D)** `2025-10-03T23:00-03:00[America/Asuncion]`
  - This option is incorrect. The date and time `2025-10-03T23:00-03:00[America/Asuncion]` does not correlate correctly with the 1 hour addition from the initial time and does not account for the DST transition.

13. The correct answer is B.

Explanation:

- **A)** `java.time.ZoneOffset` is a subclass of `java.time.ZoneId`.
  - This option is incorrect. `java.time.ZoneOffset` is not a subclass of `java.time.ZoneId`. `java.time.ZoneOffset` is a final class that extends `java.time.ZoneId` but it is not a subclass.
- **B)** `java.time.Instant` can be obtained from `java.time.ZonedDateTime`.
  - This option is correct. `java.time.Instant` can indeed be obtained from `java.time.ZonedDateTime` using the `toInstant()` method.
- **C)** `java.time.ZoneOffset` can manage DST.
  - This option is incorrect. `java.time.ZoneOffset` represents a fixed offset from UTC and does not manage Daylight Saving Time (DST). DST is managed by `java.time.ZoneId`.
- **D)** `java.time.OffsetDateTime` represents a point in time in the UTC time zone.
  - This option is incorrect. `java.time.OffsetDateTime` represents a date-time with an offset from UTC, but it does not necessarily represent a point in the UTC time zone. The offset can be any valid `ZoneOffset`.

14. The correct answer is C.

Explanation:

- **A)** `5/7/15 4:00 PM`
  - This option is incorrect. The `DateTimeFormatter.ofLocalizedTime(FormatStyle.SHORT)` method is used to format only the time portion of a `LocalDateTime` object, and it does not include the date. Therefore, the output will not include `5/7/15`.
- **B)** `5/7/15`
  - This option is incorrect. As mentioned earlier, the `DateTimeFormatter.ofLocalizedTime(FormatStyle.SHORT)` formats only the time portion and does not include the date. Thus, the output `5/7/15` is not possible.
- **C)** `4:00 PM`
  - This option is correct. The `DateTimeFormatter.ofLocalizedTime(FormatStyle.SHORT)` formats the time portion of the `LocalDateTime` object in a short style. Given the input time `16:00`, in the `Locale.ENGLISH`, the formatted output is `4:00 PM`.
- **D)** `4:00:00 PM`
  - This option is incorrect. The `DateTimeFormatter.ofLocalizedTime(FormatStyle.SHORT)` formats the time portion without including seconds. Therefore, the output will not include `4:00:00 PM`.

15. The correct answer is D.

Explanation:

- **A)** The pattern `HH:mm:ss X` is invalid.

  - This option is incorrect. The pattern `HH:mm:ss X` is valid. `HH` represents the hour of the day (00-23), `mm` represents the minute of the hour, `ss` represents the second of the minute, and `X` represents the ISO 8601 time zone offset.
- **B)** An `OffsetDateTime` is created successfully.
  - This option is incorrect. The pattern `HH:mm:ss X` is valid, but the `OffsetDateTime.parse` method requires a date and time format along with the offset. Since the input string `"11:50:20 Z"` does not contain a date part, this will cause a `DateTimeParseException`.
- **C)** `Z` is an invalid offset.
  - This option is incorrect. `Z` is a valid offset representing UTC (Coordinated Universal Time).
- **D)** An exception is thrown at runtime.
  - This option is correct. An exception is thrown at runtime because the input string `"11:50:20 Z"` does not match the expected pattern for an `OffsetDateTime`, which typically includes a date part as well as the time and offset.

Chapter TWELVE

File I/O

Chapter Content

- Basic Concepts
- Using NIO.2 Paths
- The `Files` Class
- I/O Streams
  - Stream Classes
  - `FileInputStream`
  - `FileOutputStream`
  - `FileReader`
  - `FileWriter`
  - `BufferedReader`
  - `BufferedWriter`
  - `ObjectInputStream` and `ObjectOutputStream`
  - `PrintStream`
  - `PrintWriter`
  - Standard Streams
- Copying, Moving, Deleting, and Comparing Files
- Reading and Writing Files
  - Reading Files
  - Writing Files
- Working with File Attributes
- Traversing a Directory Tree

- Serializing Data

- Reference Tables
- Key Points
- Practice Questions

Basic Concepts

Let's start by defining some concepts.

As we know, data is organized into files, directories, and file systems in a computer.

A file is a group of related data stored on a disk or other storage device. Files can contain programs, documents, images or any other types of data.

A directory, also called a folder, is a collection of files and other directories that are stored under the same name. Directories allow you to organize files in a hierarchical structure. For example:

```
documents/
  work/
    report.pdf
    presentation.ppt
  personal/
    resume.doc
    family.jpg
```

The directory at the very top of the structure is known as the root directory. In Unix systems it is represented by a forward slash (`/`), while on Windows it is identified by a drive letter followed by a colon, like `C:`.

To locate a specific file or directory, you need to specify its path, the route from the root directory to that particular item in the hierarchy. Path separators differ between operating systems. Unix uses a forward slash (`/`) while Windows uses a backslash (\).

A path can be absolute, specifying the complete route from the root:

```
/home/steve/documents/work/report.pdf
C:\Users\Steve\Documents\Work\report.pdf
```

Or it can be relative, specifying the route from the current directory, also known as the working directory:

```
documents/work/report.pdf
..\personal\resume.doc
```

Two special symbols are commonly used in relative paths: - A single dot (`.`) represents the current directory - Two dots (`..`) represent the parent of the current directory (one level up in the hierarchy)

So for example, if the current directory is `/home/steve/documents`, then: - `./work/report.pdf` is equivalent to `work/report.pdf`
- `../downloads/file.zip` refers to `/home/steve/downloads/file.zip`

In Java, you can work with the file system in two main ways:

1. Using the `java.io.File` class (legacy I/O API)
2. Using the `java.nio.file.Path` interface (NIO.2 API)

To create a `File` instance, simply pass a file or directory path to its constructor:

```
File file = new File("/home/steve/documents/work/report.pdf");
File dir = new File("C:\\Users\\Steve\\Documents");
```

Note that this does not actually create the file or directory on the disk, it just creates an object that represents that path. You can then call various methods on the `File` object to get information about the file or directory or to manipulate it.

In the newer NIO.2 (New Input/Output) API, paths are represented by the `Path` interface rather than the `File` class. You can get a `Path` instance in several ways:

```
// 1. Using the Paths helper class
Path p1 = Paths.get("/home/steve/documents/work/report.pdf");

// 2. From a File object
File file = new File("C:\\Users\\Steve\\Documents");
Path p2 = file.toPath();
```

```
// 3. By joining path strings
Path p3 = Paths.get("documents", "work", "report.pdf");
Path p4 = Paths.get("/home", "steve").resolve("documents");

// 4. From the default FileSystem
Path p5 = FileSystems.getDefault().getPath("documents/work/report.pdf");
```

You can easily convert between `File` and `Path` using the `toFile()` and `toPath()` methods:

```
File file = path.toFile();
Path path = file.toPath();
```

The `Path` interface provides similar methods to `File` but offers more flexibility and additional features for working with paths.

For example, you can extract specific parts of a path:

```
Path path = Paths.get("/home/steve/documents/work/report.pdf");

Path parent = path.getParent(); // /home/steve/documents/work
Path root = path.getRoot(); // /
Path name = path.getFileName(); // report.pdf
```

Or construct paths by joining elements:

```
Path documents = Paths.get("/home/steve/documents");
Path file = documents.resolve("work/report.pdf");
```

The resulting path doesn't have to exist, it is just an abstract representation that can be used for further processing.

In the next sections, we'll focus on the `Path` interface and the NIO.2 API.

Using NIO.2 Paths

Let's explore in more detail some of the key methods and concepts related to `Path`.

This interface provides the following methods for retrieving basic path information: - `String toString()`: Returns the string representation of the path. - `int getNameCount()`: Returns the number of name elements in the path. - `Path getName(int index)`: Returns the name element at the specified index.

Here's an example:

```
Path path = Paths.get("/home/user/documents/file.txt");
System.out.println(path.toString()); // Output: /home/user/documents/file.txt
System.out.println(path.getNameCount()); // Output: 4
System.out.println(path.getName(0)); // Output: home
System.out.println(path.getName(2)); // Output: documents
```

Additionally, there are methods for accessing the path elements: - `Path getFileName()`: Returns the filename (the last element) of the path. - `Path getRoot()`: Returns the root component of the path, or `null` if the path is relative. - `Path getParent()`: Returns the parent path, or `null` if there is no parent.

Here's an example:

```
Path path = Paths.get("/home/user/documents/file.txt");
System.out.println(path.getFileName()); // Output: file.txt
System.out.println(path.getRoot()); // Output: /
System.out.println(path.getParent()); // Output: /home/user/documents
```

The `relativize` method constructs a relative path between the current path and a given path. For example:

```
Path base = Paths.get("/home/user");
Path path = Paths.get("/home/user/documents/file.txt");
Path relativePath = base.relativize(path);
System.out.println(relativePath); // Output: documents/file.txt
```

The `normalize` method returns a path that is a normalized version of the original path, eliminating any redundant elements such as `.` (current directory) and `..` (parent directory):

```
Path path = Paths.get("/home/user/./documents/../file.txt");
Path normalizedPath = path.normalize();
System.out.println(normalizedPath); // Output: /home/user/file.txt
```

The `toRealPath` method returns the real path of an existing file in the file system, resolving any symbolic links:

```
Path path = Paths.get("/path/to/symlink");
Path realPath = path.toRealPath();
System.out.println(realPath); // Output: /actual/path/to/file
```

The `resolve` method resolves a path against the current path, allowing you to mix absolute and relative paths:

```
Path base = Paths.get("/home/user");
Path relativePath = Paths.get("documents/file.txt");
Path resolvedPath = base.resolve(relativePath);
System.out.println(resolvedPath); // Output: /home/user/documents/file.txt
```

However, if the path to be resolved is already an absolute path, it will be returned as-is:

```
Path base = Paths.get("/home/user");
Path absolutePath = Paths.get("/other/path/file.txt");
Path resolvedPath = base.resolve(absolutePath);
System.out.println(resolvedPath); // Output: /other/path/file.txt
```

The `Files` Class

The `java.nio.file.Files` class is part of the NIO.2 API. It provides a rich set of static utility methods for working with files and directories in a more concise and efficient manner compared to the legacy `File` class.

These are some of its key features:

- **Improved Exception Handling:** Many of the `Files` methods throw more specific exceptions like `NoSuchFileException`, `DirectoryNotEmptyException`, etc., making it easier to handle different error scenarios. In contrast, the `File` class methods typically return boolean values or throw more generic exceptions.
- **Symbolic Links Support:** The `Files` class has built-in support for symbolic links. You can create, detect, and resolve symbolic links using methods like `createSymbolicLink()`, `isSymbolicLink()`, and `readSymbolicLink()`.
- **Atomic Operations:** The `Files` class provides methods for performing atomic file operations. For example, `move()` with the `StandardCopyOption.ATOMIC_MOVE` option ensures that a file move operation is performed atomically.
- **File Attributes:** The `Files` class makes it easy to read and modify file attributes, such as file permissions, owners, timestamps, etc. You can use methods like `readAttributes()`, `setOwner()`, `setLastModifiedTime()`, etc.
- **Directory Walking:** The `Files.walkFileTree()` method allows you to recursively traverse a directory tree and perform actions on each file and directory encountered. This is more efficient and flexible than manually traversing the tree using the `File` class.

- **Streams and Buffers:** The `Files` class provides methods to open files as streams (`newInputStream()`, `newOutputStream()`) or buffered readers/writers (`newBufferedReader()`, `newBufferedWriter()`), making I/O operations more convenient.
- **Path Operations:** Since the `Files` class works with `Path` objects, it can perform path-related operations like resolving, normalizing, and getting path components, etc.
- **File Content Operations:** The `Files` class has methods to read and write file content in a single line of code, such as `readAllBytes()`, `readAllLines()`, `write()`, etc. This eliminates the need for manual file I/O boilerplate.

Here are some of the key methods provided by the `Files` class:

- **File Operations:**
  - `static Path createFile(Path path)`: Creates a new file.
  - `static Path createDirectory(Path path)`: Creates a new directory.
  - `static Path createDirectories(Path path)`: Creates a directory and all nonexistent parent directories.
  - `static void delete(Path path)`: Deletes a file or directory.
  - `static boolean deleteIfExists(Path path)`: Deletes a file or directory if it exists.
  - `static Path copy(Path source, Path target, CopyOption... options)`: Copies a file or directory.
  - `static Path move(Path source, Path target, CopyOption... options)`: Moves or renames a file or directory.
- **Reading and Writing:**
  - `static byte[] readAllBytes(Path path)`: Reads all bytes from a file.
  - `static String readString(Path path)`: Reads a file as a string.
  - `static List<String> readAllLines(Path path)`: Reads all lines from a file.
  - `static Path write(Path path, byte[] bytes, OpenOption... options)`: Writes bytes to a file.
  - `static Path writeString(Path path, CharSequence csq, OpenOption... options)`: Writes a string to a file.
  - `static Path write(Path path, Iterable<? extends CharSequence> lines, OpenOption... options)`: Writes lines of text to a file.
- **File Attributes:**
  - `static boolean exists(Path path, LinkOption... options)`: Checks if a file or directory exists.
  - `static boolean notExists(Path path, LinkOption... options)`: Checks if a file or directory does not exist.
  - `static boolean isReadable(Path path)`: Checks if a file is readable.
  - `static boolean isWritable(Path path)`: Checks if a file is writable.
  - `static boolean isExecutable(Path path)`: Checks if a file is executable.
  - `static boolean isDirectory(Path path, LinkOption... options)`: Checks if a path is a directory.
  - `static boolean isRegularFile(Path path, LinkOption... options)`: Checks if a path is a regular file.
  - `static long size(Path path)`: Returns the size of a file.
- **Stream Support:**
  - `static Stream<Path> list(Path dir)`: Returns a stream of entries in a directory.
  - `static Stream<Path> walk(Path start, FileVisitOption... options)`: Returns a stream that is lazily populated with `Path` by walking the file tree rooted at a given starting file.
  - `static Path walkFileTree(Path start, FileVisitor<? super Path> visitor)`: Walks a file tree.
- **Symbolic Links:**
  - `static Path createSymbolicLink(Path link, Path target, FileAttribute<?>... attrs)`: Creates a symbolic link.
  - `static Path readSymbolicLink(Path link)`: Reads the target of a symbolic link.
- **File Permissions:**
  - `static Path setPosixFilePermissions(Path path, Set<PosixFilePermission> perms)`: Sets POSIX file permissions.
  - `static Set<PosixFilePermission> getPosixFilePermissions(Path path, LinkOption... options)`:

Reads POSIX file permissions.

Several methods of the `Files` class take optional arguments that control how the operation is performed. Here are some common ones:

- `java.nio.file.LinkOption`: Specifies how symbolic links are handled. A common value is `LinkOption.NOFOLLOW_LINKS`, which indicates that symbolic links should not be followed.
- `java.nio.file.StandardCopyOption`: Controls how a file copy operation should be done. Values include `REPLACE_EXISTING` (replace the target if it exists), `COPY_ATTRIBUTES` (copy file attributes as well), `ATOMIC_MOVE` (perform an atomic move operation).
- `java.nio.file.StandardOpenOption`: Specifies options for opening a file. Common values are `CREATE` (create a new file if it doesn't exist), `APPEND` (append to the end of the file), `TRUNCATE_EXISTING` (truncate the file if it exists).
- `java.nio.file.FileVisitOption`: Used with the `Files.walkFileTree()` method to control how the file tree traversal is done. The `FOLLOW_LINKS` value indicates that symbolic links should be followed during traversal.

In the next sections, we'll review some of the methods and optional arguments of this class in more detail. But first, let's talk about I/O streams.

I/O Streams

In Java, I/O (Input/Output) streams provide a way to read data from a source or write data to a destination.

Here's an analogy to explain I/O streams. Imagine you have a water tank and you want to transfer the water to another container. You can connect a pipe between the tank and the container, and the water will flow from the tank to the container through the pipe. Similarly, I/O streams act as the pipe, allowing data to flow from a source (a file, network, or memory) to a destination (another file, network, or memory).

I/O streams can be classified into several categories.

First of all, Java provides two types of I/O streams: byte streams and character streams.

- **Byte streams**, as the name suggests, read and write data in the form of bytes (8-bit data). They are suitable for handling raw binary data, such as images, audio files, or any other type of non-text data. Examples include `InputStream` and `OutputStream`.
- **Character streams** are designed to read and write data in the form of characters (16-bit Unicode data). They are useful for handling text-based data, such as reading from or writing to text files. Examples include `Reader` and `Writer`.

I/O streams can also be classified into input streams and output streams.

- **Input streams** are used to read data from a source. They provide methods like `read()` to read bytes or characters from the input source. Examples of input stream classes in Java include `FileInputStream`, `BufferedInputStream`, `FileReader`, and `BufferedReader`.
- **Output streams** are used to write data to a destination. They provide methods like `write()` to write bytes or characters to the output destination. Examples of output stream classes in Java include `FileOutputStream`, `BufferedOutputStream`, `FileWriter`, and `BufferedWriter`.

Finally, I/O streams can be categorized into low-level streams and high-level streams.

- **Low-level streams**, also known as node streams, are directly connected to the data source or destination. They are the building blocks of I/O operations and provide basic functionality for reading from or writing to a specific source or destination. Examples of low-level streams include `FileInputStream`, `FileOutputStream`, `FileReader`, and `FileWriter`.
- **High-level streams**, also known as processing streams or filter streams, are built on top of low-level streams. They provide additional functionality and features, such as buffering, filtering, or transforming

the data as it passes through the stream. Examples of high-level streams include `BufferedInputStream`, `BufferedOutputStream`, `BufferedReader`, `BufferedWriter`, `ObjectInputStream`, and `ObjectOutputStream`.

Stream Classes

The `java.io` library defines four abstract classes that serve as the parents of all I/O stream classes:

- `InputStream`: The base class for all byte input streams.
- `OutputStream`: The base class for all byte output streams.
- `Reader`: The base class for all character input streams.
- `Writer`: The base class for all character output streams.

These abstract classes provide the fundamental methods for reading from or writing to a stream, such as `read()`, `write()`, `close()`, and more. Concrete stream classes extend these base classes to provide specific functionality.

Java provides a wide range of concrete I/O stream classes in the `java.io` package. Some commonly used classes include:

- `FileInputStream` and `FileOutputStream`: Used for reading from and writing to files as byte streams.
- `FileReader` and `FileWriter`: Used for reading from and writing to files as character streams.
- `BufferedInputStream` and `BufferedOutputStream`: Provide buffering capabilities to improve performance of byte streams.
- `BufferedReader` and `BufferedWriter`: Provide buffering capabilities and additional methods for reading and writing character streams.
- `ObjectInputStream` and `ObjectOutputStream`: Used for reading and writing Java objects to streams.
- `PrintStream` and `PrintWriter`: Provide methods for writing formatted data to a stream.

These concrete classes extend the appropriate base classes (`InputStream`, `OutputStream`, `Reader`, or `Writer`) and implement specific functionality for handling different types of data sources and destinations.

`FileInputStream`

`FileInputStream` reads bytes from a file. It inherits from `InputStream`.

It can be created either with a `File` object or a `String path`:

```
FileInputStream(File file)
FileInputStream(String path)
```

Here's how you use it:

```
try (InputStream in = new FileInputStream("/file.txt")) {
    int b;
    // -1 indicates the end of the file
    while((b = in.read()) != -1) {
        // Do something with the byte read
    }
} catch(IOException e) {
    /** ... */
}
```

There's also a `read()` method that reads bytes into an array of bytes:

```
byte[] data = new byte[1024];
int numberOfBytesRead;
while((numberOfBytesRead = in.read(data)) != -1) {
    // Do something with the array data
}
```

All the classes we'll review should be closed. Fortunately, they implement `java.lang.AutoCloseable` so they can be used in a `try-with-resources`.

Also, almost all methods of these classes throw `IOExceptions` or one of its subclasses (such as `FileNotFoundException`, which is pretty descriptive).

FileOutputStream

`FileOutputStream` writes bytes to a file. It inherits from `OutputStream`.

It can be created either with a `File` object or a `String` path and an optional `boolean` that indicates whether you want to overwrite or append to the file if it exists (it's overwritten by default):

```
FileOutputStream(File file)
FileOutputStream(File file, boolean append)
FileOutputStream(String path)
FileOutputStream(String path, boolean append)
```

Here's how you use it:

```
try (OutputStream out = new FileOutputStream("/file.txt")) {
    int b;
    // Made up method to get some data
    while((b = getData()) != -1) {
        // Writes b to the file output stream
        out.write(b);
        out.flush();
    }
} catch(IOException e) {
    /** ... */
}
```

When you write to an `OutputStream`, the data may get cached internally in memory and written to disk at a later time. If you want to make sure that all data is written to disk without having to close the `OutputStream`, you can call the `flush()` method every once in a while.

`FileOutputStream` also contains overloaded versions of `write()` that allow you to write data contained in a byte array.

FileReader

`FileReader` reads characters from a text file. It inherits from `Reader`.

It can be created either with a `File` object or a `String` path:

```
FileReader(File file)
FileReader(String path)
```

Here's how you use it:

```
try (Reader r = new FileReader("/file.txt")) {
    int c;
    // -1 indicates the end of the file
    while((c = r.read()) != -1) {
        char character = (char)c;
        // Do something with the character
    }
} catch(IOException e) {
    /** ... */
}
```

There's also a `read()` method that reads characters into an array of `chars`:

```
char[] data = new char[1024];
int numberOfCharsRead = r.read(data);
while((numberOfCharsRead = r.read(data)) != -1) {
    // Do something with the array data
}
```

`FileReader` assumes that you want to decode the characters in the file using the default character encoding of the machine your program is running on.

FileWriter

`FileWriter` writes characters to a text file. It inherits from `Writer`.

It can be created either with a `File` object or a `String` path and an optional `boolean` that indicates whether you want to overwrite or append to the file if it exists (it's overwritten by default):

```
FileWriter(File file)
FileWriter(File file, boolean append)
FileWriter(String path)
FileWriter(String path, boolean append)
```

Here's how you use it:

```
try (Writer w = new FileWriter("/file.txt")) {
    w.write('-'); // writing a character
    // writing a string
    w.write("Writing to the file...");
} catch(IOException e) {
    /** ... */
}
```

Just like an `OutputStream`, the data may get cached internally in memory and written to disk at a later time. If you want to make sure that all data is written to disk without having to close the `FileWriter`, you can call the `flush()` method every once in a while.

`FileWriter` also contains overloaded versions of `write()` that allow you to write data contained in a `char` array, or in a `String`.

`FileWriter` assumes that you want to encode the characters in the file using the default character encoding of the machine your program is running on.

BufferedReader

`BufferedReader` reads text from a character stream. Rather than read one character at a time, `BufferedReader` reads a large block at a time into a buffer. It inherits from `Reader`.

This is a wrapper class that is created by passing a `Reader` to its constructor, and optionally, the size of the buffer:

```
BufferedReader(Reader in)
BufferedReader(Reader in, int size)
```

`BufferedReader` has one extra read method (in addition to the ones inherited by `Reader`), `readLine()`. Here's how you use it:

```
try (BufferedReader br = new BufferedReader(new FileReader("/file.txt"))) {
    String line;
    // null indicates the end of the file
    while((line = br.readLine()) != null) {
```

```
        // Do something with the line
    }
} catch(IOException e) {
    /** ... */
}
```

When the `BufferedReader` is closed, it will also close the `Reader` instance it reads from.

`BufferedWriter`

`BufferedWriter` writes text to a character stream, buffering characters for efficiency. It inherits from `Writer`.

This is a wrapper class that is created by passing a `Writer` to its constructor, and optionally, the size of the buffer:

```
BufferedWriter(Writer out)
BufferedWriter(Writer out, int size)
```

`BufferedWriter` has one extra write method (in addition to the ones inherited by `Writer`), `newLine()`. Here's how you use it:

```
try (BufferedWriter bw = new BufferedWriter(new FileWriter("/file.txt"))) {
    bw.write("Writing to the file...");
    bw.newLine();
} catch(IOException e) {
    /** ... */
}
```

Since data is written to a buffer first, you can call the `flush()` method to make sure that the text written until that moment is indeed written to the disk.

When the `BufferedWriter` is closed, it will also close the `Writer` instance it writes to.

`ObjectInputStream` and `ObjectOutputStream`

The process of converting an object to a data format that can be stored (in a file, for example) is called *serialization* and converting that stored data format into an object is called *deserialization*.

If you want to serialize an object, its class must implement the `java.io.Serializable` interface, which has no methods to implement, it only tags the objects of that class as serializable.

We'll cover this process in more detail later, but right now, you have to know that `ObjectOutputStream` allows you to serialize objects to an `OutputStream` while `ObjectInputStream` allows you to deserialize objects from an `InputStream`. So both are considered wrapper classes.

Here's the constructor of the `ObjectOutputStream` class:

```
ObjectOutputStream(OutputStream out)
```

This class has methods to write many primitive types, like:

```
void writeInt(int val)
void writeBoolean(boolean val)
```

But the most useful is `writeObject(Object)`. Here's an example:

```
class Box implements java.io.Serializable {
    /** ... */
}
...
try(ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("obj.dat"))) {
    Box box = new Box();
```

```
    oos.writeObject(box);
} catch(IOException e) {
    /** ... */
}
```

To deserialize the file `obj.dat`, we use `ObjectInputStream` class. Here's its constructor:

```
ObjectInputStream(InputStream in)
```

This class has methods to read many data types, among them:

```
Object readObject() throws IOException, ClassNotFoundException
```

Notice that it returns an `Object` type. Thus, we have to cast the object explicitly. This can lead to a `ClassCastException` thrown at runtime. Note that this method also throws a `ClassNotFoundException` (a checked exception), in case the class of a serialized object cannot be found.

Here's an example:

```
try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream("obj.dat"))) {
    Box box = null;
    Object obj = ois.readObject();
    if(obj instanceof Box) {
        box = (Box)obj;
    }
} catch(IOException ioe) {
    /** ... */
} catch(ClassNotFoundException cnfe) {
    /** ... */
}
```

PrintStream

`PrintStream` is a subclass of `OutputStream` that adds functionality for printing various data types in a human-readable format. It is similar to `PrintWriter`, but it works with `OutputStreams` only. Here's a look at its constructors:

```
PrintStream(OutputStream out)
PrintStream(OutputStream out, boolean autoFlush)
PrintStream(OutputStream out, boolean autoFlush, String encoding) throws UnsupportedEncodingException
PrintStream(File file) throws FileNotFoundException
PrintStream(File file, String encoding) throws FileNotFoundException, UnsupportedEncodingException
PrintStream(String fileName) throws FileNotFoundException
PrintStream(String fileName, String encoding) throws FileNotFoundException, UnsupportedEncodingException
```

By default, it uses the default charset of the machine you're running the program, but you can specify a charset if needed.

`PrintStream` has the `write()` method like other `OutputStream` subclasses, but it overrides them to avoid throwing an `IOException`.

It also adds methods such as `print()`, `println()`, `format()`, and `printf()` for convenient output. Here's how you use this class:

```
// Opens or creates the file without automatic line flushing
// and using the default character encoding
try (PrintStream ps = new PrintStream("file.txt")) {
    ps.write("Hi".getBytes()); // Writing a String as bytes
    ps.write(100); // Writing a character as bytes
```

```
    // write the string representation of the argument
    // it has versions for all primitives, char[], String, and Object
    ps.print(true);
    ps.print(10);

    // same as print() but it also writes a line break as defined by
    // System.getProperty("line.separator") after the value
    ps.println(); // Just writes a new line
    ps.println("A new line...");

    // format() and printf() are the same methods
    // They write a formatted string using a format string,
    // its arguments and an optional Locale
    ps.format("%s %d", "Formatted string ", 1);
    ps.printf("%s %d", "Formatted string ", 2);
    ps.format(Locale.GERMAN, "%.2f", 3.1416);
    ps.printf(Locale.GERMAN, "%.3f", 3.1416);
} catch (FileNotFoundException e) {
    // if the file cannot be opened or created
}
```

You can learn more about format strings for `format()` and `printf()` in the documentation of the `java.util.Formatter` class.

PrintWriter

`PrintWriter` is a subclass of `Writer` that writes formatted data to another (wrapped) stream, even an `OutputStream`. Just look at its constructors:

```
PrintWriter(File file) throws FileNotFoundException
PrintWriter(File file, String charset) throws FileNotFoundException, UnsupportedEncodingException
PrintWriter(OutputStream out)
PrintWriter(OutputStream out, boolean autoFlush)
PrintWriter(String fileName) throws FileNotFoundException
PrintWriter(String fileName, String charset) throws FileNotFoundException, UnsupportedEncodingException
PrintWriter(Writer out)
PrintWriter(Writer out, boolean autoFlush)
```

By default, it uses the default charset of the machine you're running the program, but this class accepts the following charsets (there are other optional charsets):

- `US-ASCII`
- `ISO-8859-1`
- `UTF-8`
- `UTF-16BE`
- `UTF-16LE`
- `UTF-16`

As any `Writer`, this class has the `write()` method we've seen in other `Writer` subclasses, but it overwrites them to avoid throwing an `IOException`.

It also adds the methods `format()`, `print()`, `printf()`, `println()`.

Here's how you use this class:

```
// Opens or creates the file without automatic line flushing
// and converting characters by using the default character encoding
try(PrintWriter pw = new PrintWriter("/file.txt")) {
```

```
    pw.write("Hi"); // Writing a String
    pw.write(100); // Writing a character

    // write the string representation of the argument
    // it has versions for all primitives, char[], String, and Object
    pw.print(true);
    pw.print(10);

    // same as print() but it also writes a line break as defined by
    // System.getProperty("line.separator") after the value
    pw.println(); // Just writes a new line
    pw.println("A new line...");

    // format() and printf() are the same methods
    // They write a formatted string using a format string,
    // its arguments and an optional Locale
    pw.format("%s %d", "Formatted string ", 1);
    pw.printf("%s %d", "Formatted string ", 2);
    pw.format(Locale.GERMAN, "%.2f", 3.1416);
    pw.printf(Locale.GERMAN, "%.3f", 3.1416);
} catch(FileNotFoundException e) {
    // if the file cannot be opened or created
}
```

Just like with `PrintWriter`, you can learn more about format strings for `format()` and `printf()` in the documentation of the `java.util.Formatter` class.

Standard Streams

Java initializes and provides three stream objects as `public static` fields of the `java.lang.System` class:

- `InputStream System.in`
  The standard input stream (typically the input from the keyboard)
- `PrintStream System.out`
  The standard output stream (typically the default display output)
- `PrintStream System.err`
  The standard error output stream (typically the default error display)

Remember, `PrintStream` does exactly the same and has the same features that `PrintWriter`, it just works with `OutputStreams` only.

The following example shows how to read a single character (a byte) from the command line:

```
System.out.print("Enter a character: ");
try {
    int c = System.in.read();
} catch(IOException e) {
    System.err.println("Error: " + e);
}
```

Or to read strings:

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
String line = br.readLine();
// Or using the java.util.Scanner class
Scanner scanner = new Scanner(System.in);
String line = scanner.nextLine();
```

These streams (`System.in`, `System.out`, `System.err`) are used for basic input and output in many Java programs.

Copying, Moving, Deleting, and Comparing Files

You have previously learned that the `java.nio.file.Files` class provides various methods for file operations like copying, moving, deleting, and comparing files. Let's explore these operations in detail.

The `Files.copy()` method allows you to copy a file from one location to another. It takes a source path and a target path as parameters.

If the target file already exists, you can specify how to handle the copy operation using the `StandardCopyOption` enum.

```
Path source = Paths.get("path/to/source/file.txt");
Path target = Paths.get("path/to/target/file.txt");

// Copy the file, replacing the target file if it exists
Files.copy(source, target, StandardCopyOption.REPLACE_EXISTING);
```

The `StandardCopyOption.REPLACE_EXISTING` option indicates that if the target file already exists, it should be replaced with the source file.

You can also copy files using I/O streams. This is useful when you need more control over the copying process or when working with large files:

```
try (InputStream inputStream = new FileInputStream("source.txt");
     OutputStream outputStream = new FileOutputStream("target.txt")) {

    byte[] buffer = new byte[1024]; // Buffer size can be adjusted for performance
    int bytesRead;
    while ((bytesRead = inputStream.read(buffer)) != -1) {
        outputStream.write(buffer, 0, bytesRead);
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

In this example, we create an `InputStream` to read from the source file and an `OutputStream` to write to the target file. We use a buffer to read and write the data in chunks.

To copy a file into a directory, you can specify the target directory path and the filename.

```
Path sourceFile = Paths.get("path/to/source/file.txt");
Path targetDirectory = Paths.get("path/to/target/directory");

// Copy the file into the target directory
Files.copy(sourceFile, targetDirectory.resolve(sourceFile.getFileName()));
```

The `targetDirectory.resolve(sourceFile.getFileName())` expression creates the target path by combining the target directory path with the file name of the source file.

The `Files.move()` method allows you to move or rename a file or directory.

```
Path source = Paths.get("path/to/source/file.txt");
Path target = Paths.get("path/to/target/file.txt");

// Move the file
Files.move(source, target);
```

If the target file already exists, an exception will be thrown. You can use the `StandardCopyOption.REPLACE_EXISTING` option to replace the target file if it exists:

```
Files.move(source, target, StandardCopyOption.REPLACE_EXISTING);
```

An atomic move ensures that the move operation is performed as a single indivisible operation. It either completes successfully or fails without any partial changes:

```
Path source = Paths.get("path/to/source/file.txt");
Path target = Paths.get("path/to/target/file.txt");

// Perform an atomic move
Files.move(source, target, StandardCopyOption.ATOMIC_MOVE);
```

The `StandardCopyOption.ATOMIC_MOVE` option guarantees that the move operation is atomic, preventing data corruption during the move process.

The `Files.delete()` method allows you to delete a file or an empty directory:

```
Path path = Paths.get("path/to/file.txt");

// Delete the file
Files.delete(path);
```

If the file does not exist, a `NoSuchFileException` will be thrown.

The `Files.deleteIfExists()` method deletes the file if it exists and returns a boolean indicating whether the file was deleted:

```
Path path = Paths.get("path/to/file.txt");

// Delete the file if it exists
boolean deleted = Files.deleteIfExists(path);
```

This method does not throw an exception if the file does not exist.

The `Files.isSameFile()` method allows you to determine if two paths locate the same file in the file system:

```
Path path1 = Paths.get("path/to/file1.txt");
Path path2 = Paths.get("path/to/file2.txt");

// Check if the paths refer to the same file
boolean isSame = Files.isSameFile(path1, path2);
```

It returns `true` if the paths refer to the same file, and `false` otherwise.

The `Files.mismatch()` method compares the content of two files and returns the position of the first mismatched byte:

```
Path file1 = Paths.get("path/to/file1.txt");
Path file2 = Paths.get("path/to/file2.txt");

// Compare the content of the files
long mismatchPosition = Files.mismatch(file1, file2);
```

If the files have identical content, it returns -1. If the files have different sizes, it returns the size of the smaller file.

These are some of the key methods provided by the `Files` class for copying, moving, deleting, and comparing files in Java. They offer convenient ways to perform common file operations without the need for manual I/O stream handling.

Reading and Writing Files

Java provides several methods in the `java.nio.file.Files` class for reading from and writing to files. Let's explore some commonly used methods and techniques.

Reading Files

The `Files` class offers two convenient methods for reading the contents of a file: `readAllLines()` and `lines()`.

The `Files.readAllLines()` method reads all the lines of a file into a `List<String>`:

```
Path path = Paths.get("path/to/file.txt");

try {
    List<String> lines = Files.readAllLines(path);
    for (String line : lines) {
        System.out.println(line);
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

This method is suitable when you need to process all the lines of a file at once. However, note that it reads the entire file into memory, so it may not be efficient for large files.

On the other hand, the `Files.lines()` method returns a `Stream<String>` that allows you to process the lines of a file lazily:

```
Path path = Paths.get("path/to/file.txt");

try (Stream<String> lines = Files.lines(path)) {
    lines.forEach(System.out::println);
} catch (IOException e) {
    e.printStackTrace();
}
```

This method is more memory-efficient as it reads the lines on-demand and does not load the entire file into memory at once. It is especially useful when you need to process large files or perform operations like filtering or mapping on the lines.

However, for more control over the reading process, you can use the `Files.newBufferedReader()` method to create a `BufferedReader` instance:

```
Path path = Paths.get("path/to/file.txt");

try (BufferedReader reader = Files.newBufferedReader(path)) {
    String line;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

The `BufferedReader` provides methods such as `readLine()` to read the file line by line, allowing you to process the lines as needed.

Writing Files

The `Files` class provides methods for writing content to files, such as `write()` and `newBufferedWriter()`.

The `Files.write()` method allows you to write content to a file in a single operation:

```
Path path = Paths.get("path/to/file.txt");
String content = "Hello, World!";

try {
    Files.write(path, content.getBytes(StandardCharsets.UTF_8));
} catch (IOException e) {
    e.printStackTrace();
}
```

This method writes the specified byte array to the file. If the file already exists, it will be overwritten by default.

Additionally, you can specify additional options using the `StandardOpenOption` enum.

```
Path path = Paths.get("path/to/file.txt");
String content = "Appended content";

try {
    Files.write(path, content.getBytes(StandardCharsets.UTF_8), StandardOpenOption.APPEND);
} catch (IOException e) {
    e.printStackTrace();
}
```

The `StandardOpenOption.APPEND` option specifies that the content should be appended to the end of the file instead of overwriting it.

Other useful options include: - `StandardOpenOption.CREATE`: Creates a new file if it doesn't exist. - `StandardOpenOption.CREATE_NEW`: Creates a new file, failing if it already exists. - `StandardOpenOption.TRUNCATE_EXISTING`: Truncates the file to zero bytes if it exists.

These options give you more control over how the file is opened and written to.

However, you can also use the `Files.newBufferedWriter()` method to create a `BufferedWriter` instance:

```
Path path = Paths.get("path/to/file.txt");

try (BufferedWriter writer = Files.newBufferedWriter(path)) {
    writer.write("Hello, World!");
    writer.newLine();
    writer.write("This is a new line.");
} catch (IOException e) {
    e.printStackTrace();
}
```

The `BufferedWriter` provides methods such as `write()` and `newLine()` to write content to the file, allowing you to write line by line or in chunks.

Working with File Attributes

The `java.nio.file` package provides classes and methods for working with file attributes. File attributes are metadata associated with a file or directory, such as size, modification time, permissions, and more. You can retrieve and modify file attributes using the NIO.2 API.

Java defines several attribute and view types that represent different sets of file attributes:

BasicFileAttributes The `BasicFileAttributes` interface provides basic file attributes that are common across different file systems. It includes attributes like: - `FileTime creationTime()`: Returns the creation time of the file. - `FileTime lastModifiedTime()`: Returns the last modification time of the file. - `FileTime`

`lastAccessTime()`: Returns the last access time of the file. - `long size()`: Returns the size of the file in bytes. - `boolean isRegularFile()`, `isDirectory()`, `isSymbolicLink()`: Checks the type of the file.

DosFileAttributes The `DosFileAttributes` interface extends `BasicFileAttributes` and provides additional attributes specific to DOS/Windows file systems. It includes attributes like: - `boolean isReadOnly()`: Checks if the file is read-only. - `boolean isHidden()`: Checks if the file is hidden. - `boolean isArchive()`: Checks if the file is an archive. - `boolean isSystem()`: Checks if the file is a system file.

PosixFileAttributes The `PosixFileAttributes` interface extends `BasicFileAttributes` and provides additional attributes specific to POSIX-compliant file systems. It includes attributes like: - `UserPrincipal owner()`: Returns the owner of the file. - `GroupPrincipal group()`: Returns the group owner of the file. - `Set<PosixFilePermission> permissions()`: Returns the file permissions as a set of `PosixFilePermission`.

To retrieve file attributes, you can use the `Files.readAttributes()` method, specifying the attribute type you want to retrieve:

```
Path path = Paths.get("path/to/file.txt");

try {
    BasicFileAttributes attrs = Files.readAttributes(path, BasicFileAttributes.class);
    System.out.println("Creation Time: " + attrs.creationTime());
    System.out.println("Last Modified Time: " + attrs.lastModifiedTime());
    System.out.println("Size: " + attrs.size());
} catch (IOException e) {
    e.printStackTrace();
}
```

In this example, we retrieve the `BasicFileAttributes` of the file and access its creation time, last modified time, and size.

Additionally, you can specify the `LinkOption` to control how symbolic links are handled:

```
Path path = Paths.get("path/to/symlink.txt");

try {
    BasicFileAttributes attrs = Files.readAttributes(path, BasicFileAttributes.class, LinkOption.NOFOLLOW_LINKS);
    boolean isSymLink = attrs.isSymbolicLink();
} catch (IOException e) {
    e.printStackTrace();
}
```

By passing `LinkOption.NOFOLLOW_LINKS`, the `readAttributes()` method will not follow symbolic links and will instead return the attributes of the symbolic link itself.

To quickly check if a file is accessible for reading, writing, or executing, you can use the `Files.isReadable()`, `Files.isWritable()`, and `Files.isExecutable()` methods:

```
Path path = Paths.get("path/to/file.txt");

boolean isReadable = Files.isReadable(path);
boolean isWritable = Files.isWritable(path);
boolean isExecutable = Files.isExecutable(path);
```

These methods return `true` if the file is accessible for the respective operation, and `false` otherwise.

To modify file attributes you can use the `Files.setAttribute()` method, specifying the attribute name and value:

```
Path path = Paths.get("path/to/file.txt");
```

```
try {
    Files.setAttribute(path, "dos:readonly", true);
    Files.setAttribute(path, "dos:hidden", true);
} catch (IOException e) {
    e.printStackTrace();
}
```

In this example, we set the `readonly` and `hidden` attributes of a file on a DOS/Windows file system.

Traversing a Directory Tree

Traversing a directory tree, also known as walking a directory tree, refers to the process of recursively visiting all the subdirectories and files within a given directory. The NIO.2 API, in particular, the `Files` class, provides methods to simplify this process and allows you to perform actions on each visited file and directory.

The `Files.walk()` method is a convenient way to traverse a directory tree. It returns a `Stream<Path>` that represents the file tree rooted at the given starting directory:

```
Path startingDir = Paths.get("/path/to/directory");

try (Stream<Path> stream = Files.walk(startingDir)) {
    stream.forEach(path -> {
        // Process each path
        System.out.println(path);
    });
} catch (IOException e) {
    e.printStackTrace();
}
```

The `Files.walk()` method visits all the files and directories in the tree, including the starting directory itself. You can perform various operations on each path using the stream API, such as filtering, mapping, or collecting the paths.

You can control the depth of the traversal by passing a maximum depth value to the `Files.walk()` method:

```
Path startingDir = Paths.get("/path/to/directory");
int maxDepth = 3;

try (Stream<Path> stream = Files.walk(startingDir, maxDepth)) {
    // ...
} catch (IOException e) {
    e.printStackTrace();
}
```

In this example, the traversal will go up to a maximum depth of 3 levels below the starting directory. A depth of 0 means only the starting directory itself is visited.

By default, `Files.walk()` follows symbolic links. If you want to control this behavior, you can pass a `FileVisitOption` to the method:

```
Path startingDir = Paths.get("/path/to/directory");

try (Stream<Path> stream = Files.walk(startingDir, FileVisitOption.FOLLOW_LINKS)) {
    // ...
} catch (IOException e) {
    e.printStackTrace();
}
```

The `FileVisitOption.FOLLOW_LINKS` option specifies that symbolic links should be followed during the traversal.

However, when traversing a directory tree in Java, it's important to be aware of circular paths caused by symbolic links. A circular path occurs when a symbolic link points to a directory that is an ancestor of the link, creating an infinite loop.

To avoid circular paths, you can use the `FileVisitOption.FOLLOW_LINKS` option and implement your own cycle detection logic. Here's a complete example:

```
import java.io.IOException;
import java.nio.file.*;
import java.nio.file.attribute.BasicFileAttributes;
import java.util.EnumSet;
import java.util.HashSet;
import java.util.Set;

public class DirectoryTraversal {
    public static void main(String[] args) {
        Path startingDir = Paths.get("/path/to/directory");
        Set<Path> visitedPaths = new HashSet<>();

        try {
            Files.walkFileTree(
                            startingDir,
                            EnumSet.of(FileVisitOption.FOLLOW_LINKS),
                            Integer.MAX_VALUE, new SimpleFileVisitor<Path>() {
                @Override
                public FileVisitResult visitFile(
                        Path file, BasicFileAttributes attrs) throws IOException {
                    if (visitedPaths.contains(file)) {
                        // Circular path detected, skip processing
                        return FileVisitResult.CONTINUE;
                    }
                    visitedPaths.add(file);
                    // Process the file
                    System.out.println(file);
                    return FileVisitResult.CONTINUE;
                }

                @Override
                public FileVisitResult preVisitDirectory(
                        Path dir, BasicFileAttributes attrs) throws IOException {
                    if (visitedPaths.contains(dir)) {
                        // Circular path detected, skip processing
                        return FileVisitResult.SKIP_SUBTREE;
                    }
                    visitedPaths.add(dir);
                    // Process the directory
                    System.out.println(dir);
                    return FileVisitResult.CONTINUE;
                }

                @Override
                public FileVisitResult visitFileFailed(
```

```
                    Path file, IOException exc) throws IOException {
                System.err.println(
                    "Error visiting file: " + file + " - " + exc.getMessage()
                );
                return FileVisitResult.CONTINUE;
            }
        });
    } catch (IOException e) {
        e.printStackTrace();
    }
  }
}
```

This program demonstrates how to traverse a directory tree while avoiding circular paths caused by symbolic links. It begins by defining the starting directory specified by the `startingDir` variable and uses the `Files.walkFileTree` method to traverse the directory tree. This method is recommended because it visits all files and directories and can follow symbolic links when specified by the `FileVisitOption.FOLLOW_LINKS` option. To prevent infinite loops caused by circular paths, the program maintains a set of visited paths (`visitedPaths`). This set is used to keep track of all the directories and files that have already been visited during the traversal.

The core of the program is the implementation of a `SimpleFileVisitor`, which overrides several methods to define custom behaviors for visiting files and directories. When the program finds a file or directory, it checks the `visitedPaths` set to see if the path has already been visited. If the path is found in the set, this indicates a circular path, and the program skips further processing for that path. If the path is not in the set, it is added to the `visitedPaths` set, and the path is processed (in this case, printed to the console). This ensures that each path is processed only once, effectively preventing infinite loops.

Serializing Data

We talked about serialization before. It is the process of converting an object into a byte stream, which can be saved to a file or transmitted over a network. Deserialization is the reverse process, where the byte stream is converted back into an object. Let's explore the key concepts and techniques related to serialization in Java.

Serialization allows you to persist the state of an object and recreate it later. This is useful for: - Saving object state to a file - Sending objects over a network - Caching objects for performance

The Java Object Serialization API provides a standard mechanism for developers to handle this process.

To make a class serializable, it must implement the `java.io.Serializable` interface. This is a marker interface (it has no methods) that tells the Java runtime that the class can be serialized:

```
public class Employee implements Serializable {
    private String name;
    private int age;

    // Constructor, getters, and setters
}
```

If you try to serialize a class that doesn't implement that interface, a `java.io.NotSerializableException` (a subclass of `IOException`) will be thrown at runtime.

The `serialVersionUID` is a unique identifier for the serialized class. It's used during deserialization to verify that the sender and receiver of a serialized object have loaded classes for that object that are compatible with respect to serialization:

```
public class Employee implements Serializable {
    private static final long serialVersionUID = 1L;
```

```
    // ... rest of the class
}
```

If you don't explicitly declare a `serialVersionUID`, the Java runtime will generate one based on various aspects of your class. However, it's recommended to declare one explicitly to maintain control over class versioning.

If you have fields in your class that you don't want to be serialized (for example, sensitive data or derived data), you can mark them with the `transient` keyword:

```
public class Employee implements Serializable {
    private static final long serialVersionUID = 1L;
    private String name;
    private transient String password; // This won't be serialized
    // ... rest of the class
}
```

In summary, to ensure a class is serializable: 1. The class must be marked `Serializable`. 2. Every instance member of the class must be serializable, marked transient, or have a `null` value at the time of serialization.

After that, you can use `ObjectOutputStream` and `ObjectInputStream` for the serialization/deserialization process as shown in a previous section. Here's a complete example demonstrating this process:

```
import java.io.*;

public class SerializationDemo {
    public static void main(String[] args) {
        Person person = new Person("John Doe", 30);

        // Serialization
        try (ObjectOutputStream out =
                 new ObjectOutputStream(new FileOutputStream("person.ser"))) {
            out.writeObject(person);
            System.out.println("Person object serialized");
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Deserialization
        try (ObjectInputStream in =
                 new ObjectInputStream(new FileInputStream("person.ser"))) {
            Person deserializedPerson = (Person) in.readObject();
            System.out.println("Person object deserialized");
            System.out.println("Name: " + deserializedPerson.getName());
            System.out.println("Age: " + deserializedPerson.getAge());
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}

class Person implements Serializable {
    private static final long serialVersionUID = 1L;
    private String name;
    private int age;

    public Person(String name, int age) {
```

```
        this.name = name;
        this.age = age;
    }

    // Getters
    public String getName() { return name; }
    public int getAge() { return age; }

    // ... rest of the class
}
```

In this example, we serialize a `Person` object to a file named `person.ser` and then deserialize it back into a `Person` object.

It's important to note that when deserializing an object, the constructor and any initialization block are not executed. However, the default initialization of instance variables still occurs.

Also, unlike classes, records are automatically serializable. They implicitly implement the `Serializable` interface, so you don't need to explicitly declare it:

```
public record PersonRecord(String name, int age) /** implements Serializable */ {
    // No need to declare serialVersionUID, as it's automatically generated
}
```

Remember, records provide a compact way to declare classes that are primarily used to store data, and their built-in serializability makes them convenient for use in scenarios where object serialization is required.

Reference Tables

Here are some tables to help you review and understand the I/O stream classes and related concepts:

I/O Stream Classes Summary

Stream Type	Byte Streams	Character Streams
Input	**Abstract Classes**: `InputStream` **Concrete Classes**: `FileInputStream`, `BufferedInputStream`, `ObjectInputStream`	**Abstract Classes**: `Reader` **Concrete Classes**: `FileReader`, `BufferedReader`
Output	**Abstract Classes**: `OutputStream` **Concrete Classes**: `FileOutputStream`, `BufferedOutputStream`, `ObjectOutputStream`, `PrintStream`	**Abstract Classes**: `Writer` **Concrete Classes**: `FileWriter`, `BufferedWriter`, `PrintWriter`

File and Path Operations Comparison

Operation	File Class	NIO.2 (Path and Files)
Create File	`file.createNewFile()`	`Files.createFile(path)`
Delete File	`file.delete()`	`Files.delete(path)`
Check Existence	`file.exists()`	`Files.exists(path)`
Get Absolute Path	`file.getAbsolutePath()`	`path.toAbsolutePath()`
Check if Directory	`file.isDirectory()`	`Files.isDirectory(path)`
Check if File	`file.isFile()`	`Files.isRegularFile(path)`
List Directory Contents	`file.list()`, `file.listFiles()`	`Files.list(path)`
Create Directory	`file.mkdir()`	`Files.createDirectory(path)`

Operation	File Class	NIO.2 (Path and Files)
Create Directories	`file.mkdirs()`	`Files.createDirectories(path)`
Rename File	`file.renameTo(dest)`	`Files.move(source, target)`

Files Class Methods Summary

Method	Description
`copy()`	Copies a file to a target file
`createDirectories()`	Creates a directory and any necessary parent directories
`delete()`	Deletes a file or empty directory
`exists()`	Checks file existence
`isDirectory()`	Checks if the path is a directory
`isRegularFile()`	Checks if the path is a regular file
`move()`	Moves or renames a file
`size()`	Returns the size of a file
`readAllBytes()`	Reads all bytes from a file
`readAllLines()`	Reads all lines from a file
`walk()`	Returns a `Stream` of file tree structure
`write()`	Writes bytes or lines to a file

Common File Attributes

Attribute	`BasicFileAttributes`	`DosFileAttributes`	`PosixFileAttributes`
Creation Time			
Last Modified Time			
Last Access Time			
Size			
Is Directory			
Is Regular File			
Is Symbolic Link			
Is Hidden			
Is Read-only			
Owner			
Group			
Permissions			

StandardOpenOption Values

Option	Description
`APPEND`	Append to the end of the file if it exists
`CREATE`	Create a new file if it doesn't exist
`CREATE_NEW`	Create a new file, failing if it already exists
`DELETE_ON_CLOSE`	Delete the file when the stream is closed
`DSYNC`	Synchronize only the file's content with the underlying storage device
`READ`	Open for read access

Option	Description
`SPARSE`	Hint that a newly created file will be sparse
`SYNC`	Synchronize every update to the file's content and metadata with the underlying storage device
`TRUNCATE_EXISTING`	Truncate the file to zero bytes if it exists
`WRITE`	Open for write access

StandardCopyOption Values

Option	Description
`REPLACE_EXISTING`	Replace the target file if it exists
`COPY_ATTRIBUTES`	Copy file attributes to the target file
`ATOMIC_MOVE`	Move the file as an atomic file system operation

Key Points

- In the NIO.2 API, the key classes for file operations are:
  - `java.nio.file.Path`: Represents file and directory paths in a more flexible way (compared to the `java.io.File` class).
  - `java.nio.file.Files`: Utility class for file operations.
- There two main types of I/O streams:
  - Byte streams: Work with raw binary data (8-bit). Main abstract classes are `InputStream` and `OutputStream`.
  - Character streams: Work with text data (16-bit Unicode). Main abstract classes are `Reader` and `Writer`.
- However, streams can be classified as:
  - Input streams: Used to read data from a source.
  - Output streams: Used to write data to a destination.
  - Low-level streams: Directly connected to the data source or destination.
  - High-level streams: Built on top of other streams, providing additional functionality.
- Here are the most important stream classes:
  - `FileInputStream` and `FileOutputStream`: Read and write bytes from/to files.
  - `FileReader` and `FileWriter`: Read and write characters from/to files.
  - `BufferedReader` and `BufferedWriter`: Add buffering capability to character streams.
  - `ObjectInputStream` and `ObjectOutputStream`: Read and write serialized objects.
  - `PrintStream` and `PrintWriter`: Write formatted data to byte and character streams respectively.
- The `try-with-resources` statement should be used with I/O classes to ensure proper resource management.
- Most I/O operations can throw `IOException` or its subclasses, which need to be handled.
- Standard streams:
  - `System.in`: Standard input stream (typically keyboard input).
  - `System.out`: Standard output stream (typically console output).
  - `System.err`: Standard error stream (typically error console output).
- When working with character streams, be aware of character encoding. The default is usually the system's default encoding.

- Buffered streams (`BufferedReader`, `BufferedWriter`, etc.) can improve performance by reducing the number of I/O operations.
- The `Files` class provides methods for various file operations like copying, moving, deleting, and comparing files.
- Copying Files:
  - Use `Files.copy(source, target, CopyOption...)` to copy files.
  - `StandardCopyOption.REPLACE_EXISTING` can be used to overwrite existing files.
  - For large files, consider using I/O streams with a buffer for more control over the copying process.
- Moving Files:
  - Use `Files.move(source, target, CopyOption...)` to move or rename files.
  - `StandardCopyOption.ATOMIC_MOVE` ensures the move operation is performed as a single indivisible operation.
- Deleting Files:
  - `Files.delete(path)` deletes a file or empty directory.
  - `Files.deleteIfExists(path)` deletes a file if it exists and returns a `boolean` indicating success.
- Comparing Files:
  - `Files.isSameFile(path1, path2)` checks if two paths refer to the same file.
  - `Files.mismatch(file1, file2)` compares the content of two files and returns the position of the first mismatched byte.
- Reading Files:
  - `Files.readAllLines(path)` reads all lines of a file into a `List<String>`.
  - `Files.lines(path)` returns a `Stream<String>` for lazy processing of file lines.
  - `Files.newBufferedReader(path)` creates a `BufferedReader` for more control over reading.
- Writing Files:
  - `Files.write(path, bytes, OpenOption...)` writes content to a file in a single operation.
  - `Files.newBufferedWriter(path)` creates a `BufferedWriter` for writing line by line or in chunks.
  - Use `StandardOpenOption` enum to specify how the file should be opened or written to.
- File Attributes:
  - `BasicFileAttributes`, `DosFileAttributes`, and `PosixFileAttributes` interfaces provide access to various file attributes.
  - Use `Files.readAttributes(path, Class<A>, LinkOption...)` to retrieve file attributes.
  - `Files.setAttribute(path, attribute, value)` modifies file attributes.
- Directory Traversal:
  - `Files.walk(path, options)` returns a `Stream<Path>` for traversing a directory tree.
  - Use `FileVisitOption.FOLLOW_LINKS` to follow symbolic links during traversal.
  - Implement cycle detection logic to avoid infinite loops caused by circular symbolic links.
- Serialization allows objects to be converted to a byte stream, which can be saved or transmitted. Classes must implement the `Serializable` interface to be serializable.
- Classes must implement the `Serializable` interface to be serializable.
- Use `serialVersionUID` for version control of serialized classes.
- Mark fields as `transient` to exclude them from serialization.
- Use `ObjectOutputStream` for serialization and `ObjectInputStream` for deserialization.

- Records are automatically serializable and don't require explicit implementation of `Serializable`.

Practice Questions

1. What is the result of the following code snippet?

```
import java.nio.file.Path;
import java.nio.file.Paths;

public class PathExample {
    public static void main(String[] args) {
        Path basePath = Paths.get("/home/user");
        Path relativePath = Paths.get("documents/notes.txt");
        Path resultPath = basePath.resolve(relativePath);
        System.out.println(resultPath);
    }
}
```

A) `/home/user`
B) `/home/user/documents`
C) `/documents/notes.txt`
D) `/home/user/documents/notes.txt`

2. What is the result of the following code snippet?

```
import java.nio.file.Path;
import java.nio.file.Paths;

public class PathExample {
    public static void main(String[] args) {
        Path path = Paths.get("/home/user/../documents/./notes.txt");
        Path normalizedPath = path.normalize();
        System.out.println(normalizedPath);
    }
}
```

A) `/home/user/../documents/./notes.txt`
B) `/home/user/documents/notes.txt`
C) `/home/documents/notes.txt`
D) `/documents/notes.txt`

3. Which of the following classes is used for reading character streams in Java?

A) `FileOutputStream`
B) `FileReader`
C) `BufferedOutputStream`
D) `ObjectInputStream`

4. Which of the following code snippets correctly copies a file using the `Files` class, ensuring that an existing target file is overwritten?

A)

```
Path source = Paths.get("source.txt");
Path target = Paths.get("target.txt");
Files.copy(source, target, StandardCopyOption.ATOMIC_MOVE);
```

B)

```
Path source = Paths.get("source.txt");
Path target = Paths.get("target.txt");
Files.move(source, target, StandardCopyOption.REPLACE_EXISTING);
```

C)

```
Path source = Paths.get("source.txt");
Path target = Paths.get("target.txt");
Files.copy(source, target, StandardCopyOption.REPLACE_EXISTING);
```

D)

```
Path source = Paths.get("source.txt");
Path target = Paths.get("target.txt");
Files.copy(source, target, StandardCopyOption.APPEND);
```

5. Which of the following code snippets correctly reads all lines from a file into a `List<String>` using the `java.nio.file` API?

A)

```
Path path = Paths.get("file.txt");
List<String> lines = Files.readAllBytes(path);
```

B)

```
Path path = Paths.get("file.txt");
List<String> lines = Files.readString(path);
```

C)

```
Path path = Paths.get("file.txt");
List<String> lines = Files.lines(path);
```

D)

```
Path path = Paths.get("file.txt");
List<String> lines = Files.readAllLines(path);
```

6. Which of the following code snippets correctly writes a `List<String>` to a file using the `Files` class in Java?

A)

```
Path path = Paths.get("output.txt");
List<String> lines = Arrays.asList("line1", "line2", "line3");
Files.write(path, lines);
```

B)

```
Path path = Paths.get("output.txt");
List<String> lines = Arrays.asList("line1", "line2", "line3");
Files.writeString(path, lines);
```

C)

```
Path path = Paths.get("output.txt");
List<String> lines = Arrays.asList("line1", "line2", "line3");
Files.writeLines(path, lines);
```

D)

```
Path path = Paths.get("output.txt");
List<String> lines = Arrays.asList("line1", "line2", "line3");
Files.write(path, lines, StandardOpenOption.READ);
```

7. Which of the following methods from the `BasicFileAttributes` class retrieves the creation time of a file?

A)

```
BasicFileAttributes attrs = Files.readAttributes(path, BasicFileAttributes.class);
attrs.lastModifiedTime();
```

B)

```
BasicFileAttributes attrs = Files.readAttributes(path, BasicFileAttributes.class);
attrs.creationTime();
```

C)

```
BasicFileAttributes attrs = Files.readAttributes(path, BasicFileAttributes.class);
attrs.lastAccessTime();
```

D)

```
BasicFileAttributes attrs = Files.readAttributes(path, BasicFileAttributes.class);
attrs.size();
```

8. Which of the following code snippets correctly traverses a directory tree using the `Files.walkFileTree` method in Java?

A)

```
Path start = Paths.get("start_directory");
Files.walkFileTree(start, new SimpleFileVisitor<Path>() {
    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws IOException {
        return FileVisitResult.SKIP_SUBTREE;
    }
});
```

B)

```
Path start = Paths.get("start_directory");
Files.walkFileTree(start, new SimpleFileVisitor<Path>() {
    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws IOException {
        throw new IOException("Error visiting file");
    }
});
```

C)

```
Path start = Paths.get("start_directory");
Files.walkFileTree(start, new SimpleFileVisitor<Path>() {
    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws IOException {
        System.out.println("Visited file: " + file);
        return FileVisitResult.TERMINATE;
    }
});
```

D)

```
Path start = Paths.get("start_directory");
Files.walkFileTree(start, EnumSet.noneOf(FileVisitOption.class), Integer.MAX_VALUE, new SimpleFileVisitor<Path>()
    @Override
```

```
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws IOException {
        System.out.println("Visited file: " + file);
        return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs) throws IOException {
        return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult visitFileFailed(Path file, IOException exc) throws IOException {
        return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult postVisitDirectory(Path dir, IOException exc) throws IOException {
        return FileVisitResult.CONTINUE;
    }
});
```

9. Which of the following code snippets correctly serializes an object to a file?

A)

```
class Animal implements Serializable {
    private static final long serialVersionUID = 1L;
    private String species;
    private int age;

    public Animal(String species, int age) {
        this.species = species;
        this.age = age;
    }
}

Animal animal = new Animal("Lion", 5);
try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream("animal.ser"))) {
    ois.writeObject(animal);
} catch (IOException e) {
    e.printStackTrace();
}
```

B)

```
class Animal implements Serializable {
    private static final long serialVersionUID = 1L;
    private String species;
    private int age;

    public Animal(String species, int age) {
        this.species = species;
        this.age = age;
    }
}
```

```
Animal animal = new Animal("Lion", 5);
try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("animal.ser"))) {
    oos.writeObject(animal);
} catch (IOException e) {
    e.printStackTrace();
}
```

C)

```
class Animal {
    private String species;
    private int age;

    public Animal(String species, int age) {
        this.species = species;
        this.age = age;
    }
}

Animal animal = new Animal("Lion", 5);
try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("animal.ser"))) {
    oos.writeObject(animal);
} catch (IOException e) {
    e.printStackTrace();
}
```

D)

```
class Animal implements Serializable {
    private static final long serialVersionUID = 1L;
    private String species;
    private int age;

    public Animal(String species, int age) {
        this.species = species;
        this.age = age;
    }
}

Animal animal = new Animal("Lion", 5);
try (BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream("animal.ser"))) {
    bos.write(animal);
} catch (IOException e) {
    e.printStackTrace();
}
```

Chapter TWELVE

File I/O and Serialization

Answers

1. The correct answer is D.

Explanation:

- **A)** `/home/user`

  - This option is incorrect. The `resolve` method appends the given path to the base path. It does not return the base path alone.
- **B)** `/home/user/documents`
  - This option is incorrect. The `resolve` method includes the entire relative path provided as an argument, not just part of it.
- **C)** `/documents/notes.txt`
  - This option is incorrect. The `resolve` method combines the base path with the given relative path; it does not replace the base path with the relative path.
- **D)** `/home/user/documents/notes.txt`
  - This option is correct. The `resolve` method appends the relative path to the base path, resulting in `/home/user/documents/notes.txt`.

2. The correct answer is C.

Explanation:

- **A)** `/home/user/../documents/./notes.txt`
  - This option is incorrect. The `normalize` method removes redundant `.` and `..` elements, so it wouldn't leave the path as is.
- **B)** `/home/user/documents/notes.txt`
  - This option is incorrect. While the `.` is removed, the `..` navigates one directory up, resulting in an incorrect final path.
- **C)** `/home/documents/notes.txt`
  - This option is correct. The `normalize` method processes the path by removing the `.` and moving one directory up due to `..`, resulting in `/home/documents/notes.txt`.
- **D)** `/documents/notes.txt`
  - This option is incorrect. The `normalize` method does not completely remove the leading part of the path up to `documents`. It only processes the `.` and `..` elements.

3. The correct answer is B.

Explanation:

- **A)** `FileOutputStream`
  - This option is incorrect. `FileOutputStream` is used for writing binary data to a file, not for reading character streams.
- **B)** `FileReader`
  - This option is correct. `FileReader` is designed for reading character streams from a file, making it the appropriate class for this purpose.
- **C)** `BufferedOutputStream`
  - This option is incorrect. `BufferedOutputStream` is used to write binary data to an output stream, buffering the data for efficient writing. It is not used for reading character streams.
- **D)** `ObjectInputStream`
  - This option is incorrect. `ObjectInputStream` is used for deserializing objects from an input stream, not for reading character streams.

4. The correct answer is C.

Explanation:

- **A)**

```
Path source = Paths.get("source.txt");
Path target = Paths.get("target.txt");
Files.copy(source, target, StandardCopyOption.ATOMIC_MOVE);
```

  - This option is incorrect. `StandardCopyOption.ATOMIC_MOVE` is used for moving files atomically, not for copying. It does not ensure that an existing file is overwritten.

- **B)**

```
Path source = Paths.get("source.txt");
Path target = Paths.get("target.txt");
Files.move(source, target, StandardCopyOption.REPLACE_EXISTING);
```

- This option is incorrect. `Files.move` is used to move or rename a file, not to copy it. `StandardCopyOption.REPLACE_EXISTING` ensures the target file is overwritten during a move, not a copy.

- **C)**

```
Path source = Paths.get("source.txt");
Path target = Paths.get("target.txt");
Files.copy(source, target, StandardCopyOption.REPLACE_EXISTING);
```

- This option is correct. `Files.copy` with `StandardCopyOption.REPLACE_EXISTING` ensures the target file is overwritten if it exists, which is the correct way to copy a file with overwriting.

- **D)**

```
Path source = Paths.get("source.txt");
Path target = Paths.get("target.txt");
Files.copy(source, target, StandardCopyOption.APPEND);
```

- This option is incorrect. `StandardCopyOption.APPEND` does not exist in the `StandardCopyOption` enum, making this code snippet invalid.

5. The correct answer is D.

Explanation:

- **A)**

```
Path path = Paths.get("file.txt");
List<String> lines = Files.readAllBytes(path);
```

- This option is incorrect. `Files.readAllBytes(path)` returns a byte array, not a `List<String>`.

- **B)**

```
Path path = Paths.get("file.txt");
List<String> lines = Files.readString(path);
```

- This option is incorrect. `Files.readString(path)` returns a single `String` containing the entire content of the file, not a `List<String>`.

- **C)**

```
Path path = Paths.get("file.txt");
List<String> lines = Files.lines(path);
```

- This option is incorrect. `Files.lines(path)` returns a `Stream<String>`, not a `List<String>`. It provides a lazy-loaded stream of lines.

- **D)**

```
Path path = Paths.get("file.txt");
List<String> lines = Files.readAllLines(path);
```

- This option is correct. `Files.readAllLines(path)` reads all lines from the file and returns them as a `List<String>`, which is the desired behavior.

6. The correct answer is A.

Explanation:

- **A)**

```
Path path = Paths.get("output.txt");
List<String> lines = Arrays.asList("line1", "line2", "line3");
Files.write(path, lines);
```

- This option is correct. `Files.write(path, lines)` writes the given list of strings to the file at the specified path, creating the file if it does not exist.
- **B)**

```
Path path = Paths.get("output.txt");
List<String> lines = Arrays.asList("line1", "line2", "line3");
Files.writeString(path, lines);
```

- This option is incorrect. `Files.writeString(path, lines)` does not exist. `Files.writeString` expects a single `String` as the second argument, not a `List<String>`.
- **C)**

```
Path path = Paths.get("output.txt");
List<String> lines = Arrays.asList("line1", "line2", "line3");
Files.writeLines(path, lines);
```

- This option is incorrect. `Files.writeLines(path, lines)` does not exist. There is no such method in the `Files` class.
- **D)**

```
Path path = Paths.get("output.txt");
List<String> lines = Arrays.asList("line1", "line2", "line3");
Files.write(path, lines, StandardOpenOption.READ);
```

- This option is incorrect. `StandardOpenOption.READ` is not a valid option for writing files. It is used for reading files.

7. The correct answer is B.

Explanation:

- **A)**

```
BasicFileAttributes attrs = Files.readAttributes(path, BasicFileAttributes.class);
attrs.lastModifiedTime();
```

- This option is incorrect. `attrs.lastModifiedTime()` retrieves the last modified time of the file, not the creation time.
- **B)**

```
BasicFileAttributes attrs = Files.readAttributes(path, BasicFileAttributes.class);
attrs.creationTime();
```

- This option is correct. `attrs.creationTime()` retrieves the creation time of the file, which is the correct method from `BasicFileAttributes` for this purpose.
- **C)**

```
BasicFileAttributes attrs = Files.readAttributes(path, BasicFileAttributes.class);
attrs.lastAccessTime();
```

- This option is incorrect. `attrs.lastAccessTime()` retrieves the last access time of the file, not the creation time.
- **D)**

```
BasicFileAttributes attrs = Files.readAttributes(path, BasicFileAttributes.class);
attrs.size();
```

- This option is incorrect. `attrs.size()` retrieves the size of the file, not the creation time.

8. The correct answer is D.

Explanation:

- **A)**

```
Path start = Paths.get("start_directory");
Files.walkFileTree(start, new SimpleFileVisitor<Path>() {
    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws IOException {
        return FileVisitResult.SKIP_SUBTREE;
    }
});
```

- This option is incorrect. `FileVisitResult.SKIP_SUBTREE` will skip the traversal of the entire subtree, not allowing the complete traversal of the directory tree.

- **B)**

```
Path start = Paths.get("start_directory");
Files.walkFileTree(start, new SimpleFileVisitor<Path>() {
    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws IOException {
        throw new IOException("Error visiting file");
    }
});
```

- This option is incorrect. Throwing an `IOException` inside `visitFile` will stop the traversal due to an unhandled exception.

- **C)**

```
Path start = Paths.get("start_directory");
Files.walkFileTree(start, new SimpleFileVisitor<Path>() {
    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws IOException {
        System.out.println("Visited file: " + file);
        return FileVisitResult.TERMINATE;
    }
});
```

- This option is incorrect. The use of `FileVisitResult.TERMINATE` will stop the traversal after visiting the first file, not allowing the complete traversal of the directory tree.

- **D)**

```
Path start = Paths.get("start_directory");
Files.walkFileTree(start, EnumSet.noneOf(FileVisitOption.class), Integer.MAX_VALUE, new SimpleFileVisitor<Path>()
    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws IOException {
        System.out.println("Visited file: " + file);
        return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs) throws IOException {
```

```
        return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult visitFileFailed(Path file, IOException exc) throws IOException {
        return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult postVisitDirectory(Path dir, IOException exc) throws IOException {
        return FileVisitResult.CONTINUE;
    }
});
```

- This option is correct. It uses `Files.walkFileTree` with `SimpleFileVisitor`, specifying no special `FileVisitOption` and setting the maximum depth to `Integer.MAX_VALUE`, ensuring full traversal of the directory tree. Additionally, it correctly handles directory pre-visit, file visit, file visit failure, and directory post-visit events.

9. The correct answer is B.

Explanation:

- **A)**

```
class Animal implements Serializable {
    private static final long serialVersionUID = 1L;
    private String species;
    private int age;

    public Animal(String species, int age) {
        this.species = species;
        this.age = age;
    }
}

Animal animal = new Animal("Lion", 5);
try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream("animal.ser"))) {
    ois.writeObject(animal);
} catch (IOException e) {
    e.printStackTrace();
}
```

  - This option is incorrect. `ObjectInputStream` is used for deserialization (reading objects from a stream), not serialization. It should be `ObjectOutputStream`.

- **B)**

```
class Animal implements Serializable {
    private static final long serialVersionUID = 1L;
    private String species;
    private int age;

    public Animal(String species, int age) {
        this.species = species;
        this.age = age;
    }
}
```

```
Animal animal = new Animal("Lion", 5);
try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("animal.ser"))) {
    oos.writeObject(animal);
} catch (IOException e) {
    e.printStackTrace();
}
```

  - This option is correct. `ObjectOutputStream` is used to serialize an object to a file, which is what this code snippet does correctly.

- **C)**

```
class Animal {
    private String species;
    private int age;

    public Animal(String species, int age) {
        this.species = species;
        this.age = age;
    }
}

Animal animal = new Animal("Lion", 5);
try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("animal.ser"))) {
    oos.writeObject(animal);
} catch (IOException e) {
    e.printStackTrace();
}
```

  - This option is incorrect. The `Animal` class does not implement `Serializable`, so it cannot be serialized using `ObjectOutputStream`.

- **D)**

```
class Animal implements Serializable {
    private static final long serialVersionUID = 1L;
    private String species;
    private int age;

    public Animal(String species, int age) {
        this.species = species;
        this.age = age;
    }
}

Animal animal = new Animal("Lion", 5);
try (BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream("animal.ser"))) {
    bos.write(animal);
} catch (IOException e) {
    e.printStackTrace();
}
```

  - This option is incorrect. `BufferedOutputStream` cannot be used to write objects directly; `ObjectOutputStream` should be used for serialization.

Chapter THIRTEEN

The Java Platform Module System

Chapter Content

- Introduction
- Types of Modules
  - Named Modules
  - Automatic Modules
  - Unnamed Modules
- Creating a Module
  - Directory Structure
  - The Class Files
  - The `module-info.java` File
- Module Declaration
  - Exporting a Package
  - Access Control with Modules
  - Requiring a Module
  - Opening a Package
- Built-in Modules
  - Core Java Modules
  - JDK Modules
- Using the Command Line
  - Compiling Classes with `javac`
  - Running Classes with `java`
  - Packaging with `jar`
- Multiple Modules
  - Designing a Multi-Module Application
  - Inter-module Communication
  - Resolving Conflicts Between Modules
- Creating a Service
- Getting Module Details
  - Describing a Module
  - Listing Available Modules
  - Module Resolution
  - Using the `jar` Command
- Analyzing Dependencies With `jdeps`
- Using Module Files with `jmod`
  - The JMOD File Format
  - Operation Modes
  - Best Practices and Limitations
- Creating Runtime Images with `jlink`
  - Syntax and Options of `jlink`
  - Plugins
  - Optimizing Runtime Images
- Migrating an Application
- Key Points
- Practice Questions

Introduction

One of the most significant changes introduced in Java 9 was the Java Platform Module System (JPMS). But what exactly is JPMS, and why should we care about it?

Let's start by understanding what a module is in this context.

A module in Java is like a section in a well-organized library. Each module has a clear label (its name) and contains specific books (Java packages). However, you can't access any book without a library card (a dependency declaration) for that specific section.

In this example:

```
module com.myapp.core {
    requires java.base;
    exports com.myapp.core.api;
}
```

We're declaring a module named `com.myapp.core`. It requires the `java.base` module (like having a library card for the basic Java section) and exports the `com.myapp.core.api` package (making some of its books available to other modules).

While packages group related classes, modules take this concept further by grouping related packages and explicitly declaring their dependencies and exposed APIs.

Consider the benefits of using JPMS:

- **Improved encapsulation:** Modules allow you to hide implementation details more effectively than packages alone, reducing the risk of unintended usage of internal APIs.
- **Clearer dependencies:** Modules explicitly state dependencies, making the system's structure more apparent and helping prevent *JAR hell* of conflicting dependencies.
- **Improved performance:** The JVM can optimize startup time and memory usage because it knows exactly what code is needed.
- **Better security:** By controlling access, you can reduce the attack surface of your application.

JPMS includes:

- The `module-info.java` file: Defines your module, its dependencies, and what it exports.
- New keywords: `module`, `requires`, `exports`, `opens`, `uses`, and `provides`.
- Tools for working with modules: Like `jlink`, for creating custom runtime images.

Here's a more complex sample `module-info.java` file:

```
module com.myapp.core {
    requires java.base;
    requires java.sql;

    exports com.myapp.core.api;
    exports com.myapp.core.util to com.myapp.plugin;

    opens com.myapp.core.model;

    uses com.myapp.core.spi.Plugin;
    provides com.myapp.core.spi.Logger
        with com.myapp.core.logging.FileLogger;
}
```

This module declaration shows: - Multiple required modules. - Exporting one package to all modules and another only to a specific module. - Opening a package for reflection. - Using a service interface and providing an implementation.

While you can still use the classpath, you'd miss out on the benefits of JPMS. The classpath is like a big, unorganized pile of books, while the module path is a well-organized library with controlled access and clear dependencies.

Even in small projects, modules can improve encapsulation and maintainability. Consider this small example:

```
// In module com.myapp.core
module com.myapp.core {
    exports com.myapp.core.api;
}

package com.myapp.core.api;
public interface UserService {
    User getUser(String id);
}

package com.myapp.core.internal;
class UserServiceImpl implements UserService {
    public User getUser(String id) {
        // Implementation
    }
}

// In module com.myapp.web
module com.myapp.web {
    requires com.myapp.core;
}

package com.myapp.web;
import com.myapp.core.api.UserService;
// import com.myapp.core.internal.UserServiceImpl; // This would cause a compile-time error

public class UserController {
    private UserService userService;
    // ...
}
```

In this example, the `web` module can only access the `api` package of the `core` module, not its internal implementation.

Modules provide tools to enforce and express the architecture of your system at the language and JVM level.

Consider this: Would you rather have a big box of unsorted LEGO bricks or neatly organized sets with clear instructions? Both approaches can build amazing things, but one makes the process much smoother and less error-prone.

Types of Modules

Now that we've got a grasp on what modules are and why they're useful, let's dive into the different types of modules in JPMS. Just like how not all books in a library are created equal, not all modules are the same either. JPMS introduces three types of modules:

- Named modules
- Automatic modules
- Unnamed modules

Named Modules

Named modules are like the properly cataloged books in our library, with a clear title, author information, and a spot on the shelf. In Java terms, a named module is defined by a `module-info.java` file at the root of your module.

Here's an example of the content of this file:

```
module com.myapp.core {
    requires java.base;
    exports com.myapp.core.api;
}
```

This `module-info.java` file is the ID card of your module. It gives your module a name (`com.myapp.core` in this case), lists its dependencies (`requires java.base`), and declares what parts of itself it's willing to share with other modules (`exports com.myapp.core.api`).

Named modules are the most powerful and flexible type of module. They give you full control over your module's dependencies and what it exposes to the outside world. If you're starting a new project or refactoring an existing one to use JPMS, named modules are what you'll be working with most of the time.

Automatic Modules

But what about all those third-party libraries that haven't been modularized? This is where automatic modules come in. They're like the books in our library that don't have a proper catalog entry yet, but we still want to be able to check them out.

When you put a non-modular JAR file on the module path, the Java runtime automatically treats it as a module. This module is called an automatic module.

In this example:

```
module com.myapp.core {
    requires java.base;
    requires commons.lang; // This is an automatic module
    exports com.myapp.core.api;
}
```

`commons.lang` is an automatic module. We can require it just like we would a named module, even though it doesn't have a `module-info.java` file.

But how does Java determine the name of an automatic module? Well, the process goes something like this:

1. First, it looks for the `Automatic-Module-Name` entry in the JAR's `MANIFEST.MF` file. If it's there, that's the module name.
2. If that's not present, it derives the name from the JAR filename. It removes the file extension and version number, and replaces non-alphanumeric characters with dots.

For example: - `commons-lang3-3.14.jar` becomes the automatic module `commons.lang3` - `guava-33.2.1-jre.jar` becomes `guava`

You can see this in action using the `jar` command:

```
$ jar --describe-module --file=guava-28.0-jre.jar
No module descriptor found. Derived automatic module.

Automatic module name: guava
...
```

Unnamed Modules

Last but not least, we have unnamed modules. These are like a miscellaneous box in the library where all loose papers and bookmarks that don't fit anywhere else are stored.

When you run your application on the classpath (not the module path), all the code that is not part of a named module or automatic module ends up in one big unnamed module. This unnamed module reads all other modules, which means it can access all packages exported by all other modules.

In this command:

```
java -cp app.jar:lib/* com.myapp.Main
```

`app.jar` and everything in the `lib` directory will be part of the unnamed module.

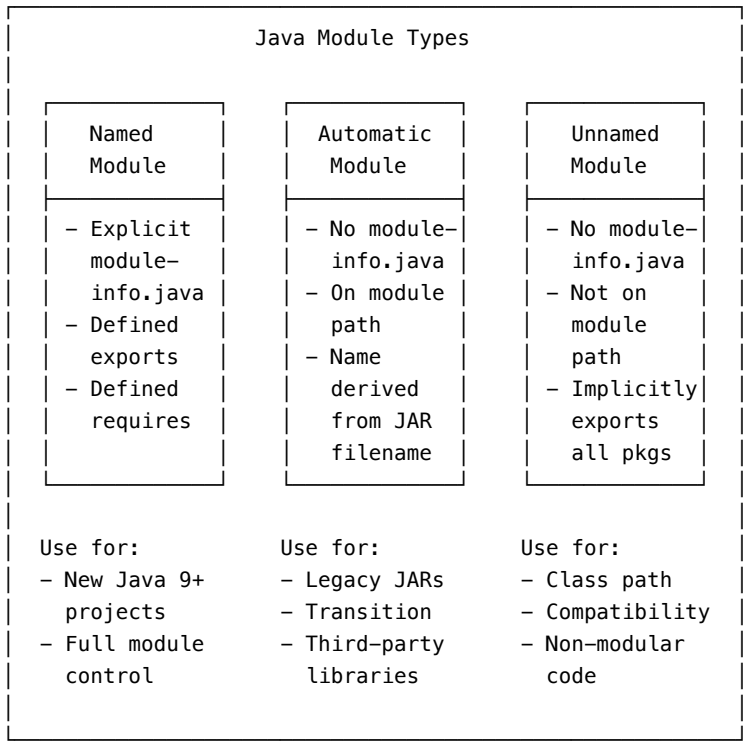
The unnamed module is important for backward compatibility, allowing existing Java code to run without modification on Java 9 and later versions. However, code in the unnamed module doesn't get the benefits of strong encapsulation that named modules provide.

Here's a quick comparison:

Module Type	Has module-info.java	On Module Path	On Classpath
Named	Yes	Yes	No
Automatic	No	Yes	No
Unnamed	No	No	Yes

Understanding these different types of modules is key to working effectively with JPMS. Named modules give you the most control and benefits, automatic modules help you integrate non-modular libraries, and unnamed modules ensure your existing code keeps running.

Here's a diagram that summarizes the types of modules:



```
Key Points:
- Named modules offer full control over exports and requires
- Automatic modules bridge between modular and non-modular code
- Unnamed modules provide backwards compatibility
```

Creating a Module

Now that we've explored the types of modules, let's create one ourselves. Creating a module is like setting up a new section in your library. We need to decide on its structure, what books (classes) it will contain, and what rules (`module-info.java`) will govern its use.

Directory Structure

The directory structure for a module is straightforward, but it's important to get it right. Here's a typical layout:

```
mymodule/
├── src/
│   ├── module-info.java
│   ├── com/
│   │   └── mycompany/
│   │       └── mymodule/
│   │           ├── MyClass.java
│   │           └── AnotherClass.java
│   └── resources/
│       └── config.properties
```

Let's break it down:

- The top-level directory (`mymodule/`) is typically named after your module.
- Inside, we have a `src/` directory. This is where all our source files live.
- The `module-info.java` file sits directly under `src/`. This is important, it defines our module.
- Our actual Java package structure (`com.mycompany.mymodule`) is represented by nested directories under `src/`.
- Resources (non-Java files) can be included in a separate directory

This structure might look familiar, it's very similar to how we organized non-modular Java projects. The key difference is the presence of the `module-info.java` file.

The Class Files

Now, let's look at what goes inside our Java files. Here's an example of what `MyClass.java` might look like:

```java
package com.mycompany.mymodule;

public class MyClass {
    public void doSomething() {
        System.out.println("MyClass is doing something!");
    }
}
```

This is just a regular Java class. However, the `package` declaration at the top is important, as it determines where this class fits in our module's structure.

Here's `AnotherClass.java`:

```java
package com.mycompany.mymodule;

public class AnotherClass {
```

```
    private MyClass myClass = new MyClass();

    public void doSomethingElse() {
        System.out.println("AnotherClass is doing something else!");
        myClass.doSomething();
    }
}
```

Again, this is a standard Java class. Notice how it can use `MyClass` without any special import because they're in the same package.

The `module-info.java` File

This file is what turns our collection of packages into a proper module. It's like the card catalog for our library section, defining what's available and what's needed.

Here's what a basic `module-info.java` might look like:

```
module com.mycompany.mymodule {
    exports com.mycompany.mymodule;
    requires java.base;
}
```

Let's break this down:

- `module com.mycompany.mymodule`: This declares our module name. By convention, this often matches our root package name.
- `exports com.mycompany.mymodule`: This line makes our package accessible to other modules. Without this, our classes would be hidden from the outside world.
- `requires java.base`: This declares a dependency on the base Java module. Actually, this line is optional, all modules implicitly require `java.base`.

But we can get more sophisticated. Let's say we want to use a logging framework and provide a service:

```
module com.mycompany.mymodule {
    exports com.mycompany.mymodule;
    requires java.base;
    requires org.apache.logging.log4j;

    provides com.mycompany.service.MyService
        with com.mycompany.mymodule.MyServiceImpl;

    uses com.mycompany.service.AnotherService;
}
```

Here, we're requiring the Log4j module, providing an implementation of `MyService`, and declaring that we'll be using `AnotherService` (which will be provided by some other module).

Isn't this `module-info.java` file a bit like the nutrition label on a food package? It tells you what's inside (exports), what it needs (requires), what it can do for you (provides), and what it expects to use (uses).

One thing to watch out for: if you're using an IDE, make sure it's set up to work with Java modules. Some IDEs might create a `module-info.java` file automatically when you create a new module, while others might require you to create it manually.

For example, assuming there's a `Main` class like this:

```
package com.mycompany.mymodule;

public class Main {
```

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

        MyClass myClass = new MyClass();
        myClass.doSomething();

        AnotherClass anotherClass = new AnotherClass();
        anotherClass.doSomethingElse();
    }
}
```

Here's how you might compile and run this module from the command line:

```
javac -d mods/com.mycompany.mymodule
    src/module-info.java
    src/com/mycompany/mymodule/*.java

java --module-path mods -m com.mycompany.mymodule/com.mycompany.mymodule.Main
```

The first command compiles our module, and the second runs it. Notice how we specify the module path (`--module-path mods`) and the main class (`-m com.mycompany.mymodule/com.mycompany.mymodule.Main`). Also, for both `javac` and `java` commands, you can use the shorter `-p` option instead of `--module-path`.

Module Declaration

Now that we've set up our module's structure, let's review the module declaration itself in the `module-info.java` file.

Exporting a Package

The `exports` keyword is used to make our module's packages accessible to other modules. Here's an example:

```
module com.mycompany.mymodule {
    exports com.mycompany.mymodule.api;
}
```

In this example, we're making the `com.mycompany.mymodule.api` package available for other modules to use. Any public types in this package can now be accessed by other modules that require it.

But what if we want to be more selective? Java modules allow for that too:

```
module com.mycompany.mymodule {
    exports com.mycompany.mymodule.api to com.mycompany.anothermodule, com.mycompany.yetanothermodule;
}
```

This declaration exports the package, but only to the specified modules. It's a way to control access to your module's internals.

Access Control with Modules

Modules add an extra layer of access control on top of Java's existing `public`, `protected`, package-private, and `private` modifiers. Here's how it works:

1. Public types in exported packages are accessible to other modules.
2. Public types in non-exported packages are only accessible within the module.
3. Protected members in exported packages are accessible in subclasses within other modules.
4. All other access rules (`protected`, package-private, `private`) still apply as usual.

Let's see this in action:

```java
// In module com.mycompany.mymodule
module com.mycompany.mymodule {
    exports com.mycompany.mymodule.api;
}

// In package com.mycompany.mymodule.api
public class PublicAPI {
    public void doSomething() { ... }
}

// In package com.mycompany.mymodule.internal
public class InternalClass {
    public void doSomethingElse() { ... }
}

// In another module
import com.mycompany.mymodule.api.PublicAPI; // This works
import com.mycompany.mymodule.internal.InternalClass; // This fails!
```

Even though `InternalClass` is public, it can't be accessed from outside the module because its package is not exported. It's like having a public reading room that's only accessible to staff members.

Requiring a Module

The `requires` keyword is how we declare dependencies on other modules. Consider this example:

```java
module com.mycompany.mymodule {
    requires java.sql;
}
```

This tells the Java runtime that our module depends on the `java.sql` module.

But what if we're building on top of another module and want to expose its functionality through our module? That's where `requires transitive` comes in:

```java
module com.mycompany.mymodule {
    requires transitive java.sql;
}
```

Now, any module that requires our module will automatically require `java.sql` too. It's like saying "if you're checking out books from our section, you'll also get a library card for the SQL section."

This is particularly useful when you're creating an API that builds on another module. Your users don't need to know about the underlying dependencies, they just require your module, and everything else comes along.

Opening a Package

Sometimes, we need to allow reflective access to a package at runtime, even if it's not exported. This is where the `opens` keyword comes in handy. Consider this example:

```java
module com.mycompany.mymodule {
    opens com.mycompany.mymodule.internal;
}
```

This allows reflective access to all types of the package at runtime, but doesn't allow compile-time access from other modules.

You can also open a package to specific modules:

```
module com.mycompany.mymodule {
    opens com.mycompany.mymodule.internal to com.mycompany.testmodule;
}
```

This is particularly useful for testing frameworks or dependency injection libraries that need to access your module's internals.

If you need to open all packages in your module for reflection, you can use the `open` keyword on the module declaration itself:

```
open module com.mycompany.mymodule {
    // module declarations
}
```

Isn't this module system a bit like setting up security clearances in a classified library? You have public sections (exported packages), restricted sections (non-exported packages), special access privileges (opens), and even transitive security clearances (requires transitive). It gives you fine-grained control over who can access what in your codebase.

Here's a more complex example putting it all together:

```
module com.mycompany.mymodule {
    exports com.mycompany.mymodule.api;
    exports com.mycompany.mymodule.util to com.mycompany.partnermodule;

    requires java.base; // This is implicit
    requires transitive com.mycompany.commonmodule;
    requires org.apache.logging.log4j;

    opens com.mycompany.mymodule.internal to org.junit.jupiter.api;
}
```

This module exports one package globally and another to a specific module, requires several modules (one transitively), and opens a package for testing.

Built-in Modules

Now that we've explored creating our own modules and services, let's look at the modules that come built into the Java platform. These built-in modules provide essential services and resources for everything else to build upon.

Core Java Modules

Modules that start with `java` are the core modules of the Java SE Platform. These modules contain the fundamental APIs that most Java applications rely on.

Here are some of the most commonly used `java` modules:

1. `java.base`: This is the foundational module of the Java SE Platform. It's automatically required by all other modules, just like how every section of a library relies on basic organizational principles.

   ```
   // You don't need to explicitly require java.base
   module com.mycompany.app {
       // java.base is implicitly required
   }
   ```

2. `java.sql`: Provides the API for accessing and processing data stored in a data source (usually a relational database) using the Java programming language.

```
module com.mycompany.app {
    requires java.sql;
}
```

3. `java.xml`: Contains the APIs for processing XML.

```
module com.mycompany.app {
    requires java.xml;
}
```

4. `java.desktop`: Defines the APIs for creating rich desktop applications, including AWT and Swing.

```
module com.mycompany.app {
    requires java.desktop;
}
```

5. `java.logging`: Provides the classes and interfaces of the Java Logging API.

```
module com.mycompany.app {
    requires java.logging;
}
```

These `java` modules provide the core functionality that most Java applications rely on. They're stable, well-documented, and form the backbone of the Java ecosystem.

JDK Modules

Modules that start with `jdk` are also part of the Java Development Kit, but they're not considered part of the core Java SE Platform specification. They provide additional tools and APIs that are useful for certain types of systems but aren't necessary for every application.

Here are some examples of `jdk` modules:

1. `jdk.httpserver`: Provides a simple HTTP server API.

```
module com.mycompany.app {
    requires jdk.httpserver;
}
```

2. `jdk.jshell`: Contains the JShell API, which allows you to create an interactive Java shell.

```
module com.mycompany.app {
    requires jdk.jshell;
}
```

3. `jdk.security.auth`: Provides implementations of the javax.security.auth.* interfaces.

```
module com.mycompany.app {
    requires jdk.security.auth;
}
```

It's important to note that while `java` modules are guaranteed to be available in all Java SE implementations, `jdk` modules might not be available. They're part of the JDK but not part of the Java SE specification. This means that if you're using a `jdk` module, your code might not be portable across all Java SE implementations.

Here's an example of how you might use both types of modules:

```
module com.mycompany.app {
    requires java.base;  // This is implicit
    requires java.sql;   // For database operations
    requires java.logging;  // For logging
    requires jdk.httpserver;  // To create a simple HTTP server
```

```
    exports com.mycompany.app.api;
}
```

In this module declaration, we're using both `java` and `jdk` modules. We're relying on core Java functionality for database operations and logging, but we're also using the JDK's simple HTTP server for some additional functionality.

Using the Command Line

While IDEs are great for productivity, understanding how to use `javac` and `java` commands is important. It's like knowing how to cook a meal from scratch instead of just reheating pre-made dishes.

There are several good reasons to learn how to compile and run Java code from the command line:

- Understanding what's happening under the hood.
- Troubleshooting build issues.
- Writing build scripts or setting up CI/CD pipelines.
- Working in environments where IDEs aren't available.

Think of it as learning to change a tire. You might not need to do it often, but when you do, you'll be glad you know how.

Compiling Classes with `javac`

Let's start with the basics. Here's how you compile a simple Java file:

```
javac MyClass.java
```

This compiles `MyClass.java` in the default package. But what about when you have packages?

```
javac com/mycompany/myapp/MyClass.java
```

This compiles `MyClass.java` in the `com.mycompany.myapp` package.

Typing out every file name can get tedious. Thankfully, you can use wildcards:

```
javac com/mycompany/myapp/*.java
```

This compiles all `.java` files in the `com/mycompany/myapp` directory.

Often, we want to keep our source files separate from our compiled classes. The `-d` option lets us specify an output directory:

```
javac -d bin com/mycompany/myapp/*.java
```

This compiles all `.java` files and puts the resulting `.class` files in the `bin` directory, maintaining the package structure.

When our code depends on external libraries, we need to tell the compiler where to find them. That's where the classpath option comes in:

```
javac -cp lib/dependency.jar com/mycompany/myapp/*.java
```

This tells the compiler to look for classes in `dependency.jar` while compiling our code. You can specify multiple JAR files or directories by separating them with a colon (`:`) on Unix-like systems or a semicolon (`;`) on Windows.

Speaking of JAR files, here's how you compile against multiple JARs:

```
javac -cp lib/dependency1.jar:lib/dependency2.jar com/mycompany/myapp/*.java
```

This compiles our code using classes from both `dependency1.jar` and `dependency2.jar`.

When working with modules, we need to specify the module path:

```
javac --module-path mods -d out src/module-info.java src/com/mycompany/myapp/*.java
```

This compiles our module, looking for dependencies in the `mods` directory and outputting to the `out` directory.

Running Classes with `java`

Once you've compiled a class, you can run it with:

```
java com.mycompany.myapp.MyClass
```

This runs `MyClass` in the `com.mycompany.myapp` package. Note that we don't include the `.class` extension.

Just like with compilation, we might need to specify a classpath when running our code:

```
java -cp bin:lib/dependency.jar com.mycompany.myapp.MyClass
```

This runs `MyClass`, looking for classes in both the `bin` directory and `dependency.jar`.

To run a modular application, we use the `--module-path` and `-m` options:

```
java --module-path out:mods -m com.mycompany.myapp/com.mycompany.myapp.MyClass
```

This runs `MyClass` from the `com.mycompany.myapp` module, looking for modules in the `out` and `mods` directories.

Packaging with `jar`

Often, you'll want to package your application into a JAR file. The `jar` command helps you do this:

```
jar -cvf myapp.jar -C bin .
```

Let's break this down: - `-c` or `--create`: Create a new archive - `-v` or `--verbose`: Generate verbose output - `-f` or `--file`: Specify the archive file name - `-C bin`: Change to the `bin` directory before adding files - `.`: Add all files in the current directory (which is now `bin`)

With a modular application, you can package your module into a modular JAR:

```
jar --create --file mods/com.mycompany.myapp.jar --main-class com.mycompany.myapp.MyClass -C out .
```

This creates a modular JAR file, optionally specifying `MyClass` as the main class.

Here's a more complex example that ties it all together:

```
# Compile the module
javac --module-path mods -d out \
    src/module-info.java \
    src/com/mycompany/myapp/*.java

# Package the module
jar --create --file mods/com.mycompany.myapp.jar \
    --main-class com.mycompany.myapp.MyClass \
    -C out .

# Run the module
java --module-path mods \
    -m com.mycompany.myapp/com.mycompany.myapp.MyClass
```

This sequence compiles the module, packages it into a JAR, and then runs it.

By convention, we store the compiled modules in a `mods` directory. When we use `--module-path mods` in our Java commands, we're telling Java to look for modules in this `mods` directory.

Multiple Modules

Up until now, we've been working with a single module, which is like organizing a single shelf in our library. But real-world applications often require multiple modules working together, more akin to organizing an entire library with multiple sections.

Designing a Multi-Module Application

Designing a multi-module application is like planning the layout of a large library. You need to think about how different sections (modules) will interact, what resources they'll share, and how to organize them for easy navigation and maintenance.

Here's a simple example of a multi-module application structure:

```
myapp/
├── core/
│   └── src/
│       ├── main/
│       │   └── java/
│       │       ├── module-info.java
│       │       └── com/mycompany/core/
│       └── test/
├── api/
│   └── src/
│       ├── main/
│       │   └── java/
│       │       ├── module-info.java
│       │       └── com/mycompany/api/
│       └── test/
└── app/
    └── src/
        ├── main/
        │   └── java/
        │       ├── module-info.java
        │       └── com/mycompany/app/
        └── test/
```

In this structure: - `core` contains the core functionality and domain logic - `api` defines the public interfaces for your application - `app` is the main application that ties everything together

When working with multiple modules, it's important to understand the dependencies between them.

Let's look at how we might define the dependencies for our example:

```
// core/src/main/java/module-info.java
module com.mycompany.core {
    exports com.mycompany.core;
}

// api/src/main/java/module-info.java
module com.mycompany.api {
    requires com.mycompany.core;
    exports com.mycompany.api;
}

// app/src/main/java/module-info.java
module com.mycompany.app {
    requires com.mycompany.core;
```

```
    requires com.mycompany.api;
}
```

In this setup, both `api` and `app` depend on `core`, and `app` also depends on `api`. This creates a hierarchy of dependencies that impacts how you develop and maintain your application:

- Changes in `core` can affect both `api` and `app`.
- Changes in `api` can affect `app`, but not `core`.
- Changes in `app` don't directly affect the other modules.

However, this dependency structure helps enforce a clean architecture, preventing lower-level modules from depending on higher-level ones.

When organizing code across modules, think about separation of concerns and information hiding. Each module should have a clear, focused purpose, and should only expose what's necessary for other modules to use.

Here's an example of how you might organize some classes:

```
// In core module
public class User { ... }
public class UserService { ... }

// In api module
public interface UserAPI { ... }

// In app module
public class UserController { ... }
```

The `core` module defines the fundamental domain objects and services. The `api` module defines the public interfaces that other parts of the application (or external systems) will use. The `app` module contains the application-specific logic that ties everything together.

This organization allows you to change the internals of the `core` module without affecting clients of the `api`, as long as the `api` remains stable.

That said, deciding on the right level of granularity for your modules can be challenging. Too few modules and you lose the benefits of modularization; too many and you introduce unnecessary complexity. Here are some best practices:

1. **Single Responsibility Principle:** Each module should have one, and only one, reason to change.
2. **Encapsulation:** Modules should hide their internals and expose only what's necessary.
3. **Stable Dependencies:** Modules should depend on modules that are more stable than they are.
4. **Reusability:** If a set of functionality might be useful in other contexts, consider making it a separate module.
5. **Size:** While there's no hard and fast rule, modules that are too large become unwieldy, while modules that are too small can lead to *dependency hell*. Aim for modules that can be reasonably understood and maintained by a small team.

Here's an example of refactoring our earlier structure, using another approach to improve granularity:

```
myapp/
├── core/
│   ├── domain/
│   └── services/
├── api/
│   ├── internal/
│   └── public/
```

```
├── infrastructure/
│   ├── persistence/
│   └── messaging/
└── app/
    ├── web/
    └── cli/
```

In this refactored structure: - We've split `core` into `domain` and `services` to separate entities from business logic. - `api` is divided into `internal` (for use within the application) and `public` (for external consumers). - We've added an `infrastructure` module to handle cross-cutting concerns. - `app` is split into `web` and `cli` for different user interfaces.

This granularity allows for more focused modules, each with a clear responsibility, while still maintaining a manageable overall structure.

Inter-module Communication

When working with multiple modules, communication between them becomes important. In Java, inter-module communication typically happens through well-defined APIs.

Here's how you might set this up:

```
// In api module
module com.mycompany.api {
    exports com.mycompany.api;
}

public interface UserService {
    User getUser(String id);
    void updateUser(User user);
}

// In core module
module com.mycompany.core {
    requires com.mycompany.api;
    provides com.mycompany.api.UserService
        with com.mycompany.core.UserServiceImpl;
}

public class UserServiceImpl implements UserService {
    public User getUser(String id) { ... }
    public void updateUser(User user) { ... }
}

// In app module
module com.mycompany.app {
    requires com.mycompany.api;
    uses com.mycompany.api.UserService;
}

public class UserController {
    @Inject
    private UserService userService;

    public void handleUserUpdate(String id, UserUpdateRequest request) {
        User user = userService.getUser(id);
```

```
        // Update user based on request
        userService.updateUser(user);
    }
}
```

In this setup: - The `api` module defines the `UserService` interface. - The `core` module provides an implementation of `UserService`. - The `app` module uses the `UserService` without knowing about its implementation.

This approach allows modules to communicate through well-defined interfaces, promoting loose coupling and making it easier to change implementations without affecting other modules.

Resolving Conflicts Between Modules

As your application grows, you might encounter conflicts between modules. Here are some common conflicts and how to resolve them:

1. **Version Conflicts:** When two modules require different versions of the same dependency. **Solution:** Use the `requires` directive with a specific version, or use build tools like Maven or Gradle to manage versions.

   ```
   module com.mycompany.moduleA {
       requires com.fasterxml.jackson.databind;
   }

   module com.mycompany.moduleB {
       requires com.fasterxml.jackson.databind@2.11.0;
   }
   ```

2. **Split Packages:** When classes in the same package are spread across multiple modules. **Solution:** Refactor your code to ensure each package is contained within a single module.

3. **Naming Conflicts:** When two modules export the same package name. **Solution:** Rename one of the packages to ensure uniqueness across your application.

4. **Cyclic Dependencies:** When modules depend on each other in a circular manner. **Solution:** Introduce a new module that both can depend on, or use the Service Provider Interface (SPI) pattern.

   ```
   // Before (cyclic dependency)
   module com.mycompany.moduleA {
       requires com.mycompany.moduleB;
   }
   module com.mycompany.moduleB {
       requires com.mycompany.moduleA;
   }

   // After (using SPI)
   module com.mycompany.api {
       exports com.mycompany.api;
   }
   module com.mycompany.moduleA {
       requires com.mycompany.api;
       provides com.mycompany.api.ServiceA with com.mycompany.moduleA.ServiceAImpl;
   }
   module com.mycompany.moduleB {
       requires com.mycompany.api;
       uses com.mycompany.api.ServiceA;
   }
   ```

To better understand the solution, let's review services in more detail.

Creating a Service

Let's dive into one of the most powerful features of the Java Module System: services. Services allow us to create flexible, extensible applications by decoupling interfaces from their implementations.

In the context of the Java Module System, a service is a well-defined set of programming interfaces and classes that provide access to some specific application functionality or feature. It's like a specialized department in our library that provides a specific service, say, book restoration.

The service model consists of three main components:

1. **The Service Provider Interface (SPI):** This is the contract that defines what the service does.
2. **The Service Provider:** This is the implementation of the SPI.
3. **The Service Consumer:** This is the code that uses the service.

This separation allows for loose coupling between modules. The consumer doesn't need to know about the specific implementation of the service, just the interface it uses.

Let's start by declaring our Service Provider Interface. We'll use the `UserService` from the previous section as our sample service:

```
// In the api module
module com.mycompany.api {
    exports com.mycompany.api;
}

package com.mycompany.api;

public interface UserService {
    User getUser(String id);
    void updateUser(User user);
}
```

This `UserService` interface defines the contract for the user management service. Any module that implements this interface can provide user management functionality.

Now that we have our Service Provider Interface, we need a way to discover and load implementations of this service. This is where a Service Locator comes in. In Java 9 and above, we can use the `ServiceLoader` class for this purpose.

Here's how we might create a `UserServiceLocator`:

```
// In the app module
module com.mycompany.app {
    requires com.mycompany.api;
    uses com.mycompany.api.UserService;
}

package com.mycompany.app;

import com.mycompany.api.UserService;
import java.util.ServiceLoader;

public class UserServiceLocator {
    private static final ServiceLoader<UserService> loader = ServiceLoader.load(UserService.class);

    public static UserService getUserService() {
        return loader.findFirst().orElseThrow(() -> new IllegalStateException("No UserService implementation found
```

```
    }
}
```

Let's break this down:

- We use the `ServiceLoader.load()` method to create a `ServiceLoader` for our `UserService` interface.
- The `getUserService()` method uses `findFirst()` to get the first available implementation of `UserService`.
- If no implementation is found, we throw an `IllegalStateException`.

Note the `uses` directive in the module declaration. This tells the module system that this module will be using the `UserService` service.

This approach provides several benefits:

- **Loose coupling:** The `app` module doesn't need to know about the specific implementation of `UserService`.
- **Flexibility:** We can easily swap out different implementations of `UserService` without changing the consuming code.
- **Extensibility:** Third-party modules can provide their own implementations of `UserService`, extending the functionality of our application.

Here's how we might use this in our `UserController`:

```
package com.mycompany.app;

public class UserController {
    private final UserService userService;

    public UserController() {
        this.userService = UserServiceLocator.getUserService();
    }

    public void handleUserUpdate(String id, UserUpdateRequest request) {
        User user = userService.getUser(id);
        // Update user based on request
        userService.updateUser(user);
    }
}
```

In this setup, `UserController` doesn't need to know anything about how `UserService` is implemented or where it comes from. It just uses the service locator to get an instance and then uses it.

However, we haven't actually provided an implementation yet. Let's do that now.

First, we'll create an implementation of our `UserService`:

```
// In the core module
package com.mycompany.core;

import com.mycompany.api.User;
import com.mycompany.api.UserService;
import java.util.HashMap;
import java.util.Map;
import java.util.UUID;

public class UserServiceImpl implements UserService {
    private final Map<String, User> users = new HashMap<>();
```

```
    @Override
    public User getUser(String id) {
        return users.get(id);
    }

    @Override
    public void updateUser(User user) {
        if (user.getId() == null) {
            user.setId(UUID.randomUUID().toString());
        }
        users.put(user.getId(), user);
    }
}
```

Now, we need to tell the module system that this implementation provides the `UserService`. We do this in the `module-info.java` file of the core module:

```
module com.mycompany.core {
    requires com.mycompany.api;
    provides com.mycompany.api.UserService with com.mycompany.core.UserServiceImpl;
}
```

The `provides ... with` clause tells the module system that this module provides an implementation of `UserService` using the `UserServiceImpl` class.

Now, when the `ServiceLoader` in our `UserServiceLocator` looks for implementations of `UserService`, it will find and use this `UserServiceImpl`.

This separation of interface and implementation gives us incredible flexibility. We could easily swap out our `UserServiceImpl` for a different implementation without having to change any of the consuming code. Maybe one that uses a database instead of an in-memory map:

```
// In the core module
package com.mycompany.core;

import com.mycompany.api.User;
import com.mycompany.api.UserService;
import java.sql.*;
import java.util.UUID;

public class DatabaseUserServiceImpl implements UserService {
    private static final String DB_URL = "jdbc:sqlite:users.db";

    @Override
    public User getUser(String id) {
        String sql = "SELECT * FROM users WHERE id = ?";
        // ...
    }

    @Override
    public void updateUser(User user) {
        String sql = "INSERT OR REPLACE INTO users(id, username, email, active) VALUES(?,?,?,?)";
        // ...
    }
}
```

Now, to use this new implementation, we only need to change the `provides` clause in our `module-info.java`

file:

```
module com.mycompany.core {
    requires com.mycompany.api;
    requires java.sql;  // We need this for JDBC
    provides com.mycompany.api.UserService with com.mycompany.core.DatabaseUserServiceImpl;
}
```

That's it! We've now swapped out our in-memory implementation for a database-backed one. The beauty of this approach is that we didn't have to change any code in the `UserController` or any other consuming classes. They're still working with the `UserService` interface, unaware that the underlying implementation has changed.

To add another implementation of the `UserService` without replacing the existing one, we'll first modify the core module to include both implementations:

```
// In the core module
module com.mycompany.core {
    requires com.mycompany.api;
    requires java.sql;  // We need this for JDBC

    provides com.mycompany.api.UserService with
        com.mycompany.core.UserServiceImpl,
        com.mycompany.core.DatabaseUserServiceImpl;
}
```

Now, the module system knows that there are two providers for `UserService`.

Next, we can update the `UserServiceLocator` to return all available implementations:

```
// In the app module
module com.mycompany.app {
    requires com.mycompany.api;
    uses com.mycompany.api.UserService;
}
```

```
package com.mycompany.app;

import com.mycompany.api.UserService;
import java.util.ServiceLoader;
import java.util.List;
import java.util.stream.Collectors;

public class UserServiceLocator {
    private static final ServiceLoader<UserService> loader = ServiceLoader.load(UserService.class);

    public static List<UserService> getUserServices() {
        return loader.stream()
                     .map(ServiceLoader.Provider::get)
                     .collect(Collectors.toList());
    }
}
```

This method returns a list of all available `UserService` implementations.

Now, we can update the `UserController` to use, for example, all available services:

```
package com.mycompany.app;
```

```java
import com.mycompany.api.User;
import com.mycompany.api.UserService;

import java.util.List;

public class UserController {
    private final List<UserService> userServices;

    public UserController() {
        this.userServices = UserServiceLocator.getUserServices();
    }

    public void handleUserUpdate(String id, UserUpdateRequest request) {
        for (UserService userService : userServices) {
            User user = userService.getUser(id);
            // Update user based on request
            userService.updateUser(user);
        }
    }
}
```

In this setup, `UserController` will iterate through all available `UserService` implementations and call the `getUser` and `updateUser` methods on each.

This service-oriented approach allows us to build more modular, flexible applications. A well-designed application using the Java Module System's service feature can easily extend and modify its functionality over time.

Getting Module Details

As your modular application grows, you might need to inspect your modules to understand their structure, dependencies, and how they're being resolved. Java provides several command-line tools to help with this.

Describing a Module

Let's start with describing a module. The `java` command with the `--describe-module` (or `-d`) option allows us to see details about a specific module:

```
java --describe-module java.sql
```

This might output something like:

```
java.sql@18.0.3
exports java.sql
exports javax.sql
requires java.logging transitive
requires java.transaction.xa transitive
requires java.base mandated
requires java.xml transitive
uses java.sql.Driver
```

This tells us what packages the module exports, what other modules it requires, and what services it uses. It's a quick way to get an overview of a module's structure and dependencies.

You can also describe modules that aren't part of the Java runtime. For example, if you have a `com.mycompany.core` module in a JAR file:

```
java --module-path mods --describe-module com.mycompany.core
```

This might output:

```
com.mycompany.core@1.0
requires java.base mandated
requires com.mycompany.api
provides com.mycompany.api.UserService with com.mycompany.core.DatabaseUserServiceImpl
```

Listing Available Modules

Sometimes, you might want to see all the modules available to your application. You can do this with the `--list-modules` option:

```
java --list-modules
```

This will list all modules in the Java runtime. If you want to include your own modules, you can use:

```
java --module-path mods --list-modules
```

This will list both the Java runtime modules and any modules in the `mods` directory (by convention, the directory where you store the compiled modules).

Module Resolution

When you're dealing with complex module dependencies, it can be helpful to see how the module system resolves these dependencies. You can do this with the `--show-module-resolution` option:

```
java --show-module-resolution --module-path mods -m com.mycompany.app/com.mycompany.app.Main
```

This will show detailed information about how each module is resolved as the application starts up. It's particularly useful for debugging issues with module dependencies.

Using the `jar` Command

While the `java` command is great for describing modules at runtime, sometimes you'll want to inspect a module without running it. The `jar` command can help with this:

```
jar --describe-module --file mods/com.mycompany.core.jar
```

This might output something like:

```
com.mycompany.core jar:file:///.../mods/com.mycompany.core.jar/!module-info.class
requires java.base mandated
requires com.mycompany.api
provides com.mycompany.api.UserService with com.mycompany.core.DatabaseUserServiceImpl
```

This provides similar information to the `java --describe-module` command, but it works directly on the JAR file without needing to set up the module path.

Here's a more complex example. Let's say we have a multi-module application and want to understand how all the pieces fit together:

```
# List all modules
java --module-path mods --list-modules

# Describe each of our modules
java --module-path mods --describe-module com.mycompany.api
java --module-path mods --describe-module com.mycompany.core
java --module-path mods --describe-module com.mycompany.app

# Show module resolution for our main application
java --show-module-resolution --module-path mods -m com.mycompany.app/com.mycompany.app.Main
```

```
# Describe our core module JAR file
jar --describe-module --file mods/com.mycompany.core.jar
```

By running these commands, you can get a comprehensive view of your application's module structure, from the high-level list of all modules, through the details of each module, to the step-by-step resolution process when you run your application.

Analyzing Dependencies With `jdeps`

`jdeps` is a tool that provides powerful capabilities for analyzing and visualizing dependencies at both the module and class level. It allows you to examine the relationships between modules, packages, and classes, enabling you to make informed decisions about the structure and organization of your codebase.

To get started with `jdeps`, let's explore its basic syntax and common options. The general format for running `jdeps` is as follows:

```
jdeps [options] path
```

Here, `path` represents the location of the Java class files, JAR files, or directories you want to analyze. The `options` allow you to customize the behavior of jdeps according to your specific needs.

These are some of the most important general options: - `-dotoutput dir or --dot-output dir`: Specifies the destination directory for DOT file output. If this option is specified, then the `jdeps` command generates one .dot file for each analyzed archive named `archive-file-name.dot` that lists the dependencies, and also a summary file named `summary.dot` that lists the dependencies among the archive files. - `-s or -summary`: Prints a dependency summary only. - `-v or -verbose`: Prints all class-level dependencies. This is equivalent to `-verbose:class -filter:none` - `-verbose:package`: Prints package-level dependencies excluding, by default, dependences within the same package. - `-verbose:class`: Prints class-level dependencies excluding, by default, dependencies within the same archive. - `-apionly or --api-only`: Restricts the analysis to APIs, for example, dependences from the signature of public and protected members of public classes including field type, method parameter types, returned type, and checked exception types. - `-jdkinternals or --jdk-internals`: Finds class-level dependences in the JDK internal APIs. By default, this option analyzes all classes specified in the `--classpath` option and input files unless you specified the `-include` option. You can't use this option with the `-p`, `-e`, and `-s` options. - `-cp path, -classpath path, or --class-path path`: Specifies where to find class files. - `--module-path module-path`: Specifies the module path. - `--add-modules module-name[, module-name...]`: Adds modules to the root set for analysis. - `-q or -quiet`: Doesn't show missing dependencies from `-generate-module-info` output.

These are the module dependence analysis options: - `-m module-name or --module module-name`: Specifies the root module for analysis. - `--generate-module-info dir`: Generates `module-info.java` under the specified directory. The specified JAR files will be analyzed. This option cannot be used with –dot-output or `--class-path` options. Use the `--generate-open-module` option for open modules. - `--generate-open-module dir`: Generates `module-info.java` for the specified JAR files under the specified directory as open modules. This option cannot be used with the `--dot-output` or `--class-path` options. - `--check module-name [, module-name...]`: Analyzes the dependence of the specified modules. It prints the module descriptor, the resulting module dependences after analysis and the graph after transition reduction. It also identifies any unused qualified exports. - `--list-deps`: Lists the module dependences and also the package names of JDK internal APIs (if referenced). This option transitively analyzes libraries on class path and module path if referenced. Use `--no-recursive` option for non-transitive dependency analysis. - `--list-reduced-deps`: Same as `--list-deps` without listing the implied reads edges from the module graph. If module M1 reads M2, and M2 requires transitive on M3, then M1 reading M3 is implied and is not shown in the graph. - `--print-module-deps`: Same as –list-reduced-deps with printing a comma-separated list of module dependences. The output can be used by jlink –add-modules to create a custom image that contains those modules and their transitive dependences. - `--ignore-missing-deps`: Ignore missing dependences.

These are the options to filter dependences: - `-p pkg_name, -package pkg_name, or --package pkg_name`: Finds dependences matching the specified package name. You can specify this option multiple times for

different packages. The `-p` and `-e` options are mutually exclusive. - `-e regex, -regex regex, or --regex regex`: Finds dependences matching the specified pattern. The `-p` and `-e` options are mutually exclusive. - `--require module-name`: Finds dependences matching the given module name (may be given multiple times). The `--package`, `--regex`, and `--require` options are mutually exclusive. - `-f regex or -filter regex`: Filters dependences matching the given pattern. If give multiple times, the last one will be selected. - `-filter:package`: Filters dependences within the same package. This is the default. - `-filter:archive`: Filters dependences within the same archive. - `-filter:module`: Filters dependences within the same module. - `-filter:none`: No `-filter:package` and `-filter:archive` filtering. Filtering specified via the `-filter` option still applies. - `--missing-deps`: Finds missing dependences. This option cannot be used with `-p`, `-e` and `-s` options.

And these are the options to filter classes to be analyzed: - `-include regex`: Restricts analysis to the classes matching pattern. This option filters the list of classes to be analyzed. It can be used together with `-p` and `-e`, which apply the pattern to the dependencies. - `-P or -profile`: Shows the profile containing a package. - `-R or --recursive`: Recursively traverses all run-time dependences. The `-R` option implies `-filter:none`. If `-p`, `-e`, or `-f` options are specified, only the matching dependences are analyzed. - `--no-recursive`: Do not recursively traverse dependences. – `I or --inverse`: Analyzes the dependences per other given options and then finds all artifacts that directly and indirectly depend on the matching nodes. This is equivalent to the inverse of the compile-time view analysis and the print dependency summary. This option must be used with the `--require`, `--package`, or `--regex` options. - `--compile-time`: Analyzes the compile-time view of transitive dependencies, such as the compile-time view of the `-R` option. Analyzes the dependences per other specified options. If a dependency is found from a directory, a JAR file or a module, all classes in that containing archive are analyzed.

One of the primary use cases of `jdeps` is analyzing module dependencies. By running `jdeps` on a module, you can obtain a detailed report of the modules it depends on and the packages it uses from each module:

```
jdeps --module-path mods --add-modules com.example.myapp mymodule.jar
```

In this example, we specify the module path using the `--module-path` option, which points to the directory containing the module definitions. The `--add-modules` option is used to specify the main module of our application. Finally, we provide the path to the JAR file representing our module.

`jdeps` will analyze the dependencies and generate a report that looks something like this:

```
mymodule.jar -> java.base
   com.example.myapp                       -> java.io
   com.example.myapp                       -> java.lang
   com.example.myapp                       -> java.util
mymodule.jar -> java.desktop
   com.example.myapp                       -> java.awt
   com.example.myapp                       -> javax.swing
```

This report shows the dependencies of `mymodule.jar` on other modules, such as `java.base` and `java.desktop`. It also lists the specific packages within `mymodule.jar` that depend on packages from those modules.

In addition to module-level analysis, `jdeps` allows you to examine dependencies at the class level. By running `jdeps` on individual class files or directories containing class files, you can gain insights into the relationships between classes and packages.

Consider this example:

```
jdeps --verbose --class-path lib/* com/example/MyClass.class
```

Here, we use the `--class-path` option to specify the classpath containing the required dependencies. The `--verbose` flag provides more detailed output, showing the specific classes and members being used.

The class-level dependency report generated by `jdeps` will include information like this:

```
com.example.MyClass -> java.lang.Object
```

```
com.example.MyClass -> java.lang.String
com.example.MyClass -> java.util.ArrayList
com.example.MyClass -> java.util.List
com.example.MyClass -> com.example.HelperClass
```

This report indicates that `MyClass` depends on classes from the `java.lang` and `java.util` packages, as well as another class named `HelperClass` from the same package.

`jdeps` also provides the ability to generate comprehensive dependency reports in various formats. By using the `--dot-output` option, you can generate a DOT file that visualizes the dependencies as a graph. This graphical representation can be extremely helpful in understanding complex dependency structures and identifying potential issues.

```
jdeps --dot-output docs --module-path mods --add-modules com.example.myapp mymodule.jar
```

In this example, `jdeps` will generate a DOT file named after the module in the `docs` directory. You can then use tools like Graphviz to render the DOT file into a visual graph.

Another useful feature of `jdeps` is its ability to identify usage of internal APIs. The `--jdk-internals` flag helps you detect and analyze the use of internal JDK APIs within your code. This is important because relying on internal APIs can lead to compatibility issues and unexpected behavior when upgrading to newer Java versions.

```
jdeps --jdk-internals --class-path lib/* com/example/MyClass.class
```

If `MyClass` uses any internal JDK APIs, `jdeps` will report them in the output, allowing you to take necessary actions to refactor or remove the dependencies on internal APIs.

If you provide the path to a JAR file or directory, `jdeps` will recursively analyze all the classes within it and generate a comprehensive dependency report.

Here's an example:

```
jdeps --recursive lib/myapp.jar
```

The `--recursive` option ensures that `jdeps` traverses all the classes and nested directories within the specified JAR file or directory, providing a complete picture of the dependencies.

`jdeps` also offers a feature called recursive dependency analysis, which allows you to understand the transitive dependencies of your code. By analyzing not only the direct dependencies but also the dependencies of those dependencies, `jdeps` helps you identify potential issues and conflicts.

Consider this example:

```
jdeps --recursive --module-path mods --add-modules com.example.myapp mymodule.jar
```

With the `--recursive` flag, `jdeps` will traverse the entire dependency graph, starting from the specified module or JAR file. It will generate a report that includes all the transitive dependencies, giving you a comprehensive view of your project's dependency structure.

Using Module Files with `jmod`

`jmod` is a command-line tool that operates on a file format called JMOD. JMOD files are similar to JAR files in the way that they package Java classes, resources, and metadata. However, JMOD files are specifically designed to work with the module system and offer additional capabilities compared to traditional JAR files.

The JMOD File Format

The JMOD file format is optimized for the JPMS and serves as a container for modular content. It encapsulates not only the compiled Java classes and resources but also includes module descriptors, native libraries, and other module-specific information. JMOD files have the `.jmod` file extension and follow a specific directory structure to organize their contents.

JMOD files do not replace JAR files.

JAR (Java Archive) files are the most common and widely used format for packaging Java classes and resources. They are essentially zip files that contain compiled Java classes, metadata, and resources. Additionally, JAR files can be placed on the classpath for easy access by Java programs.

There are some key differences between JAR and JMOD files: - **Modularity:** JMOD files are primarily used for modular Java development, whereas JAR files are used for both modular and non-modular code.

- **Native code:** JMOD files can include native libraries and executables, which is not possible with JAR files.
- **Versioning:** JMOD files support module versioning through the `--module-version` option, allowing for better version management.
- **Optimization:** JMOD files are optimized for the module system and provide better performance and encapsulation compared to JAR files.
- **Usage:** JAR files are widely used for distributing libraries and applications, while JMOD files are mainly used for creating and packaging modules.

So, when should you use JMOD files instead of JAR files? Here are some guidelines: - If you are developing a modular Java application using the JPMS, JMOD files are the recommended format for packaging your modules.

- If your module requires native libraries or executables, JMOD files provide a convenient way to include them alongside your Java code.
- If you need to create a custom runtime image or a JRE (Java Runtime Environment) specific to your application, JMOD files are used as the input to the `jlink` tool for creating optimized runtime images.

However, if you are developing a non-modular Java application or a library that needs to be compatible with older versions of Java, JAR files are still the preferred choice.

One of the key advantages of JMOD files is their ability to include native libraries and executables. This is particularly useful for modules that have platform-specific dependencies or require native code integration. By packaging native libraries within the JMOD file, the module can be easily distributed and deployed across different platforms.

JMOD files also support versioning, allowing modules to specify their version information. This is important for managing dependencies and ensuring compatibility between various versions of modules. The module descriptor in the `module-info.class` file can include version-related annotations to provide version metadata.

Operation Modes

This is the basic syntax of the `jmod` command:

```
jmod (create|extract|list|describe|hash) [options] jmod-file
```

The main operation modes are: - `create`: Creates a new JMOD archive file. - `extract`: Extracts all the files from the JMOD archive file. - `list`: Prints the names of all the entries. - `describe`: Prints the module details. - `hash`: Determines leaf modules and records the hashes of the dependencies that directly and indirectly require them.

These are the most important options: - `--class-path path`: Specifies the location of application JAR files or a directory containing classes to copy into the resulting JMOD file. - `--cmds path`: Specifies the location of native commands to copy into the resulting JMOD file. - `--config path`: Specifies the location of user-editable configuration files to copy into the resulting JMOD file. - `--dir path`: Specifies the location where jmod puts extracted files from the specified JMOD archive. - `--dry-run`: Performs a dry run of hash mode. It identifies leaf modules and their required modules without recording any hash values. - `--hash-modules regex-pattern`: Determines the leaf modules and records the hashes of the dependencies directly and indirectly requiring them, based on the module graph of the modules matching the given regex-pattern.

The hashes are recorded in the JMOD archive file being created, or a JMOD archive or modular JAR on the module path specified by the `jmod hash` command. - `--help or -h`: Prints a usage message. - `--libs path`: Specifies location of native libraries to copy into the resulting JMOD file. - `--main-class class-name`: Specifies main class to record in the `module-info.class` file. - `--module-version ersion`: Specifies the module version to record in the `module-info.class` file. - `--module-path path or -p path`: Specifies the module path. This option is required if you also specify `--hash-modules`. - `--target-platform platform`: Specifies the target platform. - `--version`: Prints version information of the `jmod` command.

Here are some examples that demonstrate the basic usage of each operation mode:

1. **Create mode:** `jmod create \ --class-path classes \ --main-class com.example.Main \ --module-version 1.0 \ --module-path lib \ mymodule.jmod`

   This command creates a new JMOD archive file named `mymodule.jmod`. It includes classes from the `classes` directory, sets the main class to `com.example.Main`, specifies the module version as 1.0, and uses the `lib` directory as the module path.

2. **Extract mode:** `jmod extract --dir extracted_files mymodule.jmod`

   This command extracts all files from `mymodule.jmod` into a directory named `extracted_files`.

3. **List mode:** `jmod list mymodule.jmod`

   This command prints the names of all entries in `mymodule.jmod`.

4. **Describe mode:** `jmod describe mymodule.jmod`

   This command prints the module details of `mymodule.jmod`.

5. **Hash mode:** `jmod hash --module-path lib \ --hash-modules java.base \ mymodule.jmod`

   This command determines leaf modules and records the hashes of dependencies that directly and indirectly require them. It uses the `lib` directory as the module path and considers modules matching the pattern `java.base`.

Best Practices and Limitations

When working with JMOD files, there are some best practices to keep in mind: - Use descriptive and meaningful names for your JMOD files, following the naming conventions for modules.

- Include a `module-info.java` file in your module's source code to define the module's name, dependencies, and exported packages.
- Organize your module's classes, resources, and native libraries in the appropriate directories within the JMOD file.
- Use the `--module-version` option when creating JMOD files to specify the version of your module.
- Store JMOD files in a separate directory structure, separate from your source code and other project artifacts.
- Use the jmod tool to create, extract, and manipulate JMOD files as needed.
- When distributing your modular application, consider using `jlink` to create optimized runtime images that include only the necessary modules.

While JMOD files offer many benefits for modular Java development, there are some limitations and considerations to keep in mind:

- JMOD files are specific to the Java Platform Module System and are not backward compatible with older versions of Java.
- Not all Java libraries and frameworks are modularized or provide JMOD files. You may need to rely on traditional JAR files for dependencies that are not yet modularized.

- The tooling and build systems for modular Java development are still evolving, and there may be some learning curve and configuration required to fully utilize JMOD files in your project.
- JMOD files are not intended to be used as a distribution format for end-users. They are typically used as an intermediate format for creating runtime images or integrating with build tools.

Creating Runtime Images with `jlink`

Traditionally, Java applications have relied on the Java Runtime Environment (JRE) to execute. The JRE includes a wide range of modules and libraries, many of which may not be necessary for a particular application. This can lead to larger distribution sizes and potentially unnecessary dependencies.

With `jlink`, we can create custom runtime images that include only the modules required by our application. These custom runtime images are self-contained and can be distributed as standalone executables. They provide several benefits, such as reducing distribution size, improving startup time, and enhancing security by minimizing the attack surface.

Syntax and Options of `jlink`

To create a custom runtime image using `jlink`, we use the following basic syntax:

```
jlink [options] --module-path <modulepath> --add-modules <modules>
```

Let's break down the key components of the `jlink` command: - `[options]`: Additional options to configure the behavior of `jlink`, such as compression, debugging, and more. - `--module-path <modulepath>`: Specifies the module path where the required modules can be found. This includes the application modules and any dependencies. - `--add-modules <modules>`: Specifies the modules to be included in the runtime image. This can be a comma-separated list of module names or the keyword `ALL-MODULE-PATH` to include all modules found on the module path.

One of the primary goals of creating custom runtime images is to minimize the size and include only the necessary modules. `jlink` provides options to create a minimal runtime that includes only the essential modules required for our application to run.

Here are some of the most important options: - `--add-modules mod [, mod...]`: Adds the named modules, `mod`, to the default set of root modules. The default set of root modules is empty. - `--bind-services`: Link service provider modules and their dependencies. - `-c ={0|1|2} or --compress={0|1|2}`: Enable compression of resources. - `--disable-plugin pluginname`: Disables the specified plug-in. - `--endian {little|big}`: Specifies the byte order of the generated image. The default value is the format of your system's architecture. - `-h or --help`: Prints the help message. - `--ignore-signing-information`: Suppresses a fatal error when signed modular JARs are linked in the runtime image. The signature-related files of the signed modular JARs aren't copied to the runtime image. - `--launcher command=module or --launcher command=module/main`: Specifies the launcher command name for the module or the command name for the module and main class (the module and the main class names are separated by a slash character). - `--limit-modules mod [, mod...]`: Limits the universe of observable modules to those in the transitive closure of the named modules, mod, plus the main module, if any, plus any further modules specified in the `--add-modules` option. - `--list-plugins`: Lists available plug-ins, which you can access through command-line options. - `-p or --module-path modulepath`: Specifies the module path. If this option is not specified, then the default module path is `$JAVA_HOME/jmods`. This directory contains the `java.base` module and the other standard and JDK modules. If this option is specified but the `java.base` module cannot be resolved from it, then the jlink command appends `$JAVA_HOME/jmods` to the module path. - `--output path`: Specifies the location of the generated runtime image. - `--suggest-providers [name, ...]`: Suggest providers that implement the given service types from the module path. - `--version`: Prints version information.

To create a minimal runtime, we can use the following command:

```
jlink --module-path <modulepath> \
      --add-modules <modules> \
```

```
      --compress 2 \
      --strip-debug \
      --no-header-files \
      --no-man-pages \
      --output <path>
```

In this command, we use several options to optimize the runtime image: - `--compress 2`: Enables compression of the generated runtime image, reducing its size. - `--strip-debug`: Removes debug information from the runtime image, further reducing its size. - `--no-header-files`: Excludes header files from the runtime image. - `--no-man-pages`: Excludes manual pages from the runtime image.

By specifying only the necessary modules with `--add-modules` and using these optimization options, we can create a minimal runtime image that is tailored to our application's specific requirements.

`jlink` also allows us to explicitly include or exclude modules from the runtime image. This gives us fine-grained control over the modules that are packaged into the image.

To include specific modules, we can use the `--add-modules` option followed by a comma-separated list of module names. For example:

```
jlink --module-path <modulepath> \
      --add-modules module1,module2,module3 \
      --output <path>
```

This command will create a runtime image that includes only `module1`, `module2`, and `module3`, along with their transitive dependencies.

On the other hand, if we want to exclude certain modules from the runtime image, we can use the `--exclude-modules` option followed by a comma-separated list of module names. For example:

```
jlink --module-path <modulepath> \
      --add-modules ALL-MODULE-PATH \
      --exclude-modules module4,module5 \
      --output <path>
```

In this case, jlink will include all modules found on the module path except for `module4` and `module5`.

Plugins

Plugins are additional components that extend the functionality of the `jlink` tool. They allow developers to customize the creation of runtime images in various ways, such as optimizing the generated image, adding or removing resources, and configuring how the image is laid out.

If you execute:

```
jlink --list-plugins
```

You'll get the list of all available plugins. For example:

- `--add-options <options>`: Prepend the specified `<options>` string, which may include whitespace, before any other options when invoking the virtual machine in the resulting image.
- `--compress <compress>`: Compression to use in compressing resources. Accepted values are `zip-[0-9]`, where `zip-0` provides no compression, and `zip-9` provides the best compression. Default is `zip-6`.
- `--exclude-files <pattern-list>`: Specify files to exclude. For example: `**.java`, `glob:/java.base/lib/client/**`
- `--exclude-jmod-section <section-name>`: Specify a JMOD section to exclude. Where `<section-name>` is `man` or `headers`.
- `--exclude-resources <pattern-list>`: Specify resources to exclude. For example: `**.jcov`, `glob:**/META-INF/**`
- `--include-locales <langtag>[,<langtag>]*`: BCP 47 language tags separated by a comma, allowing locale matching defined in RFC 4647. For example: `en`, `ja`, `*-IN`
- `--strip-debug`: Strip debug information from the output image

- `--strip-java-debug-attributes`: Strip Java debug attributes from classes in the output image
- `--strip-native-commands`: Exclude native commands (such as `java/java.exe`) from the image.
- `--vm <client|server|minimal|all>`: Select the HotSpot VM in the output image. Default is `all`.

Here are some examples of how you might use these plugins with the `jlink` command:

```
# Create a runtime image with maximum compression,
# exclude specific files, and strip debug information
jlink --module-path $JAVA_HOME/jmods \
      --add-modules java.base \
      --compress zip-9 \
      --exclude-files "**.java,glob:/java.base/lib/client/**" \
      --strip-debug \
      --output custom-runtime-image

# Create a runtime image that includes only the
# specified locales and uses the server VM
jlink --module-path $JAVA_HOME/jmods \
      --add-modules java.base \
      --include-locales en,ja \
      --vm server \
      --output custom-runtime-image
```

Optimizing Runtime Images

In addition to creating a minimal runtime image, `jlink` provides options to further optimize the generated runtime. These optimizations can help reduce the size of the runtime image and improve its performance.

One important optimization is compression. By default, `jlink` does not compress the generated runtime image. However, we can enable compression using the `--compress` option followed by a compression level. The compression level can be set to 0 (no compression), 1 (constant string sharing), or 2 (ZIP compression). For example:

```
jlink --module-path <modulepath>
      --add-modules <modules> \
      --compress 2 \
      --output <path>
```

Using `--compress 2` applies ZIP compression to the generated runtime image, significantly reducing its size.

Another optimization is to strip debug information from the runtime image. Debug information is useful during development but is not necessary for production deployments. We can remove debug information using the `--strip-debug` option:

```
jlink --module-path <modulepath>
      --add-modules <modules> \
      --strip-debug \
      --output <path>
```

This way, we can further reduce the size of the runtime image.

Migrating an Application

Migrating an existing application to use modules can be a challenging task. It's doable, but requires careful planning and execution.

Before embarking on the migration process, it's important to understand how the packages and libraries in the existing application are structured. This involves analyzing the codebase and identifying the dependencies between different parts of the application.

One approach to gain insights into the application structure is to use `jdeps`. By running `jdeps` on the application's JAR files or class files, we can generate a dependency report that provides valuable information about the relationships between packages and classes.

Here's an example of running `jdeps` on an application JAR file:

```
jdeps -s -recursive application.jar
```

The `-s` option generates a summary output, and the `-recursive` option analyzes all dependent JAR files as well.

The output of jdeps will give us an overview of the packages and their dependencies. It will highlight any dependencies on JDK internal APIs, which is important to note as these APIs may not be accessible in future Java versions.

If you want more detail, you can use the `-verbose` option:

```
jdeps -verbose application.jar
```

The output will show the dependencies between packages and classes, as well as the dependencies on external libraries.

It's important to identify and resolve any circular dependencies or unnecessary dependencies at this stage. Circular dependencies can cause issues when modularizing the application, as modules cannot have cyclic dependencies. Unnecessary dependencies can bloat the application and make it harder to modularize effectively.

Now that we have a map of the application dependencies, it's time to start planning our migration. One common strategy is to split our big project into smaller, more manageable modules. This process involves identifying logical boundaries within the application and separating the code into distinct modules based on functionality and dependencies.

JPMS gives us a few tools to ease this transition: unnamed modules and automatic modules.

Let's say we have a big monolithic application called `BigApp`. We might start by running it the traditional way on the classpath:

```
java -cp BigApp.jar:lib/* com.example.MainClass
```

This puts `BigApp` and all its dependencies into the **unnamed module**. This is how pre-JPMS applications run, everything on the classpath becomes part of the unnamed module.

As a first step toward modularization, we can move `BigApp` to the module path without defining a `module-info.java` file:

```
java --module-path BigApp.jar
     --add-modules ALL-MODULE-PATH
     com.example.MainClass
```

This turns `BigApp` into an **automatic module**. It's not a proper JPMS module yet, but it's a start, it now has a module name (derived from the JAR name) and can require other modules.

For third-party libraries that aren't yet modularized, we can use automatic modules. Let's say we're using a library named `CoolLib`. We can put it on the module path alongside our application:

```
java --module-path BigApp.jar:CoolLib.jar
     --add-modules ALL-MODULE-PATH
     com.example.MainClass
```

Now both `BigApp` and `CoolLib` become automatic modules. The module system derives their names from the JAR filenames, and they export all their packages. But remember, these are temporary solutions. Our end goal is to have proper, explicit modules with `module-info.java` files for everything.

When splitting the project into modules, it's important to consider the dependencies between the modules. Aim to minimize the coupling between modules and promote loose coupling with well-defined interfaces and APIs.

Here are some strategies for properly splitting a big project into modules:

1. **Package-based splitting:** One approach is to create modules based on the existing package structure. Each package or a group of related packages can be converted into a separate module. This helps in maintaining a clear separation of concerns and encapsulation.
2. **Layered architecture:** If the application follows a layered architecture (presentation layer, business logic layer, data access layer), each layer can be split into its own module. This allows for better modularity and easier maintenance of each layer independently.
3. **Feature-based splitting:** Another approach is to split the application based on its features or functional areas. Each major feature or functionality can be encapsulated within its own module, promoting reusability and maintainability.
4. **Dependency-based splitting:** Analyzing the dependencies between different parts of the application can help identify natural module boundaries. Strongly coupled components can be grouped together into a module, while loosely coupled components can be split into separate modules.

But you might be wondering what strategies can we take for the migration in general. Here are a few approaches:

1. **Incremental migration:** In this approach, the migration is done gradually, one module at a time. Start by identifying a suitable module to migrate first, typically one with minimal dependencies on other parts of the application. Once the module is successfully migrated, move on to the next module, and so on.
2. **Bottom-up migration:** This strategy involves starting the migration from the lowest-level modules and gradually moving up the dependency hierarchy. Begin by modularizing the modules that have no dependencies on other modules, and then proceed to modules that depend on already modularized modules.
3. **Top-down migration:** In contrast to the bottom-up approach, the top-down migration starts with the high-level modules and works its way down the dependency chain. This strategy is useful when the high-level modules have a clear separation of concerns and can be easily modularized.
4. **Parallel development:** If time and resources permit, parallel development can be employed. In this approach, a separate branch or codebase is created for the modularized version of the application, while the existing non-modularized version continues to be maintained. Development can proceed simultaneously on both versions, gradually migrating modules to the modularized branch.

For example, we might start modularizing a part of our application using a bottom-up approach:

```
module com.myapp.core {
    requires java.base;
    requires com.coollib;  // This is our automatic module
    exports com.myapp.core.api;
}
```

This `module-info.java` file defines a new module `com.myapp.core`. We're starting with a core module that likely has fewer dependencies, which is characteristic of the bottom-up approach. It requires the `java.base` module (which is implicit but we're being explicit here) and `CoolLib`, which is an automatic module. It also exports a package `com.myapp.core.api` for other modules to use.

As we create more modules, we'll need to think carefully about our module boundaries.

One important thing to keep in mind is that while we're in this phase, we might need to open up more than we'd like. For example:

```
open module com.myapp.core {
    requires java.base;
    requires com.coollib;
    exports com.myapp.core.api;
}
```

By making this an open module, we allow deep reflection into all its packages. It's not ideal for security, but it might be necessary during the migration to keep things working. As we progress in our migration, we'll want to tighten these permissions, exporting and opening only what's necessary.

Remember, migration is a process. It's okay to use unnamed and automatic modules as stepping stones. The key is to have a clear migration plan and to move steadily towards a fully modularized system. Breaking down the migration process into smaller, manageable tasks helps in tracking progress and identifying any challenges or roadblocks along the way.

Key Points

- A module in Java is a named, self-describing collection of code and data. It's defined in a `module-info.java` file.
- The main components of a module declaration are:
  - `module`: Declares the module name
  - `requires`: Specifies module dependencies
  - `exports`: Makes packages accessible to other modules
  - `provides`: Declares service implementations
  - `uses`: Indicates that a module uses a service
- There are three types of modules:
  1. Named modules: Explicitly defined with a `module-info.java` file
  2. Automatic modules: Created from JAR files placed on the module path
  3. Unnamed module: Contains all classes on the classpath
- Key benefits of JPMS include improved encapsulation, clearer dependencies, better performance, and enhanced security.
- The `exports` keyword controls which packages are accessible to other modules. You can also use `exports...to` to limit access to specific modules.
- The `requires` keyword declares module dependencies. Use `requires transitive` to make a module's dependencies available to modules that depend on it.
- The `opens` keyword allows reflective access to a package at runtime.
- Services in JPMS consist of:
  1. Service Provider Interface (SPI): Defines the service contract
  2. Service Provider: Implements the SPI
  3. Service Consumer: Uses the service
- The `provides...with` clause in a module declaration specifies that a module provides a service implementation.
- The `uses` clause indicates that a module consumes a service.
- `ServiceLoader` is used to discover and load service implementations at runtime.
- Built-in Java modules start with `java` (core SE Platform) or `jdk` (additional JDK-specific APIs).
- When designing multi-module applications, consider separation of concerns, encapsulation, stable dependencies, reusability, and appropriate module size.

- Compile modules using `javac` with the `--module-path` option. Run modular applications using `java` with `--module-path` and `-m` options.
- Use the `jar` command to package modules into modular JAR files.
- Resolve conflicts between modules by managing versions, avoiding split packages, ensuring unique package names across modules, and breaking cyclic dependencies.
- The `java` command with `--describe-module` option provides details about a specific module, including exports, requirements, and services.
- `--list-modules` option lists all available modules in the Java runtime and custom modules.
- `--show-module-resolution` option helps debug complex module dependencies by showing how each module is resolved.
- The `jar` command can be used to inspect modules without running them, using `--describe-module` option.
- `jdeps` is a tool for analyzing and visualizing dependencies at both module and class levels.
- This is its basic syntax: `jdeps [options] path`. Here are some of its most important options:
  - `--dot-output` option generates a DOT file for visualizing dependency graphs.
  - `--jdk-internals` flag helps identify usage of internal JDK APIs.
  - `--recursive` option provides transitive dependency analysis.
- JMOD files are designed for the Java Platform Module System (JPMS) and have a `.jmod` extension.
- JMOD files can include compiled classes, resources, native libraries, and module descriptors.
- `jmod` has several modes: create, extract, describe, list, and hash.
- Best practices include using descriptive names, including `module-info.java`, and organizing module contents properly.
- `jlink` creates custom runtime images that include only required modules.
- This is the basic syntax of `jlink`: `jlink [options] --module-path <modulepath> --add-modules <modules>`
- Plugins can extend `jlink` functionality for additional optimizations or customizations.
- Options like `--compress`, `--strip-debug`, `--no-header-files`, and `--no-man-pages` help optimize the runtime image.
- To migrate an application to using modules:
  - Use `jdeps` to analyze existing application structure and dependencies before migration.
  - Strategies for splitting projects into modules: package-based, layered architecture, feature-based, and dependency-based.
  - Migration approaches: incremental, bottom-up, top-down, and parallel development.
  - Unnamed modules and automatic modules can be used as temporary solutions during migration.
  - Consider using open modules during migration to allow reflection, but aim to tighten permissions as the migration progresses.
  - Migration is a process; it's okay to use unnamed and automatic modules as stepping stones towards full modularization.

Practice Questions

1. Which of the following are types of modules in the Java Platform Module System (JPMS)? (Choose all that apply.)

A) Automatic module
B) Default module
C) Unnamed module
D) Core module
E) Primary module

2. Which of the following is the correct way to declare a module named `com.example` in the Java Platform Module System (JPMS)?

A) `module com.example { export com.example.api; }`
B) `declare module com.example { }`
C) `create module com.example { requires java.base; }`
D) `module com.example { }`
E) `module com.example requires java.base;`

3. Which of the following access control statements correctly restricts access to the `com.example.internal` package so that it is only accessible to the `com.example.client` module?

A) `module com.example { exports com.example.internal to com.example.client; }`
B) `module com.example { opens com.example.internal to com.example.client; }`
C) `module com.example { requires com.example.internal; }`
D) `module com.example { provides com.example.internal to com.example.client; }`
E) `module com.example { uses com.example.internal; }`

4. Given the following module declarations, which statement is correct regarding the accessibility of the `com.example.api` package for deep reflection by the `com.example.client` module?

```
module com.example {
    exports com.example.api;
    opens com.example.internal to com.example.client;
}

module com.example.client {
    requires com.example;
}
```

A) The `com.example.client` module can access the `com.example.api` package for deep reflection.
B) The `com.example.client` module cannot access the `com.example.api` package for deep reflection.
C) The `com.example.api` package is opened to all modules for deep reflection.
D) The `com.example.internal` package is exported to the `com.example.client` module.
E) The `com.example.api` package is exported to the `com.example.client` module for deep reflection.

5. Which of the following statements is correct regarding core Java modules and their functionalities?

A) The `java.base` module provides the Swing and AWT libraries for building graphical user interfaces.
B) The `java.logging` module is responsible for handling collections, including lists, sets, and maps.
C) The `java.desktop` module provides the classes for implementing standard input and output streams.
D) The `java.xml` module includes the classes for processing XML documents.
E) The `java.naming` module provides APIs for accessing and processing annotations.

6. Which of the following command-line statements correctly compiles the module located in the `src/com.example` directory and outputs the compiled module to the `out` directory?

A) `javac -d out src/com.example/module-info.java src/com.example/com/example/*.java`

B) `javac -sourcepath src -d out com.example/module-info.java com.example/com/example/*.java`
C) `javac -d out --module-source-path src -m com.example`
D) `javac -modulepath out -d src src/com.example/module-info.java src/com.example/com/example/*.java`
E) `javac --module-path src --module com.example -d out`

7. Given the following multi-module application structure, which command compiles both modules correctly?

```
src/
├── com.foo/
│   ├── module-info.java
│   └── com/foo/Foo.java
└── com.bar/
    ├── module-info.java
    └── com/bar/Bar.java
```

A) `javac --module-source-path src -d out $(find src -name "*.java")`
B) `javac -d out --module com.foo,com.bar --module-source-path src`
C) `javac -sourcepath src -d out src/com.foo/module-info.java src/com.foo/com/foo/*.java src/com.bar/module-info.java src/com.bar/com/bar/*.java`
D) `javac -modulepath src -d out src/com.foo/*.java src/com.bar/*.java`
E) `javac --module-source-path src/com.foo,src/com.bar -d out`

8. Which of the following statements correctly specifies a service provider implementation for the service `com.example.Service` in the `module-info.java` of the `com.provider` module?

A) `requires com.example.Service with com.provider.ServiceImpl;`
B) `exports com.example.Service with com.provider.ServiceImpl;`
C) `provides com.example.Service with com.provider.ServiceImpl;`
D) `uses com.example.Service with com.provider.ServiceImpl;`

9. Which of the following command-line statements correctly describes the `com.example` module using the `--describe-module` option?

A) `java --describe-module com.example/module-info.java`
B) `javac --describe-module com.example`
C) `jar --describe-module com.example`
D) `java --describe-module com.example`

10. Which of the following command-line statements correctly uses `jdeps` to analyze the dependencies of a JAR file named `example.jar`? (Choose all that apply)

A) `jdeps --list-deps example.jar`
B) `jdeps -verbose example.jar`
C) `jdeps -s example.jar`
D) `jdeps --check example.jar`

11. Which of the following command-line statements correctly creates a JMOD file from the contents of the `mods/com.example` directory?

A) `jmod create --class-path mods/com.example --output com.example.jmod`
B) `jmod --create --class-path mods/com.example --output com.example.jmod`
C) `jmod --create --dir mods/com.example --output com.example.jmod`
D) `jmod create --dir mods/com.example --output com.example.jmod`

12. Which of the following command-line statements correctly creates a custom runtime image using the `jlink` tool with the modules `java.base` and `com.example` and outputs it to the `myimage` directory?

A) `jlink --module-path java.base:com.example --output myimage`

B) `jlink --module-path mods --add-modules java.base,com.example --output myimage`
C) `jlink --add-modules java.base,com.example --image myimage`
D) `jlink --modules java.base,com.example --dir myimage`

13. Which of the following statements is correct regarding the migration of a legacy application to the Java Platform Module System using unnamed and automatic modules?

A) An unnamed module can depend on named modules and other unnamed modules.
B) Automatic modules must have a `module-info.java` file to be placed on the module path.
C) Unnamed modules can export their packages to named modules using `module-info.java`.
D) An automatic module is created when a JAR file without a `module-info.java` is placed on the module path, and it can read all other modules.

Chapter THIRTEEN

The Java Platform Module System

Answers

1. The correct answers are A and C.

Explanation:

- **A)** Automatic module
  - This option is correct. An automatic module is created from a JAR file that is placed on the module path but does not have a module descriptor (`module-info.java`). The module system infers a module name from the JAR file name and exports all packages in the JAR.
- **B)** Default module
  - This option incorrect. There is no concept of a "default module" in JPMS. The term might be confused with unnamed modules or other types of configurations, but it is not a recognized type.
- **C)** Unnamed module
  - This option is correct. The unnamed module is a special module that includes all classes on the classpath. It does not have a module descriptor and can access other unnamed modules but cannot be required by named modules.
- **D)** Core module
  - This option is incorrect. There is no specific type called "core module" in JPMS. JPMS does not categorize modules this way.
- **E)** Primary module
  - This option is incorrect. Similar to "core module," there is no type called "primary module" in JPMS.

2. The correct answer is D.

Explanation:

- **A)** `module com.example { export com.example.api; }`
  - This option is incorrect. In this case, `export`is missing an "s". The correct syntax to export a package would be `exports com.example.api;`.
- **B)** `declare module com.example { }`
  - This option is incorrect. There is no `declare` keyword used in the JPMS for defining a module.
- **C)** `create module com.example { requires java.base; }`
  - This option is incorrect. The correct syntax does not use the `create` keyword for module declaration.
- **D)** `module com.example { }`
  - This option is correct. This is the correct way to declare a module named `com.example` without any additional requirements.
- **E)** `module com.example requires java.base;`

- This option is incorrect. The syntax is invalid because it lacks braces `{ }` to define the module body.

3. The correct answer is A.

Explanation:

- **A)** `module com.example { exports com.example.internal to com.example.client; }`
  - This option is correct. The `exports` directive with the `to` clause restricts the export of the `com.example.internal` package to only the specified module `com.example.client`.
- **B)** `module com.example { opens com.example.internal to com.example.client; }`
  - This option is incorrect. The `opens` directive is used for reflection purposes, not for compile-time access control.
- **C)** `module com.example { requires com.example.internal; }`
  - This option is incorrect. The `requires` directive is used to specify module dependencies, not to control package accessibility.
- **D)** `module com.example { provides com.example.internal to com.example.client; }`
  - This option is incorrect. The `provides` directive is used to specify service providers in the module system, not for restricting package access.
- **E)** `module com.example { uses com.example.internal; }`
  - This option is incorrect. The `uses` directive is used to specify service consumers in the module system, not for restricting package access.

4. The correct answer is B.

Explanation:

- **A)** The `com.example.client` module can access the `com.example.api` package for deep reflection.
  - This option is incorrect. The `com.example.api` package is exported, not opened, meaning it is available for use but not for deep reflection by other modules.
- **B)** The `com.example.client` module cannot access the `com.example.api` package for deep reflection.
  - This option is correct. The `com.example.api` package is not opened for deep reflection; it is only exported for use by other modules.
- **C)** The `com.example.api` package is opened to all modules for deep reflection.
  - This option is incorrect. The `com.example.api` package is exported to all modules, but it is not opened for deep reflection to any module.
- **D)** The `com.example.internal` package is exported to the `com.example.client` module.
  - This option is incorrect. The `com.example.internal` package is opened to `com.example.client` for deep reflection but not exported.
- **E)** The `com.example.api` package is exported to the `com.example.client` module for deep reflection.
  - This option is incorrect. The `com.example.api` package is exported to the `com.example.client` module, but exporting does not include deep reflection capabilities.

5. The correct answer is D.

Explanation:

- **A)** The `java.base` module provides the Swing and AWT libraries for building graphical user interfaces.
  - This option is incorrect. The `java.base` module does not provide the Swing and AWT libraries. These libraries are provided by the `java.desktop` module.
- **B)** The `java.logging` module is responsible for handling collections, including lists, sets, and maps.
  - This option is incorrect. The `java.logging` module is responsible for the logging framework in Java, not for handling collections. The collections framework is part of the `java.base` module.
- **C)** The `java.desktop` module provides the classes for implementing standard input and output streams.
  - This option is incorrect. The `java.desktop` module includes classes for building graphical user interfaces (Swing and AWT), not for standard input and output streams. Standard I/O is part of the `java.base` module.
- **D)** The `java.xml` module includes the classes for processing XML documents.

  - This option is correct. The `java.xml` module includes classes for processing XML documents, such as those for parsing and transforming XML using APIs like DOM, SAX, and StAX.
- **E)** The `java.naming` module provides APIs for accessing and processing annotations.
  - This option is incorrect. The `java.naming` module provides APIs for accessing naming and directory services (JNDI), not for processing annotations. Annotations are part of the `java.base` module.

6. The correct answer is C.

Explanation:

- **A)** `javac -d out src/com.example/module-info.java src/com.example/com/example/*.java`
  - This option is incorrect. While it correctly specifies the output directory and the source files, it does not use the `--module-source-path` option and does not specify the module name with `-m`.
- **B)** `javac -sourcepath src -d out com.example/module-info.java com.example/com/example/*.java`
  - This option is incorrect. The `-sourcepath` option is not used for module compilation. The correct option should be `--module-source-path`.
- **C)** `javac -d out --module-source-path src -m com.example`
  - This option is correct. The `javac -d out --module-source-path src -m com.example` command correctly compiles the module `com.example` located in the `src` directory and outputs the compiled classes to the `out` directory.
- **D)** `javac -modulepath out -d src src/com.example/module-info.java src/com.example/com/example/*.java`
  - This option is incorrect. The `-modulepath` option is incorrectly placed, and the source and destination directories are swapped.
- **E)** `javac --module-path src --module com.example -d out`
  - This option is incorrect. The command incorrectly uses `--module-path` instead of `--module-source-path` and the module name is specified with `--module` instead of `-m`.

7. The correct answer is A.

Explanation:

- **A)** `javac --module-source-path src -d out $(find src -name "*.java")`
  - This option is correct. The command `javac --module-source-path src -d out $(find src -name "*.java")` correctly compiles both modules by specifying the module source path and finding all Java files in the source directory.
- **B)** `javac -d out --module com.foo,com.bar --module-source-path src`
  - This option is incorrect. The `--module` option does not accept multiple modules separated by commas in this context.
- **C)** `javac -sourcepath src -d out src/com.foo/module-info.java src/com.foo/com/foo/*.java src/com.bar/module-info.java src/com.bar/com/bar/*.java`
  - This option is incorrect. Although it specifies the source files, it does not use the `--module-source-path` option and is unnecessarily verbose.
- **D)** `javac -modulepath src -d out src/com.foo/*.java src/com.bar/*.java`
  - This option is incorrect. The `-modulepath` option is misused, and the path should point to the directory containing the module source code.
- **E)** `javac --module-source-path src/com.foo,src/com.bar -d out`
  - This option is incorrect. The `--module-source-path` option should point to the base directory (`src`), not individual module directories.

8. The correct answer is C.

Explanation:

- **A)** `requires com.example.Service with com.provider.ServiceImpl;`
  - This option is incorrect. The `requires` keyword is used to declare dependencies on other modules, not for specifying service providers.
- **B)** `exports com.example.Service with com.provider.ServiceImpl;`

  - This option is incorrect. The `exports` keyword is used to make packages accessible to other modules, not for specifying service providers.
- **C)** `provides com.example.Service with com.provider.ServiceImpl;`
  - This option is correct. The `provides com.example.Service with com.provider.ServiceImpl;` statement correctly specifies that the `com.provider` module provides an implementation of the `com.example.Service`.
- **D)** `uses com.example.Service with com.provider.ServiceImpl;`
  - This option is incorrect. The `uses` keyword is used to declare that the module relies on a service but does not provide an implementation.

9. The correct answer is D.

Explanation:

- **A)** `java --describe-module com.example/module-info.java`
  - This option is incorrect. The `--describe-module` option is not used with a specific file path like `module-info.java`; it requires a module name.
- **B)** `javac --describe-module com.example`
  - This option is incorrect. The `--describe-module` option is not valid for the `javac` command; it is used with the `java` command.
- **C)** `jar --describe-module com.example`
  - This option is incorrect. The `--describe-module` option is not valid for the `jar` command; it is used with the `java` command.
- **D)** `java --describe-module com.example`
  - This option is correct. The `java --describe-module com.example` command correctly describes the module `com.example` using the `--describe-module` option.

10. The correct answers are B and C.

Explanation:

- **A)** `jdeps --list-deps example.jar`
  - This option is incorrect. The `--list-deps` option does not exist for `jdeps`.
- **B)** `jdeps -verbose example.jar`
  - This option is correct. While `-verbose` is a valid option, it provides more information.
- **C)** `jdeps -s example.jar`
  - This option is correct. The `-s` option with `jdeps` provides a summary of the dependencies of the `example.jar` file.
- **D)** `jdeps --check example.jar`
  - This option is incorrect. The `--check` option does not exist for `jdeps`.

11. The correct answer is A.

Explanation:

- **A)** `jmod create --class-path mods/com.example --output com.example.jmod`
  - This option is correct. It uses the correct syntax for the `jmod` command to create a JMOD file. The `create` operation is specified, followed by the `--class-path` option to indicate the source directory, and finally the name of the output JMOD file. This command will create a JMOD file named `com.example.jmod` using the contents of the `mods/com.example` directory.
- **B)** `jmod --create --class-path mods/com.example --output com.example.jmod`
  - This option is incorrect. The `create` operation in the `jmod` command should not be prefixed with `--`. The correct format is `jmod create`, not `jmod --create`. The rest of the command is correct, but this syntax error makes the entire command invalid.
- **C)** `jmod --create --dir mods/com.example --output com.example.jmod`
  - This option is incorrect. First, like option B, it incorrectly uses `--create` instead of `create`. Second, it uses the `--dir` option, which is not used for creating JMOD files, but for specifying the output directory when extracting files from a JMOD. When creating a JMOD file, we use `--class-path` to

specify the source directory. The `--output` option is also not a valid option for the `jmod` command.

- **D)** `jmod create --dir mods/com.example --output com.example.jmod`
  - This option is incorrect. It uses the `--dir` option instead of `--class-path` for specifying the source directory, and it incorrectly includes an `--output` option, which is not valid for the `jmod` command. When creating a JMOD file, the output file name is simply specified as the last argument, not with an `--output` option.

12. The correct answer is B.

Explanation:

- **A)** `jlink --module-path java.base:com.example --output myimage`
  - This option is incorrect. The `--module-path` option should specify the directory containing the modules, not the module names directly.
- **B)** `jlink --module-path mods --add-modules java.base,com.example --output myimage`
  - This option is correct. The command `jlink --module-path mods --add-modules java.base,com.example --output myimage` correctly specifies the module path and adds the necessary modules, outputting the custom runtime image to the `myimage` directory.
- **C)** `jlink --add-modules java.base,com.example --image myimage`
  - This option is incorrect. The `--image` option is not valid; the correct option is `--output`.
- **D)** `jlink --modules java.base,com.example --dir myimage`
  - This option is incorrect. The `--modules` option is incorrect; the correct option is `--add-modules`, and `--dir` should be `--output`.

13. The correct answer is D.

Explanation:

- **A)** An unnamed module can depend on named modules and other unnamed modules.
  - This option is incorrect. An unnamed module can depend on named modules, which is true. However, unnamed modules cannot depend on other unnamed modules. Unnamed modules are created when classes are loaded from the classpath, and they cannot read other unnamed modules. They can only read the named modules of the platform and other modules explicitly added to the module path.
- **B)** Automatic modules must have a `module-info.java` file to be placed on the module path.
  - This option is incorrect. Automatic modules do not require a `module-info.java` file. Their module name is inferred from the JAR file name.
- **C)** Unnamed modules can export their packages to named modules using `module-info.java`.
  - This option is incorrect. Unnamed modules cannot export packages because they do not use `module-info.java`.
- **D)** An automatic module is created when a JAR file without a `module-info.java` is placed on the module path, and it can read all other modules.
  - This option is correct. An automatic module is created by placing a JAR file without a `module-info.java` on the module path. This automatic module can read all other modules, both named and unnamed.

Chapter FOURTEEN

Localization

Chapter Content

- Introduction to Localization
- The `Locale` Class
  - Locale Categories
- Resource Bundles
- The `MessageFormat` Class

- The `NumberFormat` Class
  - The `DecimalFormat` Class
  - The `CompactNumberFormat` Class
- The `DateTimeFormatter` Class
- Key Points
- Practice Questions

Introduction to Localization

It's common for software applications to be used by people all around the globe who speak different languages and live in various countries and cultures. To make an application globally accessible and user-friendly, it needs to adapt to the user's language and cultural norms. This is where localization comes into play.

Localization refers to the process of designing and developing your application in a way that it can be adapted to various locales without requiring engineering changes. Think of it like creating a "world-ready" app.

A locale represents a specific geographical, political, or cultural region. It is made up of a language code, a country code, and optionally, a variant.

Here's an example of a locale representation:

`fr_CA`

Notice that: - The language part is required - The country part is optional (But if you choose to include it, it must follow the proper format) - The language part is in lower case - The country part is in upper case - The language and the country are separated by an underscore

For example, `en` represents English, `en_US` represents English as used in the United States, which might differ from `en_UK` (English as used in the United Kingdom) in things like spelling (color vs colour), vocabulary (truck vs lorry), or currency ($100 vs £100).

The `Locale` Class

Locales are represented by the `java.util.Locale` class. Basically, this class represents a language and a country although, to be precise, a locale can have the following information:

- An ISO 639 alpha-2 or alpha-3 language code, like *ja* (Japanese)
- An ISO 3166 alpha-2 country code or UN M.49 numeric-3 area code, like *JP* (Japan)
- A variant name, usually empty but can be any string.
- An ISO 15924 alpha-4 script code, like *Latn* (Latin)
- A set of extensions represented by single characters, like *u*.

There are several ways to get or create a `Locale` object in Java.

To get the default locale of the Java Virtual Machine (JVM), you use:

```
Locale locale = Locale.getDefault();
```

The `Locale` class provides several built-in constants for commonly used locales. For example:

```
Locale usLocale = Locale.US;  // English as used in the US
Locale frLocale = Locale.FRANCE; // French as used in France
```

You can also create a new `Locale` object using one of its constructors:

```
Locale(String language)
Locale(String language, String country)
Locale(String language, String country, String variant)
```

For example:

```
Locale locale = new Locale("fr", "CA", "POSIX");
```

This creates a new `Locale` object representing French as used in Canada with the POSIX variant.

The first parameter is the language code, the second is the country code, and the third (optional) is the variant code. Language codes are two or three letter lowercase codes as defined by ISO 639. Country codes are two-letter uppercase codes as defined by ISO 3166. Variants are any arbitrary value used to indicate any kind of variation, not just language variations.

You can also use the `forLanguageTag(String)` factory method. This method expects a language code, for example:

```
Locale german = Locale.forLanguageTag("de");
```

Additionally, by using `Locale.Builder`, you can set the properties you need and build the object at the end, for example:

```
Locale japan = new Locale.Builder()
                .setRegion("JP")
                .setLanguage("ja")
                .build();
```

Passing an invalid argument to any of the above constructors and methods will not throw an exception, it will just create an object with invalid options that will make your program behave incorrectly:

```
Locale badLocale = new Locale("a", "A"); // No error
System.out.println(badLocale); // It prints a_A
```

The `getDefault()` method returns the current value of the default locale for this instance of the Java Virtual Machine (JVM). The JVM sets the default locale during startup based on the host environment. It is used by many locale-sensitive methods if no locale is explicitly specified. However, it can be changed using the `setDefault(Locale)` method:

```
System.out.println(Locale.getDefault()); // Let's say it prints en_GB
Locale.setDefault(new Locale("en", "US"));
System.out.println(Locale.getDefault()); // Now prints en_US
```

Locale Categories

The `Locale.Category` enum defines two categories used to differentiate the purposes for which a `Locale` might be used: `Locale.Category.DISPLAY` and `Locale.Category.FORMAT`. These categories help specify which locale settings to apply in different contexts.

1. **Locale.Category.DISPLAY**:
   - This category is used for user interface strings, such as messages, labels, and menus.
   - It is applied when you need to localize the content that is displayed to the user.
   - For instance, when an application needs to show date, time, or currency in a localized manner, it uses the `DISPLAY` locale to ensure the formats and messages are appropriate for the user's language and region.
2. **Locale.Category.FORMAT**:
   - This category is used for formatting dates, numbers, and other values that are part of the data processing or backend systems.
   - It is applied when you need to parse, format, or validate locale-sensitive data.
   - For instance, when you need to format a date for storage or further processing, the `FORMAT` locale is used to ensure the data follows the correct localized format.

Here's an example showing how to use these categories:

```
// Setting the DISPLAY locale
Locale.setDefault(Locale.Category.DISPLAY, Locale.FRANCE);
```

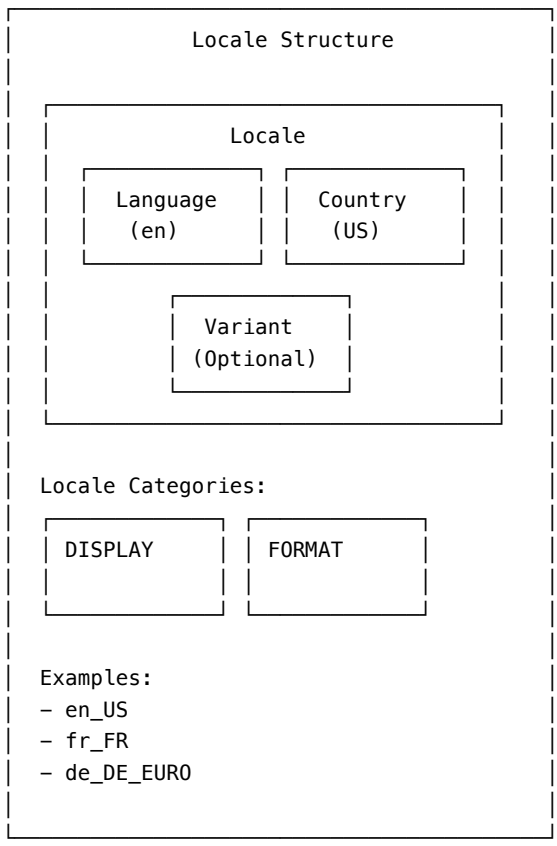
```java
// Setting the FORMAT locale
Locale.setDefault(Locale.Category.FORMAT, Locale.US);

// Getting the DISPLAY locale
Locale displayLocale = Locale.getDefault(Locale.Category.DISPLAY);
System.out.println("DISPLAY Locale: " + displayLocale);

// Getting the FORMAT locale
Locale formatLocale = Locale.getDefault(Locale.Category.FORMAT);
System.out.println("FORMAT Locale: " + formatLocale);
```

In this example, the `DISPLAY` locale is set to `Locale.FRANCE`, meaning that user interface elements will be localized for French users. The `FORMAT` locale is set to `Locale.US`, meaning that any date or number formatting will follow the conventions used in the United States.

This is the output:

```
DISPLAY Locale: fr_FR
FORMAT Locale: en_US
```

Here's a diagram that summarizes the structure of a locale:




```
Key Points:
- Language is required, country and variant are optional
- Language codes are lowercase, country codes are uppercase
- Variant is used for further distinction (such as dialect)
- DISPLAY category affects how the locale itself is displayed
- FORMAT category affects formatting of dates, numbers, etc.
```

Resource Bundles

A resource bundle is a way to organize and access locale-specific data such as messages or labels in your application. Think of it as a collection of key-value pairs, where the keys are the same across all locales but the values are locale-specific.

To support this, we have an abstract class `java.util.ResourceBundle` with two subclasses:

- `java.util.PropertyResourceBundle`: Each locale is represented by a property file. Keys and values are of type `String`.
- `java.util.ListResourceBundle`: Each locale is represented by a subclass of this class that overrides the method `Object[][] getContents()`. The returned array represents the keys and values. Keys must be of type `String`, but values can be any object.

In Java, resource bundles are typically implemented as property files. A property file is a plain text file that contains key-value pairs separated by an equals sign (=). For example:

```
greeting=Hello
farewell=Goodbye
```

The name of the property file is important. It should be in the format `<basename>_<language>_<country>_<variant>.properties` where: - `<basename>` is the name you give to the resource bundle - `<language>` is the lowercase two-letter ISO-639 language code - `<country>` is the uppercase two-letter ISO-3166 country code - `<variant>` is an optional vendor or browser-specific code

For example, we can have bundles with the following names (assuming we're working with property files, although it's the same with classes):

```
MyBundle.properties
MyBundle_en.properties
MyBundle_en_NZ.properties
MyBundle_en_US.properties
```

In this case, `MyBundle_en_US.properties` would contain messages for English as used in the United States.

To create a resource bundle, you first create the property files for each locale you want to support. Then, you use the `java.util.ResourceBundle` class to load the appropriate property file for the current locale:

```
ResourceBundle bundle = ResourceBundle.getBundle("MyBundle", locale);
```

This loads the resource bundle named `MyBundle` for the given locale. Java will search for the property file that best matches the requested locale based on the following priority:

1. Language + Country + Variant
2. Language + Country
3. Language
4. Default locale (as returned by `Locale.getDefault()`)
5. Root resource bundle (basename only, no locale info)

If no matching property file is found, a `MissingResourceException` is thrown.

Once you have a `ResourceBundle`, you can retrieve the locale-specific value for a given key using the `getString` method:

```
String greeting = bundle.getString("greeting");
```

To get an object value, use:

```
Integer num = (Integer) bundle.getObject("number");
```

It's important to note that `getString(key)` is effectively a shortcut to:

```
String val = (String) bundle.getObject(key);
```

You can use the keys of the matching resource bundle and any of its parents.

The parents of a resource bundle are the ones with the same name but fewer components. For example, the parents of `MyBundle_es_ES` are:

```
MyBundle_es
MyBundle
```

Let's assume the default locale is `en_US`, and that your program is using these and other property files, all in the default package, with the values:

```
MyBundle_en.properties
s = buddy

MyBundle_es_ES.properties
s = tío

MyBundle_es.properties
s = amigo

MyBundle.properties
hi = Hola
```

We can create a resource bundle like this:

```
Locale spain = new Locale("es", "ES");
Locale spanish = new Locale("es");

ResourceBundle rb = ResourceBundle.getBundle("MyBundle", spain);
System.out.format("%s %s\n",
    rb.getString("hi"), rb.getString("s"));

rb = ResourceBundle.getBundle("MyBundle", spanish);
System.out.format("%s %s\n",
    rb.getString("hi"), rb.getString("s"));
```

This would be the output:

```
Hola tío
Hola amigo
```

As you can see, each locale picks different values for key `s`, but they both use the same for `hi` since this key is defined in their parent.

If you don't specify a locale, the `ResourceBundle` class will use the default locale of your system:

```
ResourceBundle rb = ResourceBundle.getBundle("MyBundle");
System.out.format("%s %s\n",
    rb.getString("hi"), rb.getString("s"));
```

Since we're assuming that the default locale is `en_US`, the output is:

```
Hola buddy
```

We can also get all the keys in a resource bundle with the method `keySet()`:

```
ResourceBundle rb =
    ResourceBundle.getBundle("MyBundle", spain);
Set<String> keys = rb.keySet();
keys.stream()
```

```
    .forEach(key ->
        System.out.format("%s %s\n", key, rb.getString(key)));
```

This is the output (notice it also prints the parent key):

```
hi Hola
s tío
```

The `MessageFormat` Class

Sometimes, you need to format messages that include variable data. For example, you might want to display a personalized greeting that includes the user's name, or an error message that includes specific details about the error.

The `java.text.MessageFormat` class provides a powerful way to create localized messages by combining a pattern string with arguments. It allows you to define a message template with placeholders for variable data, and then replace those placeholders with actual values at runtime.

The key method in the `MessageFormat` class is `format`:

```
static String format(String pattern, Object... arguments)
```

This static method takes a message pattern and an array of arguments, and returns the formatted message as a string. However, there are two other `format` methods defined as instance methods:

```
final StringBuffer format(Object[] arguments, StringBuffer result, FieldPosition pos)
final StringBuffer format(Object arguments, StringBuffer result, FieldPosition pos)
```

These methods format an array of objects and append the pattern of the `MessageFormat` instance, with format elements replaced by the formatted objects, to the provided `StringBuffer`.

Additionally, its parent class, `java.text.Format`, defines another `format` method:

```
public final String format(Object obj)
```

This method formats an object to produce a string. It is equivalent to:

```
format(obj, new StringBuffer(), new FieldPosition(0)).toString();
```

The message pattern is a string that includes the static text of the message, as well as placeholders for the variable parts. The placeholders are marked with curly braces `{}` and a number that indicates the position of the argument in the argument array.

The syntax of a message format pattern follows this structure:

```
Literal text {argumentIndex,formatType,formatStyle} Literal text
```

Where:

- **Literal Text**: This is any text you want to include directly in the message. For example, in `Hello, {0}`, `Hello,` is literal text.
- **Argument Index**: `{argumentIndex}` specifies where to insert an argument. The number inside the curly braces corresponds to the position of the argument in the list you provide when formatting the message. For instance, `{0}` will be replaced by the first argument, `{1}` by the second argument, and so on.

-. **Format Type**: `{argumentIndex,formatType}` adds a format type to the argument. Common format types include: - `number`: Formats the argument as a number. - `date`: Formats the argument as a date. - `time`: Formats the argument as a time. - `choice`: Uses a choice format for the argument.

- **Format Style**: `{argumentIndex,formatType,formatStyle}` further refines the format type with a specific style. For example:
  - `number,currency`: Formats the number as a currency.

  - `date,short`: Formats the date in a short style.
  - `number,percent`: Formats the number as a percentage.

Text can be quoted using single quotes `'`, which are escaped as `''`. Any unmatched curly braces must be escaped with a single quote. For example: - `"'{0}'"` represents the literal string `"{0}"` - `"'{'"` represents the literal string `"{"` - `"'}'"` represents the literal string `"}"`

Here are some examples of valid message format patterns:

- **Simple Argument Insertion**:

  `"The value is {0}"`

  If the argument is `42`, the result will be "The value is 42".

- **Number Formatting**:

  `"The total amount is {0,number,currency}"`

  If the argument is `1234.56`, the result will be "The total amount is $1,234.56".

- **Date Formatting**:

  `"The date today is {0,date,full}"`

  If the argument is a date, the result might be "The date today is Thursday, July 25, 2024".

- **Choice Formatting**:

  `"There are {0,choice,0#no files|1#one file|1<many files}"`

  If the argument is `0`, `1`, or `2`, the results will be `There are no files`, `There is one file`, or `There are many files` respectively.

However, instead of using the static `format` method, you can create a `MessageFormat` instance by passing to the constructor of the class a pattern string and optionally a `Locale` to its constructor:

```
String pattern = "The disk \"{1}\" contains {0} file(s).";
MessageFormat messageFormat = new MessageFormat(pattern, Locale.US);
```

You can also set the `Locale` for the `MessageFormat` instance using `setLocale()`. This determines how the arguments are formatted.

In any case, to create a formatted message string, call `format()` with an array of argument objects:

```
Object[] arguments = {42, "MyDisk"};
String result = messageFormat.format(arguments);
// result: "The disk "MyDisk" contains 42 file(s)."
```

There are some rules about argument indexes: - `format()` replaces each format element `{argumentIndex}` with the corresponding object from the `arguments` array.
- If an argument index is used multiple times, all occurrences are replaced with the same formatted value. - Unused arguments are ignored. Missing or extra arguments cause an exception.

Suppose we have the following message format pattern and arguments:

```
String pattern = "Hello, {0}. Today is {1,date,long}. Your balance is {2,number,currency}. Hello again, {0}!";
Object[] arguments = {"John", new Date(), 1234.56};
```

When we format this pattern with the `arguments` array, here's how it works:

- `{0}` is replaced with `John`.
- `{1,date,long}` is replaced with the current date in long format (for example, `July 25, 2024`).
- `{2,number,currency}` is replaced with the currency formatted number (for example, `$1,234.56`).
- The second occurrence of `{0}` is also replaced with `John`.

Here is the resulting formatted message:

```
"Hello, John. Today is July 25, 2024. Your balance is $1,234.56. Hello again, John!"
```

About the format types and styles : - If a format type is not specified, a default based on the argument type is used (for example, numbers use `NumberFormat`). - Custom format types can be defined by passing a subformat pattern string, for example, `{0,number,#,##0.0}`. - The `formatStyle` is optional and can specify a predefined style like `short`, `long`, `integer`, `currency`, etc.

Consider the following message format pattern and arguments:

```
String pattern = "The item costs {0}. The item costs {0,number,currency}. Custom format: {0,number,#,##0.0}. Today
Object[] arguments = {1234.567, new Date()};
```

When formatted with the `arguments` array, it works as follows:

- `{0}` without a format type defaults to a number format (for example, `1234.567`).
- `{0,number,currency}` specifies the `number` format type with the `currency` style (for example, `$1,234.57`).
- `{0,number,#,##0.0}` uses a custom number format pattern (for example, `1,234.6`).
- `{1,date,long}` specifies the `date` format type with the `long` style (for example, `July 25, 2024`).
- `{1,date,short}` specifies the `date` format type with the `short` style (for example, `7/25/24`).

Here is the resulting formatted message:

```
"The item costs 1234.567. The item costs $1,234.57. Custom format: 1,234.6. Today is July 25, 2024. Short date: 7/25/24"
```

In addition to `MessageFormat`, there are two other subclasses of `Format`: `java.text.NumberFormat` and `java.text.DateFormat`, for formatting numbers and dates, respectively.

The format type and style values are used to create a `Format` instance for the format element. The following table shows how the values map to `Format` instances. Combinations not shown in the table are illegal. A *SubformatPattern* must be a valid pattern string for the `Format` subclass used.

FormatType	FormatStyle	Subformat Created
(none)	*(none)*	`null`
`number`	*(none)*	`NumberFormat.getInstance(getLocale())`
	`integer`	`NumberFormat.getIntegerInstance(getLocale())`
	`currency`	`NumberFormat.getCurrencyInstance(getLocale())`
	`percent`	`NumberFormat.getPercentInstance(getLocale())`
	SubformatPattern	`new DecimalFormat(subformatPattern, DecimalFormatSym-bols.getInstance(getLocale()))`
`date`	*(none)*	`DateFormat.getDateInstance(DateFormat.DEFAULT, getLocale())`
	`short`	`DateFormat.getDateInstance(DateFormat.SHORT, getLocale())`
	`medium`	`DateFormat.getDateInstance(DateFormat.DEFAULT, getLocale())`
	`long`	`DateFormat.getDateInstance(DateFormat.LONG, getLocale())`
	`full`	`DateFormat.getDateInstance(DateFormat.FULL, getLocale())`
	SubformatPattern	`new SimpleDateFormat(subformatPattern, getLocale())`
`time`	*(none)*	`DateFormat.getTimeInstance(DateFormat.DEFAULT, getLocale())`
	`short`	`DateFormat.getTimeInstance(DateFormat.SHORT, getLocale())`

FormatType	FormatStyle	Subformat Created
	`medium`	`DateFormat.getTimeInstance(DateFormat.DEFAULT, getLocale())`
	`long`	`DateFormat.getTimeInstance(DateFormat.LONG, getLocale())`
	`full`	`DateFormat.getTimeInstance(DateFormat.FULL, getLocale())`
	SubformatPattern	`new SimpleDateFormat(subformatPattern, getLocale())`
`choice`	*SubformatPattern*	`new ChoiceFormat(subformatPattern)`

In the next sections, we'll review in more detail the `NumberFormat` and `DateFormat` classes.

The `NumberFormat` Class

`NumberFormat` is the abstract base class for formatting and parsing numbers in Java. It provides a way to handle numeric values in a locale-sensitive manner, allowing you to format and parse numbers, currencies, and percentages according to the conventions of different locales.

To get a `NumberFormat` instance for a specific locale, you typically use one of its factory methods:

```
NumberFormat defaultFormat = NumberFormat.getInstance();
NumberFormat currencyFormat = NumberFormat.getCurrencyInstance();
NumberFormat percentFormat = NumberFormat.getPercentInstance();
```

These methods return a locale-specific formatter based on the default `FORMAT` locale. You can also specify a locale explicitly:

```
Locale franceLocale = new Locale("fr", "FR");
NumberFormat franceFormat = NumberFormat.getInstance(franceLocale);
```

To format a number, use one of the `format()` methods:

```
double value = 1234.56;
String formattedValue = defaultFormat.format(value);  // "1,234.56"
```

`NumberFormat` also provides methods for formatting `long` and `double` values directly into a `StringBuffer`, which can be more efficient for heavy-duty formatting:

```
abstract StringBuffer format(double number, StringBuffer toAppendTo, FieldPosition pos)
abstract StringBuffer format(long number, StringBuffer toAppendTo, FieldPosition pos)
```

On the other hand, to parse a string into a number, use the `parse()` method:

```
String text = "1,234.56";
Number number = defaultFormat.parse(text);  // Returns a Long or Double
```

The `parse()` method is locale-sensitive and will recognize the locale-specific decimal and grouping separators.

`NumberFormat` also provides several methods to control the formatting output:

- `setMinimumIntegerDigits()` and `setMaximumIntegerDigits()`: Controls the number of digits in the integer part.
- `setMinimumFractionDigits()` and `setMaximumFractionDigits()`: Controls the number of digits in the fraction part.
- `setGroupingUsed()`: Enables or disables grouping separators.
- `setRoundingMode()`: Sets the rounding mode for the formatter.

For more control, `NumberFormat` has several concrete subclasses for specific formatting needs:

- `DecimalFormat`: A general-purpose formatter for decimal numbers.
- `CompactNumberFormat`: Formats numbers in a compact, locale-sensitive way, like "1.2K" for 1200.
- `ChoiceFormat`: Allows you to map numbers to strings based on a set of ranges.

The `DecimalFormat` Class

The most commonly used subclass is `DecimalFormat`, which provides a high degree of control over the formatting pattern. You can create a `DecimalFormat` with a specific pattern and symbols:

```
DecimalFormat format = new DecimalFormat("#,##0.00", DecimalFormatSymbols.getInstance(Locale.US));
```

The second parameter (`DecimalFormatSymbols`) is optional. The pattern specifies the format of the output, with special characters representing the digit positions, decimals, grouping, etc. The `DecimalFormatSymbols` object defines the specific characters to use for these special characters based on a locale.

A `DecimalFormat` pattern contains a positive and negative subpattern, for example, `#,##0.00;(#,##0.00)`. Each subpattern has a prefix, numeric part, and suffix. The negative subpattern is optional, if absent, then the positive subpattern prefixed with the minus sign (`'-' U+002D HYPHEN-MINUS`) is used as the negative subpattern.

Many characters in a pattern are taken literally, they are matched during parsing and are unchanged in the output during formatting. On the other hand, special characters stand for other characters, strings, or classes of characters and they must be quoted.

The characters listed in the following table are used in non-localized patterns. Localized patterns use the corresponding characters taken from this formatter's `DecimalFormatSymbols` object instead, and these characters lose their special status. Two exceptions are the currency sign and quote, which are not localized:

Symbol	Location	Localized?	Meaning
`0`	Number	Yes	Digit
`#`	Number	Yes	Digit, zero shows as absent
`.`	Number	Yes	Decimal separator or monetary decimal separator
`-`	Number	Yes	Minus sign
`,`	Number	Yes	Grouping separator or monetary grouping separator
`E`	Number	Yes	Separates mantissa and exponent in scientific notation. *Need not be quoted in prefix or suffix.*
`;`	Subpattern boundary	Yes	Separates positive and negative subpatterns
`%`	Prefix or suffix	Yes	Multiply by 100 and show as percentage
`&#92;u2030`	Prefix or suffix	Yes	Multiply by 1000 and show as per mille value
`&#164;` (`&#92;u00A4`)	Prefix or suffix	No	Currency sign, replaced by currency symbol. If doubled, replaced by international currency symbol. If present in a pattern, the monetary decimal and grouping separators are used instead of the decimal and grouping separators.
`'`	Prefix or suffix	No	Used to quote special characters in a prefix or suffix, for example, `" '#' "` formats 123 to `"#123"`. To create a single quote itself, use two in a row: `"# o''clock"`.

Here are some examples:

```
// Number Formatting
DecimalFormat numberFormat = new DecimalFormat("###,###.##");
System.out.println("Number Formatting: " + numberFormat.format(1234567.89));
```

```java
// Percentage Formatting
DecimalFormat percentFormat = new DecimalFormat("##.##%");
System.out.println("Percentage Formatting: " + percentFormat.format(0.1234));

// Per Mille Formatting
DecimalFormat perMilleFormat = new DecimalFormat("##.##‰");
System.out.println("Per Mille Formatting: " + perMilleFormat.format(0.1234));

// Currency Formatting
DecimalFormat currencyFormat = new DecimalFormat("$###,###.##");
System.out.println("Currency Formatting: " + currencyFormat.format(1234.56));

// Scientific Notation
DecimalFormat scientificFormat = new DecimalFormat("0.###E0");
System.out.println("Scientific Notation: " + scientificFormat.format(12345));

// Positive and Negative Subpatterns
DecimalFormat positiveNegativeFormat = new DecimalFormat("###.##;(#.##)");
System.out.println("Positive Number: " + positiveNegativeFormat.format(1234.56));
System.out.println("Negative Number: " + positiveNegativeFormat.format(-1234.56));

// Custom Text with Numbers
DecimalFormat customTextFormat = new DecimalFormat("'Number: '###");
System.out.println("Custom Text with Numbers: " + customTextFormat.format(123));

// Custom Grouping and Decimal Separators
DecimalFormat customGroupingFormat = new DecimalFormat("'Amount: '###,###.##");
System.out.println("Custom Grouping and Decimal Separators: " + customGroupingFormat.format(1234567.89));

//  Quoting Special Characters
DecimalFormat quotingSpecialFormat = new DecimalFormat("''#''###");
System.out.println("Quoting Special Characters: " + quotingSpecialFormat.format(123));

// Integer Formatting
DecimalFormat integerFormat = new DecimalFormat("###");
System.out.println("Integer Formatting: " + integerFormat.format(1234.56));
```

This is the output:

```
Number Formatting: 1,234,567.89
Percentage Formatting: 12.34%
Per Mille Formatting: 123.4‰
Currency Formatting: $1,234.56
Scientific Notation: 1.234E4
Positive Number: 1234.56
Negative Number: (1234.56)
Custom Text with Numbers: Number: 123
Custom Grouping and Decimal Separators: Amount: 1,234,567.89
Quoting Special Characters: '123'
Integer Formatting: 1235
```

The `CompactNumberFormat` Class

`CompactNumberFormat` provides support for compact number formatting. This format is particularly useful for displaying large numbers in a more readable, locale-sensitive way.

This class supports two styles:

1. `NumberFormat.Style.SHORT`: Uses abbreviations (for example, `K` for thousand, `M` for million, `B` for billion, and so on)
2. `NumberFormat.Style.LONG`: Uses full words (for example, "thousand", "million")

To get a `CompactNumberFormat` instance, you use the `getCompactNumberInstance()` factory method:

```
NumberFormat shortFormat = NumberFormat.getCompactNumberInstance(Locale.US, NumberFormat.Style.SHORT);
NumberFormat longFormat = NumberFormat.getCompactNumberInstance(Locale.US, NumberFormat.Style.LONG);
```

Here are some examples of how `CompactNumberFormat` works:

```
double number = 1_234_567.89;

System.out.println(shortFormat.format(number));  // Output: 1M
System.out.println(longFormat.format(number));   // Output: 1 million

number = 1_234;
System.out.println(shortFormat.format(number));  // Output: 1K
System.out.println(longFormat.format(number));   // Output: 1 thousand
```

`CompactNumberFormat` automatically adjusts the number of displayed digits based on the magnitude of the number. It also handles different locales appropriately:

```
NumberFormat frFormat = NumberFormat.getCompactNumberInstance(Locale.FRANCE, NumberFormat.Style.SHORT);
System.out.println(frFormat.format(1_234_567.89));  // Output: 1 M
```

By default, `CompactNumberFormat` doesn't display fractional digits. However, you can modify this behavior using `setMinimumFractionDigits()` and `setMaximumFractionDigits()` methods.

Also, the class also supports various rounding modes by using `setRoundingMode` and the `java.math.RoundingMode` enum, with `RoundingMode.HALF_EVEN` as the default:

Enum Constant	Description
`CEILING`	Rounding mode to round towards positive infinity.
`DOWN`	Rounding mode to round towards zero.
`FLOOR`	Rounding mode to round towards negative infinity.
`HALF_DOWN`	Rounding mode to round towards "nearest neighbor" unless both neighbors are equidistant, in which case round down.
`HALF_EVEN`	Rounding mode to round towards the "nearest neighbor" unless both neighbors are equidistant, in which case, round towards the even neighbor.
`HALF_UP`	Rounding mode to round towards "nearest neighbor" unless both neighbors are equidistant, in which case round up.
`UNNECESSARY`	Rounding mode to assert that the requested operation has an exact result, hence no rounding is necessary.
`UP`	Rounding mode to round away from zero.

Here are some examples:

```
double number = 1_234_567.89;
NumberFormat shortFormat = NumberFormat.getCompactNumberInstance(Locale.US, NumberFormat.Style.SHORT);
```

```java
// Default behavior (no fractional digits)
System.out.println(shortFormat.format(number));  // Output: 1M

// Adding fractional digits
shortFormat.setMinimumFractionDigits(1);
shortFormat.setMaximumFractionDigits(2);
System.out.println(shortFormat.format(number));  // Output: 1.23M

// Changing rounding mode
shortFormat.setRoundingMode(RoundingMode.DOWN);
System.out.println(shortFormat.format(number));  // Output: 1.23M

shortFormat.setRoundingMode(RoundingMode.UP);
System.out.println(shortFormat.format(number));  // Output: 1.24M

// Behavior with smaller numbers
number = 1234.56;
System.out.println(shortFormat.format(number));  // Output: 1.24K

// Resetting to default behavior
shortFormat = NumberFormat.getCompactNumberInstance(Locale.US, NumberFormat.Style.SHORT);
System.out.println(shortFormat.format(number));  // Output: 1K
```

The `DateTimeFormatter` Class

In the The Date API chapter, we covered the `java.time.format.DateTimeFormatter` class, which provides a flexible and modern way to format dates and times represented by `java.time` classes like `LocalDate`, `LocalTime`, and `LocalDateTime`.

However, one of the key features of `DateTimeFormatter` is its ability to create localized formatters using the `ofLocalized...()` methods:

- `static DateTimeFormatter ofLocalizedDate(FormatStyle dateStyle)`: Creates a formatter that formats a date in the specified style for the current locale.
- `static DateTimeFormatter ofLocalizedTime(FormatStyle timeStyle)`: Creates a formatter that formats a time in the specified style for the current locale.
- `static DateTimeFormatter ofLocalizedDateTime(FormatStyle dateStyle, FormatStyle timeStyle)`: Creates a formatter that formats a date and time in the specified styles for the current locale.
- `static DateTimeFormatter ofLocalizedDateTime(FormatStyle dateTimeStyle)`: Creates a formatter that formats a date and time in the specified combined style for the current locale.

The `FormatStyle` enum defines the available styles: `SHORT`, `MEDIUM`, `LONG`, and `FULL`.

Here's an example of using these methods:

```java
LocalDate date = LocalDate.now();
LocalTime time = LocalTime.now();
LocalDateTime dateTime = LocalDateTime.now();

DateTimeFormatter dateFormatter = DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL);
DateTimeFormatter timeFormatter = DateTimeFormatter.ofLocalizedTime(FormatStyle.MEDIUM);
DateTimeFormatter dateTimeFormatter = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.LONG, FormatStyle.SHORT);

String formattedDate = date.format(dateFormatter);
String formattedTime = time.format(timeFormatter);
String formattedDateTime = dateTime.format(dateTimeFormatter);
```

```
System.out.println("Formatted date: " + formattedDate);
System.out.println("Formatted time: " + formattedTime);
System.out.println("Formatted date-time: " + formattedDateTime);
```

The `ofLocalized...()` methods automatically use the current locale to determine the appropriate formatting. If you want to specify a different locale, you can use the `withLocale()` method:

```
DateTimeFormatter frenchDateFormatter = DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL).withLocale(Locale.FRAN
String formattedDateInFrench = date.format(frenchDateFormatter);
System.out.println("Formatted date in French: " + formattedDateInFrench);
```

These localized formatters provide an easy way to format dates and times according to the conventions of a specific locale without having to specify the format pattern manually.

You can also use a pattern to create a `DateTimeFormatter` instance using the `ofPattern(String)` and `ofPattern(String, Locale)` methods. For example, `"d MMM uuuu"` will format `2011-12-03` as `'3 Dec 2011'`. A formatter created from a pattern can be used as many times as necessary. It is immutable and thread-safe.

Here's an example:

```
LocalDate date = LocalDate.now();
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yy MM dd");

// Format the date
String text = date.format(formatter);
System.out.println("Formatted date: " + text);

// Parse the date
LocalDate parsedDate = LocalDate.parse(text, formatter);
System.out.println("Parsed date: " + parsedDate);
```

Here's a sample output:

```
Formatted date: 24 07 25
Parsed date: 2024-07-25
```

The following table shows all the pattern letters defined (all letters 'A' to 'Z' and 'a' to 'z' are reserved):

Symbol	Meaning	Presentation	Examples
`G`	era	text	`AD`; `Anno Domini`; `A`
`u`	year	year	`2004`; `04`
`y`	year-of-era	year	`2004`; `04`
`D`	day-of-year	number	`189`
`M/L`	month-of-year	number/text	`7`; `07`; `Jul`; `July`; `J`
`d`	day-of-month	number	`10`
`g`	modified-julian-day	number	`2451334`
`Q/q`	quarter-of-year	number/text	`3`; `03`; `Q3`; `3rd quarter`
`Y`	week-based-year	year	`1996`; `96`
`w`	week-of-week-based-year	number	`27`
`W`	week-of-month	number	`4`
`E`	day-of-week	text	`Tue`; `Tuesday`; `T`
`e/c`	localized day-of-week	number/text	`2`; `02`; `Tue`; `Tuesday`; `T`
`F`	day-of-week-in-month	number	`3`
`a`	am-pm-of-day	text	`PM`
`B`	period-of-day	text	`in the morning`
`h`	clock-hour-of-am-pm (1-12)	number	`12`
`K`	hour-of-am-pm (0-11)	number	`0`

Symbol	Meaning	Presentation	Examples
k	clock-hour-of-day (1-24)	number	24
H	hour-of-day (0-23)	number	0
m	minute-of-hour	number	30
s	second-of-minute	number	55
S	fraction-of-second	fraction	978
A	milli-of-day	number	1234
n	nano-of-second	number	987654321
N	nano-of-day	number	1234000000
V	time-zone ID	zone-id	America/Los_Angeles; Z; -08:30
v	generic time-zone name	zone-name	Pacific Time; PT
z	time-zone name	zone-name	Pacific Standard Time; PST
O	localized zone-offset	offset-O	GMT+8; GMT+08:00; UTC-08:00
X	zone-offset 'Z' for zero	offset-X	Z; -08; -0830; -08:30; -083015; -08:30:15
x	zone-offset	offset-x	+0000; -08; -0830; -08:30; -083015; -08:30:15
Z	zone-offset	offset-Z	+0000; -0800; -08:00
p	pad next	pad modifier	1
'	escape for text	delimiter	
''	single quote	literal	'
[	optional section start		
]	optional section end		
#	reserved for future use		
{	reserved for future use		
}	reserved for future use		

Here are examples using many of those patterns:

```
LocalDate date = LocalDate.now();
LocalTime time = LocalTime.now();
ZonedDateTime zonedDateTime = ZonedDateTime.now();

// Era
DateTimeFormatter formatter1 = DateTimeFormatter.ofPattern("G");
System.out.println("Era: " + date.format(formatter1));

// Year
DateTimeFormatter formatter2 = DateTimeFormatter.ofPattern("yyyy");
System.out.println("Year: " + date.format(formatter2));

// Day of Year
DateTimeFormatter formatter3 = DateTimeFormatter.ofPattern("D");
System.out.println("Day of Year: " + date.format(formatter3));

// Month of Year
DateTimeFormatter formatter4 = DateTimeFormatter.ofPattern("MMMM");
System.out.println("Month of Year: " + date.format(formatter4));

// Day of Month
DateTimeFormatter formatter5 = DateTimeFormatter.ofPattern("d");
System.out.println("Day of Month: " + date.format(formatter5));

// Day of Week
DateTimeFormatter formatter6 = DateTimeFormatter.ofPattern("EEEE");
```

```
System.out.println("Day of Week: " + date.format(formatter6));

// AM/PM of Day
DateTimeFormatter formatter7 = DateTimeFormatter.ofPattern("a");
System.out.println("AM/PM of Day: " + time.format(formatter7));

// Hour of Day (0-23)
DateTimeFormatter formatter8 = DateTimeFormatter.ofPattern("H");
System.out.println("Hour of Day (0-23): " + time.format(formatter8));

// Minute of Hour
DateTimeFormatter formatter9 = DateTimeFormatter.ofPattern("m");
System.out.println("Minute of Hour: " + time.format(formatter9));

// Second of Minute
DateTimeFormatter formatter10 = DateTimeFormatter.ofPattern("s");
System.out.println("Second of Minute: " + time.format(formatter10));

// Time Zone Name
DateTimeFormatter formatter11 = DateTimeFormatter.ofPattern("z");
System.out.println("Time Zone Name: " + zonedDateTime.format(formatter11));

// ISO 8601 Time Zone
DateTimeFormatter formatter12 = DateTimeFormatter.ofPattern("X");
System.out.println("ISO 8601 Time Zone: " + zonedDateTime.format(formatter12));
```

The output should be similar to this:

```
Era: AD
Year: 2024
Day of Year: 207
Month of Year: July
Day of Month: 25
Day of Week: Thursday
AM/PM of Day: PM
Hour of Day (0-23): 20
Minute of Hour: 31
Second of Minute: 16
Time Zone Name: CDT
ISO 8601 Time Zone: -05
```

For the Java certification exam, you don't need to memorize every single pattern, but you should be familiar with the key ones that are likely to appear on the exam. Here's a list of patterns and you should concentrate on: - **Era**: `G` - **Year of Era**: `y` - **Month**: `M` - **Day of Month**: `d` - **Hour (0-23)**: `H` - **Hour (1-12)**: `h` - **Minute**: `m` - **Second**: `s` - **AM/PM**: `a` - **Week-based-year**: `Y` - **Day-of-year**: `D` - **Day-of-week**: `E`, `e` - **Period-of-day**: `B` - **Fraction-of-second**: `S` - **Nano-of-second**: `n` - **Nano-of-day**: `N` - **Time Zone ID**: `V` - **Time Zone Name**: `z` - **ISO 8601 Time Zone**: `X` - **Localized Zone Offset**: `O`

Key Points

- Localization is the process of designing and developing an application so that it can be adapted to various locales without requiring engineering changes. A locale represents a specific geographical, political, or cultural region.
- Locales are represented by the `java.util.Locale` class. You can get the default locale of the Java Virtual Machine using `Locale.getDefault()`, use built-in constants like `Locale.US`, or create a new

`Locale` object using a constructor or the `forLanguageTag()` method.

- The `Locale.Category` enum defines two categories: `DISPLAY` (for user interface elements) and `FORMAT` (for parsing and formatting data). You can set and get the default locale for each category using `Locale.setDefault()` and `Locale.getDefault()`.
- Resource bundles are used to organize and access locale-specific data. They are typically implemented as property files named in the format `<basename>_<language>_<country>_<variant>.properties`.
- The `java.util.ResourceBundle` class is used to load the appropriate property file for the current locale. You can retrieve locale-specific values using `getString()` or `getObject()`.
- The `java.text.MessageFormat` class is used to create localized messages by combining a pattern string with arguments. The pattern includes placeholders marked with `{}` and argument indexes. Format types (like `number` or `date`) and styles can be specified.
- The `java.text.Format` class has subclasses to format arguments based on their type and the specified format style. The main subclasses are `java.text.NumberFormat` for numbers and `java.text.DateFormat` for dates.
- `java.text.NumberFormat` is the abstract base class for formatting and parsing numbers in Java. It provides factory methods for getting locale-specific formatters for numbers, currencies, and percentages.
- `java.text.DecimalFormat` is a concrete subclass of `NumberFormat` that allows for more detailed control over the formatting of decimal numbers using patterns. The pattern can include special characters representing digit positions, decimals, grouping, etc.
- `java.time.format.DateTimeFormatter` is a class for formatting dates and times from the `java.time` package. It works with the modern date/time classes like `LocalDate`, `LocalTime`, `LocalDateTime`.
- `DateTimeFormatter` provides `ofLocalized...()` methods for creating locale-specific formatters in different styles, similar to `DateFormat`. It also allows creating formatters from custom patterns using `ofPattern()`.
- For the certification exam, focus on common patterns like era (`G`), year of era (`y`), month (`M`), day of month (`d`), hours (`H`, `h`), minutes (`m`), seconds (`s`), AM/PM (`a`), among others.

Practice Questions

1. Consider the following code snippet:

```
import java.util.Locale;

public class LocaleTest {
    public static void main(String[] args) {
        Locale locale1 = new Locale("fr", "CA");
        Locale locale2 = new Locale("fr", "CA", "UNIX2024");
        Locale locale3 = Locale.CANADA_FRENCH;

        System.out.println(locale1.equals(locale2));
        System.out.println(locale1.equals(locale3));
        System.out.println(locale2.equals(locale3));

        System.out.println(locale1.getDisplayName(Locale.ENGLISH));
        System.out.println(locale2.getDisplayName(Locale.ENGLISH));
        System.out.println(locale3.getDisplayName(Locale.ENGLISH));
    }
}
```

What will be the output when this code is executed?

A)

```
true
true
true
French (Canada)
French (Canada)
French (Canada)
```

B)

```
false
true
false
French (Canada)
French (Canada, UNIX2024)
French (Canada)
```

C)

```
false
true
false
French (Canada)
French (Canada, UNIX2024)
Canadian French
```

D)

```
false
false
false
French (Canada)
French (Canada, UNIX2024)
Canadian French
```

E) The code will throw a `IllegalArgumentException` because `UNIX2024` is not a valid variant.

2. Which of the following statements about `Locale` categories is correct?

A) The `Locale.Category` enum has three values: `DISPLAY`, `FORMAT`, and `LANGUAGE`.
B) The `Locale.setDefault(Locale.Category, Locale)` method can only set the default locale for the `FORMAT` category.
C) Using `Locale.getDefault(Locale.Category)` always returns the same locale regardless of the category specified.
D) The `DISPLAY` category affects the language used for displaying user interface elements, while the `FORMAT` category affects the formatting of numbers, dates, and currencies.
E) Locale categories were introduced in Java 8 to replace the older `Locale` methods.

3. Which of the following statements about Resource Bundles is correct?

A) Resource bundles can only be stored in `.properties` files.
B) The `ResourceBundle.getBundle()` method always throws a `MissingResourceException` if the requested bundle is not found.
C) When searching for a resource bundle, Java only considers the specified locale and its language.
D) If a key is not found in a specific locale's resource bundle, Java will look for it in the parent locale's bundle.
E) Resource bundles are loaded dynamically at runtime, so changes to `.properties` files are immediately reflected in the running application.

4. Consider the following code snippet:

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

public class ConfigTest {
    public static void main(String[] args) throws IOException {
        Properties props = new Properties();
        props.setProperty("color", "blue");
        props.setProperty("size", "medium");

        try (OutputStream out = new FileOutputStream("config.properties")) {
            props.store(out, "Config File");
        }

        props.clear();
        System.out.println(props.getProperty("color", "red"));

        try (InputStream in = new FileInputStream("config.properties")) {
            props.load(in);
        }

        System.out.println(props.getProperty("color", "red"));
    }
}
```

What will be the output when this code is executed?

A)

```
red
blue
```

B)

```
blue
blue
```

C)

```
red
red
```

D) The code will throw a `FileNotFoundException`.

E)

```
null
blue
```

5. Consider the following code snippet:

```
import java.text.MessageFormat;
import java.util.Date;
import java.util.Locale;

public class MessageFormatTest {
    public static void main(String[] args) {
        String pattern = "On {0, date, long}, {1} bought {2,number,integer} {3} for {4,number,currency}.";
        Object[] params = {
```

```
            new Date(),
            "Alice",
            3,
            "apples",
            19.99
        };

        MessageFormat mf = new MessageFormat(pattern, Locale.US);
        String result = mf.format(params);
        System.out.println(result);
    }
}
```

Which of the following statements about this code is correct?

A) The code will throw a `IllegalArgumentException` because the date format is invalid.
B) The output will include the date in long format, the name `"Alice"`, the number 3, the word `"apples"`, and the price in US currency format. **C)** The `{2,number,integer}` format will display 3 as `"3.0"`.
D) The code will not compile because `MessageFormat` doesn't accept a `Locale` in its constructor.
E) The `{4,number,currency}` format will always display the price in USD, regardless of the `Locale`.

6. Which of the following statements about the `NumberFormat` class in Java is correct?

A) The `NumberFormat.getCurrencyInstance()` method returns a formatter that can format monetary amounts according to the specified locale's conventions.
B) `NumberFormat` is a concrete class that can be instantiated directly using its constructor.
C) The `setMaximumFractionDigits()` method in `NumberFormat` can only accept values between 0 and 3.
D) When parsing strings, `NumberFormat` always throws a `ParseException` if the input doesn't exactly match the expected format.
E) The `NumberFormat` class can only format and parse integer values, not floating-point numbers.

7. Consider the following code snippet:

```
import java.time.LocalDateTime;
import java.time.ZoneId;
import java.time.ZonedDateTime;
import java.time.format.DateTimeFormatter;

public class DateTimeFormatterTest {
    public static void main(String[] args) {
        LocalDateTime ldt = LocalDateTime.of(2023, 6, 15, 10, 30);
        ZoneId zoneNY = ZoneId.of("America/New_York");
        ZonedDateTime zdtNY = ldt.atZone(zoneNY);

        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm z VV");
        System.out.println(formatter.format(zdtNY));

        ZoneId zoneTokyo = ZoneId.of("Asia/Tokyo");
        ZonedDateTime zdtTokyo = zdtNY.withZoneSameInstant(zoneTokyo);
        System.out.println(formatter.format(zdtTokyo));
    }
}
```

What will be the output when this code is executed?

A)

```
2023-06-15 10:30 EDT America/New_York
```

```
2023-06-15 23:30 JST Asia/Tokyo
```

B)

```
2023-06-15 10:30 EDT New_York
2023-06-15 23:30 JST Tokyo
```

C)

```
2023-06-15 10:30 -04:00 America/New_York
2023-06-15 23:30 +09:00 Asia/Tokyo
```

D)

```
2023-06-15 10:30 America/New_York
2023-06-15 23:30 Asia/Tokyo
```

E) The code will throw a `DateTimeException` because the formatter pattern is invalid.

Chapter FOURTEEN

Localization

Answers

1. The correct answer is C.

Explanation:

- **A)**

```
true
true
true
French (Canada)
French (Canada)
French (Canada)
```

- This option is incorrect because it doesn't account for the differences caused by the variant in `locale2`.
- **B)**

```
false
true
false
French (Canada)
French (Canada, UNIX2024)
French (Canada)
```

- This option is incorrect because it doesn't correctly represent the display name for `Locale.CANADA_FRENCH`.
- **C)**

```
false
true
false
French (Canada)
French (Canada, UNIX2024)
Canadian French
```

- This option is correct. Let's break it down:

1. `locale1.equals(locale2)` is `false` because `locale2` has a variant (`"UNIX2024"`) while `locale1` doesn't.
2. `locale1.equals(locale3)` is `true` because `Locale.CANADA_FRENCH` is equivalent to `new Locale("fr", "CA")`.
3. `locale2.equals(locale3)` is `false` because `locale2` has a variant while `locale3` doesn't.
4. `locale1.getDisplayName(Locale.ENGLISH)` returns `"French (Canada)"`.
5. `locale2.getDisplayName(Locale.ENGLISH)` returns `"French (Canada, UNIX2024)"`, including the variant.
6. `locale3.getDisplayName(Locale.ENGLISH)` returns `"Canadian French"`, which is the special display name for this constant.

- **D)**

```
false
false
false
French (Canada)
French (Canada, UNIX2024)
Canadian French
```

- This option is incorrect because it suggests that `locale1` and `locale3` are not equal, which they are.
- **E)** The code will throw a `IllegalArgumentException` because `UNIX2024` is not a valid variant.
  - This option is incorrect. While `UNIX2024` is not a standard ISO 639 variant code, the `Locale` constructor accepts any string as a variant without throwing an exception.

2. The correct answer is D.

Explanation:

- **A)** The `Locale.Category` enum has three values: `DISPLAY`, `FORMAT`, and `LANGUAGE`.
  - This option is incorrect. The `Locale.Category` enum has only two values: `DISPLAY` and `FORMAT`. There is no `LANGUAGE` category.
- **B)** The `Locale.setDefault(Locale.Category, Locale)` method can only set the default locale for the `FORMAT` category.
  - This option is incorrect. The `Locale.setDefault(Locale.Category, Locale)` method can set the default locale for both the `DISPLAY` and `FORMAT` categories, not just `FORMAT`.
- **C)** Using `Locale.getDefault(Locale.Category)` always returns the same locale regardless of the category specified.
  - This option is incorrect. `Locale.getDefault(Locale.Category)` can return different locales depending on the category specified. The `DISPLAY` and `FORMAT` categories can have different default locales.
- **D)** The `DISPLAY` category affects the language used for displaying user interface elements, while the `FORMAT` category affects the formatting of numbers, dates, and currencies.
  - This option is correct. The `DISPLAY` category indeed affects the language used for displaying user interface elements (like error messages or GUI labels), while the `FORMAT` category affects how numbers, dates, currencies, and other locale-sensitive data are formatted.
- **E)** Locale categories were introduced in Java 8 to replace the older `Locale` methods.
  - This option is incorrect. Locale categories were added to provide more granular control over localization aspects, complementing (not replacing) the existing `Locale` methods.

3. The correct answer is D.

Explanation:

- **A)** Resource bundles can only be stored in `.properties` files.
  - This option is incorrect. While `.properties` files are commonly used for resource bundles, Java also supports class-based resource bundles. These are Java classes that extend `ResourceBundle`

and provide localized resources programmatically.

- **B)** The `ResourceBundle.getBundle()` method always throws a `MissingResourceException` if the requested bundle is not found.
  - This option is incorrect. The `ResourceBundle.getBundle()` method does not always throw a `MissingResourceException` if the requested bundle is not found. It follows a fallback mechanism, trying to find the most specific bundle, then falling back to more general bundles, and finally to the default bundle.
- **C)** When searching for a resource bundle, Java only considers the specified locale and its language.
  - This option is incorrect. When searching for a resource bundle, Java considers not only the specified locale and its language but also the country, variant, and even the default locale. It follows a well-defined lookup procedure to find the most appropriate bundle.
- **D)** If a key is not found in a specific locale's resource bundle, Java will look for it in the parent locale's bundle.
  - This option is correct. Java implements a parent chain fallback mechanism for resource bundles. If a key is not found in the specific locale's bundle, it will look in the parent locale's bundle. For example, if a key is not found in a `fr_FR` (French France) bundle, it will look in the `fr` (French) bundle, and then in the default bundle.
- **E)** Resource bundles are loaded dynamically at runtime, so changes to `.properties` files are immediately reflected in the running application.
  - This option is incorrect. Resource bundles are typically loaded when `ResourceBundle.getBundle()` is called and then cached. Changes to .properties files are not immediately reflected in a running application. The application usually needs to be restarted or the resource bundle cache cleared for changes to take effect.

4. The correct answer is A.

Explanation:

- **A)**

```
red
blue
```

- This option is correct. Let's break down the code execution:
  1. Properties are set and stored in the `config.properties` file.
  2. `props.clear()` removes all properties from the `props` object.
  3. `System.out.println(props.getProperty("color", "red"))` prints `red` because the properties have been cleared, so it uses the default value.
  4. The properties are loaded from the file.
  5. `System.out.println(props.getProperty("color", "red"))` now prints `blue` because it's loaded from the file.
- **B)**

```
blue
blue
```

- This option is incorrect. It doesn't account for the `clear()` method call which empties the properties before the first print statement.
- **C)**

```
red
red
```

- This option is incorrect. It doesn't account for the successful loading of properties from the file before the second print statement.
- **D)** The code will throw a `FileNotFoundException`.

  - This option is incorrect. The code creates the file in the first `try-with-resources` block, so it should exist for the second block to read from.

- **E)**

```
null
blue
```

- This option is incorrect. `getProperty()` returns the default value `red` when the property is not found, not `null`.

5. The correct answer is B.

Explanation:

- **A)** The code will throw a `IllegalArgumentException` because the date format is invalid.
  - This option is incorrect. The date format `"long"` is valid in `MessageFormat` and will not throw an exception.
- **B)** The output will include the date in long format, the name `"Alice"`, the number 3, the word `"apples"`, and the price in US currency format.
  - This option is correct. The `MessageFormat` will correctly format each parameter according to the specified pattern:
  - `{0, date, long}` will format the `Date` in long format (`"June 1, 2023"`)
  - `{1}` will simply insert `"Alice"`
  - `{2,number,integer}` will format 3 as an integer
  - `{3}` will insert "apples"
  - `{4,number,currency}` will format 19.99 as currency according to the US locale (`"$19.99"`)
- **C)** The `{2,number,integer}` format will display 3 as `"3.0"`.
  - This option is incorrect. The `{2,number,integer}` format will display 3 as `"3"`, not `"3.0"`. The integer format doesn't include decimal places.
- **D)** The code will not compile because `MessageFormat` doesn't accept a `Locale` in its constructor.
  - This option is incorrect. `MessageFormat` does have a constructor that accepts a `Locale`. The code will compile successfully.
- **E)** The `{4,number,currency}` format will always display the price in USD, regardless of the `Locale`.
  - This option is incorrect. The currency format will use the locale specified in the `MessageFormat` constructor, which in this case is `Locale.US`. If a different locale were used, the currency symbol and formatting could change.

6. The correct answer is A.

Explanation:

- **A)** The `NumberFormat.getCurrencyInstance()` method returns a formatter that can format monetary amounts according to the specified locale's conventions.
  - This option is correct. The `getCurrencyInstance()` method of `NumberFormat` returns a currency formatter for the specified locale (or the default locale if none is specified). This formatter applies the appropriate currency symbol, digit grouping, and decimal separator according to the locale's conventions.
- **B)** `NumberFormat` is a concrete class that can be instantiated directly using its constructor.
  - This option is incorrect. `NumberFormat` is an abstract class and cannot be instantiated directly. Instead, you obtain instances through its static factory methods like `getInstance()`, `getCurrencyInstance()`, or `getPercentInstance()`.
- **C)** The `setMaximumFractionDigits()` method in `NumberFormat` can only accept values between 0 and 3.
  - This option is incorrect. The `setMaximumFractionDigits()` method is not limited to the range of 0 to 3.
- **D)** When parsing strings, `NumberFormat` always throws a `ParseException` if the input doesn't exactly match the expected format.

  - This option is incorrect. `NumberFormat` is generally lenient when parsing. It will attempt to parse as much of the string as it can recognize as a number, and will only throw a `ParseException` if it can't parse any part of the string as a number.
- **E)** The `NumberFormat` class can only format and parse integer values, not floating-point numbers.
  - This option is incorrect. `NumberFormat` can format and parse both integer and floating-point numbers. It provides methods like `setMaximumFractionDigits()` and `setMinimumFractionDigits()` specifically for handling decimal places in floating-point numbers.

7. The correct answer is A.

Explanation:

- **A)**

```
2023-06-15 10:30 EDT America/New_York
2023-06-15 23:30 JST Asia/Tokyo
```

- This option is correct. Let's break down the formatter pattern:
  - `"yyyy-MM-dd HH:mm"` formats the date and time
  - `"z"` outputs the short name of the zone, like EDT or JST
  - `"VV"` outputs the full time zone ID, like `America/New_York` or `Asia/Tokyo` The second line shows the correct time in Tokyo, which is 13 hours ahead of New York.
- **B)**

```
2023-06-15 10:30 EDT New_York
2023-06-15 23:30 JST Tokyo
```

- This option is incorrect. The `"VV"` pattern outputs the full time zone ID, not just the city name.
- **C)**

```
2023-06-15 10:30 -04:00 America/New_York
2023-06-15 23:30 +09:00 Asia/Tokyo
```

- This option is incorrect. The `"z"` pattern outputs the short name of the zone (EDT, JST), not the offset.
- **D)**

```
2023-06-15 10:30 America/New_York
2023-06-15 23:30 Asia/Tokyo
```

- This option is incorrect. It's missing the short zone names (EDT, JST) that the `"z"` pattern should output.
- **E)** The code will throw a `DateTimeException` because the formatter pattern is invalid.
  - This option is incorrect. The formatter pattern is valid and will not throw an exception.