Complete Java 21 Learning and Certification Guide

Table of Contents

Environment Setup and Basics

JDK Installation and Setup

Java development begins with installing the Java Development Kit (JDK). The JDK contains everything needed to develop, compile, and run Java applications.

Installing JDK 21

1. **Download JDK 21**:
   - Visit the official Oracle website (https://www.oracle.com/java/technologies/downloads/) or use OpenJDK (https://jdk.java.net/21/)
   - Select the appropriate version for your operating system
2. **Installation Steps**:
   - **Windows**: Run the installer and follow the prompts
   - **macOS**: Mount the DMG file and run the package installer
   - **Linux**: Use package manager or extract tarball

```
# Ubuntu/Debian
sudo apt update
sudo apt install openjdk-21-jdk

# Fedora/RHEL
sudo dnf install java-21-openjdk-devel
```

3. **Verify Installation**: Open a terminal/command prompt and run:

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

You should see version information confirming Java 21.

Setting Up Environment Variables

Environment variables ensure Java tools are accessible from anywhere in your system.

1. **JAVA_HOME** - Points to JDK installation directory
   - **Windows**:

     ```
     setx JAVA_HOME "C:\Program Files\Java\jdk-21"
     ```

   - **macOS/Linux** (add to ~/.bashrc, ~/.zshrc, or equivalent):

     ```
     export JAVA_HOME=/usr/lib/jvm/java-21-openjdk
     ```

2. **PATH** - Includes Java binaries
   - **Windows**:

     ```
     setx PATH "%PATH%;%JAVA_HOME%\bin"
     ```

   - **macOS/Linux**:

     ```
     export PATH=$PATH:$JAVA_HOME/bin
     ```

> **Certification Note**: Exams may test your knowledge of environment variables and their role in Java development. Remember that CLASSPATH is also important but shouldn't include the current directory (".") for production environments due to security reasons.

IDE Setup

While you can write Java code in any text editor, Integrated Development Environments (IDEs) provide powerful tools for development.

Popular Java IDEs:

1. **IntelliJ IDEA** (Recommended)
   - Download: https://www.jetbrains.com/idea/

- Available in Community (free) and Ultimate editions
- Installation: Run the installer and follow the prompts
- First-time setup:
    1. Select "New Project"
    2. Choose "Java" project
    3. Verify JDK 21 is selected (or add it if not listed)

2. **Eclipse**

- Download: https://www.eclipse.org/downloads/
- Choose "Eclipse IDE for Java Developers"
- Installation: Extract and run the Eclipse installer
- Configure for JDK 21:
    1. Window > Preferences > Java > Installed JREs
    2. Add JDK 21 and set as default
    3. Window > Preferences > Java > Compiler
    4. Set compliance level to 21

3. **VS Code with Java Extension Pack**

- Download VS Code: https://code.visualstudio.com/
- Install "Extension Pack for Java"
- Configure Java settings in settings.json

> **Common Pitfall**: Using an IDE configured for an older Java version can cause complications when using Java 21 features. Always verify your IDE is correctly configured for Java 21.

Understanding the Java Ecosystem

Key Components of the Java Platform

Java's architecture consists of several key components that work together to enable "write once, run anywhere" functionality:

1. **Java Virtual Machine (JVM)**

- Runtime environment that executes Java bytecode
- Platform-specific (different implementations for different operating systems)
- Handles memory management, garbage collection, and security
- Notable implementations: HotSpot (Oracle), OpenJ9 (Eclipse)

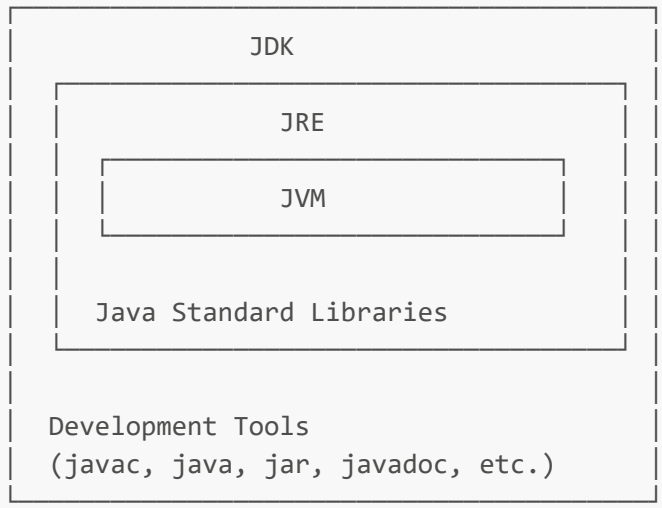
2. **Java Runtime Environment (JRE)**

- Contains the JVM
- Includes core libraries and other components needed to run Java applications
- Does not include development tools (compiler, debugger)
- End users only need the JRE to run Java applications

3. **Java Development Kit (JDK)**

- Complete development package

- Includes the JRE
- Contains development tools:
  - javac (compiler)
  - java (launcher)
  - jar (archiver)
  - javadoc (documentation generator)
  - jdb (debugger)



> **Certification Note**: Understand the distinction between these components. Know that since Java 11, the JRE is no longer distributed separately - it's part of the JDK.

How Java Works: From Source to Execution

1. **Write Source Code** (.java files)
2. **Compilation**: javac compiler converts source code to bytecode (.class files)
3. **Class Loading**: JVM loads the necessary classes
4. **Bytecode Verification**: Ensures bytecode adheres to JVM specifications (security)
5. **Just-In-Time (JIT) Compilation**: Converts frequently used bytecode to native machine code
6. **Execution**: JVM executes the program

```
Source Code (.java) → Bytecode (.class) → JVM → Execution
```

> **Technical Detail**: JIT compilation is why Java applications tend to run faster the longer they operate - more code gets optimized into native machine instructions.

Writing and Running Your First Java Program

Hello World Example

1. Create a file named `HelloWorld.java` with the following content:

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

        // Demonstrating a new Java 21 feature - String templates (preview)
        // String name = "Developer";
        // System.out.println(STR."Hello, \{name}!");
    }
}
```

2. **Compile** the program:

```
javac HelloWorld.java
```

This produces a `HelloWorld.class` file containing bytecode.

3. **Run** the program:

```
java HelloWorld
```

Understanding the Basic Structure

- `public class HelloWorld`: Class declaration, must match the filename
- `public static void main(String[] args)`: Entry point method
  - `public`: Accessible from anywhere
  - `static`: Belongs to the class, not an instance
  - `void`: Returns no value
  - `main`: Special method name recognized as the starting point
  - `String[] args`: Command-line arguments passed to the program

> **Common Pitfall**: Java is case-sensitive. `Main` is not the same as `main`.

> **Certification Note**: Know that a single Java file can contain multiple classes, but only one public class that must match the filename.

Package Structure and Organization

Java uses packages to organize classes and avoid naming conflicts.

Package Declaration and Naming Conventions

```
// File: com/example/myapp/util/StringHelper.java
package com.example.myapp.util;
```

```
public class StringHelper {
    // Class implementation
}
```

- Package names are typically lowercase
- Use reverse domain name convention (e.g., `com.company.project.module`)
- Directory structure should match package structure

Importing Classes

```
// Specific import
import java.util.ArrayList;

// Import all classes from a package (not recommended for production code)
import java.util.*;

// Static import for accessing static members without class name
import static java.lang.Math.PI;

public class ImportExample {
    public void demonstrateImports() {
        ArrayList<String> list = new ArrayList<>();
        System.out.println(PI); // Instead of Math.PI
    }
}
```

> **Best Practice**: Use specific imports rather than wildcard imports to make dependencies clear and avoid potential conflicts.

Creating a Basic Project Structure

A standard Java project structure:

```
my-java-project/
├── src/
│   ├── main/
│   │   ├── java/
│   │   │   └── com/
│   │   │       └── example/
│   │   │           ├── Main.java
│   │   │           ├── model/
│   │   │           ├── service/
│   │   │           ├── repository/
│   │   │           └── util/
│   │   └── resources/
│   └── test/
│       └── java/
│           └── com/
│               └── example/
```

```
│                   └── (test classes)
├── target/ (or build/)
│   └── (compiled classes and artifacts)
└── pom.xml (or build.gradle)
```

> **Certification Note**: The default package (no package declaration) should be avoided in real applications. Classes in the default package cannot be imported by classes in named packages.

Java Build Tools

Maven

Maven is a powerful project management tool that handles dependencies, building, testing, and deployment.

Setting Up a Maven Project

1. **Installation**:
   - Download from https://maven.apache.org/download.cgi
   - Extract and add bin directory to PATH
   - Verify with `mvn -version`

2. **Creating a New Project**:

```
mvn archetype:generate -DgroupId=com.example -DartifactId=my-app -
DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

3. **Project Structure**:

```
my-app/
├── pom.xml
└── src/
    ├── main/
    │   └── java/
    │       └── com/
    │           └── example/
    │               └── App.java
    └── test/
        └── java/
            └── com/
                └── example/
                    └── AppTest.java
```

4. **Basic pom.xml**:

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>my-app</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>my-app</name>
  <url>http://maven.apache.org</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>21</maven.compiler.source>
    <maven.compiler.target>21</maven.compiler.target>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.13.2</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

5. **Common Maven Commands**:

```
mvn compile        # Compile the source code
mvn test           # Run tests
mvn package        # Package compiled code (e.g., as JAR)
mvn clean install  # Clean previous builds and install package in local repository
```

Gradle

Gradle combines the power of Ant and Maven with a Groovy-based DSL for more flexible build scripts.

Setting Up a Gradle Project

1. **Installation**:
   - Download from https://gradle.org/install/
   - Extract and add bin directory to PATH
   - Verify with `gradle -v`

2. **Creating a New Project**:

```
mkdir my-gradle-app
cd my-gradle-app
gradle init --type java-application
```

3. **Basic build.gradle**:

```
plugins {
    id 'java'
    id 'application'
}

group = 'com.example'
version = '1.0-SNAPSHOT'
sourceCompatibility = 21

repositories {
    mavenCentral()
}

dependencies {
    testImplementation 'junit:junit:4.13.2'
}

application {
    mainClass = 'com.example.App'
}
```

4. **Common Gradle Commands**:

```
gradle build          # Build the project
gradle test           # Run tests
gradle run            # Run the application
gradle clean          # Clean build artifacts
```

> **Best Practice**: Use build tools even for small projects. They standardize the build process, manage dependencies, and make your project more portable.

> **Certification Note**: While build tools aren't extensively covered in most Java certifications, understanding project organization and the build lifecycle is important.

Java Language Fundamentals

Data Types, Variables, and Operators

Java is a strongly-typed language with two categories of data types: primitive types and reference types.

Primitive Data Types

Java has eight primitive data types:

Type	Size	Range	Default
boolean	1 bit	true/false	false
char	16 bits	'\u0000' to '\uffff' (0 to 65,535)	'\u0000'
byte	8 bits	-128 to 127	0
short	16 bits	-32,768 to 32,767	0
int	32 bits	-2^31 to 2^31-1	0
long	64 bits	-2^63 to 2^63-1	0L
float	32 bits	±3.40282347E+38F (6-7 significant decimal digits)	0.0f
double	64 bits	±1.7976931348623157E+308 (15 significant decimal digits)	0.0d

```
// Declaration and initialization
int age = 30;
double salary = 75000.50;
char grade = 'A';
boolean isEmployed = true;

// Literals with type suffixes
long population = 7800000000L; // L suffix for long
float price = 19.99F;          // F suffix for float
```

> **Certification Note**: Know the exact ranges and default values of primitive types. Understand that primitive variables store actual values, not references.

Reference Types

Reference types store references (memory addresses) to objects, not the objects themselves.

```
// Reference type examples
String name = "John";
Date birthdate = new Date();
Integer count = 42;  // Wrapper class for int
```

Key reference types:

- Class types (String, Date, etc.)
- Interface types (List, Map, etc.)
- Array types (int[], String[], etc.)
- Enum types

- Record types (Java 16+)

Variable Declaration and Initialization

```
// Declaration without initialization (uses default value)
int counter;         // default value: 0
boolean flag;        // default value: false
String message;      // default value: null

// Declaration with initialization
int quantity = 100;
final double TAX_RATE = 0.08;  // constant (cannot be changed)

// Multiple variables of the same type
int x = 1, y = 2, z = 3;
```

> **Common Pitfall**: Local variables (variables declared within a method) must be initialized before use. Instance and class variables (fields) receive default values if not explicitly initialized.

Type Conversion and Casting

Java supports two types of conversion:

1. **Widening Conversion** (Implicit/Automatic):
   - Converting from smaller to larger data type
   - No data loss possible

```
byte b = 10;
int i = b;      // Automatic conversion from byte to int
double d = i;   // Automatic conversion from int to double
```

2. **Narrowing Conversion** (Explicit/Manual):
   - Converting from larger to smaller data type
   - Potential data loss
   - Requires explicit cast

```
double price = 99.99;
int roundedPrice = (int) price;  // Explicit cast, result: 99 (decimal part lost)

long bigNumber = 1234567890123L;
int smallerNumber = (int) bigNumber;  // Potential data loss!
```

Operators

Arithmetic Operators

```
int a = 10, b = 3;
int sum = a + b;        // 13
int difference = a - b; // 7
int product = a * b;    // 30
int quotient = a / b;   // 3 (integer division truncates)
int remainder = a % b;  // 1 (modulus)

// Integer division pitfall
double result = a / b;  // 3.0 (not 3.33)
double correct = (double) a / b;  // 3.33...
```

Increment/Decrement Operators

```
int count = 5;
count++;   // Post-increment (value used, then incremented)
++count;   // Pre-increment (value incremented, then used)

int x = 5;
int y = x++;  // y = 5, x = 6
int z = ++x;  // x = 7, z = 7
```

Comparison Operators

```
int a = 10, b = 20;
boolean isEqual = (a == b);         // false
boolean isNotEqual = (a != b);      // true
boolean isGreater = (a > b);        // false
boolean isLessOrEqual = (a <= b);   // true
```

Logical Operators

```
boolean hasPermission = true;
boolean isAdmin = false;

boolean canDeleteFile = hasPermission && isAdmin;  // AND: false
boolean canReadFile = hasPermission || isAdmin;    // OR: true
boolean isRegularUser = !isAdmin;                  // NOT: true
```

> **Common Pitfall**: Short-circuit evaluation with `&&` and `||`. If the left operand of `&&` is false, the right operand is not evaluated. Similarly, if the left operand of `||` is true, the right operand is not evaluated.

Bitwise Operators

```
int a = 5;  // 101 in binary
int b = 3;  // 011 in binary

int bitwiseAnd = a & b;   // 001 (1 in decimal)
int bitwiseOr = a | b;    // 111 (7 in decimal)
int bitwiseXor = a ^ b;   // 110 (6 in decimal)
int bitwiseComplement = ~a; // 11111111111111111111111111111010 (-6 in decimal)
int leftShift = a << 1;   // 1010 (10 in decimal)
int rightShift = a >> 1;  // 010 (2 in decimal)
```

Assignment Operators

```
int value = 10;
value += 5;  // Equivalent to: value = value + 5
value *= 2;  // Equivalent to: value = value * 2
value /= 3;  // Equivalent to: value = value / 3
value %= 4;  // Equivalent to: value = value % 4
```

Ternary Operator

```
int age = 20;
String status = (age >= 18) ? "Adult" : "Minor";  // "Adult"
```

> **Certification Note**: Understand operator precedence. For example, multiplication (*) and division (/) have higher precedence than addition (+) and subtraction (-).

Control Flow

Control flow statements determine the order in which code executes, allowing for decision-making and repetition.

Decision-Making Statements

if-else Statement

```
int score = 85;

if (score >= 90) {
    System.out.println("Grade: A");
} else if (score >= 80) {
    System.out.println("Grade: B");
} else if (score >= 70) {
    System.out.println("Grade: C");
} else if (score >= 60) {
    System.out.println("Grade: D");
```

```
} else {
    System.out.println("Grade: F");
}
```

> **Common Pitfall**: Forgetting that only one block of an if-else chain will execute, even if multiple conditions are true.

switch Statement

Traditional form (pre-Java 12):

```
int day = 3;
String dayName;

switch (day) {
    case 1:
        dayName = "Monday";
        break;
    case 2:
        dayName = "Tuesday";
        break;
    case 3:
        dayName = "Wednesday";
        break;
    case 4:
        dayName = "Thursday";
        break;
    case 5:
        dayName = "Friday";
        break;
    case 6:
        dayName = "Saturday";
        break;
    case 7:
        dayName = "Sunday";
        break;
    default:
        dayName = "Invalid day";
        break;
}
```

> **Common Pitfall**: Forgetting the `break` statement causes "fall-through" behavior where execution continues to subsequent cases.

Enhanced switch expression (Java 12+):

```
int day = 3;
String dayName = switch (day) {
    case 1 -> "Monday";
```

```
    case 2 -> "Tuesday";
    case 3 -> "Wednesday";
    case 4 -> "Thursday";
    case 5 -> "Friday";
    case 6 -> "Saturday";
    case 7 -> "Sunday";
    default -> "Invalid day";
};
```

Switch with multiple case labels:

```
String dayType = switch (day) {
    case 1, 2, 3, 4, 5 -> "Weekday";
    case 6, 7 -> "Weekend";
    default -> "Invalid day";
};
```

Switch with yielding a value (Java 13+):

```
String result = switch (day) {
    case 1, 2, 3, 4, 5 -> {
        System.out.println("It's a weekday");
        yield "Work day";
    }
    case 6, 7 -> {
        System.out.println("It's the weekend");
        yield "Free day";
    }
    default -> "Invalid day";
};
```

> **Certification Note**: Know the differences between traditional switch statements and modern switch expressions. Understand that switch expressions must cover all possible values or include a default case.

Looping Statements

for Loop

```
// Simple for loop
for (int i = 0; i < 5; i++) {
    System.out.println("Iteration " + i);
}

// Multiple variables in for loop
for (int i = 0, j = 10; i < j; i++, j--) {
```

```
        System.out.println("i = " + i + ", j = " + j);
    }

    // Enhanced for loop (for-each) - Introduced in Java 5
    int[] numbers = {1, 2, 3, 4, 5};
    for (int number : numbers) {
        System.out.println(number);
    }
```

while Loop

```
int count = 0;
while (count < 5) {
    System.out.println("Count: " + count);
    count++;
}
```

do-while Loop

```
int i = 0;
do {
    System.out.println("Value of i: " + i);
    i++;
} while (i < 5);
```

> **Key Difference**: The `do-while` loop guarantees at least one execution of the loop body, even if the condition is initially false.

Branching Statements

```
// break - terminates loop
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        break;  // Exit the loop when i equals 5
    }
    System.out.println(i);
}

// continue - skips to next iteration
for (int i = 0; i < 10; i++) {
    if (i % 2 == 0) {
        continue;  // Skip even numbers
    }
    System.out.println(i);  // Prints only odd numbers
}
```

```
// labeled break and continue
outer:
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        if (i == 1 && j == 1) {
            break outer;  // Breaks out of both loops
        }
        System.out.println("i = " + i + ", j = " + j);
    }
}
```

> **Certification Note**: Understand the difference between `break` and `continue`, and know how labeled versions work with nested loops. Be aware that `break` can also be used with switch statements.

Methods and Method Overloading

Methods are blocks of code that can be called from other places in your program. They promote code reuse and help break down complex problems.

Method Declaration

```
// Basic method structure
[access modifier] [static] [final] return-type method-name([parameters]) [throws
exceptions] {
    // Method body
    [return statement]
}
```

Example:

```
public class Calculator {
    // Simple method with two parameters and a return value
    public int add(int a, int b) {
        return a + b;
    }

    // Method with no parameters and void return type
    public void displayInfo() {
        System.out.println("Calculator utility");
    }

    // Static method (belongs to class, not instances)
    public static double calculateCircleArea(double radius) {
        return Math.PI * radius * radius;
    }
}
```

Method Parameters

```
// Pass by value (primitive types)
public void incrementValue(int value) {
    value++;  // Modifies local copy, original unchanged
}

// Pass by reference value (reference types)
public void modifyList(List<String> items) {
    items.add("New Item");  // Modifies the original list
    items = new ArrayList<>();  // Reassigns local variable only, original
unchanged
}
```

> **Common Misconception**: Java is always pass-by-value. When passing objects, the value of the reference is passed, not the reference itself. This means methods can modify the object's state but cannot make the original reference point to a different object.

Method Overloading

Method overloading allows creating multiple methods with the same name but different parameter lists.

```
public class Calculator {
    // Method with two int parameters
    public int add(int a, int b) {
        return a + b;
    }

    // Overloaded method with three int parameters
    public int add(int a, int b, int c) {
        return a + b + c;
    }

    // Overloaded method with double parameters
    public double add(double a, double b) {
        return a + b;
    }

    // Overloaded method with mixed parameters
    public double add(int a, double b) {
        return a + b;
    }
}
```

> **Rules for Method Overloading**:
>
> - Methods must have different parameter lists (different number or types of parameters)
> - Return type alone is not sufficient for method overloading

> - Method overloading is determined at compile time (static binding)

```
Calculator calc = new Calculator();
calc.add(5, 10);       // Calls add(int, int)
calc.add(5, 10, 15);   // Calls add(int, int, int)
calc.add(5.5, 10.5);   // Calls add(double, double)
calc.add(5, 10.5);     // Calls add(int, double)
```

Varargs (Variable-Length Arguments)

Varargs allow methods to accept variable numbers of arguments.

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

Usage:

```
int result1 = sum(1, 2);            // 3
int result2 = sum(1, 2, 3, 4, 5);   // 15
int result3 = sum();                // 0
```

> **Rules for Varargs**:
>
> - Only one varargs parameter is allowed per method
> - The varargs parameter must be the last parameter

```
// Valid
public void printInfo(String name, int... scores) { ... }

// Invalid - varargs must be the last parameter
public void printInfo(int... scores, String name) { ... }

// Invalid - only one varargs parameter allowed
public void printInfo(String... names, int... scores) { ... }
```

> **Certification Note**: Understand how method overloading resolution works when multiple methods are applicable. The compiler chooses the most specific method that matches the arguments.

Object-Oriented Programming Principles in Java

Object-oriented programming (OOP) is a paradigm that organizes code around objects rather than actions and data rather than logic. Java is fundamentally an object-oriented language.

Four Pillars of OOP

1. **Encapsulation**: Binding data (attributes) and code (methods) together in a class, restricting direct access to some components.
2. **Inheritance**: Creating new classes (subclasses) from existing ones (superclasses), inheriting their attributes and behaviors.
3. **Polymorphism**: The ability of an object to take many forms, typically through method overriding and interfaces.
4. **Abstraction**: Hiding implementation details and showing only functionality to the user, through abstract classes and interfaces.

Encapsulation Example

```
public class BankAccount {
    // Private fields - not directly accessible outside class
    private String accountNumber;
    private double balance;

    // Public constructor
    public BankAccount(String accountNumber, double initialBalance) {
        this.accountNumber = accountNumber;
        this.balance = initialBalance;
    }

    // Public getter methods
    public String getAccountNumber() {
        return accountNumber;
    }

    public double getBalance() {
        return balance;
    }

    // Public methods to modify state in controlled ways
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
            System.out.println("Deposited: $" + amount);
        } else {
            System.out.println("Invalid deposit amount");
        }
    }

    public boolean withdraw(double amount) {
        if (amount > 0 && balance >= amount) {
            balance -= amount;
```

```
            System.out.println("Withdrawn: $" + amount);
            return true;
        } else {
            System.out.println("Invalid withdrawal or insufficient funds");
            return false;
        }
    }
}
```

Usage:

```
BankAccount account = new BankAccount("123456789", 1000.0);
System.out.println("Account: " + account.getAccountNumber());
System.out.println("Balance: $" + account.getBalance());

account.deposit(500.0);
account.withdraw(200.0);

// This is not possible due to encapsulation:
// account.balance = -1000.0;  // Compilation error: balance has private access
```

Inheritance Example

```
// Base class (superclass)
public class Vehicle {
    private String make;
    private String model;
    private int year;

    public Vehicle(String make, String model, int year) {
        this.make = make;
        this.model = model;
        this.year = year;
    }

    public void startEngine() {
        System.out.println("Vehicle engine started");
    }

    public void stopEngine() {
        System.out.println("Vehicle engine stopped");
    }

    // Getters and setters
    public String getMake() { return make; }
    public String getModel() { return model; }
    public int getYear() { return year; }
}
```

```
// Derived class (subclass)
public class Car extends Vehicle {
    private int numberOfDoors;

    public Car(String make, String model, int year, int numberOfDoors) {
        // Call superclass constructor using super
        super(make, model, year);
        this.numberOfDoors = numberOfDoors;
    }

    // Override a method from the superclass
    @Override
    public void startEngine() {
        System.out.println("Car engine started with key");
    }

    // New method specific to Car
    public void openTrunk() {
        System.out.println("Trunk opened");
    }

    public int getNumberOfDoors() { return numberOfDoors; }
}
```

Usage:

```
Car myCar = new Car("Toyota", "Camry", 2023, 4);
myCar.startEngine();  // Uses the overridden method
myCar.stopEngine();   // Inherited from Vehicle
myCar.openTrunk();    // Car-specific method

// A Vehicle reference can point to a Car object (polymorphism)
Vehicle vehicle = new Car("Honda", "Civic", 2022, 4);
vehicle.startEngine();  // Uses Car's implementation
// vehicle.openTrunk();  // Compilation error: method not found in Vehicle
```

Polymorphism Example

```
// Interface defining a common behavior
public interface AudioPlayer {
    void play();
    void stop();
    void pause();
}

// Multiple implementations of the same interface
public class CDPlayer implements AudioPlayer {
    @Override
    public void play() {
```

```
        System.out.println("CD spinning and playing");
    }

    @Override
    public void stop() {
        System.out.println("CD stopped");
    }

    @Override
    public void pause() {
        System.out.println("CD paused");
    }
}

public class MP3Player implements AudioPlayer {
    @Override
    public void play() {
        System.out.println("MP3 playing from memory");
    }

    @Override
    public void stop() {
        System.out.println("MP3 stopped");
    }

    @Override
    public void pause() {
        System.out.println("MP3 paused");
    }

    // Additional method specific to MP3Player
    public void shuffle() {
        System.out.println("MP3 playlist shuffled");
    }
}
```

Usage demonstrating polymorphism:

```
// Function accepting any AudioPlayer implementation
public void playAudio(AudioPlayer player) {
    player.play();
    // Later
    player.pause();
    // Later
    player.stop();
}

// Using the function with different implementations
playAudio(new CDPlayer());
playAudio(new MP3Player());

// Different objects, same interface, different behaviors
```

```
AudioPlayer player1 = new CDPlayer();
AudioPlayer player2 = new MP3Player();

player1.play();  // "CD spinning and playing"
player2.play();  // "MP3 playing from memory"
```

Abstraction Example

```
// Abstract class
public abstract class Shape {
    // Abstract method (no implementation)
    public abstract double calculateArea();

    // Concrete method with implementation
    public void display() {
        System.out.println("This is a shape with area: " + calculateArea());
    }
}

// Concrete implementation
public class Circle extends Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    @Override
    public double calculateArea() {
        return Math.PI * radius * radius;
    }
}

// Another concrete implementation
public class Rectangle extends Shape {
    private double length;
    private double width;

    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    @Override
    public double calculateArea() {
        return length * width;
    }
}
```

Usage:

```
// Cannot instantiate abstract class
// Shape shape = new Shape();  // Compilation error

Shape circle = new Circle(5.0);
Shape rectangle = new Rectangle(4.0, 6.0);

circle.display();     // Uses concrete method from Shape with Circle's area
rectangle.display();  // Uses concrete method from Shape with Rectangle's area
```

> **Certification Note**: Understand the differences between abstract classes and interfaces, especially after Java 8 (which added default methods to interfaces). Know when to use each approach.

Classes and Objects

In Java, a class is a blueprint for creating objects. An object is an instance of a class.

Class Declaration and Components

```
public class Student {
    // Instance variables (fields)
    private int id;
    private String name;
    private double gpa;

    // Static variable (shared across all instances)
    private static int studentCount = 0;

    // Static constant
    public static final int MAX_COURSES = 6;

    // Default constructor
    public Student() {
        studentCount++;
    }

    // Parameterized constructor
    public Student(int id, String name, double gpa) {
        this.id = id;
        this.name = name;
        this.gpa = gpa;
        studentCount++;
    }

    // Instance methods
    public void study() {
        System.out.println(name + " is studying");
    }

    public void updateGpa(double newGpa) {
        if (newGpa >= 0.0 && newGpa <= 4.0) {
```

```
            this.gpa = newGpa;
        } else {
            System.out.println("Invalid GPA value");
        }
    }

    // Static method
    public static int getStudentCount() {
        return studentCount;
    }

    // Getters and setters
    public int getId() { return id; }
    public String getName() { return name; }
    public double getGpa() { return gpa; }

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

Creating and Using Objects

```
// Creating objects using constructors
Student student1 = new Student();
Student student2 = new Student(1001, "Alice", 3.8);

// Accessing instance methods and fields
student1.setName("Bob");
student1.updateGpa(3.5);
student1.study();

System.out.println(student2.getName() + " has GPA: " + student2.getGpa());

// Accessing static method and constant
int count = Student.getStudentCount();  // 2
int maxCourses = Student.MAX_COURSES;   // 6
```

'this' Keyword

The `this` keyword refers to the current object and has several uses:

1. Differentiating instance variables from local variables:

```
public void setId(int id) {
    this.id = id;  // 'this.id' refers to the instance variable
}
```

2. Calling one constructor from another:

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

    public Person() {
        this("Unknown", 0);  // Calls the parameterized constructor
    }

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

3. Passing the current object to a method:

```
public class Customer {
    private String name;

    public void register() {
        Database.saveCustomer(this);  // Pass this Customer object
    }
}
```

Object Lifecycle

1. **Creation**: Objects are created using the `new` keyword, which allocates memory and calls a constructor.
2. **Usage**: Objects are used through references to access fields and invoke methods.
3. **Garbage Collection**: When no references to an object remain, it becomes eligible for garbage collection.

```
public void demonstrateLifecycle() {
    // Object creation
    String message = new String("Hello");

    // Object usage
    System.out.println(message.length());

    // Object eligible for garbage collection
    message = null;  // Remove reference

    // Or by going out of scope
}  // message goes out of scope
```

> **Common Pitfall**: Memory leaks can occur when objects are unintentionally kept alive by lingering references. Common sources include static fields, long-lived collections, and improper resource management.

toString, equals, and hashCode

These methods from the `Object` class are commonly overridden:

```
public class Product {
    private int id;
    private String name;
    private double price;

    // Constructor and other methods...

    // toString method for string representation
    @Override
    public String toString() {
        return "Product{id=" + id + ", name='" + name + "', price=" + price + "}";
    }

    // equals method for object comparison
    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;

        Product product = (Product) obj;
        return id == product.id &&
               Double.compare(product.price, price) == 0 &&
               Objects.equals(name, product.name);
    }

    // hashCode method for use in hash-based collections
    @Override
    public int hashCode() {
        return Objects.hash(id, name, price);
    }
}
```

> **Certification Note**: Always override `hashCode()` when overriding `equals()`. If two objects are equal according to `equals()`, they must have the same hash code, but the reverse is not necessarily true.

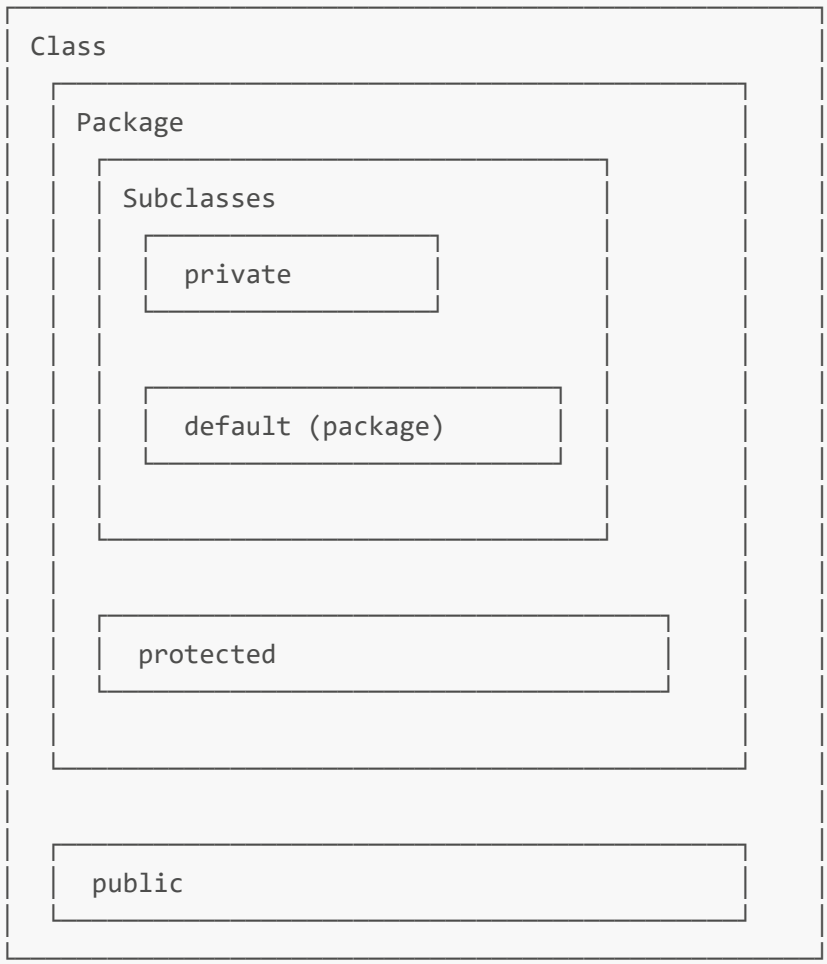
Access Modifiers and Encapsulation

Access modifiers control the visibility and accessibility of classes, variables, methods, and constructors.

Types of Access Modifiers

Modifier	Class	Package	Subclass	World

Modifier	Class	Package	Subclass	World
public	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	No
default	Yes	Yes	No	No
private	Yes	No	No	No



Examples of Access Modifier Usage

```
// File: com/example/access/AccessDemo.java
package com.example.access;

public class AccessDemo {
    public int publicVar = 1;           // Accessible from anywhere
    protected int protectedVar = 2;     // Accessible in same package or
subclasses
    int defaultVar = 3;                 // Accessible only in same package
    private int privateVar = 4;         // Accessible only in this class

    public void publicMethod() { }      // Accessible from anywhere
    protected void protectedMethod() { } // Accessible in same package or
```

```
    subclasses
        void defaultMethod() { }            // Accessible only in same package
        private void privateMethod() { }    // Accessible only in this class
    }
```

Encapsulation in Practice

Encapsulation is typically implemented by making fields private and providing public getters and setters:

```
public class Employee {
    // Private fields - encapsulated data
    private int id;
    private String name;
    private double salary;

    // Public constructors and methods provide controlled access
    public Employee(int id, String name, double salary) {
        this.id = id;
        this.name = name;
        this.salary = salary;
    }

    // Getters - provide read access
    public int getId() { return id; }
    public String getName() { return name; }
    public double getSalary() { return salary; }

    // Setters - provide write access with validation
    public void setName(String name) {
        if (name != null && !name.isEmpty()) {
            this.name = name;
        }
    }

    public void setSalary(double salary) {
        if (salary > 0) {
            this.salary = salary;
        }
    }

    // Business logic methods
    public void applyRaise(double percentage) {
        if (percentage > 0) {
            salary += salary * (percentage / 100.0);
        }
    }
}
```

> **Benefits of Encapsulation**:

> - Increased security by hiding implementation details
> - Ability to control data validation
> - Flexibility to change implementation without affecting clients
> - Helps maintain invariants (rules about object state)

> **Certification Note**: Understand that you can have access modifiers on classes, methods, constructors, and fields. Also note that nested classes can have all access modifiers, but top-level classes can only be public or package-private (default).

Inheritance, Polymorphism, and Abstraction

Inheritance in Detail

Inheritance allows a class to inherit properties and behavior from another class. Java supports single inheritance for classes (a class can extend only one class) but multiple inheritance for interfaces (a class can implement multiple interfaces).

Extending a Class

```
// Base class
public class Animal {
    protected String name;
    protected int age;

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

    public void eat() {
        System.out.println(name + " is eating");
    }

    public void sleep() {
        System.out.println(name + " is sleeping");
    }

    public String getName() { return name; }
    public int getAge() { return age; }
}

// Derived class
public class Dog extends Animal {
    private String breed;

    public Dog(String name, int age, String breed) {
        super(name, age);  // Call parent constructor
        this.breed = breed;
    }

    // New method specific to Dog
```

```
    public void bark() {
        System.out.println(name + " is barking");
    }

    // Override method from parent
    @Override
    public void eat() {
        System.out.println(name + " the dog is eating quickly");
    }

    public String getBreed() { return breed; }
}
```

Method Overriding

Method overriding occurs when a subclass provides a specific implementation of a method already defined in its superclass. The method in the subclass must have the same name, return type, and parameters.

Rules for method overriding:

- The method must have the same name and parameter list
- The return type must be the same or a subtype of the parent's return type (covariant return)
- The access level cannot be more restrictive than the parent method
- The method cannot throw new or broader checked exceptions

```
// Parent class
public class Vehicle {
    public void start() {
        System.out.println("Vehicle starting");
    }

    protected Object getDescription() {
        return "Generic vehicle";
    }
}

// Child class
public class Motorcycle extends Vehicle {
    @Override
    public void start() {  // Same name, parameters, return type
        System.out.println("Motorcycle starting - kickstart engaged");
    }

    @Override
    public String getDescription() {  // Covariant return type (String is a
subtype of Object)
        return "A two-wheeled vehicle";
    }
}
```

Constructor Chaining

When a class is instantiated, constructors are called in a chain from the top of the inheritance hierarchy down to the current class.

```
public class A {
    public A() {
        System.out.println("Constructor A");
    }
}

public class B extends A {
    public B() {
        System.out.println("Constructor B");
    }
}

public class C extends B {
    public C() {
        System.out.println("Constructor C");
    }
}

// Creating an instance of C
C c = new C();
// Output:
// Constructor A
// Constructor B
// Constructor C
```

Every constructor's first statement is either an explicit call to another constructor in the same class (`this()`) or an explicit call to a parent constructor (`super()`). If neither is provided, the compiler inserts `super()` implicitly.

The Object Class

All classes in Java implicitly extend the `Object` class if they don't extend any other class. This provides several methods to all Java objects:

```
public class Example {
    // These methods are inherited from Object and can be overridden

    @Override
    public String toString() {
        // Default implementation: getClass().getName() + "@" +
Integer.toHexString(hashCode())
        return "Custom string representation";
    }
```

```
    @Override
    public boolean equals(Object obj) {
        // Default implementation uses == (reference equality)
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
        // Custom equality logic...
        return false;
    }

    @Override
    public int hashCode() {
        // Default implementation might use memory address
        // Should be consistent with equals
        return 42;  // Not a good implementation, just an example
    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        // Default implementation creates a shallow copy
        return super.clone();
    }

    @Override
    protected void finalize() throws Throwable {
        // Called by the garbage collector before reclaiming memory
        // Deprecated in newer Java versions
        super.finalize();
    }
}
```

Polymorphism in Detail

Polymorphism allows an object to take many forms, typically through inheritance and interfaces.

Runtime Polymorphism (Dynamic Method Dispatch)

```
// Base class
public class Shape {
    public void draw() {
        System.out.println("Drawing a shape");
    }
}

// Derived classes
public class Circle extends Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a circle");
    }
}
```

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

// Usage
public void demonstratePolymorphism() {
    Shape shape1 = new Circle();      // Circle object referred by Shape reference
    Shape shape2 = new Rectangle();   // Rectangle object referred by Shape
reference

    shape1.draw();  // Calls Circle's draw() - "Drawing a circle"
    shape2.draw();  // Calls Rectangle's draw() - "Drawing a rectangle"

    // Array of different shape types
    Shape[] shapes = {new Circle(), new Rectangle(), new Shape()};
    for (Shape shape : shapes) {
        shape.draw();  // Calls appropriate draw() method for each object
    }
}
```

Polymorphism with Interfaces

```
// Interface
public interface Payable {
    double getPaymentAmount();
}

// Implementing classes
public class Employee implements Payable {
    private double salary;

    public Employee(double salary) {
        this.salary = salary;
    }

    @Override
    public double getPaymentAmount() {
        return salary;
    }
}

public class Invoice implements Payable {
    private double amount;

    public Invoice(double amount) {
        this.amount = amount;
    }
```

```
    @Override
    public double getPaymentAmount() {
        return amount;
    }
}

// Usage
public void processPayments(Payable[] items) {
    double total = 0;
    for (Payable item : items) {
        double payment = item.getPaymentAmount();  // Polymorphic call
        System.out.println("Payment: $" + payment);
        total += payment;
    }
    System.out.println("Total payments: $" + total);
}

// Creating mixed array of Payable objects
Payable[] payableItems = {
    new Employee(5000),
    new Invoice(1200),
    new Employee(6500)
};
processPayments(payableItems);
```

Abstraction in Detail

Abstraction involves hiding implementation details and showing only functionality to the user. Java provides abstraction through abstract classes and interfaces.

Abstract Classes

An abstract class cannot be instantiated and may contain abstract methods (methods without a body).

```
public abstract class Database {
    // Regular method with implementation
    public void connect() {
        System.out.println("Establishing database connection");
        // Common connection logic
    }

    // Abstract method - no implementation
    public abstract void executeQuery(String query);

    // Regular method
    public void disconnect() {
        System.out.println("Closing database connection");
    }
}

// Concrete implementation
```

```
public class MySQLDatabase extends Database {
    @Override
    public void executeQuery(String query) {
        System.out.println("Executing MySQL query: " + query);
        // MySQL-specific implementation
    }
}

// Another implementation
public class PostgreSQLDatabase extends Database {
    @Override
    public void executeQuery(String query) {
        System.out.println("Executing PostgreSQL query: " + query);
        // PostgreSQL-specific implementation
    }
}
```

Usage:

```
// Cannot instantiate abstract class
// Database db = new Database();  // Compilation error

// Can use reference of abstract type
Database mysql = new MySQLDatabase();
mysql.connect();
mysql.executeQuery("SELECT * FROM users");
mysql.disconnect();
```

Interfaces

An interface defines a contract for classes to implement. Prior to Java 8, interfaces could only contain abstract methods and constants. Starting with Java 8, interfaces can also include default and static methods, and with Java 9, private methods.

```
public interface Sortable {
    // Abstract method (implicitly public and abstract)
    void sort();

    // Constant (implicitly public, static, and final)
    int ASCENDING = 1;
    int DESCENDING = -1;

    // Default method (with implementation, Java 8+)
    default void sortUsingDefaultAlgorithm() {
        System.out.println("Using default sorting algorithm");
        sort();
    }

    // Static method (Java 8+)
```

```
    static Sortable createNaturalOrder() {
        return () -> System.out.println("Sorting in natural order");
    }

    // Private method (Java 9+) - can only be used by default or other private
methods
    private void helperMethod() {
        System.out.println("Helper method for internal use");
    }
}

// Implementing interface
public class SortableList implements Sortable {
    @Override
    public void sort() {
        System.out.println("Sorting list elements");
    }
}
```

Usage:

```
Sortable list = new SortableList();
list.sort();  // Calls implemented method
list.sortUsingDefaultAlgorithm();  // Uses default implementation

// Using static method
Sortable natural = Sortable.createNaturalOrder();
natural.sort();
```

Interfaces vs. Abstract Classes

Feature	Abstract Class	Interface
Instantiation	Cannot be instantiated	Cannot be instantiated
Inheritance	Single inheritance only	Multiple inheritance possible
Fields	Can have any kind of fields	Only constants (public static final)
Methods	Any kind of methods	Abstract, default, static, private methods
Constructor	Can have constructors	No constructors
Access Modifiers	Any access modifier	Methods implicitly public
Purpose	Partial implementation, is-a relation	Contract, can-do capability

> **When to Choose**:
>
> - Use abstract classes when classes share common implementation.
> - Use interfaces when unrelated classes need to share method signatures.

> - Abstract classes model "is-a" relationships, interfaces model "can-do" capabilities.

> **Certification Note**: Understand the differences between interfaces and abstract classes, especially the changes in interface capabilities added in Java 8 and 9. Know that a class can implement multiple interfaces but extend only one class.

Static and Instance Initialization Blocks

Static Initialization Blocks

Static initialization blocks are executed when a class is loaded, before any instance is created. They are used to initialize static variables or perform one-time setup actions.

```
public class DatabaseConnection {
    // Static variables
    private static String url;
    private static String username;
    private static String password;
    private static boolean isConfigured;

    // Static initialization block
    static {
        System.out.println("Loading database configuration...");

        // Load configuration from properties file
        Properties props = new Properties();
        try (FileInputStream fis = new FileInputStream("db.properties")) {
            props.load(fis);
            url = props.getProperty("db.url");
            username = props.getProperty("db.user");
            password = props.getProperty("db.password");
            isConfigured = true;
            System.out.println("Database configuration loaded successfully");
        } catch (IOException e) {
            System.err.println("Failed to load database configuration");
            isConfigured = false;
        }
    }

    // Another static block (multiple blocks execute in order)
    static {
        if (isConfigured) {
            System.out.println("Registering JDBC driver...");
            try {
                Class.forName("com.mysql.jdbc.Driver");
            } catch (ClassNotFoundException e) {
                System.err.println("JDBC Driver not found");
                isConfigured = false;
            }
        }
    }
```

```
    // Static method
    public static boolean isConfigured() {
        return isConfigured;
    }

    // Instance method
    public Connection getConnection() throws SQLException {
        if (!isConfigured) {
            throw new SQLException("Database not configured");
        }
        return DriverManager.getConnection(url, username, password);
    }
}
```

Key points about static initialization blocks:

- Execute when the class is loaded
- Execute in the order they appear in the class
- Can access static variables and methods only
- Cannot access instance variables or methods
- Cannot use `this` or `super` keywords
- Cannot throw checked exceptions (must handle them internally)

Instance Initialization Blocks

Instance initialization blocks are executed when an instance of a class is created, before the constructor runs.

```
public class Account {
    private int id;
    private String type;
    private double balance;
    private List<String> transactions;

    // Instance initialization block
    {
        System.out.println("Instance initialization block running");
        transactions = new ArrayList<>();
        transactions.add("Account created at: " + new Date());
    }

    // Another instance block
    {
        balance = 0.0;
        System.out.println("Initial balance set to zero");
    }

    // Default constructor
    public Account() {
        System.out.println("Default constructor running");
        type = "Checking";
    }
```

```
    // Parameterized constructor
    public Account(int id, String type, double initialDeposit) {
        System.out.println("Parameterized constructor running");
        this.id = id;
        this.type = type;
        this.balance = initialDeposit;
        transactions.add("Initial deposit: $" + initialDeposit);
    }
}
```

Using the class:

```
Account account1 = new Account();
// Output:
// Instance initialization block running
// Initial balance set to zero
// Default constructor running

Account account2 = new Account(1001, "Savings", 500.0);
// Output:
// Instance initialization block running
// Initial balance set to zero
// Parameterized constructor running
```

Key points about instance initialization blocks:

- Execute when an instance is created, before the constructor
- Execute in the order they appear in the class
- Can access both static and instance variables and methods
- Can use `this` keyword (but not `super` in the block itself)
- Can throw checked exceptions if declared in constructor's throws clause

Initialization Order

The complete initialization order when an object is created:

1. Static variables initialized with defaults
2. Static initialization blocks executed (in order of appearance)
3. Instance variables initialized with defaults
4. Instance initialization blocks executed (in order of appearance)
5. Constructor executed

```
public class InitializationOrder {
    // Static variable
    private static String staticVar = initializeStaticVar();

    // Instance variable
```

```
    private String instanceVar = initializeInstanceVar();

    // Static initialization methods
    private static String initializeStaticVar() {
        System.out.println("1. Static variable initialization");
        return "Static Value";
    }

    // Static block
    static {
        System.out.println("2. Static initialization block");
    }

    // Instance initialization methods
    private String initializeInstanceVar() {
        System.out.println("3. Instance variable initialization");
        return "Instance Value";
    }

    // Instance initialization block
    {
        System.out.println("4. Instance initialization block");
    }

    // Constructor
    public InitializationOrder() {
        System.out.println("5. Constructor execution");
    }

    public static void main(String[] args) {
        System.out.println("Main method started");
        InitializationOrder obj = new InitializationOrder();
    }
}
```

Output:

```
1. Static variable initialization
2. Static initialization block
Main method started
3. Instance variable initialization
4. Instance initialization block
5. Constructor execution
```

> **Common Pitfall**: Circular dependencies in initialization can cause unexpected behavior. For example, if a static block tries to access a static variable that's defined later in the class, the variable will exist but may not be initialized yet.

> **Certification Note**: Understand the exact order of execution for initialization blocks and constructors. Know that static blocks execute only once when the class is loaded, while instance blocks execute for

> each object creation.

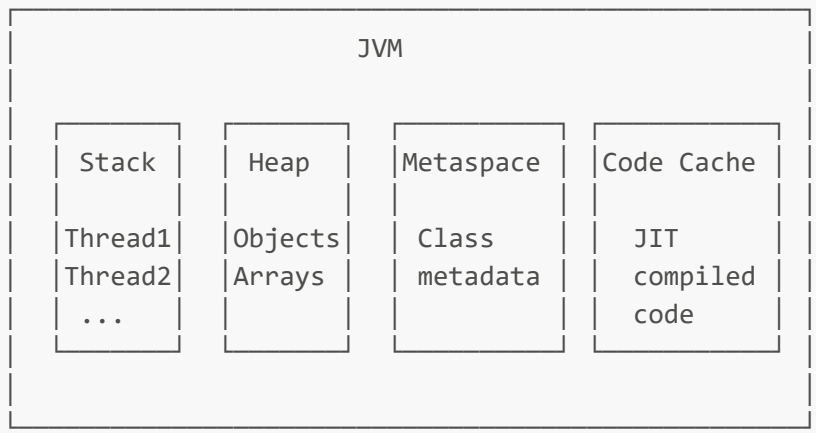
Memory Management and Garbage Collection

Java uses automatic memory management through its garbage collection (GC) system, relieving developers from manual memory allocation and deallocation.

Memory Model in Java

The JVM divides memory into several areas:

1. **Heap**:
   - Where objects are stored
   - Dynamically allocated at runtime
   - Garbage collection occurs here
   - Shared among all threads
2. **Stack**:
   - Stores method frames and local variables
   - Each thread has its own stack
   - LIFO (Last-In, First-Out) structure
   - Automatically grows and shrinks with method calls
3. **Metaspace** (formerly PermGen):
   - Stores class metadata, method bytecode, etc.
   - Dynamically sized (no fixed limit)
4. **Code Cache**:
   - Stores compiled native code generated by JIT
5. **Native Memory**:
   - Memory used for JVM internal operations



Object Lifecycle

1. **Creation**:
    - Memory allocated on the heap
    - Object fields initialized
    - Constructor executed
2. **Usage**:
    - Object referenced and used
    - Can be passed, stored, and accessed
3. **Unreachable**:
    - Object no longer has any live references
    - Becomes eligible for garbage collection
4. **Garbage Collection**:
    - Memory reclaimed by GC
    - Finalize method may be called (deprecated)

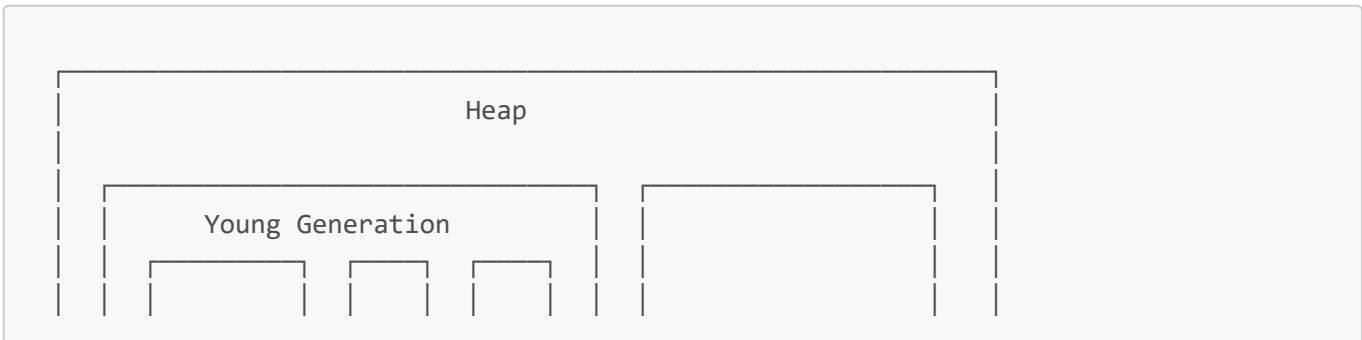
Garbage Collection Process

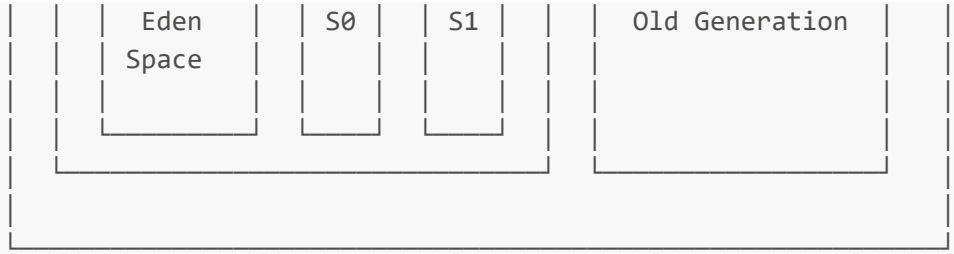
Garbage collection typically follows these steps:

1. **Mark**: Identify all reachable objects by following reference chains from "GC roots"
2. **Sweep**: Remove unreachable objects
3. **Compact** (optional): Rearrange memory to reduce fragmentation

Modern JVMs use a generational garbage collection approach:

- **Young Generation** (Eden, Survivor spaces):
    - Where new objects are allocated
    - Minor GC occurs frequently here
    - Survivors promoted to older generations
- **Old Generation**:
    - Contains long-lived objects
    - Major GC occurs less frequently
    - More expensive collection process





Making Objects Eligible for Garbage Collection

Objects become eligible for garbage collection when they are no longer reachable. This can happen several ways:

1. **Nullifying References**:

```
Person person = new Person("John");
person = null;  // Object becomes unreachable
```

2. **Reassigning References**:

```
Person person = new Person("John");
person = new Person("Alice");  // First Person object becomes unreachable
```

3. **Objects Going Out of Scope**:

```
public void method() {
    Person localPerson = new Person("John");
    // localPerson is used within the method
}  // localPerson goes out of scope, object becomes unreachable
```

4. **Island of Isolation** (Cyclic References):

```
public class Node {
    Node next;
}

Node node1 = new Node();
Node node2 = new Node();
node1.next = node2;
node2.next = node1;  // Circular reference

// Make both nodes unreachable from GC roots
node1 = null;
node2 = null;  // Both objects are now eligible for GC despite referencing each
other
```

Memory Leaks in Java

Even with automatic garbage collection, memory leaks can still occur when objects remain referenced unintentionally:

1. **Static Fields**:

```
public class Cache {
    // Static field that never gets cleared
    private static final Map<String, Object> CACHE = new HashMap<>();

    public static void store(String key, Object value) {
        CACHE.put(key, value);  // Objects never removed, potential memory leak
    }
}
```

2. **Forgotten Listeners**:

```
public class EventSource {
    private List<EventListener> listeners = new ArrayList<>();

    public void addListener(EventListener listener) {
        listeners.add(listener);
    }

    // Missing removeListener method can cause memory leaks
}
```

3. **Inner Classes and Closures**:

```
public class OuterClass {
    private byte[] largeData = new byte[100000];

    public Runnable createRunnable() {
        // Inner class holds an implicit reference to OuterClass instance
        return new Runnable() {
            @Override
            public void run() {
                System.out.println("Running");
            }
        };
    }
}
```

4. **Unclosed Resources**:

```
public void readFile(String path) throws IOException {
    FileInputStream fis = new FileInputStream(path);
    // Read file...
    // Missing fis.close() can cause resource leak
}
```

Best Practices for Memory Management

1. **Close Resources Properly**:

```
// Using try-with-resources (Java 7+)
try (FileInputStream fis = new FileInputStream("file.txt")) {
    // Use the resource
}  // Resource automatically closed
```

2. **Use Weak References for Caches**:

```
// WeakHashMap - entries removed when keys are no longer strongly reachable
Map<Key, Value> cache = new WeakHashMap<>();
```

3. **Avoid Excessive Object Creation**:

```
// Inefficient - creates many temporary String objects
String result = "";
for (int i = 0; i < 1000; i++) {
    result += i;  // Creates new String each time
}

// Better - uses StringBuilder
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 1000; i++) {
    sb.append(i);
}
String result = sb.toString();  // One String created at the end
```

4. **Consider Object Pooling for Expensive Objects**:

```
public class ConnectionPool {
    private final LinkedList<Connection> pool = new LinkedList<>();

    public Connection getConnection() {
        if (pool.isEmpty()) {
            return createNewConnection();
        } else {
```

```
                return pool.removeFirst();
            }
        }

        public void releaseConnection(Connection conn) {
            pool.addLast(conn);
        }
    }
```

5. **Use Java Mission Control and Flight Recorder** for monitoring memory usage and identifying leaks.

> **Common Pitfall**: Assuming that all objects will be garbage collected immediately after they become unreachable. Garbage collection runs on its own schedule, and unreachable objects may persist for some time.

> **Certification Note**: Understand what makes an object eligible for garbage collection and how to prevent memory leaks. Be familiar with the difference between strong, weak, soft, and phantom references (though these details are typically covered in advanced certifications).

Java Standard Library

String Handling and Text Processing

Strings are immutable sequences of characters in Java. Being immutable means that once created, a String object cannot be modified.

String Creation and Initialization

```
// String literal
String name = "John";

// Using constructor
String address = new String("123 Main St");

// String concatenation
String fullName = "John" + " " + "Doe";

// From char array
char[] chars = {'H', 'e', 'l', 'l', 'o'};
String greeting = new String(chars);

// From byte array (using default charset)
byte[] bytes = {72, 101, 108, 108, 111};
String byteGreeting = new String(bytes);
```

> **Common Pitfall**: Using `new String("literal")` creates two objects: one in the string pool and one on the heap. Prefer string literals for better memory usage.

String Manipulation Methods

```
String text = "Java Programming";

// Basic methods
int length = text.length();                      // 16
char firstChar = text.charAt(0);                 // 'J'
boolean containsJava = text.contains("Java");    // true
String upper = text.toUpperCase();               // "JAVA PROGRAMMING"
String lower = text.toLowerCase();               // "java programming"
String trimmed = "  Hello  ".trim();             // "Hello"

// Searching and extracting
int indexOfP = text.indexOf('P');                // 5
int lastA = text.lastIndexOf('a');               // 3
boolean startsWithJa = text.startsWith("Ja");    // true
boolean endsWithIng = text.endsWith("ing");      // true
String sub = text.substring(5, 16);              // "Programming"

// Replacing
String replaced = text.replace('a', 'u');        // "Juvu Progrumming"
String replacedAll = text.replaceAll("a", "A");  // "JAvA ProgrAmming"

// Splitting
String[] parts = "apple,banana,orange".split(","); // ["apple", "banana",
"orange"]

// Joining (Java 8+)
String joined = String.join("-", "2023", "05", "15"); // "2023-05-15"
```

> **Memory Efficiency**: Each String manipulation method creates a new String object. For multiple manipulations, use StringBuilder.

String Comparison

```
String s1 = "Hello";
String s2 = "Hello";
String s3 = new String("Hello");
String s4 = "HELLO";

// Reference comparison (memory address)
boolean b1 = (s1 == s2);       // true (string pool optimization)
boolean b2 = (s1 == s3);       // false (different objects)

// Content comparison
boolean b3 = s1.equals(s2);    // true
boolean b4 = s1.equals(s3);    // true
boolean b5 = s1.equals(s4);    // false (case-sensitive)
boolean b6 = s1.equalsIgnoreCase(s4);  // true (ignores case)
```

```
// Comparison for sorting
int comp1 = s1.compareTo(s2);   // 0 (equal)
int comp2 = "apple".compareTo("banana");  // negative (apple before banana)
int comp3 = "banana".compareTo("apple");  // positive (banana after apple)
```

> **Certification Note**: Always use `equals()` for string content comparison, not `==`. The `==` operator compares object references, not string content.

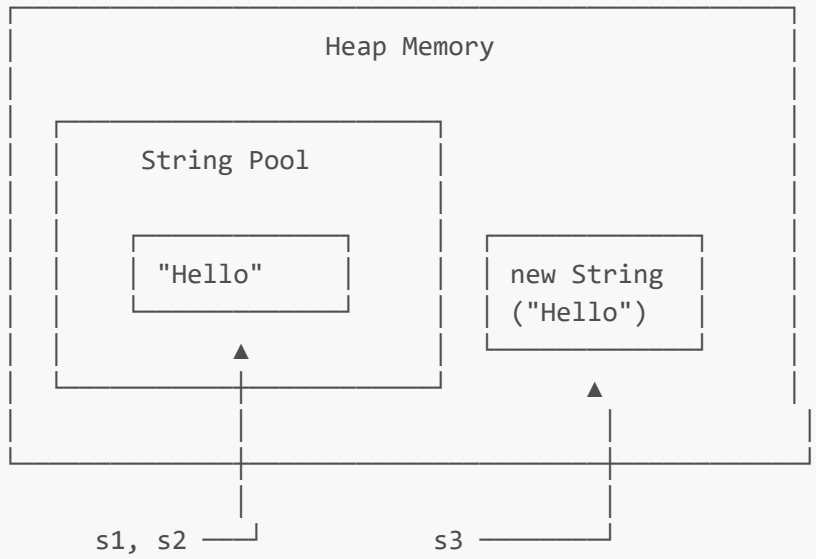
String Immutability and String Pool

String literals are stored in a special memory area called the "String Pool."

```
String s1 = "Hello";  // Creates a new string in the pool
String s2 = "Hello";  // Reuses the existing string from the pool

// Explicitly interning a string
String s3 = new String("Hello").intern();  // Forces storage in the pool
boolean sameRef = (s1 == s3);  // true after interning
```

Immutability benefits:

- Thread safety (can be safely shared between threads)
- Security (values cannot be changed unexpectedly)
- Hashcode caching (improves performance in collections)
- String pool optimization (memory efficiency)



StringBuilder and StringBuffer

For efficient string manipulation, use StringBuilder (not thread-safe) or StringBuffer (thread-safe):

```
// StringBuilder (preferred for single-threaded scenarios)
StringBuilder sb = new StringBuilder();
sb.append("Hello");
sb.append(" ");
sb.append("World");
String result = sb.toString();  // "Hello World"

// Method chaining
String chained = new StringBuilder()
    .append("Java")
    .append(" ")
    .append(17)
    .toString();  // "Java 17"

// Inserting
StringBuilder sb2 = new StringBuilder("Hello World");
sb2.insert(6, "Beautiful ");  // "Hello Beautiful World"

// Deleting
sb2.delete(6, 16);  // "Hello World"

// Replacing
sb2.replace(0, 5, "Hi");  // "Hi World"

// Reversing
sb2.reverse();  // "dlroW iH"
```

Comparing String, StringBuilder, and StringBuffer:

Feature	String	StringBuilder	StringBuffer
Mutability	Immutable	Mutable	Mutable
Thread Safety	Thread-safe	Not thread-safe	Thread-safe
Performance	Slower for concat	Fast for concat	Slower than StringBuilder
Use Case	Simple operations	Single-thread string building	Multi-thread string building

```
// Performance comparison for concatenation
String s = "";
long start = System.currentTimeMillis();
for (int i = 0; i < 100000; i++) {
    s += "a";  // Creates a new String object each time
}
System.out.println("String time: " + (System.currentTimeMillis() - start) + "ms");

StringBuilder sb = new StringBuilder();
start = System.currentTimeMillis();
for (int i = 0; i < 100000; i++) {
    sb.append("a");  // Modifies existing object
```

```
}
System.out.println("StringBuilder time: " + (System.currentTimeMillis() - start) +
"ms");
```

Text Blocks (Java 15+)

Text blocks provide a way to create multi-line strings without escape sequences:

```
// Before Java 15
String html = "<html>\n" +
              "    <body>\n" +
              "        <p>Hello, World!</p>\n" +
              "    </body>\n" +
              "</html>";

// With text blocks (Java 15+)
String html = """
              <html>
                  <body>
                      <p>Hello, World!</p>
                  </body>
              </html>
              """;
```

> **Common Pitfall**: Inconsistent indentation in text blocks can lead to unexpected whitespace in the resulting string.

String Templates (Preview feature in Java 21)

String templates are a preview feature in Java 21 that provides a more concise way to embed expressions within strings:

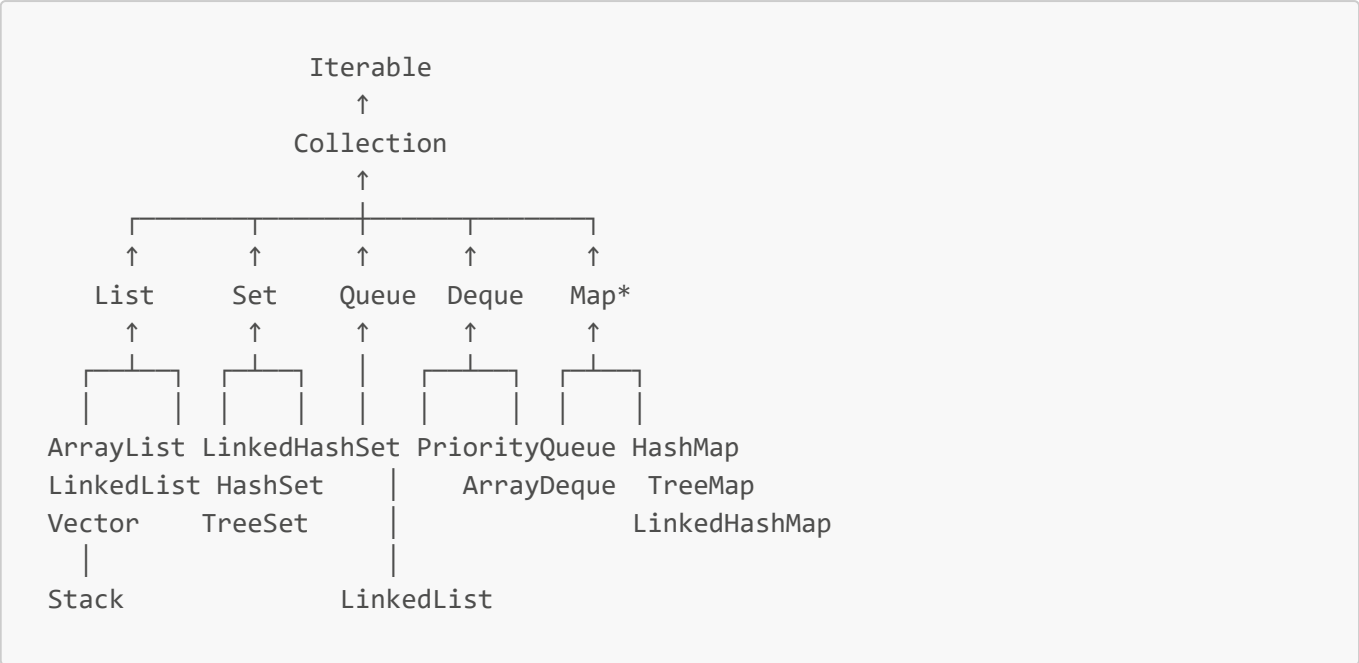
```
// Traditional string concatenation
String name = "Alice";
int age = 30;
String message = "Hello, " + name + "! You are " + age + " years old.";

// Using String templates (Java 21 preview)
String templateMessage = STR."Hello, \{name}! You are \{age} years old.";
```

> **Certification Note**: Understand the differences between String, StringBuilder, and StringBuffer, and know when to use each. Be aware of the performance implications of string concatenation in loops.

Collections Framework

The Java Collections Framework provides a unified architecture for representing and manipulating groups of objects. It includes interfaces, implementations, and algorithms to operate on collections.

Collection Hierarchy



*Map doesn't extend Collection, but is part of the Collections Framework.

Common Collection Interfaces

Collection<E>

The root interface in the collection hierarchy:

- `boolean add(E e)`: Adds an element
- `boolean remove(Object o)`: Removes an element
- `boolean contains(Object o)`: Checks if element exists
- `int size()`: Returns number of elements
- `boolean isEmpty()`: Checks if collection is empty
- `void clear()`: Removes all elements
- `Iterator<E> iterator()`: Returns an iterator

List<E>

An ordered collection (sequence) that allows duplicate elements:

- `E get(int index)`: Retrieves element at position
- `E set(int index, E element)`: Replaces element at position
- `void add(int index, E element)`: Inserts element at position
- `E remove(int index)`: Removes element at position
- `int indexOf(Object o)`: Returns first index of element
- `int lastIndexOf(Object o)`: Returns last index of element
- `List<E> subList(int fromIndex, int toIndex)`: Returns a view of portion of list

Set<E>

A collection that cannot contain duplicate elements:

- Same methods as Collection with additional behavior (no duplicates)

Queue<E>

A collection for holding elements prior to processing, typically in FIFO order:

- `boolean offer(E e)`: Adds element (returns false if full)
- `E poll()`: Retrieves and removes head (returns null if empty)
- `E peek()`: Retrieves but doesn't remove head (returns null if empty)

Deque<E>

A double-ended queue that supports element insertion and removal at both ends:

- `void addFirst(E e)`, `boolean offerFirst(E e)`: Adds at front
- `void addLast(E e)`, `boolean offerLast(E e)`: Adds at end
- `E removeFirst()`, `E pollFirst()`: Removes from front
- `E removeLast()`, `E pollLast()`: Removes from end
- `E getFirst()`, `E peekFirst()`: Retrieves from front
- `E getLast()`, `E peekLast()`: Retrieves from end

Map<K,V>

An object that maps keys to values with no duplicate keys:

- `V put(K key, V value)`: Associates value with key
- `V get(Object key)`: Returns value for key
- `V remove(Object key)`: Removes mapping for key
- `boolean containsKey(Object key)`: Checks if key exists
- `boolean containsValue(Object value)`: Checks if value exists
- `Set<K> keySet()`: Returns set view of keys
- `Collection<V> values()`: Returns collection view of values
- `Set<Map.Entry<K,V>> entrySet()`: Returns set view of key-value mappings

Common Collection Implementations

Lists

```
// ArrayList - Resizable array implementation
List<String> arrayList = new ArrayList<>();
arrayList.add("Apple");
arrayList.add("Banana");
arrayList.add("Cherry");
String second = arrayList.get(1);  // Banana
arrayList.remove(0);               // Removes Apple

// LinkedList - Doubly-linked list implementation
List<String> linkedList = new LinkedList<>();
linkedList.add("Dog");
```

```
linkedList.add("Cat");
linkedList.add(0, "Bird");  // Adds at beginning
String first = linkedList.get(0);  // Bird

// Vector - Synchronized array implementation (thread-safe)
List<String> vector = new Vector<>();
vector.add("Red");
vector.add("Green");
vector.add("Blue");

// Stack - LIFO stack (extends Vector)
Stack<Integer> stack = new Stack<>();
stack.push(1);
stack.push(2);
stack.push(3);
int top = stack.pop();     // 3
int peek = stack.peek();   // 2
boolean empty = stack.empty();  // false
```

Sets

```
// HashSet - Uses HashMap, no order guarantee, O(1) operations
Set<String> hashSet = new HashSet<>();
hashSet.add("Apple");
hashSet.add("Banana");
hashSet.add("Apple");  // Duplicate, not added
System.out.println(hashSet.size());  // 2

// LinkedHashSet - Uses LinkedHashMap, maintains insertion order
Set<String> linkedHashSet = new LinkedHashSet<>();
linkedHashSet.add("Cat");
linkedHashSet.add("Dog");
linkedHashSet.add("Bird");
// Iteration order: Cat, Dog, Bird

// TreeSet - Uses TreeMap, sorted order, O(log n) operations
Set<String> treeSet = new TreeSet<>();
treeSet.add("Zebra");
treeSet.add("Monkey");
treeSet.add("Lion");
// Iteration order: Lion, Monkey, Zebra (natural order)

// EnumSet - Specialized Set for enum types, very efficient
enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY }
EnumSet<Day> weekdays = EnumSet.range(Day.MONDAY, Day.FRIDAY);
EnumSet<Day> weekend = EnumSet.of(Day.SATURDAY, Day.SUNDAY);
```

Maps

```
// HashMap - Hash table implementation, no order guarantee
Map<String, Integer> hashMap = new HashMap<>();
hashMap.put("apple", 10);
hashMap.put("banana", 20);
hashMap.put("cherry", 30);
int value = hashMap.get("banana");  // 20
hashMap.put("banana", 25);  // Updates value
hashMap.remove("apple");    // Removes key-value pair

// LinkedHashMap - Hash table with linked list, maintains insertion order
Map<String, Double> linkedHashMap = new LinkedHashMap<>();
linkedHashMap.put("John", 3.8);
linkedHashMap.put("Alice", 4.0);
linkedHashMap.put("Bob", 3.5);
// Iteration order: John, Alice, Bob

// TreeMap - Red-black tree implementation, sorted by keys
Map<String, Integer> treeMap = new TreeMap<>();
treeMap.put("zebra", 90);
treeMap.put("monkey", 80);
treeMap.put("lion", 70);
// Iteration order: lion, monkey, zebra

// EnumMap - Specialized Map for enum keys
Map<Day, String> schedule = new EnumMap<>(Day.class);
schedule.put(Day.MONDAY, "Work");
schedule.put(Day.SATURDAY, "Relax");
```

Queues and Deques

```
// PriorityQueue - Heap implementation, natural ordering by default
Queue<Integer> priorityQueue = new PriorityQueue<>();
priorityQueue.offer(5);
priorityQueue.offer(1);
priorityQueue.offer(3);
int first = priorityQueue.poll();  // 1 (smallest element)
int peek = priorityQueue.peek();   // 3 (next smallest)

// ArrayDeque - Resizable array implementation of Deque
Deque<String> arrayDeque = new ArrayDeque<>();
arrayDeque.addFirst("First");
arrayDeque.addLast("Last");
String removeFirst = arrayDeque.removeFirst();  // "First"
arrayDeque.offerFirst("New First");
arrayDeque.offerLast("New Last");
String peekLast = arrayDeque.peekLast();        // "New Last"

// LinkedList - Also implements Deque
Deque<Integer> linkedListDeque = new LinkedList<>();
linkedListDeque.push(1);  // Adds to front
```

```
linkedListDeque.push(2);
linkedListDeque.push(3);
int popValue = linkedListDeque.pop();  // 3 (LIFO behavior)
```

Choosing the Right Collection

Collection Type	When to Use
ArrayList	Fast random access, dynamic array, frequent reads, infrequent modifications
LinkedList	Frequent insertions/deletions, especially at ends or middle
HashSet	Fast lookups, no duplicates, order not important
LinkedHashSet	Fast lookups, no duplicates, iteration order important
TreeSet	Sorted set, range operations, ordered iteration
HashMap	Fast key-based lookups, no duplicate keys, order not important
LinkedHashMap	Fast lookups, predictable iteration order
TreeMap	Sorted map, range operations, ordered key iteration
PriorityQueue	Priority-based element processing (smallest/largest first)
ArrayDeque	Efficient stack or queue operations from both ends

Iterating Collections

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

// Using Iterator
Iterator<String> iterator = names.iterator();
while (iterator.hasNext()) {
    String name = iterator.next();
    System.out.println(name);
}

// Enhanced for loop (for-each)
for (String name : names) {
    System.out.println(name);
}

// Using forEach method (Java 8+)
names.forEach(name -> System.out.println(name));

// Using forEach method with method reference (Java 8+)
names.forEach(System.out::println);

// ListIterator (for Lists only) - can iterate forwards and backwards
ListIterator<String> listIterator = names.listIterator();
while (listIterator.hasNext()) {
```

```
    int index = listIterator.nextIndex();
    String name = listIterator.next();
    System.out.println(index + ": " + name);
}

// Iterate backwards
while (listIterator.hasPrevious()) {
    System.out.println(listIterator.previous());
}

// Iterating Maps
Map<String, Integer> scores = Map.of("Alice", 95, "Bob", 85, "Charlie", 90);

// Iterating over entries
for (Map.Entry<String, Integer> entry : scores.entrySet()) {
    System.out.println(entry.getKey() + ": " + entry.getValue());
}

// Iterating over keys
for (String key : scores.keySet()) {
    System.out.println(key);
}

// Iterating over values
for (Integer value : scores.values()) {
    System.out.println(value);
}

// Using forEach (Java 8+)
scores.forEach((k, v) -> System.out.println(k + ": " + v));
```

Removing Elements During Iteration

```
List<Integer> numbers = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));

// INCORRECT: Will throw ConcurrentModificationException
for (Integer number : numbers) {
    if (number % 2 == 0) {
        numbers.remove(number);  // DON'T DO THIS
    }
}

// Correct approach 1: Using Iterator's remove method
Iterator<Integer> iterator = numbers.iterator();
while (iterator.hasNext()) {
    Integer number = iterator.next();
    if (number % 2 == 0) {
        iterator.remove();  // Safe removal during iteration
    }
}
```

```
// Correct approach 2: Using removeIf (Java 8+)
numbers.removeIf(number -> number % 2 == 0);
```

Unmodifiable Collections

Java provides several ways to create immutable or unmodifiable collections:

```
// Unmodifiable views (pre-Java 9)
List<String> mutableList = new ArrayList<>();
mutableList.add("One");
mutableList.add("Two");

List<String> unmodifiableList = Collections.unmodifiableList(mutableList);
// unmodifiableList.add("Three");  // Throws UnsupportedOperationException

// Factory methods for immutable collections (Java 9+)
List<String> immutableList = List.of("One", "Two", "Three");
Set<Integer> immutableSet = Set.of(1, 2, 3);
Map<String, Integer> immutableMap = Map.of("One", 1, "Two", 2, "Three", 3);

// For larger maps (Java 9+)
Map<String, Integer> largeMap = Map.ofEntries(
    Map.entry("One", 1),
    Map.entry("Two", 2),
    Map.entry("Three", 3),
    Map.entry("Four", 4)
);

// Collectors to unmodifiable collections (Java 10+)
List<String> unmodifiableCollectedList = Stream.of("One", "Two", "Three")
    .collect(Collectors.toUnmodifiableList());
```

> **Important**: Modifying the backing collection (`mutableList` in the example) will affect the unmodifiable view. For truly immutable collections, use the factory methods (`List.of()`, etc.).

Concurrent Collections

Java provides thread-safe collections in the `java.util.concurrent` package:

```
// ConcurrentHashMap - A thread-safe hash-based Map
Map<String, Integer> concurrentMap = new ConcurrentHashMap<>();
concurrentMap.put("One", 1);
concurrentMap.put("Two", 2);
// Safe for concurrent access from multiple threads

// CopyOnWriteArrayList - Thread-safe List with thread isolation for iterators
List<String> copyOnWriteList = new CopyOnWriteArrayList<>();
copyOnWriteList.add("One");
copyOnWriteList.add("Two");
```

```
// CopyOnWriteArraySet - Thread-safe Set based on CopyOnWriteArrayList
Set<String> copyOnWriteSet = new CopyOnWriteArraySet<>();
copyOnWriteSet.add("One");
copyOnWriteSet.add("Two");

// ConcurrentSkipListMap - Concurrent NavigableMap implementation (sorted)
NavigableMap<String, Integer> skipListMap = new ConcurrentSkipListMap<>();
skipListMap.put("Z", 26);
skipListMap.put("A", 1);
// Iteration is naturally ordered alphabetically
```

Collection Utilities

The `Collections` class provides various utility methods:

```
List<Integer> numbers = new ArrayList<>(Arrays.asList(3, 1, 4, 1, 5, 9));

// Sorting
Collections.sort(numbers);  // [1, 1, 3, 4, 5, 9]

// Binary search (on sorted list)
int index = Collections.binarySearch(numbers, 4);  // 3

// Shuffling
Collections.shuffle(numbers);  // Random order

// Frequency
int count = Collections.frequency(numbers, 1);  // 2

// Min and max
int min = Collections.min(numbers);  // 1
int max = Collections.max(numbers);  // 9

// Filling
Collections.fill(numbers, 0);  // [0, 0, 0, 0, 0, 0]

// Reversing
Collections.reverse(numbers);  // [0, 0, 0, 0, 0, 0] (no change in this case)

// Swapping
numbers = Arrays.asList(1, 2, 3, 4, 5);
Collections.swap(numbers, 0, 4);  // [5, 2, 3, 4, 1]

// Synchronized (thread-safe) wrappers
List<String> syncList = Collections.synchronizedList(new ArrayList<>());
Map<String, Integer> syncMap = Collections.synchronizedMap(new HashMap<>());
```

Arrays Utility Class

The `Arrays` class provides utility methods for arrays:

```
// Creating and initializing
int[] numbers = new int[5];
Arrays.fill(numbers, 10);  // [10, 10, 10, 10, 10]

int[] moreNumbers = {5, 2, 8, 1, 9};

// Sorting
Arrays.sort(moreNumbers);  // [1, 2, 5, 8, 9]

// Binary search (on sorted array)
int index = Arrays.binarySearch(moreNumbers, 5);  // 2

// Comparing
int[] numbers1 = {1, 2, 3};
int[] numbers2 = {1, 2, 3};
boolean equal = Arrays.equals(numbers1, numbers2);  // true

// Converting to List
String[] fruits = {"Apple", "Banana", "Cherry"};
List<String> fruitList = Arrays.asList(fruits);  // Fixed-size backed list

// Copying
int[] copy = Arrays.copyOf(moreNumbers, moreNumbers.length);
int[] partial = Arrays.copyOfRange(moreNumbers, 1, 4);  // [2, 5, 8]

// Parallel operations (Java 8+)
Arrays.parallelSort(moreNumbers);
```

> **Certification Note**: Know the performance characteristics of different collection implementations. For example, `ArrayList` provides O(1) random access but O(n) insertions/deletions in the middle, while `LinkedList` provides O(1) insertions/deletions (with a reference) but O(n) random access.

Exception Handling and Custom Exceptions

Exception handling is a mechanism to handle runtime errors that might occur during program execution. Java uses a hierarchical exception model.

Exception Hierarchy

```
              Throwable
              /      \
         Error       Exception
         /           /       \
OutOfMemoryError   IOException  RuntimeException
                   /            /        \
         FileNotFoundException  NullPointerException
ArrayIndexOutOfBoundsException
```

- **Throwable**: The root class of all errors and exceptions
- **Error**: Serious problems that applications should not try to handle (e.g., `OutOfMemoryError`)
- **Exception**: Conditions that applications might want to handle
  - **Checked Exceptions**: Must be caught or declared (e.g., `IOException`)
  - **Unchecked Exceptions**: Runtime exceptions, not required to be caught (e.g., `NullPointerException`)

Basic Exception Handling

```
// Try-catch block
try {
    FileReader file = new FileReader("file.txt");
    int data = file.read();
    file.close();
} catch (FileNotFoundException e) {
    System.err.println("File not found: " + e.getMessage());
} catch (IOException e) {
    System.err.println("Error reading file: " + e.getMessage());
}

// Multi-catch (Java 7+)
try {
    // Code that might throw exceptions
} catch (FileNotFoundException | NullPointerException e) {
    // Handle both exception types the same way
    System.err.println("Error: " + e.getMessage());
}

// Finally block (always executes)
FileReader reader = null;
try {
    reader = new FileReader("file.txt");
    // Process file
} catch (IOException e) {
    System.err.println("Error: " + e.getMessage());
} finally {
    // Clean-up code, always executes
    if (reader != null) {
        try {
            reader.close();
        } catch (IOException e) {
            System.err.println("Error closing file: " + e.getMessage());
        }
    }
}
```

Try-with-Resources (Java 7+)

The try-with-resources statement automatically closes resources that implement `AutoCloseable` or `Closeable`.

```
// Pre-Java 7 (verbose)
BufferedReader br = null;
try {
    br = new BufferedReader(new FileReader("file.txt"));
    String line = br.readLine();
    System.out.println(line);
} catch (IOException e) {
    System.err.println("Error: " + e.getMessage());
} finally {
    if (br != null) {
        try {
            br.close();
        } catch (IOException e) {
            System.err.println("Error closing reader: " + e.getMessage());
        }
    }
}

// Java 7+ (concise, with automatic resource management)
try (BufferedReader br = new BufferedReader(new FileReader("file.txt"))) {
    String line = br.readLine();
    System.out.println(line);
} catch (IOException e) {
    System.err.println("Error: " + e.getMessage());
}

// Multiple resources
try (FileInputStream input = new FileInputStream("input.txt");
     FileOutputStream output = new FileOutputStream("output.txt")) {
    // Process files
} catch (IOException e) {
    System.err.println("Error: " + e.getMessage());
}
```

> **Common Pitfall**: Forgetting that resources are closed in reverse order of their creation. If resource B depends on resource A, create A first, then B.

Exception Propagation

```
public void method1() {
    method2();
}

public void method2() {
    method3();
}
```

```
public void method3() throws IOException {
    throw new IOException("Error in method3");
}

// To call method1, you must handle or declare the IOException
public void caller() {
    try {
        method1();
    } catch (IOException e) {
        System.err.println("Caught: " + e.getMessage());
    }
}
```

Creating Custom Exceptions

```
// Checked custom exception
public class InsufficientFundsException extends Exception {
    private double amount;

    public InsufficientFundsException(String message, double amount) {
        super(message);
        this.amount = amount;
    }

    public double getAmount() {
        return amount;
    }
}

// Unchecked custom exception
public class InvalidUserException extends RuntimeException {
    private String userId;

    public InvalidUserException(String message, String userId) {
        super(message);
        this.userId = userId;
    }

    public String getUserId() {
        return userId;
    }
}
```

Usage:

```
public class BankAccount {
    private double balance;

    public void withdraw(double amount) throws InsufficientFundsException {
```

```
        if (amount > balance) {
            throw new InsufficientFundsException(
                "Not enough funds. Required: " + amount + ", Available: " +
balance,
                amount - balance
            );
        }
        balance -= amount;
    }

    public void validateUser(String userId) {
        if (userId == null || userId.isEmpty()) {
            throw new InvalidUserException("Invalid user ID: " + userId, userId);
        }
        // Proceed with validation
    }
}
```

Exception Handling Best Practices

1. **Only Catch What You Can Handle**: Don't catch exceptions you can't properly handle.
2. **Catch Specific Exceptions First**: When using multiple catch blocks, order them from most specific to most general.

```
try {
    // Code that might throw exceptions
} catch (FileNotFoundException e) {
    // Most specific
} catch (IOException e) {
    // More general
} catch (Exception e) {
    // Most general
}
```

3. **Don't Swallow Exceptions**: Always log or handle exceptions meaningfully.

```
// BAD
try {
    riskyOperation();
} catch (Exception e) {
    // Empty catch block - don't do this!
}

// GOOD
try {
    riskyOperation();
} catch (Exception e) {
    logger.error("Error during operation", e);
```

```
    // Or rethrow as a more appropriate exception
    throw new ServiceException("Service failed", e);
}
```

4. **Use Try-with-Resources**: For automatic resource management.

5. **Include Cause When Rethrowing**: Preserve the original exception's stack trace.

```
try {
    // Code that might throw exceptions
} catch (IOException e) {
    throw new ServiceException("Service failed", e);  // Include original as cause
}
```

6. **Document Exceptions in Javadoc**:

```
/**
 * Processes the file data.
 *
 * @param filePath Path to the file
 * @throws IOException If file cannot be read
 * @throws InvalidDataException If file contains invalid data
 */
public void processFile(String filePath) throws IOException, InvalidDataException
{
    // Implementation
}
```

> **Certification Note**: Understand the difference between checked and unchecked exceptions. Know that `Error` and its subclasses, along with `RuntimeException` and its subclasses, are unchecked. All other `Exception` subclasses are checked.

Java I/O and NIO.2

Java provides comprehensive I/O (Input/Output) capabilities through the `java.io` package and enhanced NIO.2 (New I/O) in the `java.nio` package.

Java I/O Overview

The I/O package provides stream-based I/O with two main hierarchies:

1. **Byte Streams**: For binary data (`InputStream`, `OutputStream`)
2. **Character Streams**: For text data (`Reader`, `Writer`)

```
InputStream                    OutputStream
   ↑                              ↑
   ├── FileInputStream            ├── FileOutputStream
```

```
    ├── ByteArrayInputStream        ├── ByteArrayOutputStream
    ├── ObjectInputStream           ├── ObjectOutputStream
    ├── PipedInputStream            ├── PipedOutputStream
    └── FilterInputStream           └── FilterOutputStream
        ↑                               ↑
        ├── BufferedInputStream         ├── BufferedOutputStream
        ├── DataInputStream             ├── DataOutputStream
        └── ...                         └── ...

Reader                          Writer
   ↑                               ↑
    ├── InputStreamReader           ├── OutputStreamWriter
    │   ↑                           │   ↑
    │   └── FileReader              │   └── FileWriter
    ├── StringReader                ├── StringWriter
    ├── CharArrayReader             ├── CharArrayWriter
    ├── PipedReader                 ├── PipedWriter
    └── BufferedReader              └── BufferedWriter
```

File I/O with Streams

Reading from a File (Byte Stream)

```
// Reading bytes from a file
try (FileInputStream fis = new FileInputStream("data.bin")) {
    int byteData;
    while ((byteData = fis.read()) != -1) {
        // Process each byte
        System.out.print(byteData + " ");
    }
} catch (IOException e) {
    System.err.println("Error: " + e.getMessage());
}
```

Writing to a File (Byte Stream)

```
// Writing bytes to a file
try (FileOutputStream fos = new FileOutputStream("output.bin")) {
    byte[] data = {65, 66, 67, 68, 69};  // ASCII for "ABCDE"
    fos.write(data);
} catch (IOException e) {
    System.err.println("Error: " + e.getMessage());
}
```

Reading Text Files (Character Stream)

```
// Reading a text file line by line
try (BufferedReader reader = new BufferedReader(new FileReader("input.txt"))) {
    String line;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
} catch (IOException e) {
    System.err.println("Error: " + e.getMessage());
}
```

Writing Text Files (Character Stream)

```
// Writing to a text file
try (BufferedWriter writer = new BufferedWriter(new FileWriter("output.txt"))) {
    writer.write("Hello, Java I/O!");
    writer.newLine();
    writer.write("This is a new line.");
} catch (IOException e) {
    System.err.println("Error: " + e.getMessage());
}
```

Buffered Streams

Buffered streams improve I/O performance by minimizing the number of system calls:

```
// Unbuffered - less efficient
try (FileInputStream fis = new FileInputStream("large-file.dat")) {
    int data;
    while ((data = fis.read()) != -1) {
        // Process each byte
    }
}

// Buffered - more efficient
try (BufferedInputStream bis = new BufferedInputStream(
        new FileInputStream("large-file.dat"))) {
    int data;
    while ((data = bis.read()) != -1) {
        // Process each byte
    }
}
```

Data Streams

For reading and writing primitive data types:

```
// Writing primitive data
try (DataOutputStream dos = new DataOutputStream(
        new FileOutputStream("data.dat"))) {
    dos.writeInt(42);
    dos.writeDouble(3.14);
    dos.writeUTF("Hello, Data Streams!");
} catch (IOException e) {
    System.err.println("Error: " + e.getMessage());
}

// Reading primitive data
try (DataInputStream dis = new DataInputStream(
        new FileInputStream("data.dat"))) {
    int intValue = dis.readInt();
    double doubleValue = dis.readDouble();
    String stringValue = dis.readUTF();

    System.out.println("Int: " + intValue);
    System.out.println("Double: " + doubleValue);
    System.out.println("String: " + stringValue);
} catch (IOException e) {
    System.err.println("Error: " + e.getMessage());
}
```

Object Serialization

For converting objects to byte streams and back:

```
// Serializable class
public class Person implements Serializable {
    private static final long serialVersionUID = 1L;  // Important for version
control

    private String name;
    private int age;
    private transient String password;  // Will not be serialized

    // Constructor, getters, setters...
}

// Serializing an object
try (ObjectOutputStream oos = new ObjectOutputStream(
        new FileOutputStream("person.ser"))) {
    Person person = new Person("John Doe", 30);
    oos.writeObject(person);
} catch (IOException e) {
    System.err.println("Error: " + e.getMessage());
}

// Deserializing an object
try (ObjectInputStream ois = new ObjectInputStream(
```

```
            new FileInputStream("person.ser"))) {
        Person person = (Person) ois.readObject();
        System.out.println("Name: " + person.getName());
        System.out.println("Age: " + person.getAge());
    } catch (IOException | ClassNotFoundException e) {
        System.err.println("Error: " + e.getMessage());
    }
```

> **Common Pitfall**: When deserializing objects, ensure the class is available in the classpath and has a compatible `serialVersionUID`.

NIO.2 (Java 7+)

NIO.2 introduced in Java 7 provides enhanced file I/O with the `java.nio.file` package.

Path and Paths

```
// Creating Path objects
Path path1 = Paths.get("data.txt");
Path path2 = Paths.get("/home", "user", "documents", "file.txt");
Path path3 = FileSystems.getDefault().getPath("config", "settings.xml");

// Path information
System.out.println("File name: " + path1.getFileName());
System.out.println("Parent: " + path1.getParent());
System.out.println("Root: " + path1.getRoot());
System.out.println("Absolute: " + path1.isAbsolute());

// Path operations
Path normalizedPath = path1.normalize();  // Removes redundancies
Path resolvedPath = path1.resolve("subdir/file.txt");  // Combines paths
Path relativePath = path1.relativize(path2);  // Creates relative path
```

Files Utility Class

```
// Checking file properties
Path filePath = Paths.get("data.txt");
boolean exists = Files.exists(filePath);
boolean isReadable = Files.isReadable(filePath);
boolean isDirectory = Files.isDirectory(filePath);

// Reading all bytes/lines at once
byte[] bytes = Files.readAllBytes(filePath);
List<String> lines = Files.readAllLines(filePath, StandardCharsets.UTF_8);

// Writing all bytes/lines at once
Path newFile = Paths.get("new-file.txt");
Files.write(newFile, "Hello, NIO.2!".getBytes());
```

```
Files.write(newFile, Arrays.asList("Line 1", "Line 2"), StandardCharsets.UTF_8);

// Copying and moving files
Path source = Paths.get("source.txt");
Path target = Paths.get("target.txt");
Files.copy(source, target, StandardCopyOption.REPLACE_EXISTING);
Files.move(source, target, StandardCopyOption.REPLACE_EXISTING);

// Deleting files
Files.delete(filePath);  // Throws exception if file doesn't exist
Files.deleteIfExists(filePath);  // No exception if file doesn't exist

// Creating directories
Path dirPath = Paths.get("nested/directories");
Files.createDirectories(dirPath);
```

File Attributes

```
// Basic file attributes
BasicFileAttributes attr = Files.readAttributes(filePath,
BasicFileAttributes.class);
System.out.println("Creation time: " + attr.creationTime());
System.out.println("Last access time: " + attr.lastAccessTime());
System.out.println("Last modified time: " + attr.lastModifiedTime());
System.out.println("Size: " + attr.size());
System.out.println("Is directory: " + attr.isDirectory());
System.out.println("Is regular file: " + attr.isRegularFile());

// Setting attributes
Files.setAttribute(filePath, "basic:lastModifiedTime",
    FileTime.fromMillis(System.currentTimeMillis()));

// POSIX file permissions (on supporting file systems)
if (FileSystems.getDefault().supportedFileAttributeViews().contains("posix")) {
    PosixFileAttributes posixAttr = Files.readAttributes(filePath,
PosixFileAttributes.class);
    Set<PosixFilePermission> permissions = posixAttr.permissions();

    // Add user write permission
    permissions.add(PosixFilePermission.OWNER_WRITE);
    Files.setPosixFilePermissions(filePath, permissions);
}
```

Directory Operations

```
// Listing directory contents
Path dir = Paths.get("src");
try (DirectoryStream<Path> stream = Files.newDirectoryStream(dir)) {
```

```
    for (Path entry : stream) {
        System.out.println(entry.getFileName());
    }
}

// Filtered directory stream
try (DirectoryStream<Path> stream =
        Files.newDirectoryStream(dir, "*.java")) {
    for (Path entry : stream) {
        System.out.println(entry.getFileName());
    }
}

// Traversing a directory tree
Files.walkFileTree(dir, new SimpleFileVisitor<Path>() {
    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) {
        System.out.println("Visited file: " + file);
        return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult visitFileFailed(Path file, IOException exc) {
        System.err.println("Failed to visit: " + file);
        return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs)
{
        System.out.println("About to visit directory: " + dir);
        return FileVisitResult.CONTINUE;
    }
});
```

File Change Notification (WatchService)

```
// Watch a directory for changes
Path dir = Paths.get("src");
WatchService watchService = FileSystems.getDefault().newWatchService();

dir.register(
    watchService,
    StandardWatchEventKinds.ENTRY_CREATE,
    StandardWatchEventKinds.ENTRY_MODIFY,
    StandardWatchEventKinds.ENTRY_DELETE
);

boolean poll = true;
while (poll) {
    WatchKey key = watchService.take();  // Blocks until events occur
```

```
        for (WatchEvent<?> event : key.pollEvents()) {
            WatchEvent.Kind<?> kind = event.kind();

            if (kind == StandardWatchEventKinds.OVERFLOW) {
                continue;  // Overflow event, some events might have been lost
            }

            @SuppressWarnings("unchecked")
            WatchEvent<Path> pathEvent = (WatchEvent<Path>) event;
            Path fileName = pathEvent.context();

            System.out.println(kind + ": " + fileName);
        }

        // Reset key and check if still valid
        boolean valid = key.reset();
        if (!valid) {
            break;  // Directory no longer accessible
        }
    }
```

> **Certification Note**: Understand the differences between traditional I/O and NIO.2, especially the performance benefits of non-blocking I/O and the convenience methods in the `Files` class.

Generics and Type Safety

Generics enable types (classes and interfaces) to be parameters when defining classes, interfaces, and methods. They provide compile-time type safety and eliminate the need for explicit casting.

Generic Classes

```
// Generic class with type parameter T
public class Box<T> {
    private T content;

    public Box(T content) {
        this.content = content;
    }

    public T getContent() {
        return content;
    }

    public void setContent(T content) {
        this.content = content;
    }
}

// Usage
Box<String> stringBox = new Box<>("Hello, Generics!");
```

```
String content = stringBox.getContent();  // No casting needed

Box<Integer> intBox = new Box<>(42);
Integer intValue = intBox.getContent();
```

Multiple Type Parameters

```
// Class with multiple type parameters
public class Pair<K, V> {
    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey() { return key; }
    public V getValue() { return value; }

    public void setKey(K key) { this.key = key; }
    public void setValue(V value) { this.value = value; }
}

// Usage
Pair<String, Integer> person = new Pair<>("John", 25);
String name = person.getKey();
Integer age = person.getValue();
```

Generic Methods

```
// Generic method
public static <T> boolean contains(T[] array, T element) {
    for (T item : array) {
        if (item.equals(element)) {
            return true;
        }
    }
    return false;
}

// Usage
Integer[] numbers = {1, 2, 3, 4, 5};
boolean hasThree = contains(numbers, 3);  // true

String[] names = {"Alice", "Bob", "Charlie"};
boolean hasJohn = contains(names, "John");  // false
```

Type Bounds

```
// Upper bound - T must be or extend Comparable
public static <T extends Comparable<T>> T findMax(T[] array) {
    if (array == null || array.length == 0) {
        return null;
    }

    T max = array[0];
    for (int i = 1; i < array.length; i++) {
        if (array[i].compareTo(max) > 0) {
            max = array[i];
        }
    }
    return max;
}

// Multiple bounds - T must implement both interfaces
public static <T extends Comparable<T> & Serializable> void process(T item) {
    // Process item that is both Comparable and Serializable
}
```

Wildcards

```
// Unbounded wildcard - any type
public static void printList(List<?> list) {
    for (Object elem : list) {
        System.out.println(elem);
    }
}

// Upper bounded wildcard - Number or any subclass
public static double sum(List<? extends Number> list) {
    double sum = 0.0;
    for (Number num : list) {
        sum += num.doubleValue();
    }
    return sum;
}

// Lower bounded wildcard - Integer or any superclass
public static void addIntegers(List<? super Integer> list) {
    list.add(10);
    list.add(20);
    // list.add("string");  // Compile error
}
```

Generic Type Erasure

Generics are implemented using type erasure, which means generic type information is removed at compile time.

```
// This code
List<String> stringList = new ArrayList<>();
stringList.add("Hello");
String str = stringList.get(0);

// Becomes (after type erasure)
List stringList = new ArrayList();
stringList.add("Hello");
String str = (String) stringList.get(0);
```

Consequences of type erasure:

1. Cannot check if an object is an instance of a generic type:

```
// Not allowed
if (obj instanceof List<String>) { ... }

// Allowed
if (obj instanceof List) { ... }
```

2. Cannot create arrays of generic types:

```
// Compile error
Box<Integer>[] boxArray = new Box<Integer>[10];

// Workaround
Box<?>[] boxArray = new Box<?>[10];
```

3. Cannot use primitive types as type arguments:

```
// Not allowed
Box<int> intBox = new Box<>(42);

// Use wrapper classes instead
Box<Integer> intBox = new Box<>(42);
```

PECS Principle: Producer Extends, Consumer Super

```
// Producer - use "extends" when you only read from the collection
public void processElements(List<? extends Number> elements) {
    // Can read elements as Numbers
```

```
    for (Number n : elements) {
        System.out.println(n.doubleValue());
    }

    // Cannot add elements (except null)
    // elements.add(new Integer(1));  // Compile error
}

// Consumer - use "super" when you only write to the collection
public void addElements(List<? super Integer> list) {
    // Can add Integers
    list.add(10);
    list.add(20);

    // Reading is limited to Object
    Object obj = list.get(0);
    // Integer i = list.get(0);  // Compile error
}
```

Type Inference (Java 7+)

Diamond operator for shorter type declarations:

```
// Pre-Java 7
Map<String, List<String>> map = new HashMap<String, List<String>>();

// Java 7+
Map<String, List<String>> map = new HashMap<>();
```

Method type inference for generic methods:

```
// Pre-Java 8
List<String> names = Arrays.<String>asList("Alice", "Bob");

// Java 8+
List<String> names = Arrays.asList("Alice", "Bob");
```

Enhanced type inference in Java 8+:

```
// Method chaining with generics
public static <T> Box<T> createBox(T t) {
    return new Box<>(t);
}

// Java 8+ can determine the return type
Box<String> stringBox = createBox("Hello");
```

> **Common Pitfall**: Using raw types (omitting the type parameter) loses all generic type safety and should be avoided:

```
// Raw type (avoid this)
List rawList = new ArrayList();
rawList.add("string");
rawList.add(42);  // No compile error, but can cause runtime issues

// Always use parameterized types
List<String> stringList = new ArrayList<>();
// stringList.add(42);  // Compile error
```

> **Certification Note**: Understand type erasure and its implications. Know how to use bounded type parameters and wildcards effectively, and remember the PECS principle (Producer Extends, Consumer Super).

Regular Expressions

Regular expressions (regex) provide a powerful way to search, extract, and manipulate text. Java supports regex through the `java.util.regex` package.

Basic Pattern Matching

```
// Check if a string matches a pattern
String text = "Java 21";
boolean matches = text.matches("Java \\d+");  // true

// Using Pattern and Matcher classes
Pattern pattern = Pattern.compile("Java (\\d+)");
Matcher matcher = pattern.matcher(text);

if (matcher.matches()) {
    String version = matcher.group(1);  // "21"
    System.out.println("Version: " + version);
}
```

Common Regex Patterns

```
// Character classes
String digits = "\\d";        // Any digit: [0-9]
String nonDigits = "\\D";     // Any non-digit: [^0-9]
String whitespace = "\\s";    // Any whitespace character
String nonWhitespace = "\\S"; // Any non-whitespace character
String wordChar = "\\w";      // Any word character: [a-zA-Z0-9_]
String nonWordChar = "\\W";   // Any non-word character

// Boundaries
```

```
String start = "^";           // Start of line
String end = "$";             // End of line
String wordBoundary = "\\b";  // Word boundary
String nonWordBoundary = "\\B"; // Non-word boundary

// Quantifiers
String zeroOrOne = "?";       // 0 or 1 occurrence
String zeroOrMore = "*";      // 0 or more occurrences
String oneOrMore = "+";       // 1 or more occurrences
String exactly3 = "{3}";      // Exactly 3 occurrences
String between3And5 = "{3,5}"; // Between 3 and 5 occurrences
String atLeast3 = "{3,}";     // At least 3 occurrences

// Groups and alternation
String group = "(...)";       // Capturing group
String alternation = "a|b";   // a or b
```

Example Patterns

```
// Email pattern
String emailRegex = "^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,}$";
String email = "user@example.com";
boolean isValid = email.matches(emailRegex);  // true

// Phone number pattern (US)
String phoneRegex = "\\(\\d{3}\\) \\d{3}-\\d{4}";
String phone = "(123) 456-7890";
boolean isValidPhone = phone.matches(phoneRegex);  // true

// Date pattern (YYYY-MM-DD)
String dateRegex = "\\d{4}-\\d{2}-\\d{2}";
String date = "2023-05-15";
boolean isValidDate = date.matches(dateRegex);  // true
```

Finding Patterns

```
String text = "Java 8 was released in 2014, Java 11 in 2018, and Java 17 in
2021.";
Pattern pattern = Pattern.compile("Java (\\d+)");
Matcher matcher = pattern.matcher(text);

// Find all occurrences
while (matcher.find()) {
    System.out.println("Found: " + matcher.group());
    System.out.println("Version: " + matcher.group(1));
    System.out.println("Start: " + matcher.start());
    System.out.println("End: " + matcher.end());
}
```

Replacing Patterns

```
// Simple replacement
String censored = "The password is 123456".replaceAll("\\d+", "******");
// "The password is ******"

// Using group references in replacement
String text = "John Smith";
String swapped = text.replaceAll("(\\w+)\\s+(\\w+)", "$2, $1");
// "Smith, John"

// Using replacement callback (Java 9+)
Pattern pattern = Pattern.compile("\\d+");
Matcher matcher = pattern.matcher("Values: 10, 20, 30");

StringBuffer result = new StringBuffer();
while (matcher.find()) {
    int value = Integer.parseInt(matcher.group());
    int doubled = value * 2;
    matcher.appendReplacement(result, String.valueOf(doubled));
}
matcher.appendTail(result);
System.out.println(result.toString());  // "Values: 20, 40, 60"
```

Splitting Strings

```
// Split by comma and optional whitespace
String text = "apple,banana, orange,grape";
String[] fruits = text.split(",\\s*");
// ["apple", "banana", "orange", "grape"]

// Split but limit results
String numbers = "1,2,3,4,5";
String[] limited = numbers.split(",", 3);
// ["1", "2", "3,4,5"]

// Split with regex
String data = "apple:12,banana:25,orange:8";
String[] items = data.split("[,:]");
// ["apple", "12", "banana", "25", "orange", "8"]
```

Flags and Options

```
// Case-insensitive matching
Pattern pattern = Pattern.compile("java", Pattern.CASE_INSENSITIVE);
Matcher matcher = pattern.matcher("Java, JAVA, java");
while (matcher.find()) {
    System.out.println("Found at: " + matcher.start());
```

```
}

// Multiple flags
Pattern multiline = Pattern.compile("^line",
    Pattern.MULTILINE | Pattern.CASE_INSENSITIVE);
Matcher m = multiline.multiline("Line 1\nline 2\nLINE 3");
while (m.find()) {
    System.out.println("Found: " + m.group());
}
```

Named Groups (Java 7+)

```
String text = "John Smith (john.smith@example.com)";
Pattern pattern = Pattern.compile("(?<name>[\\w\\s]+) \\((?<email>[\\w.@]+)\\)");
Matcher matcher = pattern.matcher(text);

if (matcher.matches()) {
    String name = matcher.group("name");   // "John Smith"
    String email = matcher.group("email"); // "john.smith@example.com"
    System.out.println("Name: " + name);
    System.out.println("Email: " + email);
}
```

Lookahead and Lookbehind

```
// Positive lookahead: match 'X' only if followed by 'Y'
Pattern pattern1 = Pattern.compile("\\w+(?=,)");
Matcher matcher1 = pattern1.matcher("apple, banana, orange");
while (matcher1.find()) {
    System.out.println(matcher1.group());  // "apple", "banana"
}

// Negative lookahead: match 'X' only if not followed by 'Y'
Pattern pattern2 = Pattern.compile("\\w+(?!,)");
Matcher matcher2 = pattern2.matcher("apple, banana, orange");
// Matches "apple" in "apple,", the "banana" in "banana,", and "orange"

// Positive lookbehind: match 'X' only if preceded by 'Y'
Pattern pattern3 = Pattern.compile("(?<=\\$)\\d+");
Matcher matcher3 = pattern3.matcher("Price: $50, Cost: $30");
while (matcher3.find()) {
    System.out.println(matcher3.group());  // "50", "30"
}

// Negative lookbehind: match 'X' only if not preceded by 'Y'
Pattern pattern4 = Pattern.compile("(?<!\\$)\\d+");
Matcher matcher4 = pattern4.matcher("Item 123: $50");
// Matches the "123" but not the "50"
```

Performance Considerations

1. **Compile patterns once, reuse them**: Compiling a regex is expensive.

```
// Inefficient
for (String input : inputs) {
    if (input.matches("complex-regex")) {
        // Process
    }
}

// Efficient
Pattern pattern = Pattern.compile("complex-regex");
for (String input : inputs) {
    if (pattern.matcher(input).matches()) {
        // Process
    }
}
```

2. **Use non-capturing groups** when you don't need the captured text:

```
// Capturing group
Pattern slow = Pattern.compile("(pattern)");

// Non-capturing group
Pattern faster = Pattern.compile("(?:pattern)");
```

3. **Avoid backtracking when possible** by using possessive quantifiers or atomic groups:

```
// Standard greedy quantifier (can backtrack)
Pattern p1 = Pattern.compile("a+");

// Possessive quantifier (no backtracking)
Pattern p2 = Pattern.compile("a++");

// Atomic group (no backtracking)
Pattern p3 = Pattern.compile("(?>a+)");
```

> **Common Pitfall**: Regular expressions can be powerful but also complex and difficult to debug. Start with simple patterns and incrementally build to more complex ones. Use regex testing tools to validate your patterns.

> **Certification Note**: Understand the basic syntax of regular expressions and how to use the Pattern and Matcher classes effectively. Be familiar with common regex patterns and flags.

Date and Time API

Java 8 introduced a new Date and Time API (`java.time` package) to address the shortcomings of the legacy date APIs.

Core Classes

1. **LocalDate**: Date without time or timezone
2. **LocalTime**: Time without date or timezone
3. **LocalDateTime**: Date and time without timezone
4. **ZonedDateTime**: Date and time with timezone
5. **Instant**: A point in time (epoch-based timestamp)
6. **Duration**: Time-based amount (seconds, nanoseconds)
7. **Period**: Date-based amount (years, months, days)

Creating Date-Time Objects

```
// Current date and time
LocalDate today = LocalDate.now();
LocalTime now = LocalTime.now();
LocalDateTime dateTime = LocalDateTime.now();
ZonedDateTime zonedDateTime = ZonedDateTime.now();
Instant instant = Instant.now();

// Creating specific dates
LocalDate date = LocalDate.of(2023, Month.MAY, 15);
LocalDate parsedDate = LocalDate.parse("2023-05-15");

// Creating specific times
LocalTime time = LocalTime.of(13, 30, 45);
LocalTime parsedTime = LocalTime.parse("13:30:45");

// Combining date and time
LocalDateTime combined = LocalDateTime.of(date, time);
LocalDateTime fromValues = LocalDateTime.of(2023, Month.MAY, 15, 13, 30, 45);

// Working with time zones
ZoneId zoneId = ZoneId.of("America/New_York");
ZonedDateTime nyTime = ZonedDateTime.of(combined, zoneId);
ZonedDateTime londonTime = ZonedDateTime.now(ZoneId.of("Europe/London"));

// Available time zones
Set<String> availableZones = ZoneId.getAvailableZoneIds();
```

Date and Time Manipulations

```
// Immutable - all operations return new objects
LocalDate today = LocalDate.now();
LocalDate tomorrow = today.plusDays(1);
LocalDate lastMonth = today.minusMonths(1);
LocalDate nextYear = today.plusYears(1);
```

```
// Chaining operations
LocalDateTime future = LocalDateTime.now()
    .plusDays(30)
    .plusHours(12)
    .minusMinutes(15);

// Using with methods to create modified copies
LocalDate firstOfMonth = today.withDayOfMonth(1);
LocalTime hourLater = LocalTime.now().withHour(LocalTime.now().getHour() + 1);

// Using TemporalAdjusters for common calculations
import static java.time.temporal.TemporalAdjusters.*;

LocalDate firstDayOfMonth = today.with(firstDayOfMonth());
LocalDate lastDayOfMonth = today.with(lastDayOfMonth());
LocalDate nextMonday = today.with(next(DayOfWeek.MONDAY));
LocalDate previousSunday = today.with(previous(DayOfWeek.SUNDAY));
```

Durations and Periods

```
// Duration - time-based amount
Duration twoHours = Duration.ofHours(2);
Duration tenMinutes = Duration.ofMinutes(10);
Duration fromDays = Duration.ofDays(1);  // 24 hours
Duration fromSeconds = Duration.ofSeconds(3600);  // 1 hour
Duration betweenTimes = Duration.between(LocalTime.of(10, 0), LocalTime.of(12,
30));

// Period - date-based amount
Period twoMonths = Period.ofMonths(2);
Period threeYears = Period.ofYears(3);
Period complex = Period.of(1, 6, 15);  // 1 year, 6 months, 15 days
Period betweenDates = Period.between(LocalDate.of(2020, 1, 1), LocalDate.of(2023,
5, 15));

// Using durations and periods
LocalDateTime start = LocalDateTime.now();
LocalDateTime end = start.plus(twoHours).plus(complex);
```

Formatting and Parsing

```
// Predefined formatters
LocalDate date = LocalDate.now();
String basicIsoDate = date.format(DateTimeFormatter.BASIC_ISO_DATE);  // 20230515
String isoDate = date.format(DateTimeFormatter.ISO_DATE);             // 2023-05-
15

// Custom patterns
```

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/yyyy");
String formatted = date.format(formatter);  // 15/05/2023
LocalDate parsed = LocalDate.parse("15/05/2023", formatter);

// Locale-specific formatting
DateTimeFormatter frenchFormatter = DateTimeFormatter
    .ofPattern("d MMMM yyyy")
    .withLocale(Locale.FRENCH);
String frenchDate = date.format(frenchFormatter);  // 15 mai 2023

// Builder pattern for complex formatters
DateTimeFormatter complexFormatter = new DateTimeFormatterBuilder()
    .appendLiteral("On ")
    .appendText(ChronoField.DAY_OF_WEEK, TextStyle.FULL)
    .appendLiteral(", ")
    .appendText(ChronoField.MONTH_OF_YEAR, TextStyle.FULL)
    .appendLiteral(" ")
    .appendValue(ChronoField.DAY_OF_MONTH)
    .appendLiteral(", ")
    .appendValue(ChronoField.YEAR)
    .toFormatter();
String prettyDate = date.format(complexFormatter);  // On Monday, May 15, 2023
```

Comparisons and Calculations

```
LocalDate date1 = LocalDate.of(2023, 5, 15);
LocalDate date2 = LocalDate.of(2023, 8, 20);

// Comparing dates
boolean isBefore = date1.isBefore(date2);  // true
boolean isAfter = date1.isAfter(date2);    // false
boolean isEqual = date1.isEqual(date2);    // false

// Calculating differences
Period period = Period.between(date1, date2);
int years = period.getYears();     // 0
int months = period.getMonths();   // 3
int days = period.getDays();       // 5

// Total days between
long daysBetween = ChronoUnit.DAYS.between(date1, date2);  // 97

// Other temporal units
long weeksBetween = ChronoUnit.WEEKS.between(date1, date2);  // 13
long monthsBetween = ChronoUnit.MONTHS.between(date1, date2);  // 3
```

Working with Legacy Date-Time API

```
// Convert from legacy to new API
Date legacyDate = new Date();
Instant instant = legacyDate.toInstant();
LocalDateTime dateTime = LocalDateTime.ofInstant(instant, ZoneId.systemDefault());
LocalDate date = dateTime.toLocalDate();

// Convert from new API to legacy
ZonedDateTime zdt = ZonedDateTime.now();
Date legacyFromZdt = Date.from(zdt.toInstant());

// Calendar to new API
Calendar calendar = Calendar.getInstance();
Instant calendarInstant = calendar.toInstant();
ZonedDateTime zdt2 = ZonedDateTime.ofInstant(calendarInstant,
calendar.getTimeZone().toZoneId());
```

Clock and Testing

```
// Using Clock for testable code
Clock clock = Clock.fixed(Instant.parse("2023-05-15T10:15:30Z"),
ZoneId.of("UTC"));
LocalDate date = LocalDate.now(clock);  // Always 2023-05-15
LocalTime time = LocalTime.now(clock);  // Always 10:15:30

// Creating a custom clock
Clock offset = Clock.offset(Clock.systemUTC(), Duration.ofHours(2));
LocalDateTime future = LocalDateTime.now(offset);  // 2 hours ahead
```

Advanced Date-Time Operations

```
// Working with different calendars
JapaneseDate japaneseDate = JapaneseDate.from(LocalDate.now());
HijrahDate hijrahDate = HijrahDate.from(LocalDate.now());
ThaiBuddhistDate buddhistDate = ThaiBuddhistDate.from(LocalDate.now());

// Finding first/last day of month
YearMonth yearMonth = YearMonth.of(2023, 5);
LocalDate firstDay = yearMonth.atDay(1);  // 2023-05-01
LocalDate lastDay = yearMonth.atEndOfMonth();  // 2023-05-31

// Working with specific years
Year year = Year.of(2023);
boolean isLeap = year.isLeap();  // false

// Working with specific months
Month month = Month.MAY;
int daysInMonth = month.length(false);  // 31 (not leap year)
```

> **Common Pitfall**: Legacy date/time classes like `java.util.Date`, `java.util.Calendar`, and `java.text.SimpleDateFormat` have design issues and should be avoided in new code in favor of the `java.time` API.

> **Certification Note**: Understand the core date-time classes and how they differ. Know how to create, manipulate, and format date-time objects using the new API. Be familiar with conversions between different types and calculations involving dates and times.

Annotations

Annotations provide metadata about code that can be processed at compile time or runtime. They are a form of declarative programming that can simplify boilerplate code and enable frameworks to process classes in specific ways.

Built-in Annotations

```
// For code that inherits methods
@Override
public String toString() {
    return "Custom string representation";
}

// To suppress compile-time warnings
@SuppressWarnings("unchecked")
public void suppressedWarning() {
    List list = new ArrayList();
    list.add("Item");
}

// To mark a method as deprecated
@Deprecated
public void oldMethod() {
    // Old implementation
}

// To document why something is deprecated (Java 9+)
@Deprecated(since = "2.0", forRemoval = true)
public void veryOldMethod() {
    // Will be removed in a future version
}

// To indicate an element should be processed by annotation tools
@Documented
@Retention(RetentionPolicy.RUNTIME)
@interface CustomAnnotation {
    // Annotation elements
}

// To specify that an annotation should be inherited by subclasses
@Inherited
@interface InheritableAnnotation {
```

```
    // Annotation elements
}

// To indicate that fields should be considered as part of the public API (Java
9+)
@Deprecated
@API(status = API.Status.DEPRECATED)  // Hypothetical API annotation
public void deprecatedPublicMethod() {
    // Old implementation
}

// To mark a type as a functional interface (Java 8+)
@FunctionalInterface
public interface MyFunction {
    void apply(String input);
    // Only one abstract method allowed in a functional interface
}

// To suppress resource warnings (Java 9+)
@SuppressWarnings("resource")
public void resourceMethod() {
    FileInputStream fis = new FileInputStream("file.txt");
    // Missing close(), but warning suppressed
}
```

Creating Custom Annotations

```
// Simple annotation with no elements
public @interface Marker {
}

// Annotation with elements
public @interface Author {
    String name();
    String date();
    int revision() default 1;
    String[] comments() default {};
}

// Usage
@Author(
    name = "John Doe",
    date = "2023-05-15",
    comments = {"Initial version", "Reviewed"}
)
public class AnnotatedClass {
    // Class implementation
}

// Annotation with a single element named "value"
public @interface Version {
```

```
    String value();
}

// Usage (simplified syntax for single "value" element)
@Version("1.0")
public class SimplifiedAnnotation {
    // Class implementation
}
```

Retention Policies

```
// SOURCE: Discarded during compilation
@Retention(RetentionPolicy.SOURCE)
public @interface CompilerInfo {
    String value();
}

// CLASS: Stored in class file but not available at runtime (default)
@Retention(RetentionPolicy.CLASS)
public @interface ClassInfo {
    String value();
}

// RUNTIME: Available at runtime through reflection
@Retention(RetentionPolicy.RUNTIME)
public @interface RuntimeInfo {
    String value();
}
```

Target Restrictions

```
// Specify where the annotation can be used
@Target(ElementType.METHOD)
public @interface MethodAnnotation {
    // Can only be used on methods
}

// Multiple targets
@Target({ElementType.FIELD, ElementType.METHOD})
public @interface FieldOrMethodAnnotation {
    // Can be used on fields or methods
}

// Available ElementType values (Java 8+)
@Target({
    ElementType.TYPE,             // Class, interface, enum
    ElementType.FIELD,            // Field
    ElementType.METHOD,           // Method
    ElementType.PARAMETER,        // Method parameter
```

```
    ElementType.CONSTRUCTOR,      // Constructor
    ElementType.LOCAL_VARIABLE,   // Local variable
    ElementType.ANNOTATION_TYPE,  // Annotation type
    ElementType.PACKAGE,          // Package
    ElementType.TYPE_PARAMETER,   // Generic type parameter
    ElementType.TYPE_USE          // Any type use
})
public @interface VersatileAnnotation {
    // Can be used almost anywhere
}
```

Repeatable Annotations (Java 8+)

```
// Container annotation
@Retention(RetentionPolicy.RUNTIME)
public @interface Schedules {
    Schedule[] value();
}

// Repeatable annotation
@Retention(RetentionPolicy.RUNTIME)
@Repeatable(Schedules.class)
public @interface Schedule {
    String dayOfMonth() default "first";
    String dayOfWeek() default "Mon";
    String hour() default "12:00";
}

// Usage
@Schedule(dayOfMonth = "last")
@Schedule(dayOfWeek = "Fri", hour = "15:00")
public class RepeatedAnnotationExample {
    // Class implementation
}
```

Processing Annotations at Runtime

```
public class AnnotationProcessor {
    public static void processClass(Class<?> clazz) {
        // Check if class has specific annotation
        if (clazz.isAnnotationPresent(Author.class)) {
            Author author = clazz.getAnnotation(Author.class);
            System.out.println("Author: " + author.name());
            System.out.println("Date: " + author.date());
            System.out.println("Revision: " + author.revision());

            System.out.println("Comments:");
            for (String comment : author.comments()) {
                System.out.println("- " + comment);
```

```
            }
        }

        // Process method annotations
        for (Method method : clazz.getDeclaredMethods()) {
            if (method.isAnnotationPresent(MethodAnnotation.class)) {
                MethodAnnotation annotation =
method.getAnnotation(MethodAnnotation.class);
                System.out.println("Method: " + method.getName() + " is
annotated");
            }
        }

        // Process repeatable annotations
        Schedule[] schedules = clazz.getAnnotationsByType(Schedule.class);
        for (Schedule schedule : schedules) {
            System.out.println("Schedule: " + schedule.dayOfWeek() + " at " +
schedule.hour());
        }
    }
}

// Usage
AnnotationProcessor.processClass(RepeatedAnnotationExample.class);
```

Annotation Processors for Compile-Time Processing

```
// Define processor
@SupportedAnnotationTypes("com.example.annotations.Processor")
@SupportedSourceVersion(SourceVersion.RELEASE_17)
public class CustomProcessor extends AbstractProcessor {
    @Override
    public boolean process(Set<? extends TypeElement> annotations,
RoundEnvironment roundEnv) {
        for (TypeElement annotation : annotations) {
            Set<? extends Element> elements =
roundEnv.getElementsAnnotatedWith(annotation);
            for (Element element : elements) {
                // Process each annotated element
                // Generate code, validate constraints, etc.
            }
        }
        return true;
    }
}
```

Type Annotations (Java 8+)

```
// Type annotation definition
@Target(ElementType.TYPE_USE)
@Retention(RetentionPolicy.RUNTIME)
public @interface NonNull {
}

// Usage on types
public class TypeAnnotationExample {
    // On a field type
    private @NonNull String name;

    // On a method return type
    public @NonNull List<String> getItems() {
        return new ArrayList<>();
    }

    // On a type parameter
    public <@NonNull T> T process(T input) {
        return input;
    }

    // On a constructor
    public TypeAnnotationExample(@NonNull String name) {
        this.name = name;
    }

    // On array dimensions
    public String @NonNull [] createArray() {
        return new String[10];
    }

    // On type casts
    public void castExample(Object obj) {
        String str = (@NonNull String) obj;
    }

    // On exception types
    public void exceptionExample() throws @NonNull IOException {
        // Method body
    }
}
```

Common Annotation Frameworks

1. **Jakarta Bean Validation**:

```
public class User {
    @NotNull
    @Size(min = 2, max = 30)
    private String username;
```

```
    @NotNull
    @Email
    private String email;

    @Min(18)
    private int age;

    // Getters and setters
}
```

2. **Lombok**:

```
@Data  // Generates getters, setters, equals, hashCode, toString
@AllArgsConstructor
@NoArgsConstructor
public class Product {
    private Long id;
    private String name;
    private double price;
}
```

3. **Spring Framework**:

```
@Controller
@RequestMapping("/users")
public class UserController {
    @Autowired
    private UserService userService;

    @GetMapping("/{id}")
    public ResponseEntity<User> getUser(@PathVariable Long id) {
        return ResponseEntity.ok(userService.findById(id));
    }

    @PostMapping
    public ResponseEntity<User> createUser(@RequestBody User user) {
        return ResponseEntity.ok(userService.save(user));
    }
}
```

Best Practices for Annotations:

- Keep annotations focused on a single concern
- Provide useful default values when appropriate
- Document annotations thoroughly, especially custom ones
- Consider retention policy carefully - use SOURCE for compile-time validation, RUNTIME only when needed
- Avoid overusing annotations that might make code harder to understand

> **Certification Note**: Understand the purpose of annotations and how to define custom annotations. Know the built-in annotations and their uses. Be familiar with retention policies and target restrictions. Understand how to process annotations at runtime using reflection.

Reflection and Dynamic Proxies

Reflection allows examination and modification of classes, interfaces, fields, methods, and constructors at runtime. Dynamic proxies enable creating implementations of interfaces at runtime.

Reflection Basics

```
// Get Class object
Class<?> stringClass = String.class;
Class<?> listClass = List.class;
Class<?> arrayClass = int[].class;

// From object instance
String str = "Hello";
Class<?> strClass = str.getClass();

// From class name
Class<?> dynamicClass = Class.forName("java.util.ArrayList");

// Class information
String className = dynamicClass.getName();
String simpleName = dynamicClass.getSimpleName();
int modifiers = dynamicClass.getModifiers();
boolean isPublic = Modifier.isPublic(modifiers);
boolean isInterface = dynamicClass.isInterface();

// Superclass and interfaces
Class<?> superclass = dynamicClass.getSuperclass();
Class<?>[] interfaces = dynamicClass.getInterfaces();

// Package information
Package pkg = dynamicClass.getPackage();
```

Working with Fields

```
class Person {
    public String name;
    private int age;
    protected boolean active;
    static final String SPECIES = "Human";

    // Constructors, methods, etc.
}

// Get fields
```

```
Class<Person> personClass = Person.class;

// Public fields only
Field[] publicFields = personClass.getFields();

// All fields (public, protected, private)
Field[] allFields = personClass.getDeclaredFields();

// Specific field
Field nameField = personClass.getField("name");
Field ageField = personClass.getDeclaredField("age");

// Get/set field values
Person person = new Person();
nameField.set(person, "John");
String name = (String) nameField.get(person);

// Accessing private fields
ageField.setAccessible(true);  // Bypass access control
ageField.set(person, 30);
int age = (int) ageField.get(person);

// Field information
String fieldName = ageField.getName();
Class<?> fieldType = ageField.getType();
int fieldModifiers = ageField.getModifiers();
boolean isPrivate = Modifier.isPrivate(fieldModifiers);
```

Working with Methods

```
// Get methods
Class<Person> personClass = Person.class;

// Public methods (including inherited)
Method[] publicMethods = personClass.getMethods();

// All declared methods (excluding inherited)
Method[] declaredMethods = personClass.getDeclaredMethods();

// Specific method
Method getAgeMethod = personClass.getDeclaredMethod("getAge");
Method setNameMethod = personClass.getDeclaredMethod("setName", String.class);

// Invoking methods
Person person = new Person();
setNameMethod.invoke(person, "John");
int age = (int) getAgeMethod.invoke(person);

// Accessing private methods
Method privateMethod = personClass.getDeclaredMethod("calculateBonus",
double.class);
```

```
privateMethod.setAccessible(true);
double bonus = (double) privateMethod.invoke(person, 1000.0);

// Method information
String methodName = setNameMethod.getName();
Class<?> returnType = setNameMethod.getReturnType();
Class<?>[] paramTypes = setNameMethod.getParameterTypes();
int methodModifiers = setNameMethod.getModifiers();
Annotation[] annotations = setNameMethod.getAnnotations();

// Parameter information (Java 8+)
Parameter[] parameters = setNameMethod.getParameters();
for (Parameter param : parameters) {
    String paramName = param.getName();  // Requires -parameters compiler flag
    Class<?> paramType = param.getType();
    boolean isVarArgs = param.isVarArgs();
}
```

Working with Constructors

```
// Get constructors
Class<Person> personClass = Person.class;

// Public constructors
Constructor<?>[] publicConstructors = personClass.getConstructors();

// All constructors
Constructor<?>[] allConstructors = personClass.getDeclaredConstructors();

// Specific constructor
Constructor<Person> defaultConstructor = personClass.getDeclaredConstructor();
Constructor<Person> paramConstructor =
personClass.getDeclaredConstructor(String.class, int.class);

// Creating instances
Person person1 = defaultConstructor.newInstance();
Person person2 = paramConstructor.newInstance("John", 30);

// Shorthand for default constructor
Person person3 = personClass.newInstance();  // Deprecated in Java 9+

// Accessing private constructors
Constructor<Person> privateConstructor =
personClass.getDeclaredConstructor(String.class);
privateConstructor.setAccessible(true);
Person person4 = privateConstructor.newInstance("Private");

// Constructor information
int constructorModifiers = paramConstructor.getModifiers();
Class<?>[] paramTypes = paramConstructor.getParameterTypes();
```

Working with Arrays

```
// Create arrays using reflection
Class<?> intArrayClass = int[].class;
int[] newArray = (int[]) Array.newInstance(int.class, 10);

// Set and get array elements
Array.set(newArray, 0, 42);
int value = (int) Array.get(newArray, 0);

// Multi-dimensional arrays
int[][] matrix = (int[][]) Array.newInstance(int.class, 3, 3);
```

Dynamic Proxies

Dynamic proxies allow creating implementations of interfaces at runtime:

```
// Interface to implement
interface UserService {
    User findById(Long id);
    List<User> findAll();
    User save(User user);
}

// Invocation handler
class LoggingInvocationHandler implements InvocationHandler {
    private final Object target;

    public LoggingInvocationHandler(Object target) {
        this.target = target;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        System.out.println("Before method: " + method.getName());

        // Invoke the actual method on the target object
        Object result = method.invoke(target, args);

        System.out.println("After method: " + method.getName());
        return result;
    }
}

// Creating and using a proxy
UserService realService = new UserServiceImpl();
UserService proxyService = (UserService) Proxy.newProxyInstance(
    UserService.class.getClassLoader(),
    new Class<?>[] { UserService.class },
```

```
        new LoggingInvocationHandler(realService)
    );

    // Using the proxy - will trigger the logging behavior
    User user = proxyService.findById(123L);
```

Common Use Cases for Reflection

1. **Testing Frameworks**: JUnit, TestNG use reflection to discover and run test methods.
2. **Dependency Injection**: Spring, Guice create and wire objects based on annotations.
3. **Object-Relational Mapping**: JPA, Hibernate map Java objects to database tables.
4. **Serialization/Deserialization**: JSON libraries like Jackson, Gson convert objects to/from JSON.
5. **Aspect-Oriented Programming**: Spring AOP uses dynamic proxies for aspect weaving.

Reflection Performance and Security Considerations

```
// Performance considerations
// 1. Cache Class objects and Method objects when possible
private static final Method METHOD_CACHE =
SomeClass.class.getDeclaredMethod("methodName");

// 2. Prefer direct calls when performance is critical
// Slow (reflection)
method.invoke(object, args);

// Fast (direct)
object.method(args);

// Security considerations
// 1. SecurityManager restrictions
SecurityManager securityManager = System.getSecurityManager();
if (securityManager != null) {
    securityManager.checkPermission(new
ReflectPermission("suppressAccessChecks"));
}

// 2. Limit accessibility changes
privateField.setAccessible(true);  // Potentially dangerous

// 3. Validate input when using reflection with user input
Method method = clazz.getMethod(userProvidedMethodName);  // Potential
vulnerability
```

Annotations and Reflection

```
// Custom annotation
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface Auditable {
    String value() default "";
}

// Class with annotated methods
class Service {
    @Auditable("critical")
    public void criticalOperation() {
        // Implementation
    }

    @Auditable
    public void regularOperation() {
        // Implementation
    }
}

// Processing annotations with reflection
Class<Service> serviceClass = Service.class;
for (Method method : serviceClass.getDeclaredMethods()) {
    if (method.isAnnotationPresent(Auditable.class)) {
        Auditable annotation = method.getAnnotation(Auditable.class);
        String value = annotation.value();
        System.out.println("Method " + method.getName() +
                           " is auditable with level: " +
                           (value.isEmpty() ? "default" : value));
    }
}
```

Module System Reflection (Java 9+)

```
// Working with modules
Class<?> clazz = String.class;
Module module = clazz.getModule();
String moduleName = module.getName();  // "java.base"

// Module descriptors
ModuleDescriptor descriptor = module.getDescriptor();
Set<String> exports = descriptor.exports().stream()
    .map(ModuleDescriptor.Exports::source)
    .collect(Collectors.toSet());

// Module requirements
Set<String> requires = descriptor.requires().stream()
    .map(ModuleDescriptor.Requires::name)
    .collect(Collectors.toSet());
```

Record Reflection (Java 16+)

```
// Working with records
record Person(String name, int age) {}

Class<Person> personClass = Person.class;
boolean isRecord = personClass.isRecord();  // true

// Get record components
RecordComponent[] components = personClass.getRecordComponents();
for (RecordComponent component : components) {
    String name = component.getName();
    Class<?> type = component.getType();
    Method accessor = component.getAccessor();
}
```

> **Common Pitfall**: Reflection is powerful but can break encapsulation, reduce performance, and bypass compiler checks. Use it judiciously and prefer non-reflective approaches when possible.

> **Certification Note**: Understand how to use reflection to inspect and manipulate classes, fields, methods, and constructors. Know how to create and use dynamic proxies. Be aware of the performance and security implications of reflection.

Java 8-21 Modern Features

Lambda Expressions and Functional Interfaces

Lambda expressions provide a concise way to represent anonymous functions. They work with functional interfaces, which are interfaces with a single abstract method.

Functional Interfaces

A functional interface has exactly one abstract method:

```
// Basic functional interface
@FunctionalInterface
interface Calculator {
    int calculate(int a, int b);
}

// Built-in functional interfaces
Runnable runnable;                // void run()
Callable<T> callable;             // T call()
Supplier<T> supplier;             // T get()
Consumer<T> consumer;             // void accept(T t)
BiConsumer<T, U> biConsumer;      // void accept(T t, U u)
Function<T, R> function;          // R apply(T t)
BiFunction<T, U, R> biFunction;   // R apply(T t, U u)
```

```
Predicate<T> predicate;           // boolean test(T t)
BiPredicate<T, U> biPredicate;    // boolean test(T t, U u)
UnaryOperator<T> unaryOperator;   // T apply(T t)
BinaryOperator<T> binaryOperator; // T apply(T t, T u)
```

Lambda Syntax

```
// Basic lambda syntax: (parameters) -> expression or statement block

// No parameters
Runnable r1 = () -> System.out.println("Hello");

// One parameter (parentheses optional)
Consumer<String> c1 = s -> System.out.println(s);
Consumer<String> c2 = (String s) -> System.out.println(s);

// Multiple parameters
BiConsumer<String, Integer> bc = (name, age) -> System.out.println(name + " is " +
age);

// With explicit type
BiFunction<Integer, Integer, Integer> bf = (Integer a, Integer b) -> a + b;

// Multiple statements in a block
Comparator<String> comp = (s1, s2) -> {
    int result = s1.length() - s2.length();
    if (result == 0) {
        result = s1.compareTo(s2);
    }
    return result;
};
```

Using Lambda Expressions

```
// With custom functional interface
Calculator adder = (a, b) -> a + b;
Calculator multiplier = (a, b) -> a * b;

int sum = adder.calculate(5, 3);        // 8
int product = multiplier.calculate(5, 3); // 15

// With built-in functional interfaces
Predicate<String> isEmpty = s -> s.isEmpty();
Predicate<String> isNotEmpty = isEmpty.negate();

Function<String, Integer> stringToLength = s -> s.length();
Function<Integer, String> intToString = i -> Integer.toString(i);
Function<String, String> combined = stringToLength.andThen(intToString);
```

```
// In collections
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
names.forEach(name -> System.out.println("Hello, " + name));

// With streams
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> doubled = numbers.stream()
                               .map(n -> n * 2)
                               .collect(Collectors.toList());
```

Method References

Method references provide a shorthand notation for lambda expressions that call a single method:

```
// Types of method references:
// 1. Static method reference: ClassName::staticMethod
// 2. Instance method of specific object: instance::method
// 3. Instance method of arbitrary object of a type: ClassName::instanceMethod
// 4. Constructor reference: ClassName::new

// Examples:
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

// Lambda
names.forEach(name -> System.out.println(name));

// Method reference
names.forEach(System.out::println);

// Static method reference
Function<String, Integer> parser = Integer::parseInt;

// Instance method reference
String prefix = "User: ";
Function<String, String> addPrefix = prefix::concat;

// Instance method of arbitrary object
Comparator<String> comparator = String::compareToIgnoreCase;

// Constructor reference
Supplier<List<String>> listSupplier = ArrayList::new;
Function<String, Integer> newInteger = Integer::new;
```

Capturing Variables

Lambda expressions can capture variables from their enclosing scope:

```
// Capturing local variables (must be effectively final)
String prefix = "User: ";
```

```
Consumer<String> printer = name -> System.out.println(prefix + name);
// prefix = "Customer: ";  // Error: Cannot modify captured variable

// Instance variables can be modified
class LambdaExample {
    private String prefix = "User: ";

    public void example() {
        Consumer<String> printer = name -> {
            System.out.println(prefix + name);
            prefix = "Modified: ";  // OK, can modify instance variables
        };

        printer.accept("Alice");  // "User: Alice"
        System.out.println(prefix);  // "Modified: "
    }
}
```

Building Complex Functions

```
// Composing functions
Function<Integer, Integer> addOne = x -> x + 1;
Function<Integer, Integer> multiplyByTwo = x -> x * 2;

// f(g(x)) - First apply g, then f
Function<Integer, Integer> addThenMultiply = multiplyByTwo.compose(addOne);
int result1 = addThenMultiply.apply(3);  // (3 + 1) * 2 = 8

// g(f(x)) - First apply f, then g
Function<Integer, Integer> multiplyThenAdd = multiplyByTwo.andThen(addOne);
int result2 = multiplyThenAdd.apply(3);  // (3 * 2) + 1 = 7

// Predicates can be combined
Predicate<String> startsWithA = s -> s.startsWith("A");
Predicate<String> endsWithX = s -> s.endsWith("x");
Predicate<String> startsWithAAndEndsWithX = startsWithA.and(endsWithX);
Predicate<String> startsWithAOrEndsWithX = startsWithA.or(endsWithX);
Predicate<String> doesNotStartWithA = startsWithA.negate();
```

Custom Functional Interfaces

```
// Trifunction (three parameters)
@FunctionalInterface
interface TriFunction<A, B, C, R> {
    R apply(A a, B b, C c);

    // Can contain default and static methods
    default <V> TriFunction<A, B, C, V> andThen(Function<? super R, ? extends V>
after) {
```

```
        Objects.requireNonNull(after);
        return (a, b, c) -> after.apply(apply(a, b, c));
    }
}

// Usage
TriFunction<Integer, Integer, Integer, Integer> sum3 = (a, b, c) -> a + b + c;
int total = sum3.apply(1, 2, 3);  // 6

// With andThen
Function<Integer, String> formatter = n -> "Result: " + n;
TriFunction<Integer, Integer, Integer, String> sumAndFormat =
sum3.andThen(formatter);
String result = sumAndFormat.apply(1, 2, 3);  // "Result: 6"
```

Exception Handling in Lambdas

```
// Without exception handling
Function<String, Integer> unsafeParser = s -> Integer.parseInt(s);  // Can throw
NumberFormatException

// With exception handling
Function<String, Integer> safeParser = s -> {
    try {
        return Integer.parseInt(s);
    } catch (NumberFormatException e) {
        return 0;  // Default value
    }
};

// Wrapping checked exceptions
@FunctionalInterface
interface ThrowingFunction<T, R, E extends Exception> {
    R apply(T t) throws E;

    static <T, R, E extends Exception> Function<T, R> unchecked(
            ThrowingFunction<T, R, E> f) {
        return t -> {
            try {
                return f.apply(t);
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
        };
    }
}

// Usage with checked exceptions
ThrowingFunction<String, String, IOException> reader = s -> {
    try (BufferedReader br = new BufferedReader(new FileReader(s))) {
        return br.readLine();
```

```
    }
};

// Convert to regular function
Function<String, String> safeReader = ThrowingFunction.unchecked(reader);
```

> **Common Pitfall**: Lambda expressions can make debugging more difficult because they don't have their own names. Consider using named methods for complex logic.

> **Certification Note**: Understand the syntax of lambda expressions and how they relate to functional interfaces. Know the built-in functional interfaces in the `java.util.function` package and when to use each one. Be familiar with method references and variable capture rules.

Stream API and Functional Programming

The Stream API allows processing collections of objects in a functional style. Streams represent a sequence of elements supporting sequential and parallel aggregate operations.

Creating Streams

```
// From collections
List<String> list = Arrays.asList("apple", "banana", "cherry");
Stream<String> streamFromList = list.stream();
Stream<String> parallelStream = list.parallelStream();

// From arrays
String[] array = {"apple", "banana", "cherry"};
Stream<String> streamFromArray = Arrays.stream(array);
Stream<Integer> streamFromIntArray = Arrays.stream(new int[]{1, 2, 3}).boxed();

// From individual values
Stream<String> streamOfValues = Stream.of("apple", "banana", "cherry");

// Empty stream
Stream<String> emptyStream = Stream.empty();

// Infinite streams
Stream<Integer> infiniteIntegers = Stream.iterate(1, n -> n + 1);
Stream<Double> infiniteRandoms = Stream.generate(Math::random);

// From files
try (Stream<String> lines = Files.lines(Paths.get("file.txt"))) {
    // Process lines
}

// From regex pattern matches
Pattern pattern = Pattern.compile(",");
Stream<String> patternStream = pattern.splitAsStream("apple,banana,cherry");
```

Stream Operations

Stream operations are divided into intermediate operations and terminal operations:

Intermediate Operations (return another stream)

```
List<String> fruits = Arrays.asList("apple", "banana", "cherry", "date",
"elderberry");

// filter - keeps elements that match the predicate
Stream<String> filtered = fruits.stream()
    .filter(s -> s.startsWith("a"));  // "apple"

// map - transforms elements
Stream<Integer> lengths = fruits.stream()
    .map(String::length);  // 5, 6, 6, 4, 11

// flatMap - transforms and flattens
List<List<Integer>> nestedLists = Arrays.asList(
    Arrays.asList(1, 2, 3),
    Arrays.asList(4, 5, 6)
);
Stream<Integer> flattened = nestedLists.stream()
    .flatMap(Collection::stream);  // 1, 2, 3, 4, 5, 6

// distinct - removes duplicates
Stream<String> distinct = Stream.of("apple", "banana", "apple", "cherry")
    .distinct();  // "apple", "banana", "cherry"

// sorted - sorts elements
Stream<String> sorted = fruits.stream()
    .sorted();  // "apple", "banana", "cherry", "date", "elderberry"
Stream<String> customSorted = fruits.stream()
    .sorted(Comparator.comparing(String::length));  // "date", "apple", "banana",
"cherry", "elderberry"

// peek - performs an action on each element without modifying the stream
Stream<String> peeked = fruits.stream()
    .peek(s -> System.out.println("Processing: " + s));

// limit - truncates the stream
Stream<String> limited = Stream.iterate(1, n -> n + 1)
    .map(String::valueOf)
    .limit(5);  // "1", "2", "3", "4", "5"

// skip - discards the first n elements
Stream<String> skipped = fruits.stream()
    .skip(2);  // "cherry", "date", "elderberry"

// takeWhile (Java 9+) - takes elements while predicate is true
Stream<String> taken = fruits.stream()
    .takeWhile(s -> s.length() < 7);  // "apple", "banana", "cherry", "date"
```

```
// dropWhile (Java 9+) - drops elements while predicate is true
Stream<String> dropped = fruits.stream()
    .dropWhile(s -> s.length() < 7);  // "elderberry"
```

Terminal Operations (produce a result or side-effect)

```
// forEach - performs an action for each element
fruits.stream()
    .forEach(System.out::println);

// forEachOrdered - like forEach but respects encounter order
fruits.parallelStream()
    .forEachOrdered(System.out::println);

// toArray - converts to array
String[] array = fruits.stream()
    .toArray(String[]::new);

// reduce - reduces to a single value
Optional<String> concatenated = fruits.stream()
    .reduce((a, b) -> a + ", " + b);  // "apple, banana, cherry, date, elderberry"

int sum = Stream.of(1, 2, 3, 4, 5)
    .reduce(0, Integer::sum);  // 15

// collect - collects to a collection or other result
List<String> collectedList = fruits.stream()
    .collect(Collectors.toList());

Set<String> collectedSet = fruits.stream()
    .collect(Collectors.toSet());

String joined = fruits.stream()
    .collect(Collectors.joining(", "));  // "apple, banana, cherry, date,
elderberry"

Map<Integer, List<String>> groupedByLength = fruits.stream()
    .collect(Collectors.groupingBy(String::length));

// min, max, count - find minimum, maximum, count elements
Optional<String> shortest = fruits.stream()
    .min(Comparator.comparing(String::length));  // "date"

Optional<String> longest = fruits.stream()
    .max(Comparator.comparing(String::length));  // "elderberry"

long count = fruits.stream()
    .count();  // 5

// anyMatch, allMatch, noneMatch - check conditions
```

```
boolean anyStartWithA = fruits.stream()
    .anyMatch(s -> s.startsWith("a"));  // true

boolean allLongerThan2 = fruits.stream()
    .allMatch(s -> s.length() > 2);  // true

boolean noneStartWithZ = fruits.stream()
    .noneMatch(s -> s.startsWith("z"));  // true

// findFirst, findAny - find elements
Optional<String> first = fruits.stream()
    .findFirst();  // "apple"

Optional<String> any = fruits.parallelStream()
    .findAny();  // Any element (non-deterministic in parallel)
```

Advanced Collectors

```
List<Person> people = Arrays.asList(
    new Person("Alice", 25),
    new Person("Bob", 30),
    new Person("Charlie", 25),
    new Person("Dave", 30)
);

// Grouping by a property
Map<Integer, List<Person>> byAge = people.stream()
    .collect(Collectors.groupingBy(Person::getAge));

// Grouping and counting
Map<Integer, Long> countByAge = people.stream()
    .collect(Collectors.groupingBy(Person::getAge, Collectors.counting()));

// Grouping and mapping
Map<Integer, List<String>> namesByAge = people.stream()
    .collect(Collectors.groupingBy(
        Person::getAge,
        Collectors.mapping(Person::getName, Collectors.toList())
    ));

// Partitioning (special case of grouping with boolean predicate)
Map<Boolean, List<Person>> partitioned = people.stream()
    .collect(Collectors.partitioningBy(p -> p.getAge() > 25));

// Summarizing statistics
IntSummaryStatistics stats = people.stream()
    .collect(Collectors.summarizingInt(Person::getAge));
// stats.getAverage(), stats.getSum(), stats.getMin(), stats.getMax(),
stats.getCount()

// Joining strings
```

```
String names = people.stream()
    .map(Person::getName)
    .collect(Collectors.joining(", "));  // "Alice, Bob, Charlie, Dave"

// Custom collector
Collector<Person, ?, Double> averageAgeCollector = Collector.of(
    () -> new double[2],  // Accumulator: [sum, count]
    (acc, p) -> {
        acc[0] += p.getAge();  // Add to sum
        acc[1]++;              // Increment count
    },
    (acc1, acc2) -> {  // Combiner for parallel streams
        acc1[0] += acc2[0];
        acc1[1] += acc2[1];
        return acc1;
    },
    acc -> acc[0] / acc[1]  // Finisher: calculate average
);

double averageAge = people.stream().collect(averageAgeCollector);  // 27.5
```

Parallel Streams

```
// Converting to parallel stream
Stream<String> parallelStream = fruits.parallelStream();
Stream<String> alsoParallel = fruits.stream().parallel();

// Example: parallel processing
long sum = Stream.iterate(1L, i -> i + 1)
    .limit(1_000_000)
    .parallel()
    .reduce(0L, Long::sum);

// Performance consideration: avoid stateful lambdas in parallel streams
List<Integer> numbers = new ArrayList<>();
// BAD: stateful lambda in parallel stream
IntStream.range(0, 10000)
    .parallel()
    .forEach(numbers::add);  // May result in incorrect results or exceptions

// GOOD: collect results instead
List<Integer> betterNumbers = IntStream.range(0, 10000)
    .parallel()
    .boxed()
    .collect(Collectors.toList());
```

Special Stream Types for Primitives

```
// IntStream, LongStream, DoubleStream
IntStream intStream = IntStream.range(1, 6);  // 1, 2, 3, 4, 5
LongStream longStream = LongStream.rangeClosed(1, 5);  // 1, 2, 3, 4, 5
DoubleStream doubleStream = DoubleStream.of(1.1, 2.2, 3.3);

// Converting between object streams and primitive streams
Stream<Integer> boxed = IntStream.range(1, 6).boxed();
IntStream unboxed = Stream.of(1, 2, 3, 4, 5).mapToInt(Integer::intValue);

// Special operations on numeric streams
int sum = IntStream.range(1, 101).sum();  // 5050
OptionalDouble avg = IntStream.range(1, 101).average();  // 50.5
IntSummaryStatistics stats = IntStream.range(1, 101).summaryStatistics();
```

Infinite Streams and Short-circuiting Operations

```
// Generate Fibonacci sequence
Stream<BigInteger> fibonacci = Stream.iterate(
    new BigInteger[] { BigInteger.ZERO, BigInteger.ONE },
    f -> new BigInteger[] { f[1], f[0].add(f[1]) }
).map(f -> f[0]);

// First 10 Fibonacci numbers
List<BigInteger> first10 = fibonacci.limit(10).collect(Collectors.toList());
// [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

// Generate primes
Stream<Integer> primes = Stream.iterate(2, n -> n + 1)
    .filter(n -> IntStream.rangeClosed(2, (int) Math.sqrt(n))
        .noneMatch(divisor -> n % divisor == 0));

// First 10 prime numbers
List<Integer> first10Primes = primes.limit(10).collect(Collectors.toList());
// [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

Real-world Stream Examples

```
// Example 1: Processing a file
try (Stream<String> lines = Files.lines(Paths.get("data.txt"))) {
    Map<String, Integer> wordFrequency = lines
        .flatMap(line -> Arrays.stream(line.toLowerCase().split("\\W+")))
        .filter(word -> !word.isEmpty())
        .collect(Collectors.groupingBy(
            Function.identity(),
            Collectors.summingInt(w -> 1)
        ));
}
```

```
// Example 2: Processing a collection of objects
class Order {
    private List<OrderItem> items;
    private Customer customer;
    // getters, setters...
}

class OrderItem {
    private Product product;
    private int quantity;
    // getters, setters...
}

class Product {
    private String name;
    private BigDecimal price;
    // getters, setters...
}

// Calculate total revenue by product
Map<String, BigDecimal> revenueByProduct = orders.stream()
    .flatMap(order -> order.getItems().stream())
    .collect(Collectors.groupingBy(
        item -> item.getProduct().getName(),
        Collectors.reducing(
            BigDecimal.ZERO,
            item -> item.getProduct().getPrice()
                .multiply(new BigDecimal(item.getQuantity())),
            BigDecimal::add
        )
    ));

// Find top 5 products by revenue
List<Map.Entry<String, BigDecimal>> top5Products =
revenueByProduct.entrySet().stream()
    .sorted(Map.Entry.<String, BigDecimal>comparingByValue().reversed())
    .limit(5)
    .collect(Collectors.toList());
```

Stream Pipeline Execution

Streams use lazy evaluation. Intermediate operations are not executed until a terminal operation is called:

```
// This demonstrates how streams are lazily evaluated
Stream<String> stream = Stream.of("apple", "banana", "cherry", "date")
    .filter(s -> {
        System.out.println("Filtering: " + s);
        return s.startsWith("a") || s.startsWith("c");
    })
    .map(s -> {
        System.out.println("Mapping: " + s);
```

```
            return s.toUpperCase();
        });

    // No output yet, because no terminal operation has been called

    // Now add a terminal operation
    List<String> result = stream.collect(Collectors.toList());

    // Output:
    // Filtering: apple
    // Mapping: apple
    // Filtering: banana
    // Filtering: cherry
    // Mapping: cherry
    // Filtering: date
```

> **Common Pitfall**: Streams can only be traversed once. Attempting to reuse a stream after a terminal operation will result in an `IllegalStateException`.

> **Certification Note**: Understand the distinction between intermediate and terminal operations. Know the behavior of common stream operations and which operations are short-circuiting. Be familiar with the collectors in the `Collectors` class and how to create custom collectors.

Optional Class

The `Optional` class introduced in Java 8 provides a container object that may or may not contain a value. It helps avoid null pointer exceptions and encourages more robust code.

Creating Optional Objects

```
// Empty Optional
Optional<String> empty = Optional.empty();

// Optional with a value
Optional<String> present = Optional.of("Hello");

// Optional that might be null
String nullableValue = getValueThatMightBeNull();
Optional<String> nullable = Optional.ofNullable(nullableValue);
```

Checking for a Value

```
// isPresent returns true if value exists
if (optional.isPresent()) {
    // Use the value
    String value = optional.get();
    System.out.println(value);
}
```

```
// isEmpty returns true if no value exists (Java 11+)
if (optional.isEmpty()) {
    System.out.println("No value present");
}
```

Getting Values from Optional

```
// get() - returns value if present, throws NoSuchElementException if empty
String value = optional.get();  // Only use when you're sure a value exists

// orElse - returns value if present, otherwise returns default
String result = optional.orElse("Default");

// orElseGet - returns value if present, otherwise invokes supplier
String computed = optional.orElseGet(() -> computeDefault());

// orElseThrow - returns value if present, otherwise throws exception
String value = optional.orElseThrow();  // Java 10+ - throws
NoSuchElementException
String value = optional.orElseThrow(() -> new CustomException("No value"));
```

Transforming Optional Values

```
// map - transforms value if present
Optional<Integer> length = optional.map(String::length);

// flatMap - transforms to another Optional
Optional<String> transformed = optional.flatMap(this::transformToOptional);

// filter - keeps value only if it satisfies predicate
Optional<String> filtered = optional.filter(s -> s.length() > 5);
```

Consuming Optional Values

```
// ifPresent - executes action if value exists
optional.ifPresent(value -> System.out.println("The value is: " + value));

// ifPresentOrElse - executes one of two actions (Java 9+)
optional.ifPresentOrElse(
    value -> System.out.println("The value is: " + value),
    () -> System.out.println("No value present")
);
```

Combining Optional Operations

```
// Chaining operations
Optional<String> name = Optional.ofNullable(getUser())
    .map(User::getAddress)
    .map(Address::getCountry)
    .map(Country::getName);

// Using or to provide an alternative (Java 9+)
Optional<String> result = optional1.or(() -> optional2);

// Transforming to stream (Java 9+)
Stream<String> stream = optional.stream();  // Empty stream if Optional is empty
```

Optional as Return Value

```
// Good use of Optional as return value
public Optional<User> findUserById(String id) {
    User user = userRepository.findById(id);
    return Optional.ofNullable(user);
}

// Using the returned Optional
Optional<User> userOpt = findUserById("123");
userOpt.ifPresent(user -> sendEmail(user));
```

Proper Usage Guidelines

```
// DO: Use Optional as return type for methods that might not return a value
public Optional<User> findUserById(String id) { ... }

// DON'T: Use Optional as parameter type
public void processUser(Optional<User> userOpt) { ... }  // Avoid this
public void processUser(User user) { ... }  // Better (check for null inside if
needed)

// DON'T: Create Optional just to unwrap it immediately
Optional<User> user = Optional.ofNullable(getUser());
if (user.isPresent()) {  // Anti-pattern
    processUser(user.get());
}
// Better:
User user = getUser();
if (user != null) {
    processUser(user);
}

// DON'T: Use Optional for collections
// Instead of Optional<List<String>>
return list != null ? list : Collections.emptyList();
```

```
// DON'T: Store Optional as a field
private Optional<String> name;  // Avoid this (Optional is not Serializable)
```

Real-world Examples with Optional

```
// Example 1: Configuration settings with default values
public class ConfigService {
    private Map<String, String> settings = new HashMap<>();

    public Optional<String> getSetting(String key) {
        return Optional.ofNullable(settings.get(key));
    }

    public String getSettingWithDefault(String key, String defaultValue) {
        return getSetting(key).orElse(defaultValue);
    }

    public int getIntSetting(String key, int defaultValue) {
        return getSetting(key)
            .map(Integer::parseInt)
            .filter(n -> n > 0)
            .orElse(defaultValue);
    }
}

// Example 2: Handling database queries
public class UserRepository {
    public Optional<User> findById(long id) {
        User user = queryDatabase(id);
        return Optional.ofNullable(user);
    }

    public List<User> findByLastName(String lastName) {
        return findById(123)
            .map(User::getFriends)
            .orElse(Collections.emptyList())
            .stream()
            .filter(friend -> lastName.equals(friend.getLastName()))
            .collect(Collectors.toList());
    }
}

// Example 3: Data processing pipeline
public class OrderProcessor {
    public Optional<OrderConfirmation> processOrder(OrderRequest request) {
        return validateOrder(request)
            .flatMap(this::checkInventory)
            .flatMap(this::processPayment)
            .map(this::generateConfirmation);
    }
```

```
    private Optional<OrderRequest> validateOrder(OrderRequest request) {
        if (request != null && request.isValid()) {
            return Optional.of(request);
        }
        return Optional.empty();
    }

    // Other methods return Optional to indicate potential failures
}
```

> **Common Pitfall**: Creating an Optional just to check if it's present and then immediately unwrap it with `get()` doesn't provide any benefits over a regular null check. Optional is most useful in API designs for return values and in fluent method chains.

> **Certification Note**: Understand the purpose of Optional and when to use it. Know the methods available on Optional and how to chain them effectively. Recognize common anti-patterns and avoid them.

Interface Enhancements

Java 8 and later versions introduced several enhancements to interfaces, making them more powerful and flexible.

Default Methods (Java 8+)

Default methods allow interfaces to provide method implementations, enabling backward compatibility when evolving interfaces.

```
// Interface with default method
public interface Vehicle {
    void accelerate();
    void brake();

    // Default method with implementation
    default void honk() {
        System.out.println("Beep!");
    }
}

// Implementing class can use default implementation
public class Car implements Vehicle {
    @Override
    public void accelerate() {
        System.out.println("Car accelerating");
    }

    @Override
    public void brake() {
        System.out.println("Car braking");
```

```
    }

    // No need to implement honk()
}

// Or override it if needed
public class Truck implements Vehicle {
    @Override
    public void accelerate() {
        System.out.println("Truck accelerating");
    }

    @Override
    public void brake() {
        System.out.println("Truck braking");
    }

    @Override
    public void honk() {
        System.out.println("HOOOONK!");
    }
}
```

Static Methods in Interfaces (Java 8+)

Interfaces can define static methods, which are not inherited by implementing classes.

```
public interface PaymentProcessor {
    void processPayment(Payment payment);

    // Static method - called on the interface, not instances
    static PaymentProcessor getDefault() {
        return new DefaultPaymentProcessor();
    }

    static boolean isValid(Payment payment) {
        return payment != null && payment.getAmount() > 0;
    }
}

// Usage
PaymentProcessor processor = PaymentProcessor.getDefault();
boolean valid = PaymentProcessor.isValid(payment);
```

Private Methods in Interfaces (Java 9+)

Private methods allow interface code reuse and better encapsulation.

```
public interface Logger {
    void log(String message);

    default void logInfo(String message) {
        log(addTimestamp("INFO: " + message));
    }

    default void logError(String message) {
        log(addTimestamp("ERROR: " + message));
    }

    // Private method for code reuse within the interface
    private String addTimestamp(String message) {
        return LocalDateTime.now() + " - " + message;
    }

    // Private static method
    private static String formatMessage(String message, String level) {
        return level + ": " + message;
    }
}
```

Resolving Conflicts with Multiple Inheritance

When a class implements multiple interfaces with conflicting default methods, the class must explicitly resolve the conflict.

```
public interface Swimmer {
    default void move() {
        System.out.println("Swimming");
    }
}

public interface Flyer {
    default void move() {
        System.out.println("Flying");
    }
}

// Must override the conflicting method
public class Duck implements Swimmer, Flyer {
    @Override
    public void move() {
        // Option 1: Call one of the interface implementations
        Swimmer.super.move();

        // Option 2: Call both
        // Swimmer.super.move();
        // Flyer.super.move();
```

```
        // Option 3: Provide a completely new implementation
        // System.out.println("Moving like a duck");
    }
}
```

Sealed Interfaces (Java 17+)

Sealed interfaces restrict which classes can implement them.

```
// Sealed interface
public sealed interface Shape permits Circle, Rectangle, Triangle {
    double area();
}

// Permitted implementations
public final class Circle implements Shape {
    private final double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    @Override
    public double area() {
        return Math.PI * radius * radius;
    }
}

public final class Rectangle implements Shape {
    private final double width;
    private final double height;

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

    @Override
    public double area() {
        return width * height;
    }
}

public non-sealed class Triangle implements Shape {
    // non-sealed allows further extension
    private final double base;
    private final double height;

    public Triangle(double base, double height) {
        this.base = base;
        this.height = height;
```

```
    }

    @Override
    public double area() {
        return 0.5 * base * height;
    }
}
```

Functional Interfaces and Default Methods

Many functional interfaces in the standard library include useful default methods:

```
// Predicate with default methods
Predicate<String> startsWithA = s -> s.startsWith("A");
Predicate<String> endsWithX = s -> s.endsWith("X");

// Combining predicates with default methods
Predicate<String> combined = startsWithA.and(endsWithX);
Predicate<String> either = startsWithA.or(endsWithX);
Predicate<String> negated = startsWithA.negate();

// Function with default methods
Function<String, Integer> lengthFunc = String::length;
Function<Integer, String> toStringFunc = Object::toString;

// Composing functions with default methods
Function<String, String> composed = lengthFunc.andThen(toStringFunc);
Function<Integer, Integer> composedReverse = lengthFunc.compose(toStringFunc);
```

> **Common Pitfall**: Default method implementations in interfaces can lead to the diamond problem when a class implements multiple interfaces with the same default method. Always be explicit about which implementation to use in such cases.

> **Certification Note**: Understand how default methods, static methods, and private methods work in interfaces. Know how to resolve conflicts when implementing multiple interfaces with the same default method.

Module System (Project Jigsaw)

The Java Platform Module System (JPMS), introduced in Java 9, provides a way to package and encapsulate Java code into modules.

Module Basics

A module is a self-describing collection of code, data, and resources. It explicitly states what it exports (makes available to other modules) and what it requires (depends on).

```

```

```
// module-info.java (placed in the root of the module)
module com.example.myapp {
    // Modules this module depends on
    requires java.base;  // Implicitly required by all modules
    requires java.sql;
    requires com.example.utils;

    // Packages this module exports
    exports com.example.myapp.api;
    exports com.example.myapp.model to com.example.client;

    // Services this module uses
    uses com.example.spi.MyService;

    // Services this module provides
    provides com.example.spi.MyService with com.example.myapp.impl.MyServiceImpl;

    // Allow reflection access to this module
    opens com.example.myapp.model;
    opens com.example.myapp.dto to com.example.serializer;
}
```

Module Types

1. **Named Modules**: Explicitly defined with `module-info.java`.
2. **Automatic Modules**: Created from JARs on the module path without a `module-info.java`.
3. **Unnamed Module**: Contains all classes on the classpath; has access to all packages of all modules.

Module Directives

```
// requires - specifies module dependencies
requires java.sql;

// requires static - compile-time only dependency
requires static org.junit.jupiter.api;

// requires transitive - dependency is also required by modules that depend on
this one
requires transitive java.xml;

// exports - makes packages accessible to all modules
exports com.example.myapp.api;

// exports to - makes packages accessible only to specific modules
exports com.example.myapp.internal to com.example.plugin;

// opens - allows deep reflection access to a package
opens com.example.myapp.model;

// opens to - allows deep reflection access only to specific modules
```

```
opens com.example.myapp.dto to com.example.json;

// uses - indicates that module uses a service
uses com.example.spi.Logger;

// provides with - specifies implementation of a service
provides com.example.spi.Logger with com.example.myapp.logging.FileLogger;
```

Creating a Modular Application

Directory Structure

```
myapp/
├── src/
│   ├── com.example.myapp/
│   │   ├── module-info.java
│   │   └── com/
│   │       └── example/
│   │           └── myapp/
│   │               ├── Main.java
│   │               └── ...
│   ├── com.example.utils/
│   │   ├── module-info.java
│   │   └── com/
│   │       └── example/
│   │           └── utils/
│   │               ├── StringUtils.java
│   │               └── ...
```

Module Definitions

```
// com.example.utils/module-info.java
module com.example.utils {
    exports com.example.utils;
}

// com.example.myapp/module-info.java
module com.example.myapp {
    requires com.example.utils;

    // Main class
    exports com.example.myapp;
}
```

Compiling and Running

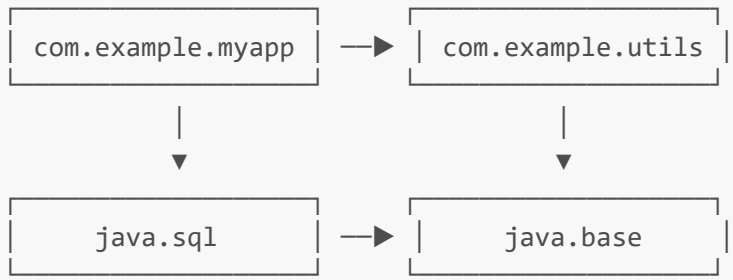
```
# Compile modules
javac -d out --module-source-path src $(find src -name "*.java")

# Run modular application
java --module-path out -m com.example.myapp/com.example.myapp.Main
```

Services and ServiceLoader

The module system integrates with the `ServiceLoader` mechanism for service-provider loading:

```
// Service interface
module com.example.api {
    exports com.example.api.service;
}

public interface Logger {
    void log(String message);
}

// Service implementation
module com.example.impl {
    requires com.example.api;
    provides com.example.api.service.Logger with com.example.impl.FileLogger;
}

public class FileLogger implements Logger {
    @Override
    public void log(String message) {
        // Implementation
    }
}

// Service consumer
module com.example.app {
    requires com.example.api;
    uses com.example.api.service.Logger;
}

public class LoggingManager {
    public void initializeLoggers() {
        ServiceLoader<Logger> loggers = ServiceLoader.load(Logger.class);
        for (Logger logger : loggers) {
            // Use logger
        }
    }
}
```

Module Resolution and Readability



- Module A reads module B if A requires B.
- Code in module A can access exported types from module B only if A reads B.
- A requires statement establishes readability, so does requires transitive.

Migration to Modules

For existing applications, Java supports incremental migration to the module system:

1. **Run on classpath**: Continue using traditional classpath, no changes needed.
2. **Use jdeps**: Analyze dependencies to identify potential module boundaries.

```
jdeps --jdk-internals MyApp.jar
jdeps --generate-module-info output MyApp.jar
```

3. **Create automatic modules**: Place JAR on module path without `module-info.java`.
4. **Add `module-info.java` descriptors**: Convert to explicit modules.

Module Patterns

1. **Aggregator Modules**: Empty modules that only require and re-export other modules.

```
module com.example.framework {
    requires transitive com.example.framework.core;
    requires transitive com.example.framework.util;
    requires transitive com.example.framework.ui;
}
```

2. **API/Implementation Separation**:

```
// API module
module com.example.api {
    exports com.example.api;
}

// Implementation module
module com.example.impl {
    requires com.example.api;
```

```
    provides com.example.api.Service with com.example.impl.ServiceImpl;
}
```

3. **Internal Packages**: Keep non-exported code truly hidden.

```
module com.example.app {
    exports com.example.app.api;  // Public API

    // Internal packages
    // com.example.app.internal (not exported)
}
```

> **Common Pitfall**: Circular module dependencies are not allowed. If module A requires module B, and module B requires module A, this creates a cycle that will result in a compilation error.

> **Certification Note**: Understand the module system concepts, including module declarations, visibility rules, and service providers. Know the different kinds of modules and how they interact.

Local Variable Type Inference (var)

Java 10 introduced local variable type inference using the `var` keyword, which allows the compiler to infer the type of local variables.

Basic Usage

```
// Instead of explicit type declaration
String message = "Hello, Java";

// Using var (compiler infers type as String)
var message = "Hello, Java";

// Complex types become more readable
Map<String, List<String>> mapOfLists = new HashMap<>();
var mapOfLists = new HashMap<String, List<String>>();

// For loop iteration
for (var entry : map.entrySet()) {
    var key = entry.getKey();
    var value = entry.getValue();
    // ...
}

// Traditional for loop
for (var i = 0; i < 10; i++) {
    // ...
}

// Try-with-resources
try (var reader = new BufferedReader(new FileReader("file.txt"))) {
```

```
    // ...
}
```

Where var Can Be Used

```
// Local variables with initializers
var count = 10;
var list = new ArrayList<String>();

// Enhanced for loop indices
for (var item : items) { ... }

// Traditional for loop control variables
for (var i = 0; i < 10; i++) { ... }

// Try-with-resources variables
try (var in = new FileInputStream("file.txt")) { ... }

// In lambda expressions (Java 11+)
var processor = (String s) -> s.length();
```

Where var Cannot Be Used

```
// Class fields
class MyClass {
    var field = "Error";  // Not allowed
}

// Method parameters
void process(var data) { ... }  // Not allowed

// Method return types
var getValue() { ... }  // Not allowed

// Lambda parameter types (without explicit type)
var processor = s -> s.length();  // Not allowed

// Without initializers
var name;  // Not allowed
name = "John";

// With null initializers
var reference = null;  // Not allowed

// Array initializers
var array = { 1, 2, 3 };  // Not allowed
var array = new int[] { 1, 2, 3 };  // Allowed
```

Type Inference Details

The compiler infers:

```
// Specific type from right-hand side
var list = new ArrayList<String>();  // ArrayList<String>
var map = Map.of("key", 42);         // Map<String, Integer>

// Most specific type when initializer is a method call
var result = getResult();            // Type depends on getResult() return type

// Anonymous classes as their own special types
var comparator = new Comparator<String>() {
    @Override
    public int compare(String s1, String s2) {
        return s1.compareTo(s2);
    }
};  // Type is anonymous subclass of Comparator<String>
```

Best Practices for var

```
// GOOD: Type is obvious from the initializer
var list = new ArrayList<String>();
var customer = customerRepository.findById(id);
var count = 0;

// GOOD: Short scope makes it easy to track type
for (var entry : map.entrySet()) {
    processEntry(entry);
}

// GOOD: Makes generic types more readable
var map = new HashMap<String, List<Customer>>();

// AVOID: Type is not obvious from initializer
var result = service.process();  // What is the return type?

// AVOID: Weakens type safety
var list = new ArrayList<>();  // Raw type ArrayList, not ArrayList<Object>

// AVOID: Makes code less self-documenting
var x = getValueSomehow();  // Unclear what x represents

// AVOID: Magic literals without context
var x = 42;  // Without context, meaning is unclear
var limit = 42;  // Better: provides context
```

Interaction with Diamond Operator

```
// With var and explicit type arguments
var list1 = new ArrayList<String>();  // Type: ArrayList<String>

// With var and diamond operator
var list2 = new ArrayList<>();  // Type: ArrayList<Object>

// Without var
ArrayList<String> list3 = new ArrayList<>();  // Type: ArrayList<String>
```

Common Examples

```
// File reading
var lines = Files.readAllLines(Path.of("data.txt"));

// String processing
var text = "1,2,3,4,5";
var parts = text.split(",");
var numbers = new ArrayList<Integer>();
for (var part : parts) {
    numbers.add(Integer.parseInt(part));
}

// Cleaner use with streams
var sum = numbers.stream()
    .filter(n -> n % 2 == 0)
    .mapToInt(Integer::intValue)
    .sum();
```

> **Common Pitfall**: Using `var` with diamond operator `<>` results in a raw type, not a generic type with an inferred parameter.

> **Certification Note**: Understand where `var` can and cannot be used. Know how type inference works and best practices for using `var` to improve code readability.

Switch Expressions and Pattern Matching

Java 12+ enhanced switch statements to also work as expressions, along with introducing pattern matching capabilities.

Traditional Switch Statements

```
// Traditional switch statement
int dayValue;
String day = "MONDAY";

switch (day) {
    case "MONDAY":
        dayValue = 1;
```

```
            break;
        case "TUESDAY":
            dayValue = 2;
            break;
        case "WEDNESDAY":
            dayValue = 3;
            break;
        // ...
        default:
            dayValue = 0;
            break;
    }
```

Switch Expressions (Java 12+)

```
// Switch expression with arrow labels
int dayValue = switch (day) {
    case "MONDAY" -> 1;
    case "TUESDAY" -> 2;
    case "WEDNESDAY" -> 3;
    case "THURSDAY" -> 4;
    case "FRIDAY" -> 5;
    case "SATURDAY", "SUNDAY" -> {
        // Block with yield
        System.out.println("Weekend!");
        yield 0;  // Return value
    }
    default -> -1;
};

// Multiple case labels
String result = switch (day) {
    case "MONDAY", "TUESDAY", "WEDNESDAY", "THURSDAY", "FRIDAY" -> "Weekday";
    case "SATURDAY", "SUNDAY" -> "Weekend";
    default -> "Invalid day";
};
```

Switch with Traditional Case Syntax and `yield` (Java 13+)

```
// Switch expression with traditional case syntax and yield
int dayValue = switch (day) {
    case "MONDAY":
    case "TUESDAY":
        yield 1;
    case "WEDNESDAY":
    case "THURSDAY":
        System.out.println("Mid-week");
        yield 2;
    case "FRIDAY":
```

```
            yield 3;
        default:
            yield -1;
    };
```

Pattern Matching for instanceof (Java 16+)

```
// Before pattern matching
if (obj instanceof String) {
    String s = (String) obj;
    if (s.length() > 5) {
        // Process s
    }
}

// With pattern matching
if (obj instanceof String s && s.length() > 5) {
    // Process s directly
}
```

Pattern Matching for Switch (Java 17+)

```
// Pattern matching in switch
Object obj = getObject();
String result = switch (obj) {
    case String s -> "String of length " + s.length();
    case Integer i -> "Integer value " + i;
    case Long l -> "Long value " + l;
    case Double d -> "Double value " + d;
    case null -> "Null value";
    default -> "Unknown type";
};
```

Sealed Types with Pattern Matching (Java 17+)

```
// Sealed hierarchy
sealed interface Shape permits Circle, Rectangle, Triangle { }
record Circle(double radius) implements Shape { }
record Rectangle(double width, double height) implements Shape { }
record Triangle(double a, double b, double c) implements Shape { }

// Exhaustive pattern matching with sealed types
double area = switch (shape) {
    case Circle c -> Math.PI * c.radius() * c.radius();
    case Rectangle r -> r.width() * r.height();
    case Triangle t -> {
```

```
            double s = (t.a() + t.b() + t.c()) / 2;
            yield Math.sqrt(s * (s - t.a()) * (s - t.b()) * (s - t.c()));
        }
        // No default needed when all types are covered
    };
```

Record Patterns (Java 19+ in preview)

```
// Record pattern (preview feature)
record Point(int x, int y) { }
record Rectangle(Point topLeft, Point bottomRight) { }

// Extracting components with pattern matching
Object obj = getShape();
if (obj instanceof Rectangle(Point(int x1, int y1), Point(int x2, int y2))) {
    int width = x2 - x1;
    int height = y2 - y1;
    System.out.println("Area: " + (width * height));
}
```

Guard Patterns (Java 19+ in preview)

```
// Using pattern matching with guards
Object obj = getValue();
String result = switch (obj) {
    case String s when s.length() > 5 -> "Long string: " + s;
    case String s -> "Short string: " + s;
    case Integer i when i > 0 -> "Positive integer: " + i;
    case Integer i -> "Non-positive integer: " + i;
    default -> "Something else";
};
```

Practical Examples

```
// HTTP response handling
HttpResponse response = client.send(request);
String message = switch (response.statusCode()) {
    case 200, 201, 202 -> "Success";
    case 400, 401, 403 -> "Client error: " + response.body();
    case 500, 502, 503 -> "Server error";
    default -> "Unknown status code: " + response.statusCode();
};

// Data processing with pattern matching
Object data = getData();
int result = switch (data) {
```

```
    case String s when s.startsWith("id:") ->
        Integer.parseInt(s.substring(3));
    case Integer i -> i;
    case Long l -> l.intValue();
    case List<?> list when !list.isEmpty() && list.get(0) instanceof Integer i ->
        i;
    default -> -1;
};
```

> **Common Pitfall**: Switch expressions require that all possible values of the input type be covered, either through cases or a default clause. Without an exhaustive coverage, a compilation error occurs.

> **Certification Note**: Understand the difference between switch statements and switch expressions. Know the syntax for arrow labels and when to use `yield`. Be familiar with pattern matching in `instanceof` and `switch`.

Text Blocks

Text blocks, introduced in Java 15, provide a way to create multi-line string literals with better readability and fewer escape sequences.

Basic Syntax

```
// Traditional multi-line string
String html = "<html>\n" +
              "    <body>\n" +
              "        <p>Hello, World!</p>\n" +
              "    </body>\n" +
              "</html>";

// Text block alternative
String html = """
              <html>
                  <body>
                      <p>Hello, World!</p>
                  </body>
              </html>
              """;
```

Indentation and Whitespace

```
// Indentation is relative to the closing delimiter
String indented = """
    This is at indentation level 1.
      This is at level 2.
        Level 3.
    Back to level 1.
    """;
```

```
// Trailing spaces can be preserved with a \ at the end of a line
String spaces = """
    This line ends with 5 spaces:     \
    Next line has no extra spaces.
    """;

// Escape sequences
String escaped = """
    This line has "quotes" in it.
    This line has a \t tab.
    This is a backslash: \\.
    """;

// Preventing a newline at the end with \
String noTrailingNewline = """
    This is a text block \
    without a newline at the end.""";
```

String Processing Methods

```
// Formatted text blocks (Java 15+)
String name = "John";
int age = 30;
String formatted = """
    Name: %s
    Age: %d
    """.formatted(name, age);

// String templates (Java 21+)
String template = STR."""
    Name: \{name}
    Age: \{age}
    """;

// Stripping indentation
String stripped = """
    Line 1
    Line 2
    Line 3
    """.stripIndent();

// Text block + String operations
String transformed = """
    <html>
        <body>
            <p>Hello</p>
        </body>
    </html>
    """
```

```
    .replaceAll("<[^>]*>", "")
    .trim();
```

Common Use Cases

```
// SQL queries
String query = """
    SELECT id, name, email
    FROM users
    WHERE status = 'active'
    ORDER BY name ASC
    LIMIT 10
    """;

// JSON
String json = """
    {
        "name": "John Doe",
        "age": 30,
        "address": {
            "street": "123 Main St",
            "city": "Anytown"
        },
        "phoneNumbers": [
            "555-1234",
            "555-5678"
        ]
    }
    """;

// XML
String xml = """
    <?xml version="1.0" encoding="UTF-8"?>
    <project>
        <groupId>com.example</groupId>
        <artifactId>demo</artifactId>
        <version>1.0.0</version>
        <properties>
            <java.version>21</java.version>
        </properties>
    </project>
    """;

// HTML
String html = """
    <!DOCTYPE html>
    <html>
    <head>
        <title>Example</title>
    </head>
    <body>
```

```
        <h1>Text Blocks Demo</h1>
        <p>This is a paragraph.</p>
    </body>
    </html>
    """;
```

> **Common Pitfall**: The indentation of a text block is determined by the position of the closing delimiter (`"""`). If you indent your code inconsistently, your text block might have unexpected whitespace.

> **Certification Note**: Understand how text blocks work, especially regarding indentation and line termination. Know when and how to use escape sequences in text blocks.

Records

Records, introduced in Java 16, provide a concise way to declare classes that are transparent carriers for immutable data.

Basic Record Declaration

```
// Simple record
record Point(int x, int y) {
    // Automatically creates:
    // - Private final fields x and y
    // - Constructor Point(int x, int y)
    // - Accessor methods x() and y()
    // - equals(), hashCode(), and toString()
}

// Usage
Point p1 = new Point(10, 20);
int x = p1.x();
int y = p1.y();
System.out.println(p1);  // Point[x=10, y=20]
```

Custom Constructors

```
// Compact constructor (implicit parameters)
record Range(int start, int end) {
    // Validation in compact constructor
    Range {
        if (end < start) {
            throw new IllegalArgumentException("End cannot be less than start");
        }
    }
}

// Canonical constructor (explicit parameters)
record User(String username, String email) {
```

```
    // Full canonical constructor
    public User(String username, String email) {
        if (username == null || email == null) {
            throw new NullPointerException("Username and email cannot be null");
        }
        this.username = username;
        this.email = email;
    }
}

// Additional constructors
record Rectangle(double width, double height) {
    // Canonical constructor with validation
    public Rectangle(double width, double height) {
        if (width <= 0 || height <= 0) {
            throw new IllegalArgumentException("Dimensions must be positive");
        }
        this.width = width;
        this.height = height;
    }

    // Additional constructor for squares
    public Rectangle(double side) {
        this(side, side);  // Calls canonical constructor
    }
}
```

Methods in Records

```
record Circle(double radius) {
    // Constant
    private static final double PI = 3.14159;

    // Instance method
    public double area() {
        return PI * radius * radius;
    }

    // Override accessor
    public double radius() {
        return Math.max(0, radius);  // Ensure non-negative
    }

    // Override toString
    @Override
    public String toString() {
        return String.format("Circle(radius=%.2f)", radius);
    }

    // Static method
    public static Circle unitCircle() {
```

```
            return new Circle(1.0);
        }
    }
```

Records and Interfaces

```
// Records can implement interfaces
interface Shape {
    double area();
    double perimeter();
}

record Square(double side) implements Shape {
    @Override
    public double area() {
        return side * side;
    }

    @Override
    public double perimeter() {
        return 4 * side;
    }
}
```

Nested Records

```
// Nested record
public class Customer {
    // Static nested record
    public static record Address(String street, String city, String zipCode) {
        // Methods for Address
    }

    private final String name;
    private final Address address;

    public Customer(String name, Address address) {
        this.name = name;
        this.address = address;
    }

    // Getters...
}

// Usage
Customer.Address address = new Customer.Address("123 Main St", "Anytown",
"12345");
Customer customer = new Customer("John Doe", address);
```

Generic Records

```
// Generic record
record Pair<K, V>(K key, V value) {
    // Methods using K and V
    public <T> Pair<T, V> withKey(T newKey) {
        return new Pair<>(newKey, value);
    }

    public <T> Pair<K, T> withValue(T newValue) {
        return new Pair<>(key, newValue);
    }
}

// Usage
Pair<String, Integer> pair = new Pair<>("age", 30);
Pair<Integer, Integer> numericPair = pair.withKey(1);
```

Record Patterns (Java 19+ in preview)

```
// Pattern matching with records
record Person(String name, int age) { }

void processPerson(Object obj) {
    if (obj instanceof Person(String name, int age)) {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
}

// Nested pattern matching
record Address(String street, String city) { }
record Customer(String name, Address address) { }

void processCustomer(Object obj) {
    if (obj instanceof Customer(String name, Address(String street, String city)))
{
        System.out.println("Customer: " + name);
        System.out.println("Lives at: " + street + ", " + city);
    }
}
```

Records vs. Classes

```
// Traditional class
public final class PointClass {
    private final int x;
    private final int y;
```

```
    public PointClass(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() { return x; }
    public int getY() { return y; }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        PointClass that = (PointClass) o;
        return x == that.x && y == that.y;
    }

    @Override
    public int hashCode() {
        return Objects.hash(x, y);
    }

    @Override
    public String toString() {
        return "PointClass[x=" + x + ", y=" + y + "]";
    }
}

// Equivalent record
record Point(int x, int y) { }
```

Limitations of Records

```
// Records cannot extend other classes
record Employee(String name, int id) extends Person { }  // Error

// Records cannot be abstract
abstract record AbstractRecord() { }  // Error

// Record fields are final and cannot be modified
record Counter(int value) {
    public void increment() {
        value++;  // Error: value is final
    }
}

// Workaround for mutable state
record Counter(int value) {
    public Counter increment() {
        return new Counter(value + 1);  // Create new instance
```

```
    }
}
```

> **Common Pitfall**: Records are primarily for immutable data carriers. If you need mutable state or complex inheritance, use traditional classes instead.

> **Certification Note**: Understand records as immutable data carriers with automatically generated methods. Know how to customize canonical constructors, add methods, and implement interfaces with records.

Sealed Classes

Sealed classes, introduced in Java 17, restrict which other classes can extend or implement them, providing more control over class hierarchies.

Basic Sealed Class Declaration

```
// Sealed class with permitted subclasses
public sealed class Shape permits Circle, Rectangle, Triangle {
    // Common methods and fields
    protected double x, y;

    public Shape(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public abstract double area();
}

// Permitted subclasses must use one of:
// 1. final - no further extension
// 2. sealed - restricted extension
// 3. non-sealed - unrestricted extension

// Option 1: final subclass
public final class Circle extends Shape {
    private final double radius;

    public Circle(double x, double y, double radius) {
        super(x, y);
        this.radius = radius;
    }

    @Override
    public double area() {
        return Math.PI * radius * radius;
    }
}

// Option 2: sealed subclass with its own permitted subclasses
```

```
public sealed class Rectangle extends Shape permits Square {
    protected double width, height;

    public Rectangle(double x, double y, double width, double height) {
        super(x, y);
        this.width = width;
        this.height = height;
    }

    @Override
    public double area() {
        return width * height;
    }
}

public final class Square extends Rectangle {
    public Square(double x, double y, double side) {
        super(x, y, side, side);
    }
}

// Option 3: non-sealed subclass allowing unrestricted extension
public non-sealed class Triangle extends Shape {
    private final double base, height;

    public Triangle(double x, double y, double base, double height) {
        super(x, y);
        this.base = base;
        this.height = height;
    }

    @Override
    public double area() {
        return 0.5 * base * height;
    }
}

// Any class can extend a non-sealed class
public class EquilateralTriangle extends Triangle {
    public EquilateralTriangle(double x, double y, double side) {
        super(x, y, side, side * Math.sqrt(3) / 2);
    }
}
```

Sealed Interfaces

```
// Sealed interface
public sealed interface Vehicle permits Car, Truck, Motorcycle {
    void drive();
    double getFuelEfficiency();
}
```

```
// Final implementation
public final class Car implements Vehicle {
    @Override
    public void drive() {
        System.out.println("Driving car");
    }

    @Override
    public double getFuelEfficiency() {
        return 30.0;  // mpg
    }
}

// Sealed implementation
public sealed interface Truck extends Vehicle permits PickupTruck, SemiTruck {
    double getCargoCapacity();
}

// Non-sealed implementation
public non-sealed interface Motorcycle extends Vehicle {
    boolean hasHandlebars();
}
```

Sealed Classes with Records

```
// Sealed hierarchy with records
public sealed interface Shape permits CircleRecord, RectangleRecord,
TriangleRecord {
    double area();
}

public record CircleRecord(double radius) implements Shape {
    @Override
    public double area() {
        return Math.PI * radius * radius;
    }
}

public record RectangleRecord(double width, double height) implements Shape {
    @Override
    public double area() {
        return width * height;
    }
}

public record TriangleRecord(double base, double height) implements Shape {
    @Override
    public double area() {
        return 0.5 * base * height;
```

```
    }
}
```

Pattern Matching with Sealed Classes

```
// Exhaustive pattern matching with sealed types
Shape shape = getShape();
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;
    // No default needed if all subclasses are covered
};
```

Benefits of Sealed Classes

1. **Controlled Extension**: You determine which classes can extend your class.
2. **Pattern Matching**: Enables exhaustive matching without default case.
3. **API Design**: Clearly communicates intended hierarchy to users.
4. **Maintenance**: Easier to change implementation when all subclasses are known.

Sealed Classes vs. Alternatives

```
// Alternative 1: Package-private superclass
// Limits extension to same package
class PackagePrivateShape {
    // Only classes in same package can extend
}

// Alternative 2: Private constructor with static factory
public class FactoryShape {
    private FactoryShape() { }

    public static FactoryShape createCircle() { ... }
    public static FactoryShape createRectangle() { ... }
}

// Alternative 3: Enum (limited to fixed instances)
public enum EnumShape {
    CIRCLE, RECTANGLE, TRIANGLE;

    public double area() {
        return switch (this) {
            case CIRCLE -> Math.PI * 5 * 5;  // Fixed size
            case RECTANGLE -> 10 * 20;
            case TRIANGLE -> 0.5 * 10 * 5;
        };
```

```
    }
}
```

> **Common Pitfall**: If a sealed class is in a different module than its permitted subclasses, you need to ensure that the subclasses can access the sealed class. This might require explicit `exports` directives in the module system.

> **Certification Note**: Understand the relationship between sealed classes and their permitted subclasses. Know the restrictions on subclasses (final, sealed, or non-sealed) and how sealed hierarchies enable exhaustive pattern matching.

Virtual Threads and Project Loom

Java 21 introduces virtual threads, lightweight threads managed by the JVM rather than the operating system. This feature, part of Project Loom, allows for high-throughput concurrency with a familiar programming model.

Creating Virtual Threads

```
// Creating a virtual thread
Thread vThread = Thread.startVirtualThread(() -> {
    System.out.println("Running in a virtual thread");
});

// Thread.Builder API
Thread.Builder builder = Thread.ofVirtual().name("worker-", 0);
Thread vThread = builder.start(() -> {
    System.out.println("Worker thread");
});

// Creating multiple virtual threads
List<Thread> threads = IntStream.range(0, 10_000)
    .mapToObj(i -> Thread.startVirtualThread(() -> {
        System.out.println("Thread " + i);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }))
    .toList();

// Wait for all to complete
for (Thread t : threads) {
    t.join();
}
```

Executors Factory Methods

```
// Using ExecutorService with virtual threads
try (ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor()) {
    IntStream.range(0, 10_000).forEach(i -> {
        executor.submit(() -> {
            // Task code
            return i * i;
        });
    });
}  // Auto-shutdown with try-with-resources

// Traditional vs. virtual thread executor
ExecutorService platformExecutor = Executors.newFixedThreadPool(100);
ExecutorService virtualExecutor = Executors.newVirtualThreadPerTaskExecutor();
```

Structured Concurrency (Java 21 Preview)

```
// Structured Concurrency API (preview)
try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
    Future<String> user = scope.fork(() -> fetchUser());
    Future<Integer> order = scope.fork(() -> fetchOrder());

    // Wait for all tasks and handle errors
    scope.join();
    scope.throwIfFailed();

    // Process results
    processUserAndOrder(user.resultNow(), order.resultNow());
}
```

Comparing Platform and Virtual Threads

```
// Platform thread - heavy, OS-level thread
Thread platformThread = Thread.ofPlatform()
    .name("platform-thread")
    .start(() -> process());

// Virtual thread - lightweight, JVM-managed
Thread virtualThread = Thread.ofVirtual()
    .name("virtual-thread")
    .start(() -> process());

// Performance comparison
void platformThreads() throws InterruptedException {
    long start = System.currentTimeMillis();
    try (var executor = Executors.newFixedThreadPool(200)) {
        for (int i = 0; i < 10_000; i++) {
            executor.submit(() -> blockingOperation());
        }
```

```
    }
    System.out.println("Platform threads took: " +
                       (System.currentTimeMillis() - start) + "ms");
}

void virtualThreads() throws InterruptedException {
    long start = System.currentTimeMillis();
    try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
        for (int i = 0; i < 10_000; i++) {
            executor.submit(() -> blockingOperation());
        }
    }
    System.out.println("Virtual threads took: " +
                       (System.currentTimeMillis() - start) + "ms");
}
```

Blocking Operations with Virtual Threads

```
// Virtual threads handle blocking operations efficiently
void handleRequest() {
    Thread.startVirtualThread(() -> {
        try {
            // Blocking database query
            Connection conn = DriverManager.getConnection(DB_URL);
            PreparedStatement stmt = conn.prepareStatement("SELECT * FROM users");
            ResultSet rs = stmt.executeQuery();

            // Thread "parks" during I/O, allowing carrier thread to run other VTs
            while (rs.next()) {
                processRow(rs);
            }

            conn.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    });
}
```

Thread-Local Variables with Virtual Threads

```
// Regular ThreadLocal works with virtual threads, but consider scope
ThreadLocal<User> userContext = new ThreadLocal<>();

void processWithContext() {
    userContext.set(new User("admin"));
    try {
        // operations using context
    } finally {
```

```
            userContext.remove();  // Clean up to avoid memory leaks
        }
    }

    // Scoped values alternative (preview)
    final ScopedValue<User> USER = ScopedValue.newInstance();

    void processScopedValue() {
        ScopedValue.where(USER, new User("admin"))
            .run(() -> {
                // Code inside has access to USER.get()
                // No need to clean up - automatically handled by scope
            });
    }
```

Best Practices for Virtual Threads

```
    // DO: Create many short-lived threads for specific tasks
    void handleRequests(List<Request> requests) {
        requests.forEach(request ->
            Thread.startVirtualThread(() -> processRequest(request))
        );
    }

    // DO: Use structured concurrency when available
    void fetchData() throws Exception {
        try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
            Future<User> user = scope.fork(() -> fetchUser());
            Future<List<Order>> orders = scope.fork(() -> fetchOrders());

            scope.join().throwIfFailed();

            processUserAndOrders(user.resultNow(), orders.resultNow());
        }
    }

    // DON'T: Create thread pools of virtual threads
    ExecutorService badPool = Executors.newFixedThreadPool(100,
        task -> Thread.startVirtualThread(task));  // Pointless limitation

    // DON'T: Use thread affinity with synchronized blocks on shared resources
    synchronized void badMethod() {  // Pins virtual thread to carrier thread
        // Blocking operation
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    // INSTEAD: Use higher-level concurrency constructs
```

```
ReentrantLock lock = new ReentrantLock();
void betterMethod() {
    lock.lock();
    try {
        // Critical section
    } finally {
        lock.unlock();
    }
}
```

Real-world Examples

```
// HTTP server with virtual threads
HttpServer server = HttpServer.create(new InetSocketAddress(8080), 0);
server.createContext("/api", exchange -> {
    Thread.startVirtualThread(() -> {
        try {
            // Blocking operations won't block the server
            String response = fetchDataFromDatabase();
            exchange.sendResponseHeaders(200, response.length());
            try (OutputStream os = exchange.getResponseBody()) {
                os.write(response.getBytes());
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    });
});
server.start();

// Web crawler with virtual threads
void crawl(List<URL> urls) {
    try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
        List<Future<Page>> futures = urls.stream()
            .map(url -> executor.submit(() -> fetchPage(url)))
            .toList();

        for (Future<Page> future : futures) {
            Page page = future.get();
            processPage(page);
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}

Page fetchPage(URL url) throws IOException {
    // Blocking connection, but efficient with virtual threads
    HttpURLConnection conn = (HttpURLConnection) url.openConnection();
    try (InputStream is = conn.getInputStream();
         BufferedReader reader = new BufferedReader(new InputStreamReader(is))) {
```

```
        StringBuilder content = new StringBuilder();
        String line;
        while ((line = reader.readLine()) != null) {
            content.append(line).append("\n");
        }

        return new Page(url, content.toString());
    }
}
```

> **Common Pitfall**: Using synchronized blocks extensively with virtual threads can negate their benefits because synchronization can pin virtual threads to carrier threads.

> **Certification Note**: Understand the differences between platform threads and virtual threads. Know how to create and use virtual threads with the Thread API and Executors. Be aware of the implications for blocking operations and thread-local variables.

Pattern Matching for instanceof and switch

Pattern matching provides a more concise and type-safe way of extracting data from objects based on their type or structure. Java has been incrementally adding pattern matching features since Java 16.

Pattern Matching for instanceof (Java 16+)

```
// Traditional approach
Object obj = getValue();
if (obj instanceof String) {
    String s = (String) obj;
    if (s.length() > 5) {
        System.out.println(s.toUpperCase());
    }
}

// With pattern matching
Object obj = getValue();
if (obj instanceof String s && s.length() > 5) {
    System.out.println(s.toUpperCase());
}
```

Scope of Pattern Variables

```
// Pattern variable scope includes else branch
if (obj instanceof String s) {
    System.out.println("String: " + s);
} else {
    // s is not in scope here
    System.out.println("Not a string");
```

```
}

// Pattern variable with condition
if (obj instanceof String s && s.length() > 0) {
    // s is in scope and non-empty
}

// Pattern variable with negation
if (!(obj instanceof String s)) {
    // s is not in scope here
} else {
    // s is in scope here
}

// Pattern in while loop
while (obj instanceof String s && !s.isEmpty()) {
    // Process s
    obj = getNext();
}
```

Pattern Matching for switch (Java 17+ Preview)

```
// Traditional switch with instanceof
Object obj = getValue();
String result;
if (obj instanceof Integer i) {
    result = "Integer: " + i;
} else if (obj instanceof String s) {
    result = "String: " + s;
} else if (obj instanceof Double d) {
    result = "Double: " + d;
} else {
    result = "Unknown";
}

// With pattern matching in switch
Object obj = getValue();
String result = switch (obj) {
    case Integer i -> "Integer: " + i;
    case String s -> "String: " + s;
    case Double d -> "Double: " + d;
    case null -> "Null value";
    default -> "Unknown";
};
```

Guards in Pattern Matching

```
// Guards with instanceof patterns
if (obj instanceof String s && s.startsWith("prefix")) {
```

```
    // s is a String starting with "prefix"
}

// Guards with switch patterns
String result = switch (obj) {
    case String s when s.length() > 5 -> "Long string: " + s;
    case String s -> "Short string: " + s;
    case Integer i when i > 0 -> "Positive: " + i;
    case Integer i -> "Zero or negative: " + i;
    default -> "Something else";
};
```

Pattern Matching with Records (Java 19+ Preview)

```
// Pattern matching with record patterns
record Point(int x, int y) { }
record Rectangle(Point topLeft, Point bottomRight) { }

// Extracting components with pattern matching
if (obj instanceof Rectangle(Point(int x1, int y1), Point(int x2, int y2))) {
    int width = x2 - x1;
    int height = y2 - y1;
    System.out.println("Width: " + width + ", Height: " + height);
}

// Switch with record patterns
Object shape = getShape();
String description = switch (shape) {
    case Circle(Point(var x, var y), var radius) ->
        "Circle at (" + x + "," + y + ") with radius " + radius;
    case Rectangle(Point(var x1, var y1), Point(var x2, var y2)) -> {
        int width = x2 - x1;
        int height = y2 - y1;
        yield "Rectangle with width " + width + " and height " + height;
    }
    default -> "Unknown shape";
};
```

Pattern Matching with Arrays (Preview)

```
// Pattern matching with arrays
Object obj = new int[] {1, 2, 3};
if (obj instanceof int[] arr) {
    System.out.println("Sum: " + Arrays.stream(arr).sum());
}

// Switch with array patterns
String result = switch (obj) {
    case String[] arr -> "String array of length " + arr.length;
```

```
    case int[] arr -> "Int array with sum " + Arrays.stream(arr).sum();
    case Point[] arr -> "Point array with " + arr.length + " points";
    default -> "Not an array";
};
```

Nested Pattern Matching

```
// Nested patterns
Object obj = getComplexObject();
if (obj instanceof Map<?, ?> map
        && map.get("name") instanceof String name
        && map.get("age") instanceof Integer age) {
    System.out.println("Name: " + name + ", Age: " + age);
}

// Deeper nesting with records
record Address(String street, String city) { }
record Person(String name, int age, Address address) { }

if (obj instanceof Person(var name, var age, Address(var street, var city))) {
    System.out.println(name + " lives at " + street + ", " + city);
}
```

Exhaustiveness in switch

```
// With sealed types, pattern matching can be exhaustive
sealed interface Shape permits Circle, Rectangle, Triangle { }
record Circle(double radius) implements Shape { }
record Rectangle(double width, double height) implements Shape { }
record Triangle(double a, double b, double c) implements Shape { }

Shape shape = getShape();
double area = switch (shape) {
    case Circle c -> Math.PI * c.radius() * c.radius();
    case Rectangle r -> r.width() * r.height();
    case Triangle t -> {
        double s = (t.a() + t.b() + t.c()) / 2;
        yield Math.sqrt(s * (s - t.a()) * (s - t.b()) * (s - t.c()));
    }
    // No default needed with sealed types if all cases are covered
};
```

Real-world Examples

```
// Processing different types of messages
Message msg = receiveMessage();
```

```
String response = switch (msg) {
    case TextMessage txt when txt.isUrgent() ->
        "URGENT: " + txt.getContent();
    case TextMessage txt ->
        "Text: " + txt.getContent();
    case ImageMessage img ->
        "Image (" + img.getWidth() + "x" + img.getHeight() + ")";
    case AudioMessage audio ->
        "Audio (" + audio.getDuration() + " seconds)";
    case null -> "Empty message";
    default -> "Unknown message type";
};

// Handling API responses
ApiResponse response = client.send(request);
Result result = switch (response) {
    case SuccessResponse(var data, var metadata) ->
        processData(data, metadata);
    case ErrorResponse(var code, var message) when code >= 500 ->
        handleServerError(code, message);
    case ErrorResponse(var code, var message) ->
        handleClientError(code, message);
    case RedirectResponse(var location) ->
        followRedirect(location);
    case null -> throw new IllegalStateException("Null response");
};
```

> **Common Pitfall**: Pattern variables in switch are implicitly final, so they cannot be modified within case blocks.

> **Certification Note**: Understand pattern matching for both `instanceof` and `switch`. Know the scope rules for pattern variables and how pattern matching interacts with sealed types to enable exhaustive checking.

Foreign Function & Memory API

The Foreign Function & Memory (FFM) API, introduced in Java 21, provides a way to call native libraries directly from Java and to safely work with native memory.

Foreign Function Interface

```
// Importing native libraries
// Getting a native linker
Linker linker = Linker.nativeLinker();

// Loading a C library
SymbolLookup stdlib = SymbolLookup.libraryLookup("c", MemorySession.global());

// Finding a native symbol
MemorySegment strchr = stdlib.lookup("strchr").orElseThrow();
```

```
// Defining function descriptor
FunctionDescriptor strchrDesc = FunctionDescriptor.of(
    CLinker.C_POINTER,     // Return type
    CLinker.C_POINTER,     // String pointer
    CLinker.C_INT          // Character to find
);

// Creating a method handle
MethodHandle strchrHandle = linker.downcallHandle(
    strchr,
    strchrDesc
);

// Using the method handle
try (MemorySession session = MemorySession.openConfined()) {
    // Allocate and initialize native string
    MemorySegment cString = session.allocateUtf8String("Hello, World!");

    // Call native function (find 'W' in string)
    MemorySegment result = (MemorySegment) strchrHandle.invoke(cString, 'W');

    // Check result and convert back to Java string
    if (!result.equals(MemorySegment.NULL)) {
        String substring = result.getUtf8String(0);
        System.out.println("Found: " + substring);  // "World!"
    }
}
```

Memory Access

```
// Allocating native memory
try (MemorySession session = MemorySession.openConfined()) {
    // Allocate 100 bytes
    MemorySegment segment = session.allocate(100);

    // Get a memory access var handle for int
    VarHandle intHandle = MemoryLayout.sequenceLayout(25, ValueLayout.JAVA_INT)
        .varHandle(int.class, MemoryLayout.PathElement.sequenceElement());

    // Write values to memory
    for (int i = 0; i < 25; i++) {
        intHandle.set(segment, (long) i, i * 10);
    }

    // Read values from memory
    for (int i = 0; i < 25; i++) {
        int value = (int) intHandle.get(segment, (long) i);
        System.out.println("Value at index " + i + ": " + value);
    }
}
```

Struct Support

```
// Defining a C struct layout
StructLayout pointLayout = MemoryLayout.structLayout(
    ValueLayout.JAVA_INT.withName("x"),
    ValueLayout.JAVA_INT.withName("y")
);

// Create var handles for struct fields
VarHandle xHandle = pointLayout.varHandle(int.class,
    PathElement.groupElement("x"));
VarHandle yHandle = pointLayout.varHandle(int.class,
    PathElement.groupElement("y"));

// Using struct in memory
try (MemorySession session = MemorySession.openConfined()) {
    // Allocate memory for struct
    MemorySegment point = session.allocate(pointLayout);

    // Set field values
    xHandle.set(point, 10);
    yHandle.set(point, 20);

    // Get field values
    int x = (int) xHandle.get(point);
    int y = (int) yHandle.get(point);

    System.out.println("Point: (" + x + ", " + y + ")");
}
```

Array Support

```
// Defining a C array layout
SequenceLayout intArrayLayout = MemoryLayout.sequenceLayout(
    10,                  // 10 elements
    ValueLayout.JAVA_INT // int type
);

// Create a var handle for array elements
VarHandle intElemHandle = intArrayLayout.varHandle(int.class,
    PathElement.sequenceElement());

// Using array in memory
try (MemorySession session = MemorySession.openConfined()) {
    // Allocate memory for array
    MemorySegment array = session.allocate(intArrayLayout);

    // Set array elements
    for (int i = 0; i < 10; i++) {
        intElemHandle.set(array, (long) i, i * i);
```

```
    }

    // Get array elements
    for (int i = 0; i < 10; i++) {
        int value = (int) intElemHandle.get(array, (long) i);
        System.out.println("Array[" + i + "] = " + value);
    }
}
```

Callback Support

```
// Defining a callback function
class Callbacks {
    static int compare(MemorySegment a, MemorySegment b) {
        return Integer.compare(a.get(ValueLayout.JAVA_INT, 0),
                               b.get(ValueLayout.JAVA_INT, 0));
    }
}

// Creating a function pointer for the callback
FunctionDescriptor compareDesc = FunctionDescriptor.of(
    ValueLayout.JAVA_INT,    // Return type
    ValueLayout.ADDRESS,     // Pointer to first int
    ValueLayout.ADDRESS      // Pointer to second int
);

MethodHandle compareHandle = MethodHandles.lookup()
    .findStatic(Callbacks.class, "compare",
                MethodType.methodType(int.class, MemorySegment.class,
MemorySegment.class));

// Creating a native function pointer
try (MemorySession session = MemorySession.openConfined()) {
    MemorySegment compareFuncPtr = linker.upcallStub(
        compareHandle,
        compareDesc,
        session
    );

    // qsort example using the callback
    MemorySegment qsort = stdlib.lookup("qsort").orElseThrow();
    FunctionDescriptor qsortDesc = FunctionDescriptor.ofVoid(
        ValueLayout.ADDRESS,     // Array pointer
        ValueLayout.JAVA_LONG,   // Number of elements
        ValueLayout.JAVA_LONG,   // Size of each element
        ValueLayout.ADDRESS      // Compare function pointer
    );

    MethodHandle qsortHandle = linker.downcallHandle(qsort, qsortDesc);

    // Create and fill an array to sort
```

```
        MemorySegment array = session.allocateArray(ValueLayout.JAVA_INT,
            new int[] {5, 3, 8, 1, 7, 2, 9, 4, 6, 0});

        // Call qsort
        qsortHandle.invoke(array, 10L, 4L, compareFuncPtr);

        // Print sorted array
        for (int i = 0; i < 10; i++) {
            System.out.print(array.getAtIndex(ValueLayout.JAVA_INT, i) + " ");
        }
        // Output: 0 1 2 3 4 5 6 7 8 9
    }
```

Safety Features

```
// Memory session with confined scope
try (MemorySession session = MemorySession.openConfined()) {
    MemorySegment segment = session.allocate(100);

    // Memory is only valid within this session
    processSegment(segment);

} // Memory automatically freed here

// Attempting to use after session is closed throws IllegalStateException
// processSegment(segment);  // Error!

// Shared sessions for multithreaded access
try (MemorySession sharedSession = MemorySession.openShared()) {
    MemorySegment sharedSegment = sharedSession.allocate(1024);

    // Can be safely accessed from multiple threads
    CompletableFuture<Void> future1 = CompletableFuture.runAsync(() -> {
        processSegmentConcurrently(sharedSegment, 0, 512);
    });

    CompletableFuture<Void> future2 = CompletableFuture.runAsync(() -> {
        processSegmentConcurrently(sharedSegment, 512, 1024);
    });

    CompletableFuture.allOf(future1, future2).join();
}

// Global session for permanent native memory
MemorySession globalSession = MemorySession.global();
MemorySegment globalSegment = globalSession.allocate(100);
// This memory stays allocated until the program exits
```

Real-world Examples

```
// Example: Using native image processing library
// Loading a C library for image processing
SymbolLookup imageLib = SymbolLookup.libraryLookup("imageprocessing", global());

// Looking up a function to resize an image
MemorySegment resizeFunc = imageLib.lookup("resize_image").orElseThrow();

// Function descriptor
FunctionDescriptor resizeDesc = FunctionDescriptor.of(
    ValueLayout.ADDRESS,     // Return: new image pointer
    ValueLayout.ADDRESS,     // Input: original image
    ValueLayout.JAVA_INT,    // Input: new width
    ValueLayout.JAVA_INT     // Input: new height
);

// Method handle
MethodHandle resizeHandle = linker.downcallHandle(resizeFunc, resizeDesc);

// Example: Reading system information using native APIs
// Load Windows kernel32.dll
SymbolLookup kernel32 = SymbolLookup.libraryLookup("kernel32", global());

// Get system info function
MemorySegment getSysInfoFunc = kernel32.lookup("GetSystemInfo").orElseThrow();

// System info struct layout
StructLayout sysInfoLayout = MemoryLayout.structLayout(
    ValueLayout.JAVA_INT.withName("wProcessorArchitecture"),
    ValueLayout.JAVA_INT.withName("wReserved"),
    ValueLayout.JAVA_INT.withName("dwPageSize"),
    ValueLayout.ADDRESS.withName("lpMinimumApplicationAddress"),
    ValueLayout.ADDRESS.withName("lpMaximumApplicationAddress"),
    ValueLayout.JAVA_LONG.withName("dwActiveProcessorMask"),
    ValueLayout.JAVA_INT.withName("dwNumberOfProcessors"),
    ValueLayout.JAVA_INT.withName("dwProcessorType"),
    ValueLayout.JAVA_INT.withName("dwAllocationGranularity"),
    ValueLayout.JAVA_SHORT.withName("wProcessorLevel"),
    ValueLayout.JAVA_SHORT.withName("wProcessorRevision")
);

// Function descriptor (void function with pointer to struct)
FunctionDescriptor getSysInfoDesc =
FunctionDescriptor.ofVoid(ValueLayout.ADDRESS);

// Method handle
MethodHandle getSysInfoHandle = linker.downcallHandle(getSysInfoFunc,
getSysInfoDesc);

// Get system info
try (MemorySession session = MemorySession.openConfined()) {
    MemorySegment sysInfo = session.allocate(sysInfoLayout);
    getSysInfoHandle.invoke(sysInfo);
```

```
        // Access struct fields
        int pageSize = sysInfo.get(ValueLayout.JAVA_INT,
            sysInfoLayout.byteOffset(PathElement.groupElement("dwPageSize")));
        int processors = sysInfo.get(ValueLayout.JAVA_INT,

    sysInfoLayout.byteOffset(PathElement.groupElement("dwNumberOfProcessors")));

        System.out.println("Page size: " + pageSize);
        System.out.println("Number of processors: " + processors);
    }
```

> **Common Pitfall**: Memory segments are only valid within the lifetime of their memory session. Using segments after their session is closed results in runtime exceptions.

> **Certification Note**: Understand the basic concepts of the Foreign Function & Memory API. Know how to allocate native memory, access it safely, call native functions, and create callbacks. Understand the role of memory sessions in resource management.

Structured Concurrency

Structured Concurrency is a Java 21 preview feature that simplifies multithreaded programming by ensuring that tasks executed in different threads have a well-defined lifecycle tied to a controlling scope.

Basic Concepts

```
// Structured task scope
try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
    // Fork tasks
    Future<String> user = scope.fork(() -> fetchUser(userId));
    Future<List<Order>> orders = scope.fork(() -> fetchOrders(userId));

    // Wait for all tasks to complete
    scope.join();

    // Check for exceptions
    scope.throwIfFailed(e -> new RuntimeException("Task failed", e));

    // Use results
    processUserOrders(user.resultNow(), orders.resultNow());
}  // All subtasks complete before exiting the block
```

Different Task Scope Types

```
// ShutdownOnFailure - cancels remaining tasks if any task fails
try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
    Future<String> task1 = scope.fork(() -> criticalOperation());
    Future<Integer> task2 = scope.fork(() -> anotherOperation());
```

```
        scope.join();           // Wait for all tasks
        scope.throwIfFailed();  // Throw if any task failed

        String result1 = task1.resultNow();
        Integer result2 = task2.resultNow();
    }

    // ShutdownOnSuccess - cancels remaining tasks when any task succeeds
    try (var scope = new StructuredTaskScope.ShutdownOnSuccess<String>()) {
        scope.fork(() -> findFirstMatch("source1"));
        scope.fork(() -> findFirstMatch("source2"));
        scope.fork(() -> findFirstMatch("source3"));

        // Get the first successful result
        String result = scope.join().result();

        System.out.println("First success: " + result);
    }

    // Custom policy
    class CustomTaskScope<T> extends StructuredTaskScope<T> {
        private List<T> successResults = new ArrayList<>();
        private List<Throwable> failures = new ArrayList<>();

        @Override
        protected void handleComplete(Future<T> future) {
            try {
                successResults.add(future.resultNow());
            } catch (Exception e) {
                failures.add(e);
                // Custom policy: shut down after 3 failures
                if (failures.size() >= 3) {
                    shutdown();
                }
            }
        }

        public List<T> successResults() {
            return successResults;
        }

        public List<Throwable> failures() {
            return failures;
        }
    }

    try (var scope = new CustomTaskScope<String>()) {
        // Fork multiple tasks
        for (int i = 0; i < 10; i++) {
            final int id = i;
            scope.fork(() -> processItem(id));
        }

        scope.join();
```

```
    System.out.println("Successful results: " + scope.successResults());
    System.out.println("Failures: " + scope.failures());
}
```

Error Handling Patterns

```
// Collect exceptions and handle them together
try (var scope = new StructuredTaskScope<Object>() {
    private final List<Throwable> exceptions = new ArrayList<>();

    @Override
    protected void handleComplete(Future<Object> future) {
        try {
            future.resultNow();  // Will throw if failed
        } catch (Exception e) {
            exceptions.add(e);
        }
    }

    public List<Throwable> exceptions() {
        return exceptions;
    }
}) {
    // Fork tasks
    scope.fork(() -> task1());
    scope.fork(() -> task2());
    scope.fork(() -> task3());

    scope.join();

    if (!scope.exceptions().isEmpty()) {
        // Log all exceptions
        for (Throwable e : scope.exceptions()) {
            logger.error("Task failed", e);
        }
        throw new AggregateException("Multiple tasks failed", scope.exceptions());
    }
}

// Handle subtask cancellation
try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
    Future<String> task = scope.fork(() -> {
        try {
            return slowOperation();
        } catch (InterruptedException e) {
            // Handle cancellation gracefully
            cleanup();
            throw e;  // Re-throw to propagate the cancellation
        }
    });
```

```
    // Another task - if this fails, the previous task will be cancelled
    Future<Integer> result = scope.fork(() -> computeResult());

    scope.join();
    scope.throwIfFailed();
}
```

Composing Scopes

```
// Nested task scopes
Result performOperation() throws Exception {
    try (var outerScope = new StructuredTaskScope.ShutdownOnFailure()) {
        // First set of parallel tasks
        Future<Data> data = outerScope.fork(() -> fetchData());
        Future<Config> config = outerScope.fork(() -> fetchConfig());

        outerScope.join();
        outerScope.throwIfFailed();

        // Process initial results and fork more tasks
        Data dataResult = data.resultNow();
        Config configResult = config.resultNow();

        try (var innerScope = new StructuredTaskScope.ShutdownOnFailure()) {
            Future<ProcessedData> processed = innerScope.fork(
                () -> processData(dataResult, configResult));
            Future<Metadata> metadata = innerScope.fork(
                () -> generateMetadata(dataResult));

            innerScope.join();
            innerScope.throwIfFailed();

            return new Result(processed.resultNow(), metadata.resultNow());
        }
    }
}
```

Integration with Existing Code

```
// Using with ExecutorService
ExecutorService executor = Executors.newFixedThreadPool(4);

try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
    // Tasks using the executor
    Future<String> task1 = scope.fork(() -> {
        return executor.submit(() -> computeValue()).get();
    });
```

```
    Future<Integer> task2 = scope.fork(() -> {
        return executor.submit(() -> countItems()).get();
    });

    scope.join();
    scope.throwIfFailed();

    // Process results
    processResults(task1.resultNow(), task2.resultNow());
} finally {
    executor.shutdown();
}

// Using with CompletableFuture
try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
    // Adapt CompletableFuture to StructuredTaskScope
    Future<String> task = scope.fork(() -> {
        CompletableFuture<String> future = asyncOperation();
        return future.get();  // Block until the CompletableFuture completes
    });

    scope.join();
    scope.throwIfFailed();

    String result = task.resultNow();
}
```

Timeouts and Cancellation

```
// Setting a timeout for all tasks
try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
    Future<String> task1 = scope.fork(() -> slowOperation1());
    Future<String> task2 = scope.fork(() -> slowOperation2());

    // Wait for all tasks to complete, but no longer than 5 seconds
    try {
        scope.joinUntil(Instant.now().plusSeconds(5));
    } catch (TimeoutException e) {
        System.out.println("Operation timed out, cancelling tasks");
        // Tasks are automatically cancelled when the scope is closed
    }

    // Check if tasks completed successfully
    if (task1.state() == Future.State.SUCCESS &&
        task2.state() == Future.State.SUCCESS) {
        return new Result(task1.resultNow(), task2.resultNow());
    } else {
        return new TimeoutResult();
    }
}
```

```
// Manually cancelling tasks
try (var scope = new StructuredTaskScope<String>()) {
    List<Future<String>> tasks = new ArrayList<>();

    // Fork multiple tasks
    for (int i = 0; i < 10; i++) {
        tasks.add(scope.fork(() -> processItem(i)));
    }

    // Wait for some condition
    while (!shouldStop()) {
        Thread.sleep(100);

        // Check if any task completed with a specific result
        for (Future<String> task : tasks) {
            if (task.state() == Future.State.SUCCESS &&
                isTargetResult(task.resultNow())) {
                // Found what we're looking for, cancel remaining work
                scope.shutdown();
                return task.resultNow();
            }
        }
    }

    scope.join();
    // Process available results...
}
```

Real-world Examples

```
// Web service requests with fallback
String fetchDataWithFallback() throws Exception {
    try (var scope = new StructuredTaskScope.ShutdownOnSuccess<String>()) {
        // Try multiple services in parallel
        scope.fork(() -> primaryService.fetchData());
        scope.fork(() -> backupService1.fetchData());
        scope.fork(() -> backupService2.fetchData());

        // Get first successful result
        try {
            return scope.joinUntil(Instant.now().plusSeconds(5)).result();
        } catch (TimeoutException e) {
            throw new ServiceException("All services timed out");
        }
    }
}

// Coordinated database operations
void processTransaction(Order order) throws Exception {
    try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
        // Parallel database operations
```

```
        Future<Customer> customer = scope.fork(() ->
customerDao.find(order.getCustomerId()));
        Future<List<Product>> products = scope.fork(() ->
            productDao.findAllById(order.getProductIds()));
        Future<PaymentStatus> payment = scope.fork(() ->
            paymentService.processPayment(order.getPaymentDetails()));

        scope.join();
        scope.throwIfFailed(e -> new TransactionException("Transaction failed",
e));

        // All operations succeeded, complete the transaction
        Customer c = customer.resultNow();
        List<Product> p = products.resultNow();
        PaymentStatus status = payment.resultNow();

        if (status.isSuccessful()) {
            orderDao.save(new CompletedOrder(order, c, p, status));
            notificationService.sendConfirmation(c.getEmail(), order);
        } else {
            throw new PaymentFailedException(status.getReason());
        }
    }
}
```

> **Common Pitfall**: Tasks must not hold locks or resources when the scope is closed, as this could lead to deadlocks or resource leaks. Always ensure proper cleanup in task code, especially when handling cancellation.

> **Certification Note**: Understand the concept of structured concurrency and how it enforces a parent-child relationship between tasks. Know the different types of task scopes and their behavior regarding success, failure, and cancellation. Be familiar with how to integrate structured concurrency with existing concurrency constructs.

Concurrency and Multithreading

Thread Creation and Lifecycle

Java supports multithreading, allowing multiple threads of execution to run concurrently within a single program.

Creating Threads

```
// Method 1: Extending Thread class
class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println("Thread running: " + Thread.currentThread().getName());
    }
```

```
}

// Usage
MyThread thread1 = new MyThread();
thread1.start();  // Creates and starts the thread

// Method 2: Implementing Runnable interface (preferred)
class MyRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println("Runnable running: " +
Thread.currentThread().getName());
    }
}

// Usage
Thread thread2 = new Thread(new MyRunnable());
thread2.start();  // Creates and starts the thread

// Method 3: Using lambda expression (Java 8+)
Thread thread3 = new Thread(() -> {
    System.out.println("Lambda running: " + Thread.currentThread().getName());
});
thread3.start();

// Method 4: Using virtual threads (Java 21+)
Thread vThread = Thread.startVirtualThread(() -> {
    System.out.println("Virtual thread running");
});
```

Thread Lifecycle

A thread can be in one of these states:

1. **NEW**: Created but not started
2. **RUNNABLE**: Executing or ready to execute
3. **BLOCKED**: Waiting for a monitor lock
4. **WAITING**: Waiting indefinitely for another thread
5. **TIMED_WAITING**: Waiting for a specified time
6. **TERMINATED**: Completed execution

```
Thread thread = new Thread(() -> {
    try {
        Thread.sleep(2000);  // TIMED_WAITING state
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
});

System.out.println("State after creation: " + thread.getState());  // NEW
```

```
thread.start();
System.out.println("State after start: " + thread.getState());     // RUNNABLE

Thread.sleep(100);  // Give time for the thread to start sleeping
System.out.println("State during sleep: " + thread.getState());    //
TIMED_WAITING

thread.join();  // Wait for thread to complete
System.out.println("State after completion: " + thread.getState()); // TERMINATED
```

Thread Methods and Properties

```
// Creating a thread with name
Thread thread = new Thread(() -> {
    // Thread body
}, "WorkerThread");

// Setting thread properties
thread.setDaemon(true);  // Daemon threads don't prevent JVM exit
thread.setPriority(Thread.MAX_PRIORITY);  // 10 (highest)
// Thread.NORM_PRIORITY = 5, Thread.MIN_PRIORITY = 1

// Starting the thread
thread.start();

// Getting thread information
String name = thread.getName();
int priority = thread.getPriority();
boolean isDaemon = thread.isDaemon();
Thread.State state = thread.getState();
long id = thread.getId();

// Joining threads (waiting for completion)
try {
    thread.join();  // Wait indefinitely
    thread.join(1000);  // Wait up to 1 second
} catch (InterruptedException e) {
    Thread.currentThread().interrupt();
}

// Checking if thread is alive
boolean isAlive = thread.isAlive();

// Current thread reference
Thread currentThread = Thread.currentThread();

// Sleeping
try {
    Thread.sleep(1000);  // Current thread sleeps for 1 second
} catch (InterruptedException e) {
    Thread.currentThread().interrupt();
```

```
}

// Yielding
Thread.yield();  // Hint to scheduler to let other threads run
```

Interrupting Threads

```
// Thread that handles interruption
Thread thread = new Thread(() -> {
    try {
        while (!Thread.currentThread().isInterrupted()) {
            // Do work
            System.out.println("Working...");
            Thread.sleep(1000);  // Interruptible operation
        }
    } catch (InterruptedException e) {
        // Restore interrupt status
        Thread.currentThread().interrupt();
        System.out.println("Thread was interrupted");
    } finally {
        System.out.println("Cleanup resources");
    }
});

thread.start();
Thread.sleep(3000);  // Let it work for 3 seconds
thread.interrupt();  // Request interruption
```

Daemon Threads

```
// Daemon threads terminate when all non-daemon threads exit
Thread daemonThread = new Thread(() -> {
    while (true) {
        try {
            System.out.println("Daemon working...");
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            break;
        }
    }
});

daemonThread.setDaemon(true);
daemonThread.start();

// Main thread sleeps and exits
Thread.sleep(5000);
System.out.println("Main thread exiting");
// Daemon thread will be terminated when JVM exits
```

Thread Groups

```
// Creating a thread group
ThreadGroup group = new ThreadGroup("WorkerGroup");

// Creating threads in the group
Thread t1 = new Thread(group, () -> { /* work */ }, "Worker1");
Thread t2 = new Thread(group, () -> { /* work */ }, "Worker2");

t1.start();
t2.start();

// Getting active threads in group
int activeCount = group.activeCount();
Thread[] threads = new Thread[activeCount];
group.enumerate(threads);

// Interrupting all threads in group
group.interrupt();
```

ThreadLocal Variables

```
// ThreadLocal provides thread-isolated variables
ThreadLocal<Integer> threadId = ThreadLocal.withInitial(() -> 0);

// Using ThreadLocal
class Worker implements Runnable {
    private static int nextId = 0;

    @Override
    public void run() {
        // Each thread gets its own copy
        threadId.set(nextId++);

        System.out.println("Thread " + Thread.currentThread().getName() +
                           " has id " + threadId.get());

        // Clean up ThreadLocal to avoid memory leaks
        threadId.remove();
    }
}

// Using in multiple threads
Thread t1 = new Thread(new Worker(), "T1");
Thread t2 = new Thread(new Worker(), "T2");
t1.start();
t2.start();
```

> **Common Pitfall**: Failing to handle `InterruptedException` properly can make threads unresponsive to interruption. Always restore the interrupt status when catching `InterruptedException`.

> **Certification Note**: Understand thread states and transitions between them. Know how to create, start, and manage threads. Be familiar with thread priorities and daemon threads. Understand thread interruption and how to handle it correctly.

Synchronization and Thread Safety

Thread safety is about ensuring that code behaves correctly when accessed by multiple threads simultaneously. Synchronization helps achieve thread safety by controlling access to shared resources.

Synchronized Methods

```
class Counter {
    private int count = 0;

    // Synchronized instance method
    public synchronized void increment() {
        count++;
    }

    // Synchronized static method
    public static synchronized void staticMethod() {
        // Synchronized on Counter.class
    }

    public int getCount() {
        return count;
    }
}

// Using synchronized methods
Counter counter = new Counter();
Thread t1 = new Thread(() -> {
    for (int i = 0; i < 10000; i++) {
        counter.increment();
    }
});

Thread t2 = new Thread(() -> {
    for (int i = 0; i < 10000; i++) {
        counter.increment();
    }
});

t1.start();
t2.start();
t1.join();
t2.join();

System.out.println("Final count: " + counter.getCount());  // 20000
```

Synchronized Blocks

```
class Buffer {
    private final List<String> data = new ArrayList<>();
    private final Object lock = new Object();  // Dedicated lock object

    public void add(String item) {
        // Synchronized on the instance itself
        synchronized (this) {
            data.add(item);
        }
    }

    public void process() {
        // Synchronized on a dedicated lock object
        synchronized (lock) {
            // Critical section
            for (String item : data) {
                // Process each item
            }
        }
    }

    public void staticOperation() {
        // Synchronized on class object
        synchronized (Buffer.class) {
            // Critical static section
        }
    }
}
```

Volatile Keyword

The `volatile` keyword ensures that a variable is always read from and written to main memory, making it visible to all threads.

```
class StatusChecker implements Runnable {
    // Without volatile, this might not be seen by other threads
    private volatile boolean running = true;

    public void stop() {
        running = false;
    }

    @Override
    public void run() {
        while (running) {
            // Check status
```

```
        }
        System.out.println("Stopped");
    }
}

// Usage
StatusChecker checker = new StatusChecker();
Thread thread = new Thread(checker);
thread.start();

// Later, from another thread
checker.stop();  // Guaranteed to be visible to the running thread
```

Key points about `volatile`:

- Guarantees visibility of changes to variables across threads
- Does not provide atomicity for compound operations
- Prevents reordering of instructions by the compiler or CPU
- Lighter weight than synchronization but less powerful

The Happens-Before Relationship

Java Memory Model defines "happens-before" relationships that guarantee visibility of actions between threads:

```
class HappensBefore {
    private int x = 0;
    private volatile boolean flag = false;

    public void writer() {
        x = 42;         // Write to x
        flag = true;    // Write to flag
    }

    public void reader() {
        if (flag) {     // Read from flag
            // If flag is true, x is guaranteed to be 42
            System.out.println(x);
        }
    }
}
```

Key happens-before guarantees:

- Program order: Each action in a thread happens-before every subsequent action in that thread
- Monitor lock: Releasing a lock happens-before acquiring the same lock
- Volatile: Writing to a volatile field happens-before reading that field
- Thread start: `start()` happens-before any actions in the started thread

- Thread termination: All actions in a thread happen-before another thread detects its termination via `join()`
- Transitivity: If A happens-before B and B happens-before C, then A happens-before C

Atomic Classes

Atomic classes in `java.util.concurrent.atomic` provide non-blocking thread-safe operations:

```
import java.util.concurrent.atomic.*;

class AtomicCounter {
    private AtomicInteger count = new AtomicInteger(0);

    public void increment() {
        count.incrementAndGet();  // Atomic operation
    }

    public int getCount() {
        return count.get();
    }

    public void complexUpdate() {
        // Compare-and-set pattern for more complex updates
        int current;
        do {
            current = count.get();
        } while (!count.compareAndSet(current, current + 2));

        // Or using updateAndGet (Java 8+)
        count.updateAndGet(n -> n + 2);
    }
}

// Usage with multiple threads
AtomicCounter counter = new AtomicCounter();
Thread t1 = new Thread(() -> {
    for (int i = 0; i < 10000; i++) {
        counter.increment();
    }
});

Thread t2 = new Thread(() -> {
    for (int i = 0; i < 10000; i++) {
        counter.increment();
    }
});

t1.start();
t2.start();
t1.join();
t2.join();
```

```
System.out.println("Final count: " + counter.getCount());  // Always 20000
```

Common atomic classes:

- `AtomicInteger`, `AtomicLong`, `AtomicBoolean`
- `AtomicReference<V>` for reference types
- `AtomicIntegerArray`, `AtomicLongArray`, `AtomicReferenceArray<V>`
- `AtomicIntegerFieldUpdater`, `AtomicLongFieldUpdater`, `AtomicReferenceFieldUpdater`
- `DoubleAdder`, `LongAdder` (Java 8+) for high-contention scenarios

Thread Confinement

Thread confinement is a strategy to ensure thread safety by confining data to one thread:

```
// ThreadLocal - each thread has its own copy
ThreadLocal<SimpleDateFormat> dateFormatter = ThreadLocal.withInitial(
    () -> new SimpleDateFormat("yyyy-MM-dd")
);

// Usage in multiple threads
String formatDate(Date date) {
    return dateFormatter.get().format(date);
}

// Stack confinement - local variables are confined to their thread
void processLocal() {
    // Local variables are thread safe
    int counter = 0;
    for (int i = 0; i < 10; i++) {
        counter++;
    }
    // counter is only visible to this thread
}
```

Immutability

Immutable objects are inherently thread-safe:

```
// Immutable class
public final class Point {
    private final int x;
    private final int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
```

```
    public int getX() { return x; }
    public int getY() { return y; }

    // Create new object instead of modifying
    public Point moveBy(int dx, int dy) {
        return new Point(x + dx, y + dy);
    }
}
```

Rules for immutability:

1. No methods that modify state
2. Class is final to prevent overriding
3. All fields are final
4. Proper construction (no leaking this reference)
5. Deep immutability for any contained mutable objects

Locking Strategies

```
// Intrinsic locks (synchronized)
class BankAccount {
    private double balance;

    public synchronized void deposit(double amount) {
        balance += amount;
    }

    public synchronized void withdraw(double amount) {
        if (balance >= amount) {
            balance -= amount;
        }
    }
}

// Explicit locks (ReentrantLock)
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

class BankAccountWithLock {
    private final Lock lock = new ReentrantLock();
    private double balance;

    public void deposit(double amount) {
        lock.lock();
        try {
            balance += amount;
        } finally {
            lock.unlock();  // Always unlock in finally block
        }
    }
```

```
    public void withdraw(double amount) {
        lock.lock();
        try {
            if (balance >= amount) {
                balance -= amount;
            }
        } finally {
            lock.unlock();
        }
    }

    // Trylock pattern
    public boolean tryTransfer(BankAccountWithLock to, double amount, long
timeout) {
        try {
            if (lock.tryLock(timeout, TimeUnit.MILLISECONDS)) {
                try {
                    if (to.lock.tryLock(timeout, TimeUnit.MILLISECONDS)) {
                        try {
                            if (balance >= amount) {
                                balance -= amount;
                                to.balance += amount;
                                return true;
                            }
                            return false;
                        } finally {
                            to.lock.unlock();
                        }
                    }
                } finally {
                    lock.unlock();
                }
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
        return false;
    }
}
```

Lock types:

- `ReentrantLock`: Standard exclusive lock with same semantics as `synchronized`
- `ReadWriteLock`: Allows multiple readers or one writer
- `StampedLock` (Java 8+): Supports optimistic reads, can convert to read/write lock

Memory Consistency Errors

Memory consistency errors occur when different threads have inconsistent views of the same data:

```
class UnsafeCounter {
    private int count = 0;

    // Not synchronized - may lead to lost updates
    public void increment() {
        count++;  // Not atomic: read, increment, write
    }

    public int getCount() {
        return count;
    }
}

// Race condition example
UnsafeCounter counter = new UnsafeCounter();
Thread t1 = new Thread(() -> {
    for (int i = 0; i < 10000; i++) {
        counter.increment();
    }
});

Thread t2 = new Thread(() -> {
    for (int i = 0; i < 10000; i++) {
        counter.increment();
    }
});

t1.start();
t2.start();
t1.join();
t2.join();

// May print less than 20000 due to lost updates
System.out.println("Final count: " + counter.getCount());
```

Common concurrency issues:

- Race conditions: Result depends on thread execution order
- Deadlocks: Two or more threads waiting for each other
- Livelocks: Threads keep changing state but make no progress
- Starvation: A thread is unable to gain access to a shared resource
- Thread interference: Interleaved operations on shared data

Monitor Pattern

The monitor pattern provides mutual exclusion and condition synchronization:

```
// Basic monitor pattern
class MessageQueue {
    private final Queue<String> queue = new LinkedList<>();
```

```
    private final int capacity;

    public MessageQueue(int capacity) {
        this.capacity = capacity;
    }

    // Producer method
    public synchronized void put(String message) throws InterruptedException {
        while (queue.size() == capacity) {
            wait();  // Wait for space
        }
        queue.add(message);
        notifyAll();  // Notify waiting consumers
    }

    // Consumer method
    public synchronized String take() throws InterruptedException {
        while (queue.isEmpty()) {
            wait();  // Wait for messages
        }
        String message = queue.remove();
        notifyAll();  // Notify waiting producers
        return message;
    }
}
```

Key monitor operations:

- `wait()`: Releases lock and waits for notification
- `notify()`: Wakes up one waiting thread
- `notifyAll()`: Wakes up all waiting threads

When using these methods:

- Must hold the lock on the object
- Always call within a loop to recheck condition (to handle spurious wakeups)
- `wait()` throws `InterruptedException`, which must be handled

Deadlock Prevention

```
// Deadlock-prone code
class DeadlockRisk {
    private final Object resourceA = new Object();
    private final Object resourceB = new Object();

    public void methodA() {
        synchronized (resourceA) {
            System.out.println("Thread holds resource A");
            // Potential deadlock if another thread holds B and wants A
            synchronized (resourceB) {
                System.out.println("Thread holds both resources");
```

```
            }
        }
    }

    public void methodB() {
        synchronized (resourceB) {
            System.out.println("Thread holds resource B");
            // Potential deadlock if another thread holds A and wants B
            synchronized (resourceA) {
                System.out.println("Thread holds both resources");
            }
        }
    }
}

// Deadlock prevention: Consistent lock ordering
class DeadlockSafe {
    private final Object resourceA = new Object();
    private final Object resourceB = new Object();

    public void methodA() {
        synchronized (resourceA) {
            synchronized (resourceB) {
                // Safe: consistent lock ordering
            }
        }
    }

    public void methodB() {
        synchronized (resourceA) {  // Same order as methodA
            synchronized (resourceB) {
                // Safe: consistent lock ordering
            }
        }
    }
}

// Using tryLock to avoid deadlock
class TimeoutBasedDeadlockAvoidance {
    private final Lock lockA = new ReentrantLock();
    private final Lock lockB = new ReentrantLock();

    public boolean operation() {
        try {
            if (lockA.tryLock(1, TimeUnit.SECONDS)) {
                try {
                    if (lockB.tryLock(1, TimeUnit.SECONDS)) {
                        try {
                            // Operation with both locks
                            return true;
                        } finally {
                            lockB.unlock();
                        }
                    }
```

```
                } finally {
                    lockA.unlock();
                }
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
        return false;  // Couldn't acquire both locks
    }
}
```

Deadlock prevention strategies:

1. Lock ordering: Always acquire locks in the same global order
2. Lock timeout: Use `tryLock()` with timeout
3. Deadlock detection: Use thread dumps and monitoring tools
4. Lock hierarchy: Document and enforce a lock acquisition hierarchy
5. Avoid nested locks: Reduce the need for multiple locks

> **Common Pitfall**: Forgetting to release locks in `finally` blocks can lead to resource leaks and potential deadlocks.

> **Certification Note**: Understand thread safety mechanisms, including synchronized methods and blocks, volatile variables, and atomic classes. Know about common concurrency issues like race conditions and deadlocks, and how to prevent them.

java.util.concurrent Framework

The `java.util.concurrent` package provides high-level concurrency utilities that make developing thread-safe applications easier.

Executors and Thread Pools

Thread pools manage a collection of worker threads to execute tasks:

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

// Creating different types of thread pools
ExecutorService singleThreadExecutor = Executors.newSingleThreadExecutor();
ExecutorService fixedThreadPool = Executors.newFixedThreadPool(4);
ExecutorService cachedThreadPool = Executors.newCachedThreadPool();
ScheduledExecutorService scheduledPool = Executors.newScheduledThreadPool(2);

// Virtual thread executor (Java 21+)
ExecutorService virtualExecutor = Executors.newVirtualThreadPerTaskExecutor();

// Submitting tasks
Future<String> future = fixedThreadPool.submit(() -> {
    // Task that returns a result
    return "Task completed";
```

```
});

// Executing tasks without return value
fixedThreadPool.execute(() -> {
    System.out.println("Task running in thread: " +
                       Thread.currentThread().getName());
});

// Scheduling tasks
scheduledPool.schedule(() -> {
    System.out.println("Delayed task");
}, 1, TimeUnit.SECONDS);

scheduledPool.scheduleAtFixedRate(() -> {
    System.out.println("Periodic task");
}, 1, 5, TimeUnit.SECONDS);  // Initial delay 1s, then every 5s

scheduledPool.scheduleWithFixedDelay(() -> {
    System.out.println("Fixed delay task");
}, 0, 1, TimeUnit.SECONDS);  // Start immediately, then 1s after completion

// Shutting down executors
fixedThreadPool.shutdown();  // Allows previously submitted tasks to execute
try {
    if (!fixedThreadPool.awaitTermination(1, TimeUnit.MINUTES)) {
        fixedThreadPool.shutdownNow();  // Cancel running tasks
    }
} catch (InterruptedException e) {
    fixedThreadPool.shutdownNow();
    Thread.currentThread().interrupt();
}
```

Always use try-with-resources or ensure proper shutdown:

```
// Executor with try-with-resources (Java 19+)
try (ExecutorService executor = Executors.newFixedThreadPool(4)) {
    // Submit tasks here
}  // Automatically calls shutdown()
```

Future and CompletableFuture

Futures represent the result of asynchronous computations:

```
// Using Future
ExecutorService executor = Executors.newFixedThreadPool(4);
Future<Integer> future = executor.submit(() -> {
    Thread.sleep(1000);
    return 42;
});
```

```
// Check if completed
boolean isDone = future.isDone();
boolean isCancelled = future.isCancelled();

// Get result (blocking)
try {
    Integer result = future.get();  // Blocks until result is available
    Integer resultWithTimeout = future.get(2, TimeUnit.SECONDS);  // With timeout
} catch (InterruptedException e) {
    Thread.currentThread().interrupt();
} catch (ExecutionException e) {
    // Task threw an exception
    Throwable cause = e.getCause();
} catch (TimeoutException e) {
    // Timeout waiting for result
    future.cancel(true);  // Attempt to cancel task
}

// CompletableFuture (Java 8+) - more powerful Future
CompletableFuture<String> cf = CompletableFuture.supplyAsync(() -> {
    // Async computation
    return "Result";
}, executor);

// Transformation and chaining
CompletableFuture<Integer> lengthFuture = cf.thenApply(String::length);
CompletableFuture<Void> printFuture = cf.thenAccept(System.out::println);
CompletableFuture<Void> chainedFuture = cf.thenRun(() -> {
    System.out.println("Completed");
});

// Async versions
CompletableFuture<Integer> asyncLengthFuture = cf.thenApplyAsync(String::length);

// Combining futures
CompletableFuture<String> first = CompletableFuture.supplyAsync(() -> "Hello");
CompletableFuture<String> second = CompletableFuture.supplyAsync(() -> "World");

CompletableFuture<String> combined = first.thenCombine(
    second,
    (r1, r2) -> r1 + " " + r2
);  // "Hello World"

// First completing future
CompletableFuture<String> fastest = CompletableFuture.anyOf(
    CompletableFuture.supplyAsync(() -> slowOperation()),
    CompletableFuture.supplyAsync(() -> fastOperation())
).thenApply(result -> (String) result);

// Waiting for all futures
CompletableFuture<Void> allDone = CompletableFuture.allOf(
    CompletableFuture.runAsync(() -> task1()),
    CompletableFuture.runAsync(() -> task2()),
```

```
    CompletableFuture.runAsync(() -> task3())
);

// Get results from multiple futures
List<CompletableFuture<String>> futures = Arrays.asList(
    CompletableFuture.supplyAsync(() -> "One"),
    CompletableFuture.supplyAsync(() -> "Two"),
    CompletableFuture.supplyAsync(() -> "Three")
);

CompletableFuture<List<String>> allResults = CompletableFuture.allOf(
    futures.toArray(new CompletableFuture[0])
).thenApply(v ->
    futures.stream()
           .map(CompletableFuture::join)  // Safe after allOf
           .collect(Collectors.toList())
);

// Error handling
CompletableFuture<String> handled = CompletableFuture.supplyAsync(() -> {
    if (Math.random() < 0.5) throw new RuntimeException("Error");
    return "Success";
}).exceptionally(ex -> {
    System.err.println("Exception: " + ex.getMessage());
    return "Default value";
}).handle((result, ex) -> {
    if (ex != null) return "Handled: " + ex.getMessage();
    return "Result: " + result;
});

// Timeout (Java 9+)
CompletableFuture<String> withTimeout = CompletableFuture.supplyAsync(() -> {
    try {
        Thread.sleep(2000);
        return "Completed";
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
        return "Interrupted";
    }
}).orTimeout(1, TimeUnit.SECONDS)
  .exceptionally(ex -> "Timeout");
```

Concurrent Collections

Thread-safe collections designed for concurrent access:

```
// ConcurrentHashMap - high concurrency, segments
ConcurrentMap<String, Integer> concurrentMap = new ConcurrentHashMap<>();
concurrentMap.put("one", 1);
concurrentMap.put("two", 2);
```

```
// Atomic operations
concurrentMap.putIfAbsent("three", 3);  // Only put if key doesn't exist
concurrentMap.replace("two", 2, 22);    // Replace only if current value matches

// Bulk operations
concurrentMap.forEach((k, v) -> System.out.println(k + ": " + v));
int sum = concurrentMap.reduceValues(1, Integer::sum);

// ConcurrentLinkedQueue - non-blocking queue
Queue<String> concurrentQueue = new ConcurrentLinkedQueue<>();
concurrentQueue.offer("one");
concurrentQueue.poll();  // Returns and removes head, or null if empty

// BlockingQueue - blocks if empty/full
BlockingQueue<String> blockingQueue = new LinkedBlockingQueue<>(10);  // Capacity
10
blockingQueue.put("item");  // Blocks if queue is full
String item = blockingQueue.take();  // Blocks if queue is empty

// Other blocking queue implementations:
BlockingQueue<Integer> arrayBlockingQueue = new ArrayBlockingQueue<>(100);
BlockingQueue<Integer> delayQueue = new DelayQueue<>();  // Elements implement
Delayed
BlockingQueue<Integer> priorityBlockingQueue = new PriorityBlockingQueue<>();  //
Natural ordering

// Deques
Deque<String> concurrentDeque = new ConcurrentLinkedDeque<>();
BlockingDeque<String> blockingDeque = new LinkedBlockingDeque<>();

// CopyOnWrite collections - thread-safe, but expensive for modifications
List<String> copyOnWriteList = new CopyOnWriteArrayList<>();
Set<String> copyOnWriteSet = new CopyOnWriteArraySet<>();

// Interfaces with CAS operation for concurrent sets
NavigableSet<String> concurrentSkipListSet = new ConcurrentSkipListSet<>();
```

Synchronizers

High-level synchronization tools for coordinating threads:

```
// CountDownLatch - one-time barrier
CountDownLatch startSignal = new CountDownLatch(1);  // Count of 1
CountDownLatch doneSignal = new CountDownLatch(10);  // Count of 10 (for 10
workers)

// Worker thread
Runnable worker = () -> {
    try {
        startSignal.await();  // Wait for start signal
        doWork();
```

```
            doneSignal.countDown();  // Signal completion
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    };

    // Control thread
    for (int i = 0; i < 10; i++) {
        new Thread(worker).start();
    }

    prepareWork();
    startSignal.countDown();  // Start all workers
    doneSignal.await();       // Wait for all workers to finish
    System.out.println("All workers completed");

    // CyclicBarrier - reusable barrier
    CyclicBarrier barrier = new CyclicBarrier(3, () -> {
        // This runs when barrier is tripped
        System.out.println("All threads reached the barrier");
    });

    Runnable barrierAction = () -> {
        try {
            System.out.println(Thread.currentThread().getName() + " waiting at
    barrier");
            barrier.await();  // Wait for all threads to reach this point
            System.out.println(Thread.currentThread().getName() + " continuing");
        } catch (InterruptedException | BrokenBarrierException e) {
            Thread.currentThread().interrupt();
        }
    };

    for (int i = 0; i < 3; i++) {
        new Thread(barrierAction).start();
    }

    // Phaser - dynamic number of parties
    Phaser phaser = new Phaser(3);  // 3 parties

    Runnable phaserAction = () -> {
        phaser.register();  // Add this thread as a party

        try {
            System.out.println("Phase 1");
            phaser.arriveAndAwaitAdvance();  // Wait for phase 1 completion

            System.out.println("Phase 2");
            phaser.arriveAndAwaitAdvance();  // Wait for phase 2 completion

            System.out.println("Phase 3");
            phaser.arriveAndDeregister();    // Complete and deregister
        } catch (Exception e) {
            phaser.forceTermination();
```

```
    }
};

for (int i = 0; i < 3; i++) {
    new Thread(phaserAction).start();
}

// Semaphore - controls access to a resource
Semaphore semaphore = new Semaphore(3);  // 3 permits

Runnable semaphoreAction = () -> {
    try {
        semaphore.acquire();  // Get a permit (blocks if none available)
        try {
            accessResource();  // Only 3 threads can execute this at once
        } finally {
            semaphore.release();  // Return the permit
        }
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
};

// Exchanger - exchanges objects between two threads
Exchanger<List<Integer>> exchanger = new Exchanger<>();

Thread producerThread = new Thread(() -> {
    List<Integer> buffer = new ArrayList<>();
    try {
        while (!Thread.interrupted()) {
            // Fill buffer
            for (int i = 0; i < 10; i++) {
                buffer.add(i);
            }

            // Exchange full buffer for empty one
            buffer = exchanger.exchange(buffer);
        }
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
});

Thread consumerThread = new Thread(() -> {
    List<Integer> buffer = new ArrayList<>();
    try {
        while (!Thread.interrupted()) {
            // Get full buffer from producer
            buffer = exchanger.exchange(buffer);

            // Process buffer
            for (Integer i : buffer) {
                process(i);
            }
```

```
            buffer.clear();  // Empty the buffer
        }
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
});

producerThread.start();
consumerThread.start();
```

Locks and Conditions

More flexible locking mechanisms than `synchronized`:

```
import java.util.concurrent.locks.*;

// ReentrantLock - basic lock with same behavior as synchronized
ReentrantLock lock = new ReentrantLock();
lock.lock();  // Acquire lock
try {
    // Critical section
} finally {
    lock.unlock();  // Always release in finally block
}

// Condition variables
ReentrantLock conditionLock = new ReentrantLock();
Condition notFull = conditionLock.newCondition();
Condition notEmpty = conditionLock.newCondition();

// Producer-consumer pattern
void put(T item) throws InterruptedException {
    conditionLock.lock();
    try {
        while (isFull()) {
            notFull.await();  // Wait for space
        }
        add(item);
        notEmpty.signal();  // Notify consumers
    } finally {
        conditionLock.unlock();
    }
}

T take() throws InterruptedException {
    conditionLock.lock();
    try {
        while (isEmpty()) {
            notEmpty.await();  // Wait for items
        }
```

```
            T item = remove();
            notFull.signal();  // Notify producers
            return item;
        } finally {
            conditionLock.unlock();
        }
    }

    // ReadWriteLock - allows multiple readers or one writer
    ReadWriteLock rwLock = new ReentrantReadWriteLock();
    Lock readLock = rwLock.readLock();
    Lock writeLock = rwLock.writeLock();

    // Reading (multiple threads can hold read lock)
    readLock.lock();
    try {
        // Read shared data
    } finally {
        readLock.unlock();
    }

    // Writing (exclusive access)
    writeLock.lock();
    try {
        // Modify shared data
    } finally {
        writeLock.unlock();
    }

    // StampedLock (Java 8+) - optimistic reading
    StampedLock stampedLock = new StampedLock();

    // Optimistic read
    long stamp = stampedLock.tryOptimisticRead();
    // Read values...
    if (!stampedLock.validate(stamp)) {
        // Someone wrote data, fallback to read lock
        stamp = stampedLock.readLock();
        try {
            // Re-read values
        } finally {
            stampedLock.unlockRead(stamp);
        }
    }

    // Write lock
    long writeStamp = stampedLock.writeLock();
    try {
        // Write values
    } finally {
        stampedLock.unlockWrite(writeStamp);
    }

    // Converting a read lock to a write lock
```

```
long convertStamp = stampedLock.readLock();
try {
    // Read values

    // Try to convert to write lock
    long writeStamp = stampedLock.tryConvertToWriteLock(convertStamp);
    if (writeStamp != 0L) {
        // Successfully converted
        convertStamp = writeStamp;
        // Write values
    } else {
        // Unable to convert, acquire write lock explicitly
        stampedLock.unlockRead(convertStamp);
        convertStamp = stampedLock.writeLock();
        // Write values
    }
} finally {
    stampedLock.unlock(convertStamp);
}
```

Fork/Join Framework

The Fork/Join framework is designed for parallel recursive divide-and-conquer tasks:

```
import java.util.concurrent.*;
import java.util.concurrent.RecursiveTask;

// Task that returns a result
class SumTask extends RecursiveTask<Long> {
    private final long[] array;
    private final int start;
    private final int end;
    private static final int THRESHOLD = 10000;

    public SumTask(long[] array, int start, int end) {
        this.array = array;
        this.start = start;
        this.end = end;
    }

    @Override
    protected Long compute() {
        int length = end - start;
        if (length <= THRESHOLD) {
            // Sequential computation for small enough tasks
            long sum = 0;
            for (int i = start; i < end; i++) {
                sum += array[i];
            }
            return sum;
        } else {
```

```
            // Split task into subtasks
            int mid = start + length / 2;
            SumTask leftTask = new SumTask(array, start, mid);
            SumTask rightTask = new SumTask(array, mid, end);

            // Fork right task asynchronously
            rightTask.fork();

            // Compute left task in current thread
            Long leftResult = leftTask.compute();

            // Join right task (wait for result)
            Long rightResult = rightTask.join();

            // Combine results
            return leftResult + rightResult;
        }
    }
}

// Task with no result
class SortTask extends RecursiveAction {
    private final int[] array;
    private final int start;
    private final int end;
    private static final int THRESHOLD = 100;

    public SortTask(int[] array, int start, int end) {
        this.array = array;
        this.start = start;
        this.end = end;
    }

    @Override
    protected void compute() {
        if (end - start <= THRESHOLD) {
            Arrays.sort(array, start, end);
        } else {
            int mid = start + (end - start) / 2;

            // Create and fork subtasks
            SortTask leftTask = new SortTask(array, start, mid);
            SortTask rightTask = new SortTask(array, mid, end);

            invokeAll(leftTask, rightTask);

            // Merge results (already sorted in-place)
            merge(array, start, mid, end);
        }
    }

    private void merge(int[] array, int start, int mid, int end) {
        // Merge implementation
    }
```

```
}

// Using the Fork/Join tasks
ForkJoinPool pool = ForkJoinPool.commonPool();  // Use common pool

// Example summing a large array
long[] numbers = new long[100_000_000];
// Fill array with values...

SumTask task = new SumTask(numbers, 0, numbers.length);
long sum = pool.invoke(task);
System.out.println("Sum: " + sum);

// Example sorting a large array
int[] arrayToSort = new int[1_000_000];
// Fill array with values...

SortTask sortTask = new SortTask(arrayToSort, 0, arrayToSort.length);
pool.invoke(sortTask);
```

Key points about Fork/Join:

- Use work-stealing for load balancing (idle threads steal tasks from busy threads)
- Best for CPU-bound tasks that can be recursively decomposed
- Avoid synchronization, blocking, or I/O in tasks
- Typically use the common pool unless you have specific requirements
- Follow the pattern: split the work, process the subtasks, combine the results
- Join operation can only be called after fork

CompletionService

Manages completion of asynchronous tasks:

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

ExecutorService executor = Executors.newFixedThreadPool(4);
CompletionService<Integer> completionService = new ExecutorCompletionService<>
(executor);

// Submit tasks
for (int i = 0; i < 10; i++) {
    final int id = i;
    completionService.submit(() -> {
        // Simulating tasks taking different times
        Thread.sleep(new Random().nextInt(1000));
        return id;
    });
}

// Process results in completion order
try {
```

```
    for (int i = 0; i < 10; i++) {
        Future<Integer> future = completionService.take();  // Blocks until a task
completes
        Integer result = future.get();  // Will not block since task is complete
        System.out.println("Completed: " + result);
    }
} catch (InterruptedException | ExecutionException e) {
    Thread.currentThread().interrupt();
}

// Always shut down the executor
executor.shutdown();
```

ThreadLocalRandom

A random number generator isolated to the current thread:

```
// Instead of shared Random (which can be a contention point)
Random random = new Random();  // Potentially shared across threads
int value = random.nextInt(100);

// Use ThreadLocalRandom
int value = ThreadLocalRandom.current().nextInt(100);
double doubleValue = ThreadLocalRandom.current().nextDouble();
long longValue = ThreadLocalRandom.current().nextLong(10, 100);  // Range: [10,
100)
```

> **Common Pitfall**: Using thread pools without proper shutdown can lead to resource leaks and prevent the JVM from exiting. Always shut down executors when they're no longer needed.

> **Certification Note**: Understand the core components of the `java.util.concurrent` framework, including executors, futures, concurrent collections, and synchronizers. Know how to use these components to solve common concurrent programming challenges.

CompletableFuture and Asynchronous Programming

Advanced CompletableFuture Patterns

```
// Composing dependent async operations
CompletableFuture<User> userFuture = getUserAsync(userId);

CompletableFuture<List<Order>> ordersFuture = userFuture
    .thenCompose(user -> getOrdersAsync(user.getId()));

// Parallel operations
CompletableFuture<User> userFuture = getUserAsync(userId);
CompletableFuture<List<Product>> recommendationsFuture =
getRecommendationsAsync(userId);
```

```
CompletableFuture<UserProfile> profileFuture = userFuture
    .thenCombine(recommendationsFuture, (user, recommendations) -> {
        return new UserProfile(user, recommendations);
    });

// Processing multiple futures
List<CompletableFuture<Item>> futures = itemIds.stream()
    .map(this::getItemAsync)
    .collect(Collectors.toList());

// Combined future that completes when all complete
CompletableFuture<Void> allFuture = CompletableFuture.allOf(
    futures.toArray(new CompletableFuture[0])
);

// Transform to get results
CompletableFuture<List<Item>> itemsFuture = allFuture.thenApply(v ->
    futures.stream()
        .map(CompletableFuture::join)  // Safe after allOf
        .collect(Collectors.toList())
);

// First completing future
CompletableFuture<String> future1 = getDataFromSource1();
CompletableFuture<String> future2 = getDataFromSource2();

CompletableFuture<String> firstFuture = future1
    .applyToEither(future2, result -> "Result: " + result);

// Async exception handling
CompletableFuture<Data> dataFuture = CompletableFuture
    .supplyAsync(() -> {
        if (Math.random() < 0.5) {
            throw new RuntimeException("Fetch failed");
        }
        return new Data("success");
    })
    .exceptionally(ex -> {
        log.error("Error fetching data", ex);
        return new Data("default");  // Fallback value
    })
    .handle((data, ex) -> {
        if (ex != null) {
            log.error("Error in pipeline", ex);
            return new Data("handled");
        }
        return data;
    });

// Timeouts (Java 9+)
CompletableFuture<String> futureWithTimeout = getData()
    .orTimeout(1, TimeUnit.SECONDS)
    .exceptionally(ex -> {
        if (ex instanceof TimeoutException) {
```

```
            return "Timeout occurred";
        }
        return "Other error: " + ex.getMessage();
    });

// Completing with a default after timeout (Java 9+)
CompletableFuture<String> futureWithDefault = getData()
    .completeOnTimeout("Default value", 1, TimeUnit.SECONDS);

// Cancellation
CompletableFuture<String> future = longRunningOperation();
boolean cancelled = future.cancel(true);

// Manual completion
CompletableFuture<String> manualFuture = new CompletableFuture<>();
// Later...
manualFuture.complete("Result");
// Or if an error occurs
manualFuture.completeExceptionally(new RuntimeException("Failed"));

// Running a task after others complete (regardless of success/failure)
CompletableFuture<Void> future = CompletableFuture.allOf(future1, future2)
    .whenComplete((result, ex) -> {
        if (ex != null) {
            log.error("At least one operation failed", ex);
        } else {
            log.info("All operations completed successfully");
        }
    })
    .thenRun(() -> {
        cleanupResources();
    });
```

Asynchronous HTTP Client (Java 11+)

```
import java.net.http.*;
import java.net.URI;
import java.time.Duration;

// Create HTTP client
HttpClient client = HttpClient.newBuilder()
    .connectTimeout(Duration.ofSeconds(10))
    .build();

// Synchronous request
HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create("https://api.example.com/data"))
    .header("Accept", "application/json")
    .GET()
    .build();
```

```
HttpResponse<String> response = client.send(request,
HttpResponse.BodyHandlers.ofString());
System.out.println("Status: " + response.statusCode());
System.out.println("Body: " + response.body());

// Asynchronous request
CompletableFuture<HttpResponse<String>> futureResponse = client.sendAsync(
    request, HttpResponse.BodyHandlers.ofString());

futureResponse
    .thenApply(HttpResponse::body)
    .thenAccept(System.out::println)
    .exceptionally(ex -> {
        System.err.println("Error: " + ex.getMessage());
        return null;
    });

// POST request with JSON body
HttpRequest postRequest = HttpRequest.newBuilder()
    .uri(URI.create("https://api.example.com/create"))
    .header("Content-Type", "application/json")
    .POST(HttpRequest.BodyPublishers.ofString("{\"name\":\"John\",\"age\":30}"))
    .build();

client.sendAsync(postRequest, HttpResponse.BodyHandlers.ofString())
    .thenApply(response -> {
        if (response.statusCode() == 201) {
            return response.body();
        } else {
            throw new RuntimeException("Failed: " + response.statusCode());
        }
    })
    .thenAccept(body -> System.out.println("Created: " + body))
    .exceptionally(ex -> {
        System.err.println("Error: " + ex.getMessage());
        return null;
    });
```

Custom Thread Factory

```
class CustomThreadFactory implements ThreadFactory {
    private final String namePrefix;
    private final AtomicInteger counter = new AtomicInteger(1);

    public CustomThreadFactory(String namePrefix) {
        this.namePrefix = namePrefix;
    }

    @Override
    public Thread newThread(Runnable r) {
        Thread thread = new Thread(r, namePrefix + "-" +
```

```
counter.getAndIncrement());

        // Set thread properties
        thread.setPriority(Thread.NORM_PRIORITY);
        thread.setDaemon(false);

        // Uncaught exception handler
        thread.setUncaughtExceptionHandler((t, e) -> {
            System.err.println("Thread " + t.getName() + " threw exception: " +
e.getMessage());
            e.printStackTrace();
        });

        return thread;
    }
}

// Using the custom thread factory
ExecutorService executor = Executors.newFixedThreadPool(5,
    new CustomThreadFactory("worker"));
```

Reactive Streams

Reactive Streams is an API for asynchronous stream processing with non-blocking back pressure.

```
import java.util.concurrent.Flow.*;

// Publisher implementation
class SimplePublisher implements Publisher<Integer> {
    private final List<Integer> data;

    public SimplePublisher(List<Integer> data) {
        this.data = new ArrayList<>(data);
    }

    @Override
    public void subscribe(Subscriber<? super Integer> subscriber) {
        subscriber.onSubscribe(new SimpleSubscription(subscriber, data));
    }

    static class SimpleSubscription implements Subscription {
        private final Subscriber<? super Integer> subscriber;
        private final List<Integer> data;
        private int position = 0;
        private boolean cancelled = false;

        public SimpleSubscription(Subscriber<? super Integer> subscriber,
List<Integer> data) {
            this.subscriber = subscriber;
            this.data = data;
        }
```

```
        @Override
        public void request(long n) {
            if (cancelled) return;

            for (int i = 0; i < n && position < data.size(); i++) {
                subscriber.onNext(data.get(position++));
            }

            if (position >= data.size()) {
                subscriber.onComplete();
            }
        }

        @Override
        public void cancel() {
            cancelled = true;
        }
    }
}

// Subscriber implementation
class SimpleSubscriber implements Subscriber<Integer> {
    private Subscription subscription;
    private final int bufferSize;
    private int count = 0;

    public SimpleSubscriber(int bufferSize) {
        this.bufferSize = bufferSize;
    }

    @Override
    public void onSubscribe(Subscription subscription) {
        this.subscription = subscription;
        subscription.request(bufferSize);  // Initial request
    }

    @Override
    public void onNext(Integer item) {
        System.out.println("Received: " + item);
        count++;

        // Request more items when buffer is half consumed
        if (count >= bufferSize / 2) {
            subscription.request(count);
            count = 0;
        }
    }

    @Override
    public void onError(Throwable throwable) {
        System.err.println("Error: " + throwable.getMessage());
    }
```

```
    @Override
    public void onComplete() {
        System.out.println("Completed");
    }
}

// Processor (both subscriber and publisher)
class SimpleProcessor implements Processor<Integer, Integer> {
    private Subscription subscription;
    private List<Subscriber<? super Integer>> subscribers = new ArrayList<>();

    @Override
    public void subscribe(Subscriber<? super Integer> subscriber) {
        subscribers.add(subscriber);
        subscriber.onSubscribe(new SimpleSubscription(subscriber));
    }

    @Override
    public void onSubscribe(Subscription subscription) {
        this.subscription = subscription;
        subscription.request(Long.MAX_VALUE);  // Request all items
    }

    @Override
    public void onNext(Integer item) {
        // Process and forward to subscribers
        int transformed = item * 2;

        for (Subscriber<? super Integer> subscriber : subscribers) {
            subscriber.onNext(transformed);
        }
    }

    @Override
    public void onError(Throwable throwable) {
        for (Subscriber<? super Integer> subscriber : subscribers) {
            subscriber.onError(throwable);
        }
    }

    @Override
    public void onComplete() {
        for (Subscriber<? super Integer> subscriber : subscribers) {
            subscriber.onComplete();
        }
    }

    class SimpleSubscription implements Subscription {
        private final Subscriber<? super Integer> subscriber;

        public SimpleSubscription(Subscriber<? super Integer> subscriber) {
            this.subscriber = subscriber;
        }
```

```
        @Override
        public void request(long n) {
            // Forward request to upstream
            if (subscription != null) {
                subscription.request(n);
            }
        }

        @Override
        public void cancel() {
            subscribers.remove(subscriber);
        }
    }
}

// Usage
List<Integer> data = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
Publisher<Integer> publisher = new SimplePublisher(data);
Processor<Integer, Integer> processor = new SimpleProcessor();
Subscriber<Integer> subscriber = new SimpleSubscriber(4);

publisher.subscribe(processor);
processor.subscribe(subscriber);
```

SubmissionPublisher (Java 9+) - standard implementation of Publisher:

```
import java.util.concurrent.SubmissionPublisher;

try (SubmissionPublisher<Integer> publisher = new SubmissionPublisher<>()) {
    // Subscribe
    publisher.subscribe(new SimpleSubscriber(4));

    // Publish items
    for (int i = 0; i < 10; i++) {
        publisher.submit(i);
        Thread.sleep(100);  // Give time to process
    }
}  // Auto-closes and calls onComplete
```

> **Common Pitfall**: CompletableFuture callbacks run on the thread that completes the future, which might not be the intended thread for UI updates or thread-sensitive operations. Use `thenAcceptAsync` or similar methods to control the execution thread.

> **Certification Note**: Understand advanced patterns with CompletableFuture, including composing and combining futures, handling exceptions, and working with timeouts. Be familiar with the reactive streams API and its components: Publisher, Subscriber, Subscription, and Processor.

Virtual Threads and Project Loom

Java 21 introduces virtual threads, lightweight threads that significantly improve throughput for applications with many concurrent tasks.

Creating Virtual Threads

```
// Start a virtual thread directly
Thread vt = Thread.startVirtualThread(() -> {
    System.out.println("Running in virtual thread: " + Thread.currentThread());
});

// Create but don't start
Thread vt2 = Thread.ofVirtual().name("worker").unstarted(() -> {
    // Task code
});
vt2.start();

// Using builder
Thread.Builder builder = Thread.ofVirtual().name("worker-", 0);
Thread vt3 = builder.start(() -> {
    System.out.println("Worker thread");
});

// Creating many virtual threads
List<Thread> threads = IntStream.range(0, 10_000)
    .mapToObj(i -> Thread.startVirtualThread(() -> {
        System.out.println("Thread " + i);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }))
    .toList();

// Wait for all to complete
for (Thread t : threads) {
    t.join();
}
```

Executors and Virtual Threads

```
// Virtual thread per task executor
try (ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor()) {
    IntStream.range(0, 10_000).forEach(i -> {
        executor.submit(() -> {
            Thread.sleep(1000);
            return i;
        });
    });
}  // Auto-shutdown with try-with-resources
```

```
// Submit tasks and collect results
try (ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor()) {
    List<Future<Integer>> futures = IntStream.range(0, 100)
        .mapToObj(i -> executor.submit(() -> {
            // Perform task
            return processData(i);
        }))
        .collect(Collectors.toList());

    // Gather results
    List<Integer> results = new ArrayList<>();
    for (Future<Integer> future : futures) {
        results.add(future.get());
    }
}
```

Blocking Operations with Virtual Threads

```
// JDBC operations with virtual threads
try (ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor()) {
    executor.submit(() -> {
        try (Connection conn = DriverManager.getConnection(DB_URL)) {
            // Virtual thread "parks" during I/O, doesn't block platform thread
            PreparedStatement stmt = conn.prepareStatement("SELECT * FROM users");
            ResultSet rs = stmt.executeQuery();

            while (rs.next()) {
                processRow(rs);
            }
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    });
}

// HTTP requests with virtual threads
try (ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor()) {
    List<Future<String>> futures = urls.stream()
        .map(url -> executor.submit(() -> {
            HttpClient client = HttpClient.newHttpClient();
            HttpRequest request = HttpRequest.newBuilder()
                .uri(URI.create(url))
                .build();

            // Blocks the virtual thread, but not platform thread
            HttpResponse<String> response = client.send(
                request, HttpResponse.BodyHandlers.ofString());

            return response.body();
        }))
```

```
            .collect(Collectors.toList());

        // Process results
        for (Future<String> future : futures) {
            process(future.get());
        }
    }
```

Structured Concurrency

Structured Concurrency (preview feature in Java 21) makes it easier to manage concurrent tasks:

```
// Using StructuredTaskScope
try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
    Future<User> user = scope.fork(() -> fetchUser(userId));
    Future<List<Order>> orders = scope.fork(() -> fetchOrders(userId));

    // Join and check for exceptions
    scope.join();
    scope.throwIfFailed();

    // Process results
    User u = user.resultNow();
    List<Order> o = orders.resultNow();
    return new UserOrders(u, o);
}

// ShutdownOnSuccess - complete when first task succeeds
try (var scope = new StructuredTaskScope.ShutdownOnSuccess<String>()) {
    // Try multiple data sources in parallel
    scope.fork(() -> fetchFromPrimarySource());
    scope.fork(() -> fetchFromSecondarySource());
    scope.fork(() -> fetchFromTertiarySource());

    // Get the first successful result
    return scope.join().result();
}

// Custom scope for handling partial results
class CustomScope<T> extends StructuredTaskScope<T> {
    private final List<T> successes = Collections.synchronizedList(new ArrayList<>
());
    private final List<Throwable> failures = Collections.synchronizedList(new
ArrayList<>());

    @Override
    protected void handleComplete(Future<T> future) {
        try {
            successes.add(future.resultNow());
        } catch (Exception e) {
            failures.add(e);
```

```
        }
    }

    public List<T> successes() {
        return successes;
    }

    public List<Throwable> failures() {
        return failures;
    }
}

// Using the custom scope
try (var scope = new CustomScope<String>()) {
    // Fork tasks
    for (int i = 0; i < 10; i++) {
        scope.fork(() -> processPartition(i));
    }

    // Wait for all tasks to complete
    scope.join();

    // Process results
    List<String> results = scope.successes();
    List<Throwable> errors = scope.failures();

    if (!errors.isEmpty()) {
        log.warn("Some tasks failed: " + errors.size());
    }

    return results;
}
```

Best Practices for Virtual Threads

```
// DO: Create many short-lived threads for specific tasks
// Each HTTP request gets its own virtual thread
server.createContext("/api", exchange -> {
    Thread.startVirtualThread(() -> {
        try {
            String response = processRequest(exchange);
            exchange.sendResponseHeaders(200, response.length());
            try (OutputStream os = exchange.getResponseBody()) {
                os.write(response.getBytes());
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    });
});
```

```
// DON'T: Create thread pools of virtual threads
// This defeats the purpose of virtual threads
ExecutorService badPool = Executors.newFixedThreadPool(100,
    task -> Thread.startVirtualThread(task));  // Pointless limitation

// Instead, use virtual thread per task executor
ExecutorService goodPool = Executors.newVirtualThreadPerTaskExecutor();

// DON'T: Use thread affinity with synchronized blocks on shared resources
synchronized void blockingMethod() {  // Pins virtual thread to carrier thread
    // Long-running or blocking operation
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
}

// INSTEAD: Use higher-level concurrency constructs for coordination
ReentrantLock lock = new ReentrantLock();
void betterMethod() {
    lock.lock();
    try {
        // Critical section (keep it short)
    } finally {
        lock.unlock();
    }
}
```

Monitoring and Debugging Virtual Threads

```
// Getting thread information
Thread.Builder builder = Thread.ofVirtual().name("task-");
Thread vt = builder.start(() -> {
    // Task code
});

System.out.println("Is virtual: " + vt.isVirtual());
System.out.println("Thread name: " + vt.getName());
System.out.println("Thread state: " + vt.getState());

// JFR events for virtual threads (available with JDK Flight Recorder)
// Start recording from command line:
// java -XX:StartFlightRecording=filename=recording.jfr,settings=profile YourApp

// Thread dumps containing virtual threads
// jcmd <pid> Thread.dump_to_file -format=json /path/to/dump.json

// JMX MBean for monitoring thread counts
ThreadMXBean threadBean = ManagementFactory.getThreadMXBean();
int threadCount = threadBean.getThreadCount();  // Platform threads
```

```
// Note: Virtual threads may not be included in standard JMX counters

// Custom monitoring for virtual threads
AtomicInteger activeVirtualThreads = new AtomicInteger(0);

Runnable monitoredTask = () -> {
    activeVirtualThreads.incrementAndGet();
    try {
        // Task code
    } finally {
        activeVirtualThreads.decrementAndGet();
    }
};

// Start monitored virtual threads
for (int i = 0; i < 1000; i++) {
    Thread.startVirtualThread(monitoredTask);
}

// Report metrics
System.out.println("Active virtual threads: " + activeVirtualThreads.get());
```

> **Common Pitfall**: Synchronized methods and blocks with virtual threads can cause pinning, which eliminates the performance benefits of virtual threads. Prefer using explicit locks (ReentrantLock) for virtual thread synchronization.

> **Certification Note**: Understand how to create and use virtual threads, and how they differ from platform threads. Know the best practices for virtual thread usage, including avoiding thread pools and minimizing pinning. Be familiar with structured concurrency and how it simplifies concurrent task management.

Java I/O and Network Programming

Java I/O Operations

File and Directory Operations

```
import java.io.File;
import java.nio.file.*;

// File class (legacy)
File file = new File("data.txt");
boolean exists = file.exists();
boolean isFile = file.isFile();
boolean isDirectory = file.isDirectory();
boolean created = file.createNewFile();
boolean deleted = file.delete();
long size = file.length();
long lastModified = file.lastModified();
```

```
// Creating directories
File dir = new File("mydir");
boolean dirCreated = dir.mkdir();  // Create single directory
File deepDir = new File("path/to/dir");
boolean deepDirCreated = deepDir.mkdirs();  // Create parent directories too

// Listing directory contents
File[] files = dir.listFiles();
File[] txtFiles = dir.listFiles((d, name) -> name.endsWith(".txt"));

// NIO.2 Path (Java 7+)
Path path = Paths.get("data.txt");
Path absolute = path.toAbsolutePath();
Path normalized = path.normalize();
Path parent = path.getParent();
Path fileName = path.getFileName();
Path subpath = path.subpath(0, 1);

// Path resolution and relativization
Path base = Paths.get("/home/user");
Path resolved = base.resolve("documents/file.txt");  //
/home/user/documents/file.txt
Path rel = base.relativize(resolved);  // documents/file.txt

// Path operations with NIO.2
boolean pathExists = Files.exists(path);
boolean isRegularFile = Files.isRegularFile(path);
boolean isDir = Files.isDirectory(path);
boolean isReadable = Files.isReadable(path);
boolean isWritable = Files.isWritable(path);
boolean isExecutable = Files.isExecutable(path);

// Creating files and directories
Path newFile = Files.createFile(Paths.get("newfile.txt"));
Path newDir = Files.createDirectory(Paths.get("newdir"));
Path newDirs = Files.createDirectories(Paths.get("path/to/deep/dir"));

// File attributes
BasicFileAttributes attrs = Files.readAttributes(path, BasicFileAttributes.class);
System.out.println("Creation time: " + attrs.creationTime());
System.out.println("Last access time: " + attrs.lastAccessTime());
System.out.println("Last modified time: " + attrs.lastModifiedTime());
System.out.println("Size: " + attrs.size());
System.out.println("Is directory: " + attrs.isDirectory());
System.out.println("Is regular file: " + attrs.isRegularFile());
System.out.println("Is symbolic link: " + attrs.isSymbolicLink());

// Setting file times
FileTime now = FileTime.fromMillis(System.currentTimeMillis());
Files.setLastModifiedTime(path, now);

// Copy, move, delete operations
Path source = Paths.get("source.txt");
```

```
Path target = Paths.get("target.txt");

Files.copy(source, target, StandardCopyOption.REPLACE_EXISTING);
Files.move(source, target, StandardCopyOption.REPLACE_EXISTING);
Files.delete(path);  // Throws exception if file doesn't exist
Files.deleteIfExists(path);  // No exception if file doesn't exist

// Temporary files and directories
Path tempDir = Files.createTempDirectory("prefix-");
Path tempFile = Files.createTempFile("prefix-", "-suffix");

// Walking a directory tree
Path startDir = Paths.get("src");
Files.walkFileTree(startDir, new SimpleFileVisitor<Path>() {
    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) {
        System.out.println(file);
        return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult visitFileFailed(Path file, IOException exc) {
        System.err.println("Failed to visit: " + file);
        return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs)
{
        System.out.println("About to visit directory: " + dir);
        return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult postVisitDirectory(Path dir, IOException exc) {
        System.out.println("Finished directory: " + dir);
        return FileVisitResult.CONTINUE;
    }
});

// Finding files
PathMatcher matcher = FileSystems.getDefault().getPathMatcher("glob:*.
{java,class}");
Files.find(startDir, Integer.MAX_VALUE,
    (path, attrs) -> matcher.matches(path.getFileName()))
    .forEach(System.out::println);

// Streams of paths
try (Stream<Path> paths = Files.list(startDir)) {
    paths.filter(Files::isRegularFile)
         .forEach(System.out::println);
}
```

File I/O: Byte Streams

```
// FileInputStream - reading bytes
try (FileInputStream fis = new FileInputStream("input.bin")) {
    byte[] buffer = new byte[1024];
    int bytesRead;

    while ((bytesRead = fis.read(buffer)) != -1) {
        // Process bytes in buffer up to bytesRead
        processBytes(buffer, bytesRead);
    }
}

// FileOutputStream - writing bytes
try (FileOutputStream fos = new FileOutputStream("output.bin")) {
    byte[] data = {1, 2, 3, 4, 5};
    fos.write(data);

    // Append mode
    try (FileOutputStream appendFos = new FileOutputStream("output.bin", true)) {
        appendFos.write(new byte[]{6, 7, 8});
    }
}

// BufferedInputStream - buffered reading for efficiency
try (BufferedInputStream bis = new BufferedInputStream(
        new FileInputStream("large-file.bin"))) {
    byte[] buffer = new byte[4096];
    int bytesRead;

    while ((bytesRead = bis.read(buffer)) != -1) {
        // Process bytes
    }
}

// BufferedOutputStream - buffered writing for efficiency
try (BufferedOutputStream bos = new BufferedOutputStream(
        new FileOutputStream("output.bin"))) {
    for (int i = 0; i < 10000; i++) {
        bos.write(i & 0xFF);  // Write low 8 bits of each int
    }
    // Data is buffered and flushed automatically when closed
    // or can be manually flushed:
    bos.flush();
}

// DataInputStream/DataOutputStream - reading/writing primitive data types
try (DataOutputStream dos = new DataOutputStream(
        new FileOutputStream("data.bin"))) {
    dos.writeInt(42);
    dos.writeDouble(3.14);
    dos.writeUTF("Hello, DataOutputStream!");
```

```
}

try (DataInputStream dis = new DataInputStream(
        new FileInputStream("data.bin"))) {
    int intValue = dis.readInt();
    double doubleValue = dis.readDouble();
    String stringValue = dis.readUTF();
}

// ByteArrayInputStream/ByteArrayOutputStream - working with byte arrays
ByteArrayOutputStream baos = new ByteArrayOutputStream();
try (DataOutputStream dos = new DataOutputStream(baos)) {
    dos.writeInt(123);
    dos.writeUTF("Test");
}
byte[] bytes = baos.toByteArray();

ByteArrayInputStream bais = new ByteArrayInputStream(bytes);
try (DataInputStream dis = new DataInputStream(bais)) {
    int value = dis.readInt();
    String text = dis.readUTF();
}
```

File I/O: Character Streams

```
// FileReader - reading characters
try (FileReader reader = new FileReader("text.txt")) {
    char[] buffer = new char[1024];
    int charsRead;

    while ((charsRead = reader.read(buffer)) != -1) {
        // Process characters
        System.out.print(new String(buffer, 0, charsRead));
    }
}

// FileWriter - writing characters
try (FileWriter writer = new FileWriter("output.txt")) {
    writer.write("Hello, FileWriter!");

    // Append mode
    try (FileWriter appendWriter = new FileWriter("output.txt", true)) {
        appendWriter.write("\nAppended text");
    }
}

// BufferedReader - efficient character reading with line support
try (BufferedReader reader = new BufferedReader(
        new FileReader("text.txt"))) {
    String line;
    while ((line = reader.readLine()) != null) {
```

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

    // BufferedWriter - efficient character writing
    try (BufferedWriter writer = new BufferedWriter(
            new FileWriter("output.txt"))) {
        writer.write("Line 1");
        writer.newLine();  // Platform-specific line separator
        writer.write("Line 2");
    }

    // PrintWriter - formatted text output
    try (PrintWriter writer = new PrintWriter(
            new BufferedWriter(new FileWriter("formatted.txt")))) {
        writer.println("Hello, PrintWriter!");
        writer.printf("Formatted value: %d, %s%n", 42, "test");

        // Check for errors
        if (writer.checkError()) {
            System.err.println("Error writing to file");
        }
    }

    // StringReader/StringWriter - working with strings
    StringWriter stringWriter = new StringWriter();
    try (PrintWriter writer = new PrintWriter(stringWriter)) {
        writer.println("Line 1");
        writer.println("Line 2");
    }
    String result = stringWriter.toString();

    StringReader stringReader = new StringReader(result);
    try (BufferedReader reader = new BufferedReader(stringReader)) {
        String line;
        while ((line = reader.readLine()) != null) {
            System.out.println(line);
        }
    }

    // Reading with specific character encoding
    try (BufferedReader reader = new BufferedReader(
            new InputStreamReader(
                new FileInputStream("utf8-file.txt"),
                StandardCharsets.UTF_8))) {
        String line;
        while ((line = reader.readLine()) != null) {
            // Process UTF-8 encoded content
        }
    }

    // Writing with specific character encoding
    try (Writer writer = new BufferedWriter(
            new OutputStreamWriter(
```

```
            new FileOutputStream("output.txt"),
            StandardCharsets.UTF_16))) {
    writer.write("Text with UTF-16 encoding");
}
```

NIO File I/O

```
import java.nio.file.*;
import java.nio.charset.StandardCharsets;

// Reading all bytes/lines at once
byte[] allBytes = Files.readAllBytes(Paths.get("data.bin"));
List<String> allLines = Files.readAllLines(Paths.get("text.txt"));

// Writing all bytes/lines at once
Files.write(Paths.get("output.bin"), new byte[]{1, 2, 3});
Files.write(Paths.get("output.txt"), Arrays.asList("Line 1", "Line 2"));

// Reading as a stream (Java 8+)
try (Stream<String> lines = Files.lines(Paths.get("large-file.txt"))) {
    lines.filter(line -> line.contains("pattern"))
         .map(String::trim)
         .forEach(System.out::println);
}

// Reading and writing with options
byte[] bytes = Files.readAllBytes(Paths.get("file.txt"));
Files.write(Paths.get("copy.txt"), bytes,
    StandardOpenOption.CREATE,
    StandardOpenOption.TRUNCATE_EXISTING);

// Appending to a file
Files.write(Paths.get("log.txt"),
    Collections.singletonList("New log entry"),
    StandardCharsets.UTF_8,
    StandardOpenOption.CREATE,
    StandardOpenOption.APPEND);

// Reading/writing using BufferedReader/BufferedWriter
try (BufferedReader reader = Files.newBufferedReader(
        Paths.get("input.txt"), StandardCharsets.UTF_8)) {
    String line;
    while ((line = reader.readLine()) != null) {
        // Process line
    }
}

try (BufferedWriter writer = Files.newBufferedWriter(
        Paths.get("output.txt"), StandardCharsets.UTF_8)) {
    writer.write("Line 1");
    writer.newLine();
```

```
    writer.write("Line 2");
}

// Copying with NIO
Files.copy(Paths.get("source.txt"), Paths.get("destination.txt"),
    StandardCopyOption.REPLACE_EXISTING);

// Copying from InputStream to a file
try (InputStream is = new URL("https://example.com/file.txt").openStream()) {
    Files.copy(is, Paths.get("downloaded.txt"),
StandardCopyOption.REPLACE_EXISTING);
}

// Copying from a file to OutputStream
try (OutputStream os = new FileOutputStream("copy.txt")) {
    Files.copy(Paths.get("source.txt"), os);
}
```

NIO.2 Channel I/O

```
import java.nio.channels.*;
import java.nio.ByteBuffer;

// FileChannel - reading
try (FileChannel channel = FileChannel.open(
        Paths.get("data.bin"), StandardOpenOption.READ)) {
    ByteBuffer buffer = ByteBuffer.allocate(1024);

    while (channel.read(buffer) != -1) {
        buffer.flip();  // Prepare buffer for reading

        while (buffer.hasRemaining()) {
            byte b = buffer.get();
            // Process byte
        }

        buffer.clear();  // Prepare buffer for writing
    }
}

// FileChannel - writing
try (FileChannel channel = FileChannel.open(
        Paths.get("output.bin"),
        StandardOpenOption.CREATE,
        StandardOpenOption.WRITE)) {
    ByteBuffer buffer = ByteBuffer.allocate(1024);

    // Fill the buffer
    for (int i = 0; i < 100; i++) {
        buffer.put((byte) i);
    }
```

```
    buffer.flip();  // Prepare buffer for reading

    while (buffer.hasRemaining()) {
        channel.write(buffer);
    }
}

// Using direct ByteBuffer (off-heap)
ByteBuffer directBuffer = ByteBuffer.allocateDirect(1024);
directBuffer.putInt(42);
directBuffer.putDouble(3.14);
directBuffer.flip();

// Memory-mapped file
try (FileChannel channel = FileChannel.open(
        Paths.get("large-file.bin"), StandardOpenOption.READ,
StandardOpenOption.WRITE)) {

    // Map a 1MB region starting at position 0
    MappedByteBuffer mappedBuffer = channel.map(
        FileChannel.MapMode.READ_WRITE, 0, 1024 * 1024);

    // Work directly with memory-mapped file
    mappedBuffer.putInt(0, 123);  // Write directly to first 4 bytes
    int value = mappedBuffer.getInt(1024);  // Read directly from position 1024

    // Changes are automatically persisted
    mappedBuffer.force();  // Ensure all changes are written to disk
}

// Channel transfers (zero-copy when possible)
try (FileChannel sourceChannel = FileChannel.open(Paths.get("source.bin"),
StandardOpenOption.READ);
     FileChannel targetChannel = FileChannel.open(Paths.get("target.bin"),
         StandardOpenOption.CREATE, StandardOpenOption.WRITE)) {

    long size = sourceChannel.size();
    long position = 0;
    long count;

    while (position < size) {
        count = sourceChannel.transferTo(position, 1024 * 1024, targetChannel);
        position += count;
    }
}

// Scatter/Gather
try (FileChannel channel = FileChannel.open(Paths.get("data.bin"),
StandardOpenOption.READ)) {
    // Scatter: read into multiple buffers
    ByteBuffer headerBuffer = ByteBuffer.allocate(128);
    ByteBuffer bodyBuffer = ByteBuffer.allocate(1024);
```

```
    ByteBuffer[] bufferArray = {headerBuffer, bodyBuffer};
    channel.read(bufferArray);

    headerBuffer.flip();
    bodyBuffer.flip();

    // Process header and body separately
}

try (FileChannel channel = FileChannel.open(Paths.get("output.bin"),
        StandardOpenOption.CREATE, StandardOpenOption.WRITE)) {

    // Gather: write from multiple buffers
    ByteBuffer headerBuffer = ByteBuffer.allocate(128);
    ByteBuffer bodyBuffer = ByteBuffer.allocate(1024);

    // Fill buffers with data
    fillBuffer(headerBuffer);
    fillBuffer(bodyBuffer);

    headerBuffer.flip();
    bodyBuffer.flip();

    ByteBuffer[] bufferArray = {headerBuffer, bodyBuffer};
    channel.write(bufferArray);
}
```

Serialization

```
import java.io.*;

// Serializable class
class Person implements Serializable {
    // serialVersionUID helps with version compatibility
    private static final long serialVersionUID = 1L;

    private String name;
    private int age;
    private transient String password;  // transient fields aren't serialized

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

    // Custom serialization
    private void writeObject(ObjectOutputStream out) throws IOException {
        out.defaultWriteObject();  // Default serialization
        out.writeObject("Extra data");  // Custom data
    }
```

```
    // Custom deserialization
    private void readObject(ObjectInputStream in) throws IOException,
ClassNotFoundException {
        in.defaultReadObject();  // Default deserialization
        String extra = (String) in.readObject();  // Read custom data
        this.password = "Default password";  // Reset transient fields
    }
}

// Serializing objects
try (ObjectOutputStream oos = new ObjectOutputStream(
        new FileOutputStream("person.ser"))) {
    Person person = new Person("John Doe", 30, "secret");
    oos.writeObject(person);  // Serialize object

    List<Person> people = Arrays.asList(
        new Person("Alice", 25, "pwd1"),
        new Person("Bob", 35, "pwd2")
    );
    oos.writeObject(people);  // Serialize collection
}

// Deserializing objects
try (ObjectInputStream ois = new ObjectInputStream(
        new FileInputStream("person.ser"))) {
    Person person = (Person) ois.readObject();
    @SuppressWarnings("unchecked")
    List<Person> people = (List<Person>) ois.readObject();
}

// Externalizable for full control over serialization
class ExternalizableExample implements Externalizable {
    private int id;
    private String name;
    private List<String> data;

    // Required no-arg constructor for Externalizable
    public ExternalizableExample() {}

    public ExternalizableExample(int id, String name, List<String> data) {
        this.id = id;
        this.name = name;
        this.data = data;
    }

    @Override
    public void writeExternal(ObjectOutput out) throws IOException {
        out.writeInt(id);
        out.writeUTF(name);
        out.writeInt(data.size());
        for (String item : data) {
            out.writeUTF(item);
        }
```

```
    }

    @Override
    public void readExternal(ObjectInput in) throws IOException,
ClassNotFoundException {
        id = in.readInt();
        name = in.readUTF();
        int size = in.readInt();
        data = new ArrayList<>(size);
        for (int i = 0; i < size; i++) {
            data.add(in.readUTF());
        }
    }
}
```

Working with ZIP Files

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

// Creating a ZIP file
try (ZipOutputStream zos = new ZipOutputStream(
        new FileOutputStream("archive.zip"))) {

    // Add file entry
    Path file1 = Paths.get("file1.txt");
    ZipEntry entry1 = new ZipEntry(file1.getFileName().toString());
    zos.putNextEntry(entry1);
    Files.copy(file1, zos);
    zos.closeEntry();

    // Add another file entry
    Path file2 = Paths.get("file2.txt");
    ZipEntry entry2 = new ZipEntry(file2.getFileName().toString());
    zos.putNextEntry(entry2);
    Files.copy(file2, zos);
    zos.closeEntry();

    // Add directory entry with files
    Path dir = Paths.get("dir");
    if (Files.isDirectory(dir)) {
        Files.walk(dir)
            .filter(path -> !Files.isDirectory(path))
            .forEach(path -> {
                ZipEntry entry = new ZipEntry(dir.relativize(path).toString());
                try {
                    zos.putNextEntry(entry);
                    Files.copy(path, zos);
                    zos.closeEntry();
                } catch (IOException e) {
                    e.printStackTrace();
```

```
                }
            });
    }
}

// Reading a ZIP file
try (ZipFile zipFile = new ZipFile("archive.zip")) {
    Enumeration<? extends ZipEntry> entries = zipFile.entries();
    while (entries.hasMoreElements()) {
        ZipEntry entry = entries.nextElement();
        System.out.println("Entry: " + entry.getName());

        if (!entry.isDirectory()) {
            try (InputStream is = zipFile.getInputStream(entry)) {
                // Process file contents
                byte[] buffer = new byte[1024];
                int bytesRead;
                while ((bytesRead = is.read(buffer)) != -1) {
                    // Process bytes
                }
            }
        }
    }
}

// Unzipping a ZIP file
Path destDir = Paths.get("unzipped");
Files.createDirectories(destDir);

try (ZipInputStream zis = new ZipInputStream(
        new FileInputStream("archive.zip"))) {
    ZipEntry entry;
    while ((entry = zis.getNextEntry()) != null) {
        Path entryPath = destDir.resolve(entry.getName());

        if (entry.isDirectory()) {
            Files.createDirectories(entryPath);
        } else {
            // Create parent directories if they don't exist
            Files.createDirectories(entryPath.getParent());

            // Write file content
            try (OutputStream os = new FileOutputStream(entryPath.toFile())) {
                byte[] buffer = new byte[1024];
                int bytesRead;
                while ((bytesRead = zis.read(buffer)) != -1) {
                    os.write(buffer, 0, bytesRead);
                }
            }
        }
        zis.closeEntry();
    }
}
```

File Change Notification

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

// WatchService for monitoring file changes
Path dirToWatch = Paths.get("watched-dir");
try (WatchService watchService = FileSystems.getDefault().newWatchService()) {
    // Register for create, modify, delete events
    dirToWatch.register(
        watchService,
        StandardWatchEventKinds.ENTRY_CREATE,
        StandardWatchEventKinds.ENTRY_MODIFY,
        StandardWatchEventKinds.ENTRY_DELETE
    );

    System.out.println("Watching directory: " + dirToWatch);

    while (true) {
        // Wait for key to be signaled
        WatchKey key;
        try {
            key = watchService.take();  // Blocks until events
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            break;
        }

        // Process events
        for (WatchEvent<?> event : key.pollEvents()) {
            WatchEvent.Kind<?> kind = event.kind();

            // Overflow event - some events may have been lost
            if (kind == StandardWatchEventKinds.OVERFLOW) {
                continue;
            }

            @SuppressWarnings("unchecked")
            WatchEvent<Path> pathEvent = (WatchEvent<Path>) event;
            Path fileName = pathEvent.context();
            Path fullPath = dirToWatch.resolve(fileName);

            System.out.printf("Event %s on %s%n", kind, fullPath);
        }

        // Reset key and check if still valid
        boolean valid = key.reset();
        if (!valid) {
            // Directory is no longer accessible
            break;
        }
    }
```

```
}

// Alternative with polling timeout
try (WatchService watchService = FileSystems.getDefault().newWatchService()) {
    dirToWatch.register(watchService, StandardWatchEventKinds.ENTRY_MODIFY);

    while (!Thread.currentThread().isInterrupted()) {
        // Wait with timeout
        WatchKey key = watchService.poll(10, TimeUnit.SECONDS);
        if (key == null) {
            // Timeout occurred, no events
            continue;
        }

        // Process events...
        // Reset key...
    }
}

// Recursive directory watching (requires manual implementation)
Files.walkFileTree(dirToWatch, new SimpleFileVisitor<Path>() {
    @Override
    public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs)
            throws IOException {
        dir.register(watchService,
            StandardWatchEventKinds.ENTRY_CREATE,
            StandardWatchEventKinds.ENTRY_MODIFY,
            StandardWatchEventKinds.ENTRY_DELETE);
        return FileVisitResult.CONTINUE;
    }
});
```

> **Common Pitfall**: Not properly closing I/O resources. Always use try-with-resources to ensure streams and channels are closed, even when exceptions occur.

> **Certification Note**: Understand the differences between the various I/O classes in Java and when to use each. Know how to perform common file operations using both legacy I/O and NIO.2. Be familiar with the byte and character stream hierarchies.

Network Programming

URL and URLConnection

```
import java.net.*;
import java.io.*;

// Working with URLs
URL url = new URL("https://example.com/path?query=value#fragment");
System.out.println("Protocol: " + url.getProtocol());
System.out.println("Host: " + url.getHost());
System.out.println("Port: " + url.getPort());  // -1 if default port
```

```
System.out.println("Default port: " + url.getDefaultPort());
System.out.println("Path: " + url.getPath());
System.out.println("Query: " + url.getQuery());
System.out.println("Fragment: " + url.getRef());

// Reading from a URL
try (InputStream is = url.openStream();
     BufferedReader reader = new BufferedReader(new InputStreamReader(is))) {
    String line;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
}

// Using URLConnection for more control
URLConnection connection = url.openConnection();
connection.setRequestProperty("User-Agent", "Java Application");
connection.setConnectTimeout(5000);
connection.setReadTimeout(5000);
connection.connect();

// Reading response headers
Map<String, List<String>> headers = connection.getHeaderFields();
for (Map.Entry<String, List<String>> entry : headers.entrySet()) {
    System.out.println(entry.getKey() + ": " + entry.getValue());
}

System.out.println("Content type: " + connection.getContentType());
System.out.println("Content length: " + connection.getContentLength());
System.out.println("Last modified: " + new Date(connection.getLastModified()));

// Reading response
try (InputStream is = connection.getInputStream();
     BufferedReader reader = new BufferedReader(new InputStreamReader(is))) {
    StringBuilder response = new StringBuilder();
    String line;
    while ((line = reader.readLine()) != null) {
        response.append(line).append('\n');
    }
    System.out.println(response.toString());
}

// HTTP-specific functionality with HttpURLConnection
HttpURLConnection httpConnection = (HttpURLConnection) url.openConnection();
httpConnection.setRequestMethod("GET");
httpConnection.setInstanceFollowRedirects(true);
httpConnection.connect();

int responseCode = httpConnection.getResponseCode();
String responseMessage = httpConnection.getResponseMessage();
System.out.println(responseCode + " " + responseMessage);

// POST request with HttpURLConnection
URL postUrl = new URL("https://example.com/api");
```

```
HttpURLConnection postConnection = (HttpURLConnection) postUrl.openConnection();
postConnection.setRequestMethod("POST");
postConnection.setDoOutput(true);
postConnection.setRequestProperty("Content-Type", "application/json");

// Write request body
String requestBody = "{\"name\":\"John\",\"age\":30}";
try (OutputStream os = postConnection.getOutputStream();
     BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(os))) {
    writer.write(requestBody);
}

// Read response
int postResponseCode = postConnection.getResponseCode();
if (postResponseCode >= 200 && postResponseCode < 300) {
    try (BufferedReader reader = new BufferedReader(
            new InputStreamReader(postConnection.getInputStream()))) {
        // Process successful response
    }
} else {
    try (BufferedReader reader = new BufferedReader(
            new InputStreamReader(postConnection.getErrorStream()))) {
        // Process error response
    }
}

// Always disconnect when done
postConnection.disconnect();
```

HTTP Client (Java 11+)

```
import java.net.http.*;
import java.time.Duration;

// Create HTTP client
HttpClient client = HttpClient.newBuilder()
    .version(HttpClient.Version.HTTP_2)  // Prefer HTTP/2
    .connectTimeout(Duration.ofSeconds(10))
    .followRedirects(HttpClient.Redirect.NORMAL)
    .build();

// Simple GET request
HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create("https://example.com/api"))
    .header("User-Agent", "Java HttpClient")
    .GET()  // Default method is GET
    .build();

// Synchronous send
HttpResponse<String> response = client.send(request,
HttpResponse.BodyHandlers.ofString());
```

```
System.out.println("Status: " + response.statusCode());
System.out.println("Headers: " + response.headers());
System.out.println("Body: " + response.body());

// Asynchronous send
CompletableFuture<HttpResponse<String>> futureResponse =
    client.sendAsync(request, HttpResponse.BodyHandlers.ofString());

futureResponse
    .thenApply(HttpResponse::body)
    .thenAccept(System.out::println)
    .join();  // Wait for completion

// POST request with JSON body
HttpRequest postRequest = HttpRequest.newBuilder()
    .uri(URI.create("https://example.com/api/create"))
    .header("Content-Type", "application/json")
    .POST(HttpRequest.BodyPublishers.ofString("{\"name\":\"John\",\"age\":30}"))
    .build();

HttpResponse<String> postResponse = client.send(postRequest,
    HttpResponse.BodyHandlers.ofString());

// Different response body handlers
HttpResponse<byte[]> binaryResponse = client.send(request,
    HttpResponse.BodyHandlers.ofByteArray());

HttpResponse<Path> fileResponse = client.send(request,
    HttpResponse.BodyHandlers.ofFile(Paths.get("response.txt")));

HttpResponse<Stream<String>> streamResponse = client.send(request,
    HttpResponse.BodyHandlers.ofLines());

// Custom response handler
HttpResponse<JsonNode> jsonResponse = client.send(request, responseInfo -> {
    HttpResponse.BodySubscriber<InputStream> inputStream =
        HttpResponse.BodySubscribers.ofInputStream();

    return HttpResponse.BodySubscribers.mapping(
        inputStream,
        is -> {
            try (InputStream stream = is) {
                return new ObjectMapper().readTree(stream);
            } catch (IOException e) {
                throw new UncheckedIOException(e);
            }
        });
});

// HTTP/2 server push
HttpRequest pushRequest = HttpRequest.newBuilder()
    .uri(URI.create("https://example.com/index.html"))
    .build();
```

```
client.sendAsync(pushRequest, HttpResponse.BodyHandlers.ofString(),
    (req, pushPromiseHandler) -> {
        pushPromiseHandler.accept(
            HttpResponse.BodyHandlers.ofFile(
                Paths.get("pushed-" + req.uri().getPath().replaceAll("/", "-")))
        );
    }).join();

// Handling authentication
HttpClient authClient = HttpClient.newBuilder()
    .authenticator(new Authenticator() {
        @Override
        protected PasswordAuthentication getPasswordAuthentication() {
            return new PasswordAuthentication(
                "username", "password".toCharArray());
        }
    })
    .build();

// Setting cookies
HttpClient cookieClient = HttpClient.newBuilder()
    .cookieHandler(new CookieManager(null, CookiePolicy.ACCEPT_ALL))
    .build();

// Configuring proxy
HttpClient proxyClient = HttpClient.newBuilder()
    .proxy(ProxySelector.of(new InetSocketAddress("proxy.example.com", 8080)))
    .build();
```

Socket Programming

```
import java.net.*;
import java.io.*;

// Client socket
try (Socket socket = new Socket("localhost", 8080)) {
    System.out.println("Connected to server");

    // Set socket options
    socket.setSoTimeout(5000);  // Read timeout in milliseconds
    socket.setKeepAlive(true);
    socket.setTcpNoDelay(true);  // Disable Nagle's algorithm

    // Get socket info
    System.out.println("Local port: " + socket.getLocalPort());
    System.out.println("Remote address: " + socket.getInetAddress());
    System.out.println("Remote port: " + socket.getPort());

    // Send data
    try (PrintWriter out = new PrintWriter(socket.getOutputStream(), true)) {
        out.println("Hello, server!");
```

```
    }

    // Receive data
    try (BufferedReader in = new BufferedReader(
            new InputStreamReader(socket.getInputStream()))) {
        String response = in.readLine();
        System.out.println("Server response: " + response);
    }
} catch (ConnectException e) {
    System.err.println("Connection refused: " + e.getMessage());
} catch (SocketTimeoutException e) {
    System.err.println("Connection timeout: " + e.getMessage());
}

// Server socket
try (ServerSocket serverSocket = new ServerSocket(8080)) {
    System.out.println("Server started on port " + serverSocket.getLocalPort());

    // Set socket options
    serverSocket.setSoTimeout(60000);  // Accept timeout
    serverSocket.setReuseAddress(true);

    while (true) {
        try {
            // Wait for client connection
            Socket clientSocket = serverSocket.accept();
            System.out.println("Client connected: " +
                clientSocket.getInetAddress());

            // Handle client connection in a new thread
            new Thread(() -> handleClient(clientSocket)).start();
        } catch (SocketTimeoutException e) {
            System.out.println("Accept timeout, checking shutdown condition...");
            // Check if server should continue running
            if (shouldShutdown()) {
                break;
            }
        }
    }
}

// Client handler method
void handleClient(Socket clientSocket) {
    try (
        BufferedReader in = new BufferedReader(
            new InputStreamReader(clientSocket.getInputStream()));
        PrintWriter out = new PrintWriter(
            clientSocket.getOutputStream(), true)
    ) {
        String inputLine = in.readLine();
        System.out.println("Received: " + inputLine);

        // Echo back the input
        out.println("Echo: " + inputLine);
```

```
        } catch (IOException e) {
            System.err.println("Error handling client: " + e.getMessage());
        } finally {
            try {
                clientSocket.close();
            } catch (IOException e) {
                System.err.println("Error closing client socket: " + e.getMessage());
            }
        }
    }
```

Multicast Sockets

```
// MulticastSocket for group communication
InetAddress group = InetAddress.getByName("230.0.0.1");
int port = 4446;

// Sender
try (MulticastSocket socket = new MulticastSocket()) {
    socket.setTimeToLive(1);  // Limit to local network

    String message = "Hello, multicast world!";
    byte[] buffer = message.getBytes();

    DatagramPacket packet = new DatagramPacket(
        buffer, buffer.length, group, port);

    socket.send(packet);
    System.out.println("Multicast message sent");
}

// Receiver
try (MulticastSocket socket = new MulticastSocket(port)) {
    // Join the multicast group
    socket.joinGroup(group);

    byte[] buffer = new byte[1024];
    DatagramPacket packet = new DatagramPacket(buffer, buffer.length);

    // Wait for packet
    socket.receive(packet);

    String received = new String(packet.getData(), 0, packet.getLength());
    System.out.println("Received: " + received);

    // Leave the group when done
    socket.leaveGroup(group);
}
```

Non-blocking I/O with NIO

```
import java.nio.channels.*;
import java.nio.*;
import java.util.*;

// NIO Server with Selector
public class NioServer {
    private final ServerSocketChannel serverChannel;
    private final Selector selector;

    public NioServer(int port) throws IOException {
        // Create server socket channel
        serverChannel = ServerSocketChannel.open();
        serverChannel.socket().bind(new InetSocketAddress(port));
        serverChannel.configureBlocking(false);  // Non-blocking mode

        // Create selector and register server channel
        selector = Selector.open();
        serverChannel.register(selector, SelectionKey.OP_ACCEPT);

        System.out.println("NIO Server started on port " + port);
    }

    public void start() throws IOException {
        while (true) {
            // Wait for events
            int readyChannels = selector.select();
            if (readyChannels == 0) continue;

            // Process selected keys
            Set<SelectionKey> selectedKeys = selector.selectedKeys();
            Iterator<SelectionKey> keyIterator = selectedKeys.iterator();

            while (keyIterator.hasNext()) {
                SelectionKey key = keyIterator.next();

                if (key.isAcceptable()) {
                    handleAccept();
                } else if (key.isReadable()) {
                    handleRead(key);
                }

                keyIterator.remove();  // Remove the key from the selected set
            }
        }
    }

    private void handleAccept() throws IOException {
        // Accept the connection
        SocketChannel client = serverChannel.accept();
        client.configureBlocking(false);

        // Register for reading
```

```
            client.register(selector, SelectionKey.OP_READ);
            System.out.println("Accepted connection from " +
    client.getRemoteAddress());
        }

        private void handleRead(SelectionKey key) throws IOException {
            SocketChannel client = (SocketChannel) key.channel();
            ByteBuffer buffer = ByteBuffer.allocate(1024);

            try {
                int bytesRead = client.read(buffer);

                if (bytesRead == -1) {
                    // End of stream, close the connection
                    key.cancel();
                    client.close();
                    System.out.println("Connection closed");
                    return;
                }

                buffer.flip();  // Prepare for reading

                byte[] bytes = new byte[buffer.remaining()];
                buffer.get(bytes);
                String message = new String(bytes);

                System.out.println("Received: " + message.trim());

                // Echo back
                buffer.clear();
                buffer.put(("Echo: " + message).getBytes());
                buffer.flip();
                client.write(buffer);

            } catch (IOException e) {
                // Connection problem, close the channel
                key.cancel();
                client.close();
                System.out.println("Connection error: " + e.getMessage());
            }
        }

        public static void main(String[] args) throws IOException {
            NioServer server = new NioServer(8080);
            server.start();
        }
    }

    // NIO Client
    public class NioClient {
        public static void main(String[] args) throws IOException {
            SocketChannel channel = SocketChannel.open();
            channel.configureBlocking(false);
```

```
        // Connect to server
        channel.connect(new InetSocketAddress("localhost", 8080));

        // Wait for connection to complete
        while (!channel.finishConnect()) {
            System.out.println("Connecting...");
            Thread.sleep(100);
        }

        // Send message
        ByteBuffer buffer = ByteBuffer.wrap("Hello, NIO Server!".getBytes());
        channel.write(buffer);

        // Read response
        buffer.clear();
        int bytesRead = channel.read(buffer);

        if (bytesRead > 0) {
            buffer.flip();
            byte[] bytes = new byte[buffer.remaining()];
            buffer.get(bytes);
            System.out.println("Server response: " + new String(bytes));
        }

        channel.close();
    }
}
```

Datagram Sockets (UDP)

```
// UDP Server
try (DatagramSocket socket = new DatagramSocket(9876)) {
    byte[] receiveBuffer = new byte[1024];

    while (true) {
        DatagramPacket receivePacket = new DatagramPacket(
            receiveBuffer, receiveBuffer.length);

        // Wait for packet
        socket.receive(receivePacket);

        String message = new String(
            receivePacket.getData(), 0, receivePacket.getLength());
        System.out.println("Received: " + message);

        // Get sender address and port
        InetAddress clientAddress = receivePacket.getAddress();
        int clientPort = receivePacket.getPort();

        // Prepare response
        String response = "Echo: " + message;
```

```
            byte[] sendBuffer = response.getBytes();

            DatagramPacket sendPacket = new DatagramPacket(
                sendBuffer, sendBuffer.length, clientAddress, clientPort);

            // Send response
            socket.send(sendPacket);
        }
    }

    // UDP Client
    try (DatagramSocket socket = new DatagramSocket()) {
        InetAddress serverAddress = InetAddress.getByName("localhost");
        int serverPort = 9876;

        // Prepare message
        String message = "Hello, UDP Server!";
        byte[] sendBuffer = message.getBytes();

        DatagramPacket sendPacket = new DatagramPacket(
            sendBuffer, sendBuffer.length, serverAddress, serverPort);

        // Send packet
        socket.send(sendPacket);

        // Prepare to receive response
        byte[] receiveBuffer = new byte[1024];
        DatagramPacket receivePacket = new DatagramPacket(
            receiveBuffer, receiveBuffer.length);

        // Set timeout
        socket.setSoTimeout(5000);

        // Wait for response
        socket.receive(receivePacket);

        String response = new String(
            receivePacket.getData(), 0, receivePacket.getLength());
        System.out.println("Server response: " + response);
    }
```

Asynchronous I/O with CompletableFuture

```
// Asynchronous network operations with CompletableFuture
public class AsyncHttpClient {
    private final HttpClient client = HttpClient.newBuilder()
        .version(HttpClient.Version.HTTP_2)
        .build();

    public CompletableFuture<String> fetchAsync(String uri) {
        HttpRequest request = HttpRequest.newBuilder()
```

```
            .uri(URI.create(uri))
            .build();

        return client.sendAsync(request, HttpResponse.BodyHandlers.ofString())
            .thenApply(HttpResponse::body);
    }

    public void fetchMultiple(List<String> urls) {
        List<CompletableFuture<String>> futures = urls.stream()
            .map(this::fetchAsync)
            .collect(Collectors.toList());

        // Wait for all to complete
        CompletableFuture<Void> allFutures = CompletableFuture.allOf(
            futures.toArray(new CompletableFuture[0]));

        // Process when all complete
        allFutures.thenRun(() -> {
            System.out.println("All requests completed");

            for (int i = 0; i < urls.size(); i++) {
                try {
                    String result = futures.get(i).get();
                    System.out.println("Result from " + urls.get(i) +
                        " (" + result.length() + " bytes)");
                } catch (Exception e) {
                    System.err.println("Error for " + urls.get(i) +
                        ": " + e.getMessage());
                }
            }
        });

        // Wait for completion
        try {
            allFutures.get(30, TimeUnit.SECONDS);
        } catch (Exception e) {
            System.err.println("Error waiting for requests: " + e.getMessage());
        }
    }

    public static void main(String[] args) {
        AsyncHttpClient client = new AsyncHttpClient();
        List<String> urls = Arrays.asList(
            "https://example.com",
            "https://example.org",
            "https://example.net"
        );
        client.fetchMultiple(urls);
    }
}
```

> **Common Pitfall**: Not properly handling errors and timeouts in network code can lead to resource leaks and hanging connections. Always set appropriate timeouts and handle all possible exceptions.

> **Certification Note**: Understand the different network programming APIs in Java, including URLs, sockets, and the HTTP client. Know the differences between blocking and non-blocking I/O, and between connection-oriented (TCP) and connectionless (UDP) protocols.

Database Access with JDBC and JPA

JDBC (Java Database Connectivity)

JDBC Architecture

JDBC provides a standard API for connecting to relational databases from Java applications. The architecture consists of:

1. **JDBC Application**: Your Java code that uses JDBC API
2. **JDBC API**: The interfaces and classes defined in `java.sql` and `javax.sql`
3. **JDBC Driver Manager**: Manages database drivers
4. **JDBC Drivers**: Database-specific implementations that connect to actual databases

Establishing Database Connections

```
import java.sql.*;

// Traditional DriverManager approach
String url = "jdbc:mysql://localhost:3306/mydb";
String username = "user";
String password = "password";

// Load driver (optional since JDBC 4.0)
try {
    Class.forName("com.mysql.cj.jdbc.Driver");
} catch (ClassNotFoundException e) {
    System.err.println("Driver not found: " + e.getMessage());
    return;
}

// Creating a connection
try (Connection connection = DriverManager.getConnection(url, username, password))
{
    System.out.println("Database connected!");

    // Set connection properties
    connection.setAutoCommit(false);
    connection.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);

    // Get database metadata
    DatabaseMetaData metaData = connection.getMetaData();
    System.out.println("Database: " + metaData.getDatabaseProductName() +
```

```
                        " " + metaData.getDatabaseProductVersion());
    System.out.println("Driver: " + metaData.getDriverName() +
                        " " + metaData.getDriverVersion());
} catch (SQLException e) {
    System.err.println("Connection error: " + e.getMessage());
}

// Connection pool approach with DataSource
import javax.sql.DataSource;
import com.zaxxer.hikari.HikariConfig;
import com.zaxxer.hikari.HikariDataSource;

// Set up connection pool
HikariConfig config = new HikariConfig();
config.setJdbcUrl(url);
config.setUsername(username);
config.setPassword(password);
config.setMaximumPoolSize(10);
config.setMinimumIdle(5);
config.setIdleTimeout(30000);
config.setConnectionTimeout(10000);

DataSource dataSource = new HikariDataSource(config);

// Get connection from pool
try (Connection connection = dataSource.getConnection()) {
    // Use connection
}
```

Executing SQL Statements

```
// Statement - for basic SQL
try (Connection connection = dataSource.getConnection();
     Statement statement = connection.createStatement()) {

    // Execute query - returns ResultSet
    ResultSet resultSet = statement.executeQuery("SELECT * FROM users");

    // Execute update - returns affected row count
    int rowsAffected = statement.executeUpdate(
        "UPDATE users SET active = true WHERE last_login < '2023-01-01'");

    // Execute any SQL - returns true if result is ResultSet, false otherwise
    boolean isResultSet = statement.execute("SELECT * FROM users");
    if (isResultSet) {
        ResultSet rs = statement.getResultSet();
        // Process result set
    } else {
        int count = statement.getUpdateCount();
        // Process update count
    }
```

```
    // Batch updates
    statement.addBatch("INSERT INTO logs VALUES('log1', NOW())");
    statement.addBatch("INSERT INTO logs VALUES('log2', NOW())");
    statement.addBatch("UPDATE stats SET count = count + 2");
    int[] batchResults = statement.executeBatch();
}

// PreparedStatement - for parameterized SQL
String insertSql = "INSERT INTO users (username, email, created_at) VALUES (?, ?,
?)";
try (Connection connection = dataSource.getConnection();
     PreparedStatement preparedStatement = connection.prepareStatement(
         insertSql, Statement.RETURN_GENERATED_KEYS)) {

    // Set parameters (1-based indexing)
    preparedStatement.setString(1, "johndoe");
    preparedStatement.setString(2, "john@example.com");
    preparedStatement.setTimestamp(3, Timestamp.valueOf(LocalDateTime.now()));

    // Execute
    int rowsAffected = preparedStatement.executeUpdate();

    // Get generated keys
    try (ResultSet generatedKeys = preparedStatement.getGeneratedKeys()) {
        if (generatedKeys.next()) {
            long id = generatedKeys.getLong(1);
            System.out.println("Generated ID: " + id);
        }
    }

    // Reuse prepared statement with new parameters
    preparedStatement.clearParameters();
    preparedStatement.setString(1, "janedoe");
    preparedStatement.setString(2, "jane@example.com");
    preparedStatement.setTimestamp(3, Timestamp.valueOf(LocalDateTime.now()));
    preparedStatement.executeUpdate();
}

// CallableStatement - for stored procedures
String callSql = "{call create_user(?, ?, ?)}";
try (Connection connection = dataSource.getConnection();
     CallableStatement callableStatement = connection.prepareCall(callSql)) {

    // Set input parameters
    callableStatement.setString(1, "newuser");
    callableStatement.setString(2, "password123");

    // Register output parameter
    callableStatement.registerOutParameter(3, Types.INTEGER);

    // Execute procedure
    callableStatement.execute();
```

```
    // Get output parameter
    int userId = callableStatement.getInt(3);
    System.out.println("Created user with ID: " + userId);
}
```

Processing ResultSet

```
// Reading data from ResultSet
String sql = "SELECT id, username, email, created_at FROM users";
try (Connection connection = dataSource.getConnection();
     Statement statement = connection.createStatement();
     ResultSet resultSet = statement.executeQuery(sql)) {

    // Get column count and names
    ResultSetMetaData metaData = resultSet.getMetaData();
    int columnCount = metaData.getColumnCount();

    for (int i = 1; i <= columnCount; i++) {
        System.out.printf("Column %d: %s (%s)%n",
            i,
            metaData.getColumnName(i),
            metaData.getColumnTypeName(i));
    }

    // Process rows
    while (resultSet.next()) {
        // Get by column index (1-based)
        long id = resultSet.getLong(1);

        // Get by column name
        String username = resultSet.getString("username");
        String email = resultSet.getString("email");

        // Handle null values
        if (resultSet.wasNull()) {
            email = "N/A";
        }

        // Get date/time
        Timestamp createdAt = resultSet.getTimestamp("created_at");
        LocalDateTime localDateTime = createdAt.toLocalDateTime();

        System.out.printf("User: %d, %s, %s, %s%n",
            id, username, email, localDateTime);
    }
}

// Scrollable ResultSet
try (Connection connection = dataSource.getConnection();
     Statement statement = connection.createStatement(
         ResultSet.TYPE_SCROLL_INSENSITIVE,  // Scrollable
```

```
             ResultSet.CONCUR_READ_ONLY)) {      // Read-only

        ResultSet resultSet = statement.executeQuery("SELECT * FROM products");

        // Move to last row
        resultSet.last();
        int rowCount = resultSet.getRow();
        System.out.println("Total rows: " + rowCount);

        // Move back to beginning
        resultSet.beforeFirst();

        // Forward iteration
        while (resultSet.next()) {
            // Process row
        }

        // Backward iteration
        while (resultSet.previous()) {
            // Process row in reverse
        }

        // Jump to specific row
        resultSet.absolute(5);  // Move to row 5
        resultSet.relative(-2);  // Move 2 rows back
    }

    // Updatable ResultSet
    try (Connection connection = dataSource.getConnection();
         Statement statement = connection.createStatement(
             ResultSet.TYPE_SCROLL_INSENSITIVE,
             ResultSet.CONCUR_UPDATABLE)) {

        ResultSet resultSet = statement.executeQuery(
            "SELECT id, name, price FROM products");

        // Update current row
        while (resultSet.next()) {
            double currentPrice = resultSet.getDouble("price");
            if (currentPrice < 10.0) {
                resultSet.updateDouble("price", currentPrice * 1.1);
                resultSet.updateRow();  // Commit the update
            }
        }

        // Insert new row
        resultSet.moveToInsertRow();
        resultSet.updateString("name", "New Product");
        resultSet.updateDouble("price", 19.99);
        resultSet.insertRow();

        // Delete row
        resultSet.absolute(3);  // Move to row 3
```

```
    resultSet.deleteRow();
}
```

Transaction Management

```
// Manual transaction management
try (Connection connection = dataSource.getConnection()) {
    try {
        // Disable auto-commit to start transaction
        connection.setAutoCommit(false);

        // Execute multiple updates within transaction
        try (Statement statement = connection.createStatement()) {
            statement.executeUpdate("UPDATE accounts SET balance = balance - 100
WHERE id = 1");
            statement.executeUpdate("UPDATE accounts SET balance = balance + 100
WHERE id = 2");
        }

        // Check business rules
        try (Statement statement = connection.createStatement();
             ResultSet resultSet = statement.executeQuery(
                 "SELECT balance FROM accounts WHERE id = 1")) {
            if (resultSet.next() && resultSet.getDouble("balance") < 0) {
                throw new SQLException("Insufficient funds");
            }
        }

        // Commit transaction
        connection.commit();
        System.out.println("Transaction committed successfully");
    } catch (Exception e) {
        // Rollback transaction on any error
        connection.rollback();
        System.err.println("Transaction rolled back: " + e.getMessage());
    } finally {
        // Restore auto-commit
        connection.setAutoCommit(true);
    }
}

// Savepoints
try (Connection connection = dataSource.getConnection()) {
    connection.setAutoCommit(false);

    try {
        // First operation
        updateAccount(connection, 1, -100);

        // Set savepoint after first operation
        Savepoint savepoint = connection.setSavepoint("AfterFirstUpdate");
```

```
            try {
                // Second operation
                updateAccount(connection, 2, 100);

                // Check if second account exists
                if (!accountExists(connection, 2)) {
                    // Rollback to savepoint - undo only second operation
                    connection.rollback(savepoint);
                    System.out.println("Rolled back to savepoint");

                    // Try alternative operation
                    updateAccount(connection, 3, 100);
                }

                // Commit transaction
                connection.commit();
            } catch (Exception e) {
                // Rollback to savepoint
                connection.rollback(savepoint);
                System.err.println("Partial rollback: " + e.getMessage());
                connection.commit();  // Commit the first operation
            }
        } catch (Exception e) {
            // Full rollback
            connection.rollback();
            System.err.println("Full rollback: " + e.getMessage());
        } finally {
            connection.setAutoCommit(true);
        }
    }

    // Helper methods
    private void updateAccount(Connection conn, int accountId, double amount)
            throws SQLException {
        String sql = "UPDATE accounts SET balance = balance + ? WHERE id = ?";
        try (PreparedStatement ps = conn.prepareStatement(sql)) {
            ps.setDouble(1, amount);
            ps.setInt(2, accountId);
            ps.executeUpdate();
        }
    }

    private boolean accountExists(Connection conn, int accountId)
            throws SQLException {
        String sql = "SELECT 1 FROM accounts WHERE id = ?";
        try (PreparedStatement ps = conn.prepareStatement(sql)) {
            ps.setInt(1, accountId);
            try (ResultSet rs = ps.executeQuery()) {
                return rs.next();
            }
        }
    }
```

Batch Processing

```
// Batch inserts with PreparedStatement
String insertSql = "INSERT INTO orders (customer_id, product_id, quantity) VALUES
(?, ?, ?)";
try (Connection connection = dataSource.getConnection();
     PreparedStatement ps = connection.prepareStatement(insertSql)) {

    // Disable auto-commit for better batch performance
    connection.setAutoCommit(false);

    // Add multiple orders to batch
    for (Order order : orders) {
        ps.setInt(1, order.getCustomerId());
        ps.setInt(2, order.getProductId());
        ps.setInt(3, order.getQuantity());
        ps.addBatch();

        // Execute every 1000 items
        if (++count % 1000 == 0) {
            ps.executeBatch();
            ps.clearBatch();
        }
    }

    // Execute remaining items
    if (count % 1000 != 0) {
        ps.executeBatch();
    }

    // Commit transaction
    connection.commit();
} catch (SQLException e) {
    if (connection != null) {
        connection.rollback();
    }
    throw e;
} finally {
    if (connection != null) {
        connection.setAutoCommit(true);
    }
}
```

Handling SQL Exceptions

```
try (Connection connection = dataSource.getConnection();
     Statement statement = connection.createStatement()) {

    statement.executeUpdate("INSERT INTO users (username) VALUES
```

```
('existing_user')");
} catch (SQLException e) {
    // Get error details
    int errorCode = e.getErrorCode();
    String sqlState = e.getSQLState();

    System.err.println("SQL Error: " + e.getMessage());
    System.err.println("Vendor Error Code: " + errorCode);
    System.err.println("SQL State: " + sqlState);

    // Handle specific errors
    if (sqlState.equals("23505") || // PostgreSQL unique violation
        errorCode == 1062) {        // MySQL duplicate entry
        System.err.println("Duplicate username detected");
    } else if (sqlState.startsWith("08")) {
        System.err.println("Database connection issue");
    }

    // Print full exception chain
    Throwable cause = e.getCause();
    while (cause != null) {
        System.err.println("Caused by: " + cause.getMessage());
        cause = cause.getCause();
    }
}
```

Working with Large Objects (BLOBs and CLOBs)

```
// Storing a BLOB (Binary Large Object)
String insertSql = "INSERT INTO documents (name, content) VALUES (?, ?)";
try (Connection connection = dataSource.getConnection();
     PreparedStatement ps = connection.prepareStatement(insertSql)) {

    ps.setString(1, "report.pdf");

    // From file
    try (FileInputStream fis = new FileInputStream("report.pdf")) {
        ps.setBinaryStream(2, fis);
        ps.executeUpdate();
    }

    // Alternative: from byte array
    byte[] documentBytes = Files.readAllBytes(Paths.get("report.pdf"));
    ps.setBytes(2, documentBytes);
    ps.executeUpdate();
}

// Retrieving a BLOB
String selectSql = "SELECT content FROM documents WHERE id = ?";
try (Connection connection = dataSource.getConnection();
     PreparedStatement ps = connection.prepareStatement(selectSql)) {
```

```
    ps.setInt(1, documentId);

    try (ResultSet rs = ps.executeQuery()) {
        if (rs.next()) {
            // Method 1: Get as InputStream
            try (InputStream is = rs.getBinaryStream("content");
                 FileOutputStream fos = new
FileOutputStream("downloaded_report.pdf")) {

                byte[] buffer = new byte[4096];
                int bytesRead;
                while ((bytesRead = is.read(buffer)) != -1) {
                    fos.write(buffer, 0, bytesRead);
                }
            }

            // Method 2: Get as byte array (for smaller BLOBs)
            byte[] content = rs.getBytes("content");
            Files.write(Paths.get("downloaded_report2.pdf"), content);

            // Method 3: Use java.sql.Blob
            Blob blob = rs.getBlob("content");
            try (InputStream is = blob.getBinaryStream();
                 FileOutputStream fos = new
FileOutputStream("downloaded_report3.pdf")) {

                byte[] buffer = new byte[4096];
                int bytesRead;
                while ((bytesRead = is.read(buffer)) != -1) {
                    fos.write(buffer, 0, bytesRead);
                }
            } finally {
                blob.free();  // Release resources
            }
        }
    }
}

// Working with CLOB (Character Large Object)
String insertClob = "INSERT INTO articles (title, content) VALUES (?, ?)";
try (Connection connection = dataSource.getConnection();
     PreparedStatement ps = connection.prepareStatement(insertClob)) {

    ps.setString(1, "Java 21 Features");

    // From character stream
    try (Reader reader = new FileReader("article.txt")) {
        ps.setCharacterStream(2, reader);
        ps.executeUpdate();
    }

    // Alternative: from String
    String content = Files.readString(Paths.get("article.txt"));
```

```
    ps.setString(2, content);
    ps.executeUpdate();
}

// Retrieving a CLOB
String selectClob = "SELECT content FROM articles WHERE id = ?";
try (Connection connection = dataSource.getConnection();
     PreparedStatement ps = connection.prepareStatement(selectClob)) {

    ps.setInt(1, articleId);

    try (ResultSet rs = ps.executeQuery()) {
        if (rs.next()) {
            // Method 1: Get as Reader
            try (Reader reader = rs.getCharacterStream("content");
                 Writer writer = new FileWriter("downloaded_article.txt")) {

                char[] buffer = new char[4096];
                int charsRead;
                while ((charsRead = reader.read(buffer)) != -1) {
                    writer.write(buffer, 0, charsRead);
                }
            }

            // Method 2: Get as String (for smaller CLOBs)
            String content = rs.getString("content");
            Files.writeString(Paths.get("downloaded_article2.txt"), content);

            // Method 3: Use java.sql.Clob
            Clob clob = rs.getClob("content");
            try (Reader reader = clob.getCharacterStream();
                 Writer writer = new FileWriter("downloaded_article3.txt")) {

                char[] buffer = new char[4096];
                int charsRead;
                while ((charsRead = reader.read(buffer)) != -1) {
                    writer.write(buffer, 0, charsRead);
                }
            } finally {
                clob.free();  // Release resources
            }
        }
    }
}
```

Working with Dates and Times

```
// Inserting dates and times
String insertSql = "INSERT INTO events (name, event_date, start_time, duration,
created_at) " +
                   "VALUES (?, ?, ?, ?, ?)";
```

```
try (Connection connection = dataSource.getConnection();
     PreparedStatement ps = connection.prepareStatement(insertSql)) {

    ps.setString(1, "Conference");

    // Set date (java.sql.Date maps to SQL DATE)
    LocalDate eventDate = LocalDate.of(2023, 10, 15);
    ps.setDate(2, java.sql.Date.valueOf(eventDate));

    // Set time (java.sql.Time maps to SQL TIME)
    LocalTime startTime = LocalTime.of(9, 30);
    ps.setTime(3, java.sql.Time.valueOf(startTime));

    // Set duration (as interval or number)
    ps.setInt(4, 120); // 120 minutes

    // Set timestamp (java.sql.Timestamp maps to SQL TIMESTAMP)
    LocalDateTime createdAt = LocalDateTime.now();
    ps.setTimestamp(5, java.sql.Timestamp.valueOf(createdAt));

    ps.executeUpdate();
}

// Retrieving dates and times
String selectSql = "SELECT event_date, start_time, duration, created_at FROM
events WHERE id = ?";
try (Connection connection = dataSource.getConnection();
     PreparedStatement ps = connection.prepareStatement(selectSql)) {

    ps.setInt(1, eventId);

    try (ResultSet rs = ps.executeQuery()) {
        if (rs.next()) {
            // Get SQL DATE as LocalDate
            java.sql.Date sqlDate = rs.getDate("event_date");
            LocalDate eventDate = sqlDate.toLocalDate();

            // Get SQL TIME as LocalTime
            java.sql.Time sqlTime = rs.getTime("start_time");
            LocalTime startTime = sqlTime.toLocalTime();

            // Get duration as integer
            int durationMinutes = rs.getInt("duration");
            Duration duration = Duration.ofMinutes(durationMinutes);

            // Get SQL TIMESTAMP as LocalDateTime
            java.sql.Timestamp sqlTimestamp = rs.getTimestamp("created_at");
            LocalDateTime createdAt = sqlTimestamp.toLocalDateTime();

            System.out.println("Event: " + eventDate + " at " + startTime +
                               ", duration: " + duration +
                               ", created: " + createdAt);
        }
    }
```

```
}

// Working with time zones
String tzSql = "INSERT INTO meetings (name, start_time) VALUES (?, ?)";
try (Connection connection = dataSource.getConnection();
     PreparedStatement ps = connection.prepareStatement(tzSql)) {

    ps.setString(1, "Global Meeting");

    // Store as timestamp with time zone
    ZonedDateTime meetingTime = ZonedDateTime.of(
        LocalDateTime.of(2023, 10, 15, 14, 0),
        ZoneId.of("America/New_York"));

    // Convert to UTC for storage (database dependent)
    Instant instant = meetingTime.toInstant();
    ps.setTimestamp(2, Timestamp.from(instant));

    ps.executeUpdate();
}
```

Advanced JDBC Features

```
// Retrieving database metadata
try (Connection connection = dataSource.getConnection()) {
    DatabaseMetaData metadata = connection.getMetaData();

    // Database information
    System.out.println("Database: " + metadata.getDatabaseProductName() +
                       " " + metadata.getDatabaseProductVersion());
    System.out.println("JDBC: " + metadata.getJDBCMajorVersion() +
                       "." + metadata.getJDBCMinorVersion());

    // Check feature support
    System.out.println("Supports transactions: " +
metadata.supportsTransactions());
    System.out.println("Supports batch updates: " +
metadata.supportsBatchUpdates());

    // Get tables
    try (ResultSet tables = metadata.getTables(null, "public", "%", new String[]
{"TABLE"})) {
        System.out.println("Tables:");
        while (tables.next()) {
            String tableName = tables.getString("TABLE_NAME");
            System.out.println("  " + tableName);

            // Get columns for this table
            try (ResultSet columns = metadata.getColumns(null, "public",
tableName, "%")) {
                while (columns.next()) {
```

```
                    String columnName = columns.getString("COLUMN_NAME");
                    String typeName = columns.getString("TYPE_NAME");
                    int columnSize = columns.getInt("COLUMN_SIZE");
                    boolean nullable = columns.getInt("NULLABLE") ==
DatabaseMetaData.columnNullable;

                    System.out.printf("    %s: %s(%d)%s%n",
                        columnName, typeName, columnSize, nullable ? " NULL" : "
NOT NULL");
                }
            }

            // Get primary keys
            try (ResultSet primaryKeys = metadata.getPrimaryKeys(null, "public",
tableName)) {
                if (primaryKeys.next()) {
                    System.out.print("    Primary Key: ");
                    do {
                        System.out.print(primaryKeys.getString("COLUMN_NAME") + "
");
                    } while (primaryKeys.next());
                    System.out.println();
                }
            }
        }
    }

    // Get procedures
    try (ResultSet procedures = metadata.getProcedures(null, "public", "%")) {
        System.out.println("Stored Procedures:");
        while (procedures.next()) {
            System.out.println("  " + procedures.getString("PROCEDURE_NAME"));
        }
    }
}

// RowSet - disconnected ResultSet (javax.sql)
import javax.sql.rowset.*;

// Connected RowSet (JdbcRowSet)
try (JdbcRowSet rowSet = RowSetProvider.newFactory().createJdbcRowSet()) {
    rowSet.setUrl(url);
    rowSet.setUsername(username);
    rowSet.setPassword(password);
    rowSet.setCommand("SELECT * FROM employees WHERE department = ?");
    rowSet.setString(1, "Engineering");
    rowSet.execute();

    while (rowSet.next()) {
        System.out.println("Employee: " + rowSet.getString("name"));
    }
}

// Disconnected RowSet (CachedRowSet)
```

```
try (CachedRowSet cachedRowSet = RowSetProvider.newFactory().createCachedRowSet())
{
    // Initialize and populate from database
    cachedRowSet.setUrl(url);
    cachedRowSet.setUsername(username);
    cachedRowSet.setPassword(password);
    cachedRowSet.setCommand("SELECT id, name, salary FROM employees");
    cachedRowSet.execute();

    // Work with data while disconnected
    while (cachedRowSet.next()) {
        double currentSalary = cachedRowSet.getDouble("salary");
        if (currentSalary < 50000) {
            cachedRowSet.updateDouble("salary", currentSalary * 1.1);
            cachedRowSet.updateRow();
        }
    }

    // Reconnect and synchronize changes
    cachedRowSet.acceptChanges(dataSource.getConnection());
}
```

Connection Pooling

```
// Using HikariCP (the most popular connection pool)
import com.zaxxer.hikari.HikariConfig;
import com.zaxxer.hikari.HikariDataSource;

// Set up the connection pool
HikariConfig config = new HikariConfig();
config.setJdbcUrl("jdbc:mysql://localhost:3306/mydb");
config.setUsername("user");
config.setPassword("password");

// Pool configuration
config.setMaximumPoolSize(10);
config.setMinimumIdle(5);
config.setIdleTimeout(30000);
config.setMaxLifetime(1800000);
config.setConnectionTimeout(5000);
config.setPoolName("MyHikariPool");

// Optional: Connection testing
config.setConnectionTestQuery("SELECT 1");
config.setValidationTimeout(5000);

// Create the DataSource
HikariDataSource dataSource = new HikariDataSource(config);

// Using the pool
try (Connection connection = dataSource.getConnection()) {
```

```
    // Use connection normally
    // Connection automatically returns to pool on close()
}

// Close the pool when application stops
dataSource.close();

// Using JNDI in an application server
import javax.naming.InitialContext;
import javax.sql.DataSource;

// Look up DataSource from JNDI
InitialContext ctx = new InitialContext();
DataSource jndiDataSource = (DataSource) ctx.lookup("java:comp/env/jdbc/MyDB");

// Get connection from the pool
try (Connection conn = jndiDataSource.getConnection()) {
    // Use connection
}
```

> **Common Pitfall**: Not closing JDBC resources (Connections, Statements, ResultSets) can lead to serious connection leaks and database performance issues.

> **Certification Note**: Understand the core JDBC components, including DriverManager, Connection, Statement types, and ResultSet. Know how to handle database transactions and work with connection pools.

JPA (Java Persistence API)

JPA Architecture

JPA is a specification for object-relational mapping (ORM) in Java. It provides a higher-level abstraction over JDBC:

1. **Entity Manager Factory**: Creates entity managers
2. **Entity Manager**: Core interface for persistence operations
3. **Persistence Context**: Set of managed entity instances
4. **Entity**: Mapped Java class representing database table
5. **Query**: Interface for executing database queries
6. **Criteria API**: Type-safe way to build queries
7. **Persistence Provider**: Implementation of JPA (e.g., Hibernate, EclipseLink)

Setting Up JPA with Hibernate

```

// Maven dependencies
// <dependency>
//     <groupId>org.hibernate</groupId>
//     <artifactId>hibernate-core</artifactId>
//     <version>6.2.0.Final</version>
// </dependency>
```

```
// <dependency>
//     <groupId>jakarta.persistence</groupId>
//     <artifactId>jakarta.persistence-api</artifactId>
//     <version>3.1.0</version>
// </dependency>

// persistence.xml configuration (src/main/resources/META-INF/persistence.xml)
<persistence xmlns="https://jakarta.ee/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence

https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd"
             version="3.0">
    <persistence-unit name="MyPersistenceUnit" transaction-type="RESOURCE_LOCAL">
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
        <class>com.example.entity.Employee</class>
        <class>com.example.entity.Department</class>
        <properties>
            <!-- Database connection -->
            <property name="jakarta.persistence.jdbc.driver"
value="com.mysql.cj.jdbc.Driver"/>
            <property name="jakarta.persistence.jdbc.url"
value="jdbc:mysql://localhost:3306/mydb"/>
            <property name="jakarta.persistence.jdbc.user" value="user"/>
            <property name="jakarta.persistence.jdbc.password" value="password"/>

            <!-- Hibernate properties -->
            <property name="hibernate.dialect"
value="org.hibernate.dialect.MySQLDialect"/>
            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.format_sql" value="true"/>
            <property name="hibernate.hbm2ddl.auto" value="update"/>

            <!-- Connection pooling -->
            <property name="hibernate.connection.provider_class"

value="org.hibernate.hikaricp.internal.HikariCPConnectionProvider"/>
            <property name="hibernate.hikari.maximumPoolSize" value="10"/>
        </properties>
    </persistence-unit>
</persistence>
```

Entity Mapping

```
import jakarta.persistence.*;
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;

@Entity
@Table(name = "employees")
```

```
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "first_name", nullable = false, length = 50)
    private String firstName;

    @Column(name = "last_name", nullable = false, length = 50)
    private String lastName;

    @Column(unique = true)
    private String email;

    // Basic types
    private int age;
    private double salary;
    private boolean active;

    // Date/Time
    @Column(name = "hire_date")
    private LocalDate hireDate;

    // Enumerated type
    @Enumerated(EnumType.STRING)
    @Column(name = "employment_type")
    private EmploymentType employmentType;

    // Embedded object
    @Embedded
    private Address address;

    // Many-to-one relationship
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "department_id")
    private Department department;

    // One-to-many relationship
    @OneToMany(mappedBy = "employee", cascade = CascadeType.ALL, orphanRemoval =
true)
    private List<SkillCertification> certifications = new ArrayList<>();

    // Large objects
    @Lob
    @Column(columnDefinition = "TEXT")
    private String biography;

    @Lob
    @Column(columnDefinition = "BLOB")
    private byte[] photo;

    // Transient (not persisted)
    @Transient
    private int yearsOfService;
```

```
    // Version for optimistic locking
    @Version
    private Long version;

    // Constructors, getters, setters, etc.

    // Convenience method for bidirectional relationship
    public void addCertification(SkillCertification certification) {
        certifications.add(certification);
        certification.setEmployee(this);
    }

    public void removeCertification(SkillCertification certification) {
        certifications.remove(certification);
        certification.setEmployee(null);
    }

    // Enum class for employment type
    public enum EmploymentType {
        FULL_TIME, PART_TIME, CONTRACT, INTERN
    }
}

// Embeddable class
@Embeddable
public class Address {
    private String street;
    private String city;

    @Column(name = "zip_code", length = 10)
    private String zipCode;

    // Constructors, getters, setters
}

// Department entity
@Entity
@Table(name = "departments")
public class Department {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, unique = true)
    private String name;

    @OneToMany(mappedBy = "department")
    private List<Employee> employees = new ArrayList<>();

    // Constructors, getters, setters
}

// Skill certification entity
```

```
@Entity
@Table(name = "skill_certifications")
public class SkillCertification {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String name;

    @Column(name = "certification_date")
    private LocalDate certificationDate;

    @ManyToOne
    @JoinColumn(name = "employee_id")
    private Employee employee;

    // Constructors, getters, setters
}
```

EntityManager Operations

```
import jakarta.persistence.*;
import java.util.List;

// Creating EntityManagerFactory (once for application)
EntityManagerFactory emf =
Persistence.createEntityManagerFactory("MyPersistenceUnit");

// Basic CRUD operations
try {
    // Create EntityManager
    EntityManager em = emf.createEntityManager();

    try {
        // Start a transaction
        em.getTransaction().begin();

        // CREATE - Persist a new entity
        Employee employee = new Employee();
        employee.setFirstName("John");
        employee.setLastName("Doe");
        employee.setEmail("john.doe@example.com");
        employee.setAge(35);
        employee.setSalary(75000.0);
        employee.setActive(true);
        employee.setHireDate(LocalDate.of(2020, 5, 15));
        employee.setEmploymentType(Employee.EmploymentType.FULL_TIME);

        // Set embedded object
        Address address = new Address();
```

```
            address.setStreet("123 Main St");
            address.setCity("Springfield");
            address.setZipCode("12345");
            employee.setAddress(address);

            // Persist
            em.persist(employee);

            // Commit the transaction
            em.getTransaction().commit();
            System.out.println("Employee created with ID: " + employee.getId());

            // Start a new transaction
            em.getTransaction().begin();

            // READ - Find by primary key
            Employee foundEmployee = em.find(Employee.class, employee.getId());
            System.out.println("Found: " + foundEmployee.getFirstName() +
                               " " + foundEmployee.getLastName());

            // UPDATE - Modify an entity
            foundEmployee.setSalary(80000.0);

            // No need to call persist() on a managed entity
            // Changes are tracked automatically

            // DELETE - Remove an entity
            // em.remove(foundEmployee);

            // Commit the transaction
            em.getTransaction().commit();
        } catch (Exception e) {
            // Rollback on error
            if (em.getTransaction().isActive()) {
                em.getTransaction().rollback();
            }
            throw e;
        } finally {
            // Close the EntityManager
            em.close();
        }
    } finally {
        // Close the EntityManagerFactory when application ends
        emf.close();
    }
```

JPQL (JPA Query Language)

```
// Simple query
EntityManager em = emf.createEntityManager();
TypedQuery<Employee> query = em.createQuery(
```

```
    "SELECT e FROM Employee e WHERE e.salary > :minSalary", Employee.class);
query.setParameter("minSalary", 70000.0);
List<Employee> highPaidEmployees = query.getResultList();

// Single result
TypedQuery<Employee> singleQuery = em.createQuery(
    "SELECT e FROM Employee e WHERE e.email = :email", Employee.class);
singleQuery.setParameter("email", "john.doe@example.com");
try {
    Employee employee = singleQuery.getSingleResult();
    System.out.println("Found: " + employee.getFirstName());
} catch (NoResultException e) {
    System.out.println("Employee not found");
} catch (NonUniqueResultException e) {
    System.out.println("More than one employee found");
}

// JOIN with relationship
TypedQuery<Employee> joinQuery = em.createQuery(
    "SELECT e FROM Employee e JOIN e.department d WHERE d.name = :deptName",
    Employee.class);
joinQuery.setParameter("deptName", "Engineering");
List<Employee> engineeringEmployees = joinQuery.getResultList();

// Projection (specific fields)
TypedQuery<Object[]> projectionQuery = em.createQuery(
    "SELECT e.id, e.firstName, e.lastName FROM Employee e WHERE e.active = true",
    Object[].class);
List<Object[]> activeEmployeeDetails = projectionQuery.getResultList();
for (Object[] result : activeEmployeeDetails) {
    Long id = (Long) result[0];
    String firstName = (String) result[1];
    String lastName = (String) result[2];
    System.out.println(id + ": " + firstName + " " + lastName);
}

// Using constructor expressions
TypedQuery<EmployeeDTO> dtoQuery = em.createQuery(
    "SELECT new com.example.dto.EmployeeDTO(e.id, e.firstName, e.lastName,
e.salary) " +
    "FROM Employee e", EmployeeDTO.class);
List<EmployeeDTO> employeeDTOs = dtoQuery.getResultList();

// Aggregate functions
TypedQuery<Double> avgQuery = em.createQuery(
    "SELECT AVG(e.salary) FROM Employee e WHERE e.department.id = :deptId",
    Double.class);
avgQuery.setParameter("deptId", 1L);
Double averageSalary = avgQuery.getSingleResult();

// GROUP BY and HAVING
TypedQuery<Object[]> groupQuery = em.createQuery(
    "SELECT d.name, COUNT(e), AVG(e.salary) " +
    "FROM Department d JOIN d.employees e " +
```

```
    "GROUP BY d.name " +
    "HAVING COUNT(e) > 5",
    Object[].class);
List<Object[]> departmentStats = groupQuery.getResultList();
for (Object[] result : departmentStats) {
    System.out.println("Department: " + result[0] +
                       ", Employees: " + result[1] +
                       ", Avg Salary: " + result[2]);
}

// ORDER BY
TypedQuery<Employee> orderQuery = em.createQuery(
    "SELECT e FROM Employee e ORDER BY e.salary DESC", Employee.class);
List<Employee> sortedEmployees = orderQuery.getResultList();

// Pagination
TypedQuery<Employee> pagedQuery = em.createQuery(
    "SELECT e FROM Employee e ORDER BY e.id", Employee.class);
pagedQuery.setFirstResult(0);  // Skip 0 results (first page)
pagedQuery.setMaxResults(10);  // Get 10 results per page
List<Employee> firstPageEmployees = pagedQuery.getResultList();

// UPDATE query
em.getTransaction().begin();
int updatedCount = em.createQuery(
    "UPDATE Employee e SET e.active = false WHERE e.hireDate < :date")
    .setParameter("date", LocalDate.of(2010, 1, 1))
    .executeUpdate();
em.getTransaction().commit();
System.out.println("Updated " + updatedCount + " employees");

// DELETE query
em.getTransaction().begin();
int deletedCount = em.createQuery(
    "DELETE FROM SkillCertification c WHERE c.certificationDate < :date")
    .setParameter("date", LocalDate.of(2015, 1, 1))
    .executeUpdate();
em.getTransaction().commit();
System.out.println("Deleted " + deletedCount + " certifications");
```

Criteria API

```
import jakarta.persistence.criteria.*;

// Setup
CriteriaBuilder cb = em.getCriteriaBuilder();

// Simple query
CriteriaQuery<Employee> cq = cb.createQuery(Employee.class);
Root<Employee> employee = cq.from(Employee.class);
cq.select(employee);
```

```
List<Employee> allEmployees = em.createQuery(cq).getResultList();

// Query with WHERE clause
CriteriaQuery<Employee> query = cb.createQuery(Employee.class);
Root<Employee> e = query.from(Employee.class);
query.select(e)
     .where(cb.equal(e.get("employmentType"), Employee.EmploymentType.FULL_TIME));
List<Employee> fullTimeEmployees = em.createQuery(query).getResultList();

// Multiple conditions
CriteriaQuery<Employee> complexQuery = cb.createQuery(Employee.class);
Root<Employee> emp = complexQuery.from(Employee.class);
Predicate salaryCondition = cb.greaterThan(emp.get("salary"), 50000.0);
Predicate activeCondition = cb.equal(emp.get("active"), true);
complexQuery.select(emp)
            .where(cb.and(salaryCondition, activeCondition));
List<Employee> employees = em.createQuery(complexQuery).getResultList();

// Join
CriteriaQuery<Employee> joinQuery = cb.createQuery(Employee.class);
Root<Employee> employeeRoot = joinQuery.from(Employee.class);
Join<Employee, Department> departmentJoin = employeeRoot.join("department");
joinQuery.select(employeeRoot)
         .where(cb.equal(departmentJoin.get("name"), "Engineering"));
List<Employee> engineeringEmployees = em.createQuery(joinQuery).getResultList();

// Projection
CriteriaQuery<Object[]> projectionQuery = cb.createQuery(Object[].class);
Root<Employee> empRoot = projectionQuery.from(Employee.class);
projectionQuery.multiselect(
    empRoot.get("id"),
    empRoot.get("firstName"),
    empRoot.get("lastName")
);
List<Object[]> employeeDetails = em.createQuery(projectionQuery).getResultList();

// Aggregate function
CriteriaQuery<Double> avgQuery = cb.createQuery(Double.class);
Root<Employee> avgRoot = avgQuery.from(Employee.class);
avgQuery.select(cb.avg(avgRoot.get("salary")));
Double avgSalary = em.createQuery(avgQuery).getSingleResult();

// Group By
CriteriaQuery<Object[]> groupQuery = cb.createQuery(Object[].class);
Root<Employee> groupRoot = groupQuery.from(Employee.class);
Join<Employee, Department> deptJoin = groupRoot.join("department");
groupQuery.multiselect(
    deptJoin.get("name"),
    cb.count(groupRoot),
    cb.avg(groupRoot.get("salary"))
);
groupQuery.groupBy(deptJoin.get("name"));
groupQuery.having(cb.gt(cb.count(groupRoot), 5L));
List<Object[]> deptStats = em.createQuery(groupQuery).getResultList();
```

```
// Order By
CriteriaQuery<Employee> orderQuery = cb.createQuery(Employee.class);
Root<Employee> orderRoot = orderQuery.from(Employee.class);
orderQuery.select(orderRoot);
orderQuery.orderBy(cb.desc(orderRoot.get("salary")));
List<Employee> sortedEmployees = em.createQuery(orderQuery).getResultList();

// Subqueries
CriteriaQuery<Employee> subQuery = cb.createQuery(Employee.class);
Root<Employee> subRoot = subQuery.from(Employee.class);

// Create subquery to find average salary
Subquery<Double> sq = subQuery.subquery(Double.class);
Root<Employee> sqRoot = sq.from(Employee.class);
sq.select(cb.avg(sqRoot.get("salary")));

// Main query: find employees with salary > average
subQuery.select(subRoot)
        .where(cb.gt(subRoot.get("salary"), sq));
List<Employee> aboveAvgEmployees = em.createQuery(subQuery).getResultList();
```

Named Queries

```
// Define named queries in the entity class

@Entity
@NamedQueries({
    @NamedQuery(
        name = "Employee.findAll",
        query = "SELECT e FROM Employee e"
    ),
    @NamedQuery(
        name = "Employee.findByDepartment",
        query = "SELECT e FROM Employee e WHERE e.department.id = :deptId"
    ),
    @NamedQuery(
        name = "Employee.updateSalary",
        query = "UPDATE Employee e SET e.salary = e.salary * :factor " +
                "WHERE e.department.id = :deptId"
    )
})
public class Employee {
    // Class definition...
}

// Using named queries
TypedQuery<Employee> allQuery = em.createNamedQuery("Employee.findAll",
Employee.class);
List<Employee> allEmployees = allQuery.getResultList();
```

```
TypedQuery<Employee> deptQuery = em.createNamedQuery("Employee.findByDepartment",
Employee.class);
deptQuery.setParameter("deptId", 1L);
List<Employee> deptEmployees = deptQuery.getResultList();

// Update using named query
em.getTransaction().begin();
int updated = em.createNamedQuery("Employee.updateSalary")
               .setParameter("factor", 1.1)  // 10% increase
               .setParameter("deptId", 1L)
               .executeUpdate();
em.getTransaction().commit();
```

Native SQL Queries

```
// Native SQL (when JPA queries are insufficient)
Query nativeQuery = em.createNativeQuery(
    "SELECT e.id, e.first_name, e.last_name, d.name as dept_name " +
    "FROM employees e " +
    "JOIN departments d ON e.department_id = d.id " +
    "WHERE e.hire_date > ?");
nativeQuery.setParameter(1, java.sql.Date.valueOf(LocalDate.of(2020, 1, 1)));
List<Object[]> results = nativeQuery.getResultList();
for (Object[] result : results) {
    System.out.println("ID: " + result[0] +
                       ", Name: " + result[1] + " " + result[2] +
                       ", Department: " + result[3]);
}

// Native SQL mapped to entity
Query entityNativeQuery = em.createNativeQuery(
    "SELECT * FROM employees WHERE salary > :minSalary", Employee.class);
entityNativeQuery.setParameter("minSalary", 60000.0);
List<Employee> highPaidEmployees = entityNativeQuery.getResultList();

// Complex SQL with result set mapping
@SqlResultSetMapping(
    name = "EmployeeWithDepartmentMapping",
    entities = {
        @EntityResult(
            entityClass = Employee.class,
            fields = {
                @FieldResult(name = "id", column = "emp_id"),
                @FieldResult(name = "firstName", column = "first_name"),
                @FieldResult(name = "lastName", column = "last_name"),
                @FieldResult(name = "salary", column = "salary")
            }
        )
    },
    columns = {
        @ColumnResult(name = "dept_name")
```

```
    }
)

// Using the mapping
Query mappedQuery = em.createNativeQuery(
    "SELECT e.id as emp_id, e.first_name, e.last_name, e.salary, " +
    "d.name as dept_name " +
    "FROM employees e " +
    "JOIN departments d ON e.department_id = d.id " +
    "WHERE e.salary > :minSalary",
    "EmployeeWithDepartmentMapping");
mappedQuery.setParameter("minSalary", 70000.0);
List<Object[]> mappedResults = mappedQuery.getResultList();
```

Entity Lifecycle and Callbacks

```
@Entity
public class AuditedEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    // Other fields...

    private LocalDateTime createdAt;
    private LocalDateTime updatedAt;
    private String createdBy;
    private String updatedBy;

    // Lifecycle callbacks
    @PrePersist
    public void onPrePersist() {
        createdAt = LocalDateTime.now();
        updatedAt = createdAt;
        createdBy = getCurrentUser();
        updatedBy = createdBy;

        System.out.println("Entity is about to be persisted: " + this);
    }

    @PostPersist
    public void onPostPersist() {
        System.out.println("Entity was persisted with ID: " + id);
    }

    @PreUpdate
    public void onPreUpdate() {
        updatedAt = LocalDateTime.now();
        updatedBy = getCurrentUser();

        System.out.println("Entity is about to be updated: " + this);
```

```
    }

    @PostUpdate
    public void onPostUpdate() {
        System.out.println("Entity was updated: " + this);
    }

    @PreRemove
    public void onPreRemove() {
        System.out.println("Entity is about to be deleted: " + this);
    }

    @PostRemove
    public void onPostRemove() {
        System.out.println("Entity was deleted: " + this);
    }

    @PostLoad
    public void onPostLoad() {
        System.out.println("Entity was loaded: " + this);
    }

    private String getCurrentUser() {
        // In a real application, this would get the current user
        return "system";
    }
}
```

Entity Relationships

```
// ONE-TO-ONE Relationship
@Entity
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    // Other fields...

    // Owning side of one-to-one relationship
    @OneToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    @JoinColumn(name = "parking_spot_id", unique = true)
    private ParkingSpot parkingSpot;

    // Getters and setters...
}

@Entity
@Table(name = "parking_spots")
public class ParkingSpot {
    @Id
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String spotNumber;

    // Inverse side of one-to-one relationship
    @OneToOne(mappedBy = "parkingSpot")
    private Employee employee;

    // Getters and setters...
}

// ONE-TO-MANY / MANY-TO-ONE Relationship
@Entity
@Table(name = "departments")
public class Department {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    // One-to-many relationship
    @OneToMany(mappedBy = "department", cascade = CascadeType.ALL, orphanRemoval =
true)
    private List<Employee> employees = new ArrayList<>();

    // Convenience methods
    public void addEmployee(Employee employee) {
        employees.add(employee);
        employee.setDepartment(this);
    }

    public void removeEmployee(Employee employee) {
        employees.remove(employee);
        employee.setDepartment(null);
    }

    // Getters and setters...
}

@Entity
@Table(name = "employees")
public class Employee {
    @Id
    @GeneratedValue
    private Long id;

    // Other fields...

    // Many-to-one relationship
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "department_id")
    private Department department;
```

```
    // Getters and setters...
}

// MANY-TO-MANY Relationship
@Entity
@Table(name = "employees")
public class Employee {
    @Id
    @GeneratedValue
    private Long id;

    // Other fields...

    // Many-to-many relationship
    @ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
    @JoinTable(
        name = "employee_project",
        joinColumns = @JoinColumn(name = "employee_id"),
        inverseJoinColumns = @JoinColumn(name = "project_id")
    )
    private Set<Project> projects = new HashSet<>();

    // Convenience methods
    public void addProject(Project project) {
        projects.add(project);
        project.getEmployees().add(this);
    }

    public void removeProject(Project project) {
        projects.remove(project);
        project.getEmployees().remove(this);
    }

    // Getters and setters...
}

@Entity
@Table(name = "projects")
public class Project {
    @Id
    @GeneratedValue
    private Long id;

    private String name;

    // Many-to-many relationship (inverse side)
    @ManyToMany(mappedBy = "projects")
    private Set<Employee> employees = new HashSet<>();

    // Getters and setters...
}

// Many-to-many with attributes (using join entity)
```

```java
@Entity
@Table(name = "employees")
public class Employee {
    @Id
    @GeneratedValue
    private Long id;

    // Other fields...

    // One-to-many to join entity
    @OneToMany(mappedBy = "employee", cascade = CascadeType.ALL, orphanRemoval =
true)
    private Set<EmployeeProject> employeeProjects = new HashSet<>();

    // Convenience methods...
}

@Entity
@Table(name = "projects")
public class Project {
    @Id
    @GeneratedValue
    private Long id;

    private String name;

    // One-to-many to join entity
    @OneToMany(mappedBy = "project", cascade = CascadeType.ALL, orphanRemoval =
true)
    private Set<EmployeeProject> employeeProjects = new HashSet<>();

    // Convenience methods...
}

@Entity
@Table(name = "employee_project")
public class EmployeeProject {
    @EmbeddedId
    private EmployeeProjectId id;

    @ManyToOne(fetch = FetchType.LAZY)
    @MapsId("employeeId")
    private Employee employee;

    @ManyToOne(fetch = FetchType.LAZY)
    @MapsId("projectId")
    private Project project;

    private LocalDate assignedDate;
    private String role;

    // Constructors, getters, setters...
}
```

```
@Embeddable
public class EmployeeProjectId implements Serializable {
    private Long employeeId;
    private Long projectId;

    // Constructors, equals, hashCode...
}
```

Inheritance Strategies

```
// 1. Single Table Strategy (default)
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "vehicle_type", discriminatorType =
DiscriminatorType.STRING)
public abstract class Vehicle {
    @Id
    @GeneratedValue
    private Long id;

    private String manufacturer;
    private String model;
    private int year;

    // Common fields and methods...
}

@Entity
@DiscriminatorValue("CAR")
public class Car extends Vehicle {
    private int numberOfDoors;
    private String fuelType;

    // Car-specific fields and methods...
}

@Entity
@DiscriminatorValue("TRUCK")
public class Truck extends Vehicle {
    private double payloadCapacity;
    private int numberOfAxles;

    // Truck-specific fields and methods...
}

// 2. Joined Table Strategy
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Payment {
    @Id
    @GeneratedValue
```

```
    private Long id;

    private double amount;
    private LocalDateTime paymentDate;

    // Common fields and methods...
}

@Entity
@Table(name = "credit_card_payments")
public class CreditCardPayment extends Payment {
    private String cardNumber;
    private String cardHolderName;
    private LocalDate expirationDate;

    // Credit card specific fields and methods...
}

@Entity
@Table(name = "bank_transfer_payments")
public class BankTransferPayment extends Payment {
    private String accountNumber;
    private String bankName;
    private String referenceNumber;

    // Bank transfer specific fields and methods...
}

// 3. Table Per Class Strategy
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Person {
    @Id
    @GeneratedValue
    private Long id;

    private String name;
    private LocalDate birthDate;

    // Common fields and methods...
}

@Entity
@Table(name = "employees")
public class Employee extends Person {
    private String employeeNumber;
    private String department;
    private double salary;

    // Employee-specific fields and methods...
}

@Entity
@Table(name = "customers")
```

```
public class Customer extends Person {
    private String customerNumber;
    private String address;
    private double creditScore;

    // Customer-specific fields and methods...
}

// 4. Mapped Superclass (not an entity)
@MappedSuperclass
public abstract class BaseEntity {
    @Id
    @GeneratedValue
    private Long id;

    @Column(name = "created_at")
    private LocalDateTime createdAt;

    @Column(name = "updated_at")
    private LocalDateTime updatedAt;

    @PrePersist
    public void prePersist() {
        createdAt = LocalDateTime.now();
        updatedAt = createdAt;
    }

    @PreUpdate
    public void preUpdate() {
        updatedAt = LocalDateTime.now();
    }

    // Getters and setters...
}

@Entity
@Table(name = "products")
public class Product extends BaseEntity {
    private String name;
    private double price;

    // Product-specific fields and methods...
}
```

Transaction Management

```
// Explicit transaction management
EntityManagerFactory emf =
Persistence.createEntityManagerFactory("MyPersistenceUnit");
EntityManager em = emf.createEntityManager();
```

```
try {
    // Start transaction
    em.getTransaction().begin();

    // Perform operations
    Department dept = em.find(Department.class, 1L);
    Employee emp = new Employee();
    emp.setFirstName("Jane");
    emp.setLastName("Smith");
    emp.setDepartment(dept);
    em.persist(emp);

    // Commit transaction
    em.getTransaction().commit();
} catch (Exception e) {
    // Rollback on error
    if (em.getTransaction().isActive()) {
        em.getTransaction().rollback();
    }
    throw e;
} finally {
    // Close resources
    em.close();
    emf.close();
}

// Using transaction boundary methods
public Employee createEmployee(Employee employee) {
    EntityManager em = emf.createEntityManager();
    EntityTransaction tx = em.getTransaction();

    try {
        tx.begin();
        em.persist(employee);
        tx.commit();
        return employee;
    } catch (Exception e) {
        if (tx != null && tx.isActive()) {
            tx.rollback();
        }
        throw e;
    } finally {
        em.close();
    }
}

// Integration with Jakarta Transactions API (JTA)
@PersistenceContext
private EntityManager em;

@Inject
private UserTransaction userTransaction;

public void transferBetweenDepartments(Long employeeId, Long newDepartmentId)
```

```
    throws Exception {

    userTransaction.begin();

    try {
        Employee employee = em.find(Employee.class, employeeId);
        Department newDept = em.find(Department.class, newDepartmentId);

        // Update employee
        employee.setDepartment(newDept);

        // Additional complex operations...

        userTransaction.commit();
    } catch (Exception e) {
        userTransaction.rollback();
        throw e;
    }
}
```

EntityManager Caching

```
// First-level (Persistence Context) Cache
EntityManager em = emf.createEntityManager();

// First find - hits the database
Employee emp1 = em.find(Employee.class, 1L);

// Second find - retrieves from persistence context cache
Employee emp2 = em.find(Employee.class, 1L);
System.out.println("Same reference: " + (emp1 == emp2));  // true

// Clear cache
em.clear();

// After clear - hits the database again
Employee emp3 = em.find(Employee.class, 1L);
System.out.println("Same reference after clear: " + (emp1 == emp3));  // false

// Detach specific entity
em.detach(emp3);

// After detach - hits the database again
Employee emp4 = em.find(Employee.class, 1L);
System.out.println("Same reference after detach: " + (emp3 == emp4));  // false

// Second-level Cache (shared across EntityManagers)
// Configuration in persistence.xml
<property name="hibernate.cache.use_second_level_cache" value="true"/>
<property name="hibernate.cache.region.factory_class"
          value="org.hibernate.cache.jcache.JCacheRegionFactory"/>
```

```
<property name="hibernate.javax.cache.provider"
          value="org.ehcache.jsr107.EhcacheCachingProvider"/>

// Entity configured for second-level cache
@Entity
@Cacheable
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
public class Department {
    // Entity definition...
}

// Using the cache
EntityManager em1 = emf.createEntityManager();
Department dept1 = em1.find(Department.class, 1L);  // Database hit
em1.close();

EntityManager em2 = emf.createEntityManager();
Department dept2 = em2.find(Department.class, 1L);  // Retrieved from second-level
cache
em2.close();

// Query cache
em.createQuery("SELECT d FROM Department d WHERE d.name = :name")
  .setParameter("name", "HR")
  .setHint("org.hibernate.cacheable", "true")
  .getResultList();
```

> **Common Pitfall**: Misunderstanding the difference between managed and detached entities. Changes to managed entities are automatically persisted during transaction commit, but changes to detached entities are not.

> **Certification Note**: Understand the core JPA concepts, including entity mapping, relationships, inheritance strategies, and JPQL. Know how to manage the EntityManager lifecycle and transactions.

Testing in Java

JUnit 5 Framework

Setting Up JUnit 5

```
// Maven dependencies
// <dependencies>
//     <dependency>
//         <groupId>org.junit.jupiter</groupId>
//         <artifactId>junit-jupiter</artifactId>
//         <version>5.9.2</version>
//         <scope>test</scope>
//     </dependency>
// </dependencies>
```

```
// Gradle dependencies
// testImplementation 'org.junit.jupiter:junit-jupiter:5.9.2'
// testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.9.2'
```

Basic Test Structure

```
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;

public class CalculatorTest {

    private Calculator calculator;

    @BeforeEach
    void setUp() {
        // This runs before each test
        calculator = new Calculator();
    }

    @AfterEach
    void tearDown() {
        // This runs after each test
        calculator = null;
    }

    @BeforeAll
    static void setUpAll() {
        // This runs once before all tests in this class
        System.out.println("Starting Calculator tests");
    }

    @AfterAll
    static void tearDownAll() {
        // This runs once after all tests in this class
        System.out.println("Finished Calculator tests");
    }

    @Test
    void testAddition() {
        // Given
        int a = 5;
        int b = 3;

        // When
        int result = calculator.add(a, b);

        // Then
        assertEquals(8, result, "5 + 3 should equal 8");
    }

    @Test
```

```
    void testDivision() {
        // Given
        int a = 10;
        int b = 2;

        // When
        int result = calculator.divide(a, b);

        // Then
        assertEquals(5, result);
    }

    @Test
    void testDivisionByZero() {
        // Given
        int a = 10;
        int b = 0;

        // Then
        assertThrows(ArithmeticException.class, () -> {
            // When
            calculator.divide(a, b);
        });
    }
}

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

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

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

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

Assertions

```
import static org.junit.jupiter.api.Assertions.*;

@Test
void testAssertions() {
```

```
        // Simple assertions
        assertEquals(4, 2 + 2);
        assertEquals(4, 2 + 2, "Optional failure message");
        assertNotEquals(5, 2 + 2);

        // Boolean assertions
        assertTrue(3 > 2);
        assertFalse(2 > 3);

        // Null checks
        String str = null;
        assertNull(str);

        str = "Hello";
        assertNotNull(str);

        // Same object check
        String s1 = "Hello";
        String s2 = s1;
        assertSame(s1, s2);

        String s3 = new String("Hello");
        assertNotSame(s1, s3);

        // Array equality
        assertArrayEquals(new int[] {1, 2, 3}, new int[] {1, 2, 3});

        // Floating point comparisons (with delta)
        assertEquals(0.1 + 0.2, 0.3, 0.000001);

        // Exception testing
        Exception exception = assertThrows(ArithmeticException.class, () -> {
            int result = 1 / 0;
        });
        assertEquals("/ by zero", exception.getMessage());

        // Asserting no exception is thrown
        assertDoesNotThrow(() -> {
            int result = 1 / 1;
        });

        // Failing a test
        // fail("Not implemented yet");

        // Grouped assertions (all executed, failures grouped)
        assertAll("address",
            () -> assertEquals("John", person.getFirstName()),
            () -> assertEquals("Doe", person.getLastName()),
            () -> assertEquals(30, person.getAge())
        );

        // Timeout assertion
        assertTimeout(Duration.ofMillis(100), () -> {
            // Code that should execute in less than 100ms
```

```
            Thread.sleep(50);
        });

        // Timeout assertion that preemptively aborts
        assertTimeoutPreemptively(Duration.ofMillis(100), () -> {
            // Code that should execute in less than 100ms
            Thread.sleep(50);
        });
    }
```

Test Lifecycle and Organization

```
// Test lifecycle annotations
@BeforeAll     // Executed once before all test methods (must be static)
@AfterAll      // Executed once after all test methods (must be static)
@BeforeEach    // Executed before each test method
@AfterEach     // Executed after each test method
@Test          // Marks a method as a test
@Disabled      // Disables a test (skipped during execution)
@DisplayName   // Provides a custom name for a test method or class

// Test organization
@Nested        // Indicates a nested test class (for hierarchical tests)
@Tag           // Tags tests for filtering (e.g., "unit", "integration")
@TestMethodOrder // Specifies the order of test methods
@TestInstance  // Defines test instance lifecycle (PER_METHOD or PER_CLASS)

// Example of nested tests
@DisplayName("Stack Tests")
class StackTest {

    Stack<String> stack;

    @BeforeEach
    void setUp() {
        stack = new Stack<>();
    }

    @Nested
    @DisplayName("when empty")
    class WhenEmpty {

        @Test
        @DisplayName("is empty")
        void isEmpty() {
            assertTrue(stack.isEmpty());
        }

        @Test
        @DisplayName("throws EmptyStackException on pop")
        void throwsExceptionOnPop() {
```

```
            assertThrows(EmptyStackException.class, () -> stack.pop());
        }
    }

    @Nested
    @DisplayName("when one element")
    class WhenOneElement {

        @BeforeEach
        void addElement() {
            stack.push("sample");
        }

        @Test
        @DisplayName("is not empty")
        void isNotEmpty() {
            assertFalse(stack.isEmpty());
        }

        @Test
        @DisplayName("returns element on pop")
        void returnElementOnPop() {
            assertEquals("sample", stack.pop());
        }
    }
}

// Test method ordering
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
class OrderedTest {

    @Test
    @Order(1)
    void firstTest() {
        // Executed first
    }

    @Test
    @Order(2)
    void secondTest() {
        // Executed second
    }

    @Test
    @Order(3)
    void thirdTest() {
        // Executed third
    }
}

// Test instance lifecycle
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
class SharedStateTest {
```

```
    private int counter = 0;

    // With PER_CLASS, @BeforeAll and @AfterAll don't need to be static
    @BeforeAll
    void setUp() {
        counter = 0;
    }

    @Test
    void firstTest() {
        counter++;
        assertEquals(1, counter);
    }

    @Test
    void secondTest() {
        counter++;
        assertEquals(2, counter);  // Passes with PER_CLASS, fails with PER_METHOD
    }
}
```

Parameterized Tests

```
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.*;

class ParameterizedTests {

    // Simple value source
    @ParameterizedTest
    @ValueSource(ints = {1, 2, 3, 4, 5})
    void testWithValueSource(int number) {
        assertTrue(number > 0);
    }

    // Multiple value sources
    @ParameterizedTest
    @ValueSource(strings = {"apple", "banana", "cherry"})
    void testWithStringSource(String fruit) {
        assertNotNull(fruit);
        assertTrue(fruit.length() > 3);
    }

    // Null and empty sources
    @ParameterizedTest
    @NullSource
    @EmptySource
    @ValueSource(strings = {" ", "   "})
    void testWithNullAndEmptySources(String text) {
        assertTrue(text == null || text.trim().isEmpty());
    }
```

```java
// ENUM source
@ParameterizedTest
@EnumSource(Month.class)
void testWithEnumSource(Month month) {
    assertNotNull(month);
}

// Filtered ENUM source
@ParameterizedTest
@EnumSource(value = Month.class, names = {"JANUARY", "FEBRUARY", "MARCH"})
void testWithFilteredEnumSource(Month month) {
    assertTrue(month.getValue() <= 3);
}

// CSV source
@ParameterizedTest
@CsvSource({
    "apple, 5",
    "banana, 6",
    "cherry, 6"
})
void testWithCsvSource(String fruit, int length) {
    assertEquals(length, fruit.length());
}

// CSV file source
@ParameterizedTest
@CsvFileSource(resources = "/test-data.csv", numLinesToSkip = 1)
void testWithCsvFileSource(String input, int expected) {
    assertEquals(expected, input.length());
}

// Method source
@ParameterizedTest
@MethodSource("stringProvider")
void testWithMethodSource(String input) {
    assertNotNull(input);
}

static Stream<String> stringProvider() {
    return Stream.of("apple", "banana", "cherry");
}

// Method source with complex objects
@ParameterizedTest
@MethodSource("personProvider")
void testPersonProperties(Person person, boolean isAdult) {
    assertEquals(isAdult, person.getAge() >= 18);
}

static Stream<Arguments> personProvider() {
    return Stream.of(
        Arguments.of(new Person("John", 20), true),
```

```
            Arguments.of(new Person("Alice", 16), false),
            Arguments.of(new Person("Bob", 18), true)
        );
    }

    // Custom ArgumentsProvider
    @ParameterizedTest
    @ArgumentsSource(CustomArgumentsProvider.class)
    void testWithArgumentsSource(String input) {
        assertNotNull(input);
    }

    static class CustomArgumentsProvider implements ArgumentsProvider {
        @Override
        public Stream<? extends Arguments> provideArguments(ExtensionContext
context) {
            return Stream.of("one", "two", "three")
                .map(Arguments::of);
        }
    }
}
```

Test Extensions

```
import org.junit.jupiter.api.extension.*;

// Custom extension for timing tests
class TimingExtension implements BeforeTestExecutionCallback,
AfterTestExecutionCallback {

    private static final Logger logger =
Logger.getLogger(TimingExtension.class.getName());

    @Override
    public void beforeTestExecution(ExtensionContext context) {
        context.getStore(ExtensionContext.Namespace.create(getClass(),
context.getRequiredTestMethod()))
            .put("start", System.currentTimeMillis());
    }

    @Override
    public void afterTestExecution(ExtensionContext context) {
        long start =
context.getStore(ExtensionContext.Namespace.create(getClass(),
context.getRequiredTestMethod()))
            .get("start", long.class);
        long duration = System.currentTimeMillis() - start;

        logger.info(() -> String.format("Test [%s] took %d ms.",
            context.getDisplayName(), duration));
    }
```

```
}

// Using the extension with a test
@ExtendWith(TimingExtension.class)
class ExtendedTest {

    @Test
    void testWithTiming() throws InterruptedException {
        Thread.sleep(100);  // Simulate work
        assertTrue(true);
    }
}

// Built-in extensions
@ExtendWith(TempDirectory.class)  // Creates temporary directories
class TempDirectoryTest {

    @Test
    void testWithTempDirectory(@TempDir Path tempDir) {
        Path file = tempDir.resolve("test.txt");
        // Use temporary file
    }
}

// Conditional test execution
@EnabledOnOs(OS.LINUX)
@Test
void testOnlyOnLinux() {
    // This test runs only on Linux
}

@DisabledOnOs(OS.WINDOWS)
@Test
void testExceptOnWindows() {
    // This test runs on all OSes except Windows
}

@EnabledOnJre(JRE.JAVA_17)
@Test
void testOnlyOnJava17() {
    // This test runs only on Java 17
}

@EnabledIfSystemProperty(named = "os.arch", matches = ".*64.*")
@Test
void testOnlyOn64BitArchitecture() {
    // This test runs only on 64-bit architecture
}

@EnabledIfEnvironmentVariable(named = "ENV", matches = "dev")
@Test
void testOnlyInDevEnvironment() {
    // This test runs only when ENV=dev
}
```

```
// Custom condition
@Test
@EnabledIf("customCondition")
void testEnabledByCustomCondition() {
    // Test that runs only if customCondition method returns true
}

boolean customCondition() {
    return System.currentTimeMillis() % 2 == 0;  // 50% chance of running
}
```

Advanced Testing Features

```
// Timeout testing
@Test
@Timeout(value = 500, unit = TimeUnit.MILLISECONDS)
void testTimeout() throws InterruptedException {
    // Test fails if it exceeds 500ms
    Thread.sleep(100);
    assertTrue(true);
}

// Repeated tests
@RepeatedTest(5)
void repeatedTest() {
    assertTrue(true);
}

// Repeated test with custom display name
@RepeatedTest(value = 5, name = "{displayName}
{currentRepetition}/{totalRepetitions}")
@DisplayName("Repeating Test")
void customRepeatDisplayName() {
    assertTrue(true);
}

// Repeated test with RepetitionInfo
@RepeatedTest(5)
void repeatedTestWithInfo(RepetitionInfo info) {
    System.out.println("Repetition: " + info.getCurrentRepetition() +
                       " of " + info.getTotalRepetitions());
    assertTrue(true);
}

// Dynamic tests
@TestFactory
Collection<DynamicTest> dynamicTests() {
    return Arrays.asList(
        DynamicTest.dynamicTest("First dynamic test", () -> assertTrue(true)),
        DynamicTest.dynamicTest("Second dynamic test", () -> assertEquals(4, 2 *
```

```
2))
    );
}

// Generated dynamic tests
@TestFactory
Stream<DynamicTest> generateDynamicTests() {
    return IntStream.range(1, 6)
        .mapToObj(n -> DynamicTest.dynamicTest("Test " + n,
            () -> assertTrue(n % n == 0)));
}

// Assumptions
@Test
void testOnlyOnDevelopmentMachine() {
    assumeTrue("DEV".equals(System.getenv("ENVIRONMENT")));
    // Test code that runs only if assumption is true
}

@Test
void testWithAssumptions() {
    // Set up test
    int x = 5;
    assumingThat(x > 10, () -> {
        // This code runs only if x > 10
        fail("Should not reach here if x <= 10");
    });

    // This code runs regardless
    assertTrue(x < 10);
}
```

Testing Best Practices

```
// 1. One assertion per test (conceptually)
@Test
void userRegistration_ValidInput_Success() {
    // Given
    UserRegistrationRequest request = new UserRegistrationRequest(
        "john@example.com", "password123", "John", "Doe");

    // When
    UserResponse response = userService.registerUser(request);

    // Then
    assertNotNull(response);
    assertEquals("john@example.com", response.getEmail());
    assertTrue(response.isActive());
}

// Better approach - separate conceptual validations
```

```java
@Test
void userRegistration_ValidInput_ReturnsUserWithCorrectEmail() {
    // Given
    UserRegistrationRequest request = new UserRegistrationRequest(
        "john@example.com", "password123", "John", "Doe");

    // When
    UserResponse response = userService.registerUser(request);

    // Then
    assertEquals("john@example.com", response.getEmail());
}

@Test
void userRegistration_ValidInput_ReturnsActiveUser() {
    // Given
    UserRegistrationRequest request = new UserRegistrationRequest(
        "john@example.com", "password123", "John", "Doe");

    // When
    UserResponse response = userService.registerUser(request);

    // Then
    assertTrue(response.isActive());
}

// 2. Arrange-Act-Assert (Given-When-Then) pattern
@Test
void withdrawFromAccount_SufficientFunds_ReducesBalance() {
    // Arrange (Given)
    Account account = new Account(500.0);

    // Act (When)
    boolean result = account.withdraw(300.0);

    // Assert (Then)
    assertTrue(result);
    assertEquals(200.0, account.getBalance(), 0.001);
}

// 3. Descriptive test names
@Test
void withdraw_AmountLessThanBalance_ReturnsTrue() {
    // Test code...
}

@Test
void withdraw_AmountGreaterThanBalance_ReturnsFalse() {
    // Test code...
}

// 4. Reusable test fixtures
public class OrderServiceTest {
```

```
    private OrderService orderService;
    private CustomerRepository customerRepository;
    private ProductRepository productRepository;

    @BeforeEach
    void setUp() {
        customerRepository = mock(CustomerRepository.class);
        productRepository = mock(ProductRepository.class);
        orderService = new OrderService(customerRepository, productRepository);

        // Common test data
        when(customerRepository.findById(1L)).thenReturn(
            Optional.of(new Customer(1L, "John Doe", "Premium")));

        when(productRepository.findById(101L)).thenReturn(
            Optional.of(new Product(101L, "Test Product", 100.0, 10)));
    }

    // Tests can now use the fixtures...
}

// 5. Test doubles (using Mockito)
@Test
void placeOrder_ValidOrder_ReturnsOrderConfirmation() {
    // Given
    OrderRequest request = new OrderRequest(1L, 101L, 2);

    // When
    OrderConfirmation confirmation = orderService.placeOrder(request);

    // Then
    assertNotNull(confirmation);
    assertEquals(200.0, confirmation.getTotalPrice());

    // Verify interactions with mocks
    verify(customerRepository).findById(1L);
    verify(productRepository).findById(101L);
    verify(productRepository).updateStock(101L, 8);
}

// 6. Exception testing
@Test
void divide_ByZero_ThrowsArithmeticException() {
    // Given
    Calculator calculator = new Calculator();

    // When & Then
    ArithmeticException exception = assertThrows(
        ArithmeticException.class,
        () -> calculator.divide(10, 0)
    );

    assertEquals("/ by zero", exception.getMessage());
}
```

```
// 7. Boundary value testing
@ParameterizedTest
@ValueSource(ints = {0, 1, 99, 100})
void validateAge_BoundaryValues_ValidForValidRange(int age) {
    // Given
    UserValidator validator = new UserValidator();

    // When & Then
    assertTrue(validator.isValidAge(age));
}

@ParameterizedTest
@ValueSource(ints = {-1, 101})
void validateAge_BoundaryValues_InvalidForOutOfRange(int age) {
    // Given
    UserValidator validator = new UserValidator();

    // When & Then
    assertFalse(validator.isValidAge(age));
}

// 8. Clean tests - no control logic
@Test
void authenticate_ValidCredentials_ReturnsToken() {
    // Given
    String username = "admin";
    String password = "correct_password";
    when(userRepository.findByUsername(username))
        .thenReturn(Optional.of(new User(username, hash(password))));

    // When
    String token = authService.authenticate(username, password);

    // Then
    assertNotNull(token);
    assertTrue(token.length() > 10);
}

// Avoid control logic in tests:
@Test
void authenticate_ValidCredentials_ReturnsToken_BAD() {
    // Given
    String username = "admin";
    String password = "correct_password";
    User user = null;

    // Bad practice: avoid if statements and control logic in tests
    if (shouldMockRepository()) {
        user = new User(username, hash(password));
        when(userRepository.findByUsername(username))
            .thenReturn(Optional.of(user));
    } else {
        userRepository.save(new User(username, hash(password)));
```

```
    }

    // When
    String token = authService.authenticate(username, password);

    // Then
    assertNotNull(token);

    // Bad practice: avoid if-else in assertions
    if (isJwt(token)) {
        assertTrue(token.contains("."));
    } else {
        assertTrue(token.length() > 10);
    }
}

// 9. Test isolation
@Test
void depositMoney_PositiveAmount_IncreasesBalance() {
    // Given - each test has its own account instance
    Account account = new Account(100.0);

    // When
    account.deposit(50.0);

    // Then
    assertEquals(150.0, account.getBalance(), 0.001);

    // No test pollution - other tests start with fresh Account instances
}

// 10. Test coverage for edge cases
@Test
void login_EmptyUsername_ThrowsValidationException() {
    // Given
    LoginRequest request = new LoginRequest("", "password");

    // When & Then
    ValidationException exception = assertThrows(
        ValidationException.class,
        () -> authService.login(request)
    );
    assertTrue(exception.getMessage().contains("username"));
}

@Test
void login_NullPassword_ThrowsValidationException() {
    // Given
    LoginRequest request = new LoginRequest("admin", null);

    // When & Then
    ValidationException exception = assertThrows(
        ValidationException.class,
        () -> authService.login(request)
```

```
        );
        assertTrue(exception.getMessage().contains("password"));
    }
```

Mocking Frameworks

Mockito

```
// Maven dependency:
// <dependency>
//     <groupId>org.mockito</groupId>
//     <artifactId>mockito-core</artifactId>
//     <version>5.2.0</version>
//     <scope>test</scope>
// </dependency>
// <dependency>
//     <groupId>org.mockito</groupId>
//     <artifactId>mockito-junit-jupiter</artifactId>
//     <version>5.2.0</version>
//     <scope>test</scope>
// </dependency>

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.Mock;
import org.mockito.InjectMocks;
import org.mockito.junit.jupiter.MockitoExtension;
import static org.mockito.Mockito.*;

@ExtendWith(MockitoExtension.class)
class UserServiceTest {

    @Mock
    private UserRepository userRepository;

    @Mock
    private EmailService emailService;

    @InjectMocks
    private UserService userService;

    @Test
    void registerUser_ValidUser_Success() {
        // Given
        User user = new User("test@example.com", "password");

when(userRepository.findByEmail("test@example.com")).thenReturn(Optional.empty());
        when(userRepository.save(any(User.class))).thenReturn(user);

        // When
        User registered = userService.registerUser(user);
```

```
        // Then
        assertNotNull(registered);

        // Verify interactions
        verify(userRepository).findByEmail("test@example.com");
        verify(userRepository).save(user);
        verify(emailService).sendWelcomeEmail("test@example.com");
    }

    @Test
    void registerUser_DuplicateEmail_ThrowsException() {
        // Given
        User user = new User("existing@example.com", "password");
        when(userRepository.findByEmail("existing@example.com"))
            .thenReturn(Optional.of(new User("existing@example.com", "oldpass")));

        // When & Then
        assertThrows(UserAlreadyExistsException.class, () -> {
            userService.registerUser(user);
        });

        // Verify no save happened
        verify(userRepository, never()).save(any(User.class));
        verify(emailService, never()).sendWelcomeEmail(anyString());
    }

    @Test
    void deactivateUser_UserExists_Deactivated() {
        // Given
        User user = new User("test@example.com", "password");
        user.setActive(true);
        when(userRepository.findById(1L)).thenReturn(Optional.of(user));

        // When
        boolean result = userService.deactivateUser(1L);

        // Then
        assertTrue(result);
        assertFalse(user.isActive());
        verify(userRepository).save(user);
    }

    // Using argument captors
    @Test
    void updateUserProfile_ValidData_SavesUpdatedUser() {
        // Given
        User existingUser = new User("test@example.com", "password");
        existingUser.setId(1L);
        existingUser.setName("Old Name");

        ProfileUpdateRequest request = new ProfileUpdateRequest("New Name", "New
Bio");
```

```
        when(userRepository.findById(1L)).thenReturn(Optional.of(existingUser));
        when(userRepository.save(any(User.class))).thenAnswer(i ->
i.getArgument(0));

        // Argument captor to inspect saved user
        ArgumentCaptor<User> userCaptor = ArgumentCaptor.forClass(User.class);

        // When
        User result = userService.updateProfile(1L, request);

        // Then
        assertNotNull(result);
        assertEquals("New Name", result.getName());

        verify(userRepository).save(userCaptor.capture());
        User savedUser = userCaptor.getValue();
        assertEquals(1L, savedUser.getId());
        assertEquals("New Name", savedUser.getName());
        assertEquals("New Bio", savedUser.getBio());
        assertEquals("test@example.com", savedUser.getEmail());  // Should not
change
    }

    // Handling void methods
    @Test
    void deleteUser_UserExists_CallsRepositoryAndService() {
        // Given
        User user = new User("test@example.com", "password");
        when(userRepository.findById(1L)).thenReturn(Optional.of(user));

        // Configure void method
        doNothing().when(emailService).sendAccountDeletionEmail(anyString());

        // When
        userService.deleteUser(1L);

        // Then
        verify(userRepository).delete(user);
        verify(emailService).sendAccountDeletionEmail("test@example.com");
    }

    // Mock behavior with exceptions
    @Test
    void findUser_RepositoryException_HandlesGracefully() {
        // Given
        when(userRepository.findById(anyLong())).thenThrow(new
DataAccessException("DB Error"));

        // When & Then
        assertThrows(ServiceException.class, () -> userService.findUser(1L));
    }

    // Mock static methods (Mockito 3.4.0+)
    @Test
```

```
    void validateEmail_UsesStaticUtility() {
        try (MockedStatic<EmailValidator> validator =
mockStatic(EmailValidator.class)) {
            // Given
            validator.when(() ->
EmailValidator.isValid("good@example.com")).thenReturn(true);
            validator.when(() ->
EmailValidator.isValid("bad@example")).thenReturn(false);

            // When & Then
            assertTrue(userService.validateEmail("good@example.com"));
            assertFalse(userService.validateEmail("bad@example"));

            // Verify the static method was called
            validator.verify(() -> EmailValidator.isValid("good@example.com"));
            validator.verify(() -> EmailValidator.isValid("bad@example"));
        }
    }

    // Mock final classes/methods (Mockito 2.1.0+)
    @Test
    void processWith_FinalClass_Mocked() {
        // Mockito can now mock final classes and methods by default
        FinalClass finalClass = mock(FinalClass.class);
        when(finalClass.calculate()).thenReturn(42);

        assertEquals(42, finalClass.calculate());
    }

    // Spy - partial mocking
    @Test
    void withSpy_PartiallyMocked() {
        List<String> realList = new ArrayList<>();
        List<String> spyList = spy(realList);

        // Real method executes
        spyList.add("one");
        spyList.add("two");

        // Still a real list underneath
        assertEquals(2, spyList.size());
        assertEquals("one", spyList.get(0));

        // Can stub specific methods
        doReturn(100).when(spyList).size();
        assertEquals(100, spyList.size());

        // But other methods still work normally
        assertEquals("one", spyList.get(0));
    }
}
```

Integration Testing

```
// Creating an integration test with Spring Boot
@SpringBootTest
class UserServiceIntegrationTest {

    @Autowired
    private UserService userService;

    @Autowired
    private UserRepository userRepository;

    @MockBean
    private EmailService emailService;  // Mock external service

    @BeforeEach
    void setUp() {
        // Clear database before each test
        userRepository.deleteAll();

        // Set up mocked dependencies
        when(emailService.sendWelcomeEmail(anyString())).thenReturn(true);
    }

    @Test
    void registerUser_NewUser_SavedToDatabase() {
        // Given
        UserRegistrationRequest request = new UserRegistrationRequest(
            "test@example.com", "password", "Test User");

        // When
        User registered = userService.registerUser(request);

        // Then
        assertNotNull(registered);
        assertNotNull(registered.getId());
        assertEquals("test@example.com", registered.getEmail());

        // Verify database state
        Optional<User> found = userRepository.findByEmail("test@example.com");
        assertTrue(found.isPresent());
        assertEquals("Test User", found.get().getName());

        // Verify external service interactions
        verify(emailService).sendWelcomeEmail("test@example.com");
    }

    @Test
    void findUsers_MultipleCriteria_ReturnsMatchingUsers() {
        // Given - set up test data
        userRepository.save(new User("user1@example.com", "pass", "User One",
25));
```

```
        userRepository.save(new User("user2@example.com", "pass", "User Two",
30));
        userRepository.save(new User("admin@example.com", "pass", "Admin", 35));

        // When
        List<User> users = userService.findUsersByAgeRange(25, 32);

        // Then
        assertEquals(2, users.size());
        assertTrue(users.stream().anyMatch(u ->
u.getEmail().equals("user1@example.com")));
        assertTrue(users.stream().anyMatch(u ->
u.getEmail().equals("user2@example.com")));
    }
}

// Database integration testing with test containers
@Testcontainers
@SpringBootTest
class DatabaseIntegrationTest {

    @Container
    static PostgreSQLContainer<?> postgres = new PostgreSQLContainer<>
("postgres:14")
        .withDatabaseName("testdb")
        .withUsername("test")
        .withPassword("test");

    @DynamicPropertySource
    static void registerPgProperties(DynamicPropertyRegistry registry) {
        registry.add("spring.datasource.url", postgres::getJdbcUrl);
        registry.add("spring.datasource.username", postgres::getUsername);
        registry.add("spring.datasource.password", postgres::getPassword);
    }

    @Autowired
    private ProductRepository productRepository;

    @Test
    void productRepository_SaveAndFind_Success() {
        // Given
        Product product = new Product("Test Product", 99.99, 10);

        // When
        Product saved = productRepository.save(product);

        // Then
        assertNotNull(saved.getId());

        // Verify retrieve
        Optional<Product> retrieved = productRepository.findById(saved.getId());
        assertTrue(retrieved.isPresent());
        assertEquals("Test Product", retrieved.get().getName());
    }
```

```
}

// REST API integration testing
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
class UserControllerIntegrationTest {

    @LocalServerPort
    private int port;

    @Autowired
    private TestRestTemplate restTemplate;

    @Autowired
    private UserRepository userRepository;

    @BeforeEach
    void setUp() {
        userRepository.deleteAll();
    }

    @Test
    void registerUser_ValidRequest_ReturnsCreatedUser() {
        // Given
        UserRegistrationRequest request = new UserRegistrationRequest(
            "test@example.com", "password", "Test User");

        // When
        ResponseEntity<UserResponse> response = restTemplate.postForEntity(
            "http://localhost:" + port + "/api/users/register",
            request,
            UserResponse.class);

        // Then
        assertEquals(HttpStatus.CREATED, response.getStatusCode());
        assertNotNull(response.getBody());
        assertEquals("test@example.com", response.getBody().getEmail());
        assertEquals("Test User", response.getBody().getName());
    }

    @Test
    void getUser_ExistingUser_ReturnsUser() {
        // Given
        User user = userRepository.save(new User("existing@example.com",
"password", "Existing User"));

        // When
        ResponseEntity<UserResponse> response = restTemplate.getForEntity(
            "http://localhost:" + port + "/api/users/" + user.getId(),
            UserResponse.class);

        // Then
        assertEquals(HttpStatus.OK, response.getStatusCode());
        assertNotNull(response.getBody());
        assertEquals("existing@example.com", response.getBody().getEmail());
```

```
    }

    @Test
    void getUser_NonExistingUser_ReturnsNotFound() {
        // When
        ResponseEntity<String> response = restTemplate.getForEntity(
            "http://localhost:" + port + "/api/users/999",
            String.class);

        // Then
        assertEquals(HttpStatus.NOT_FOUND, response.getStatusCode());
    }
}
```

Performance Testing with JMH

```
// Maven dependency:
// <dependency>
//     <groupId>org.openjdk.jmh</groupId>
//     <artifactId>jmh-core</artifactId>
//     <version>1.36</version>
//     <scope>test</scope>
// </dependency>
// <dependency>
//     <groupId>org.openjdk.jmh</groupId>
//     <artifactId>jmh-generator-annprocess</artifactId>
//     <version>1.36</version>
//     <scope>test</scope>
// </dependency>

import org.openjdk.jmh.annotations.*;
import org.openjdk.jmh.infra.Blackhole;
import org.openjdk.jmh.runner.Runner;
import org.openjdk.jmh.runner.options.Options;
import org.openjdk.jmh.runner.options.OptionsBuilder;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.concurrent.TimeUnit;

@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MICROSECONDS)
@State(Scope.Thread)
@Fork(1)
@Warmup(iterations = 3)
@Measurement(iterations = 5)
public class ListPerformanceTest {

    @Param({"10", "100", "1000"})
    private int size;
```

```
    private List<Integer> arrayList;
    private List<Integer> linkedList;

    @Setup
    public void setup() {
        arrayList = new ArrayList<>();
        linkedList = new LinkedList<>();

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

    @Benchmark
    public void arrayListGet(Blackhole blackhole) {
        for (int i = 0; i < size; i++) {
            blackhole.consume(arrayList.get(i));
        }
    }

    @Benchmark
    public void linkedListGet(Blackhole blackhole) {
        for (int i = 0; i < size; i++) {
            blackhole.consume(linkedList.get(i));
        }
    }

    @Benchmark
    public List<Integer> arrayListAdd() {
        List<Integer> list = new ArrayList<>();
        for (int i = 0; i < size; i++) {
            list.add(i);
        }
        return list;
    }

    @Benchmark
    public List<Integer> linkedListAdd() {
        List<Integer> list = new LinkedList<>();
        for (int i = 0; i < size; i++) {
            list.add(i);
        }
        return list;
    }

    @Benchmark
    public List<Integer> arrayListAddFirst() {
        List<Integer> list = new ArrayList<>();
        for (int i = 0; i < size; i++) {
            list.add(0, i);
        }
        return list;
```

```
    }

    @Benchmark
    public List<Integer> linkedListAddFirst() {
        List<Integer> list = new LinkedList<>();
        for (int i = 0; i < size; i++) {
            list.add(0, i);
        }
        return list;
    }

    public static void main(String[] args) throws Exception {
        Options options = new OptionsBuilder()
            .include(ListPerformanceTest.class.getSimpleName())
            .build();
        new Runner(options).run();
    }
}
```

> **Common Pitfall**: Writing tests that depend on execution order or external state. Tests should be isolated and independent of each other.

> **Certification Note**: Understand the basics of JUnit 5, including annotations, assertions, and lifecycle methods. Know how to write parameterized tests and how to use mocking frameworks like Mockito.

Certification-Specific Preparation

Exam Format and Structure

Java Certification Types

1. **Oracle Certified Professional: Java SE 17 Developer**
    - Primary certification for Java developers
    - Tests core Java SE 17 features and APIs
    - Prerequisite: None (previous "Associate" level no longer required)
    - Cost: $245 USD (as of 2023)
    - Time: 90 minutes
    - Questions: 50 multiple-choice and drag-and-drop
    - Passing score: 68% (34/50 correct answers)
2. **Oracle Certified Professional: Java SE 21 Developer** (when available)
    - Similar to Java SE 17 certification with updated content for Java 21
    - Will cover additional features introduced between Java 17 and 21
    - Likely similar exam format and passing requirements
3. **Oracle Certified Professional: Java EE 7 Application Developer**
    - For enterprise Java development

- Covers Jakarta EE (formerly Java EE) technologies
- Prerequisite: Java SE certification
- Focus on enterprise APIs like Servlets, JSP, JPA, etc.

Exam Content Areas

For the Java SE 17/21 Developer exam, topics include:

1. **Java Fundamentals (15-20%)**
   - Object-oriented concepts
   - Java syntax and structure
   - Modifiers, inheritance, overriding
   - Exception handling
   - Java packages and modules
2. **Java Object-Oriented Programming (15-20%)**
   - Encapsulation, inheritance, polymorphism
   - Abstract classes and interfaces
   - Inner classes and enums
   - Records, sealed classes
3. **Java Data Types and Collections (15-20%)**
   - Primitives and wrappers
   - String and StringBuilder
   - Collections framework
   - Generics
4. **Java Stream API and Functional Interfaces (15-20%)**
   - Functional interfaces and lambda expressions
   - Built-in functional interfaces
   - Stream operations
   - Method references
5. **Java Platform Module System (10-15%)**
   - Module declarations and directives
   - Service provider interfaces
   - Module resolution
6. **Concurrency and Exception Handling (10-15%)**
   - Threads and Executors
   - Synchronization
   - Fork/Join framework
   - Virtual threads (Java 21)
7. **Java I/O and NIO.2 (5-10%)**

- File operations
- Reading/writing data
- Serialization

Question Types

1. **Multiple Choice**
   - Single correct answer from several options
   - Multiple correct answers (select all that apply)
2. **Drag and Drop**
   - Arrange code fragments in correct order
   - Match options to corresponding categories
3. **Code Snippets**
   - Analyze code and determine output
   - Identify compilation errors
   - Choose code snippets that achieve a specified goal

Question Types and Common Traps

Multiple Choice Examples

```
Q: Which statement about the following code is true?

public class Test {
    public static void main(String[] args) {
        var list = List.of(1, 2, 3);
        list.add(4);
        System.out.println(list);
    }
}

A. The code prints [1, 2, 3, 4]
B. The code prints [1, 2, 3]
C. The code does not compile
D. The code throws an UnsupportedOperationException at runtime
```

Correct Answer: D - The `List.of()` method returns an immutable list, so attempting to add an element throws an UnsupportedOperationException.

Select All That Apply Example

```
Q: Which of the following are valid declarations of a functional interface?
(Choose all that apply)
```

```
A. interface Func { void apply(); }
B. @FunctionalInterface interface Func { void apply(); }
C. @FunctionalInterface interface Func { void apply(); int count(); }
D. interface Func { void apply(); default void log() { } }
E. interface Func extends Runnable { void apply(); }
```

Correct Answers: A, B, D - A functional interface must have exactly one abstract method, but can have any number of default or static methods.

Drag and Drop Example

```
Q: Arrange the code fragments to create a valid method that returns the sum of
elements in a list that are greater than the threshold:

1. public static int sumGreaterThan(List<Integer> numbers, int threshold) {
2. return numbers.stream()
3. .filter(n -> n > threshold)
4. .mapToInt(Integer::intValue)
5. .map(n -> n * n)
6. .sum();
7. }
```

Correct Order: 1, 2, 3, 4, 6, 7 (leaving out 5, which would square the values instead of summing them)

Common Traps and Pitfalls

1. **Operator Precedence**

```
int x = 5;
int y = 10;
int z = ++x + y--;
System.out.println(x + "," + y + "," + z);
```

   Output: `6,9,16` (++x increments x before the addition, y-- decrements y after the addition)

2. **String References vs. Content**

```
String s1 = "Java";
String s2 = "Java";
String s3 = new String("Java");
String s4 = s3.intern();

System.out.println(s1 == s2);        // true
System.out.println(s1 == s3);        // false
System.out.println(s1.equals(s3));   // true
System.out.println(s1 == s4);        // true
```

3. **Overriding vs. Overloading**

```
class Animal {
    public void makeSound() { System.out.println("Animal sound"); }
    public Number getWeight() { return 0; }
}

class Dog extends Animal {
    // Overriding - same signature, covariant return type allowed
    @Override
    public void makeSound() { System.out.println("Woof"); }
    @Override
    public Integer getWeight() { return 20; }  // Covariant return

    // Overloading - different parameter types
    public void makeSound(String intensity) {
        System.out.println("Woof " + intensity);
    }
}
```

4. **Generic Type Erasure**

```
// Cannot overload with same erased signature
public void process(List<String> strings) { }
public void process(List<Integer> integers) { }  // Compile error

// Cannot create arrays of generic types
List<String>[] stringLists = new List<String>[10];  // Compile error
```

5. **Lambda Variable Capture**

```
int x = 10;
Runnable r = () -> {
    System.out.println(x);  // x is effectively final
    // x++;  // Would cause compile error
};
```

6. **Access Modifiers**

```
package pkg1;
public class Parent {
    public void publicMethod() { }
    protected void protectedMethod() { }
    void packageMethod() { }
    private void privateMethod() { }
}
```

```
package pkg2;
import pkg1.Parent;
public class Child extends Parent {
    public void test() {
        publicMethod();     // OK
        protectedMethod();  // OK - accessible in subclass
        // packageMethod(); // Error - different package
        // privateMethod(); // Error - private to Parent
    }
}
```

7. **Exception Handling**

```
// Must catch more specific exceptions first
try {
    // code that might throw exceptions
} catch (IOException e) {
    // handle IOException
} catch (Exception e) {
    // handle other exceptions
}

// Must catch checked exceptions
public void readFile(String path) {
    // FileReader constructor throws FileNotFoundException
    // Either surround with try-catch or declare in method signature
    FileReader reader = new FileReader(path);  // Compilation error
}
```

8. **Collection Modification During Iteration**

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

// ConcurrentModificationException
for (String s : list) {
    if (s.equals("B")) {
        list.remove(s);  // Don't modify during iteration
    }
}

// Correct approaches:
// 1. Use Iterator's remove method
Iterator<String> it = list.iterator();
while (it.hasNext()) {
    if (it.next().equals("B")) {
```

```
            it.remove();  // OK
        }
    }

    // 2. Use removeIf (Java 8+)
    list.removeIf(s -> s.equals("B"));  // OK
```

Study Strategies and Time Management

Study Strategies

Study Strategies

1. **Structured Learning Plan**
    - Create a detailed study schedule
    - Allocate specific time periods for each topic area
    - Cover all topics in the exam objectives
2. **Hands-On Practice**
    - Write code for each concept you learn
    - Compile and run examples to verify understanding
    - Create mini-projects to apply multiple concepts together
3. **Use Multiple Resources**
    - Official documentation (Java Language Specification, JavaDocs)
    - Certification study guides
    - Online courses and tutorials
    - Practice exams from reputable sources
4. **Active Learning Techniques**
    - Take handwritten notes
    - Explain concepts to others (rubber duck debugging)
    - Create flashcards for key concepts and APIs
    - Draw diagrams for complex relationships
5. **Review and Repetition**
    - Regularly review previously studied topics
    - Focus more on challenging concepts
    - Use spaced repetition for better retention
    - Revise common pitfalls and trick scenarios

Sample Study Timeline (12-week plan)

Weeks 1-2: Java Fundamentals

- Java basics and syntax
- OOP concepts
- Packages and modules

- Exception handling

Weeks 3-4: Data Types and Collections

- Primitives and wrappers
- String handling
- Collections framework
- Generics

Weeks 5-6: Functional Programming

- Lambda expressions
- Functional interfaces
- Stream API
- Method references

Weeks 7-8: Concurrency and I/O

- Threads and Executors
- Synchronization
- File I/O and NIO.2
- Serialization

Weeks 9-10: Advanced Features

- Java Platform Module System
- Records and sealed classes
- Pattern matching
- Virtual threads

Weeks 11-12: Final Preparation

- Complete practice exams
- Review weak areas
- Timed test simulations
- Last-minute review of tricky topics

Time Management During the Exam

1. **Initial Scan**
    - Quickly review all questions (2-3 minutes)
    - Note difficulty level of each question
    - Plan your approach based on complexity
2. **Three-Pass Strategy**
    - First pass: Answer easy questions immediately (30 minutes)
    - Second pass: Tackle moderate questions (30 minutes)
    - Third pass: Focus on difficult questions (20 minutes)
    - Final pass: Review marked questions (10 minutes)

3. **Question Time Allocation**
    - Easy questions: ~1 minute
    - Moderate questions: ~2 minutes
    - Difficult questions: ~3 minutes
4. **Flag for Review**
    - If you're unsure about an answer, make your best guess and flag for review
    - Don't leave any questions unanswered
    - Return to flagged questions if time permits
5. **Stay Calm and Focused**
    - Manage your stress with deep breathing
    - If stuck on a question, move on and return later
    - Watch the clock, but don't let it rush your decisions

Practice Questions with Explanations

Question 1: Lambda Expressions and Functional Interfaces

```
interface StringProcessor {
    String process(String input);
}

class Processor {
    public static void printProcessed(String str, StringProcessor p) {
        System.out.println(p.process(str));
    }
}

public class Test {
    public static void main(String[] args) {
        String text = "Hello";

        // Which lambda can be used with printProcessed?
        Processor.printProcessed(text, /* Lambda expression here */);
    }
}
```

Options:

1. `text -> text.toUpperCase()`
2. `(String s) -> { s.toUpperCase(); }`
3. `s -> s.toUpperCase()`
4. `s -> { return s.toUpperCase(); }`
5. `String s -> s.toUpperCase()`

Answer: Options 3 and 4 are valid lambda expressions that match the StringProcessor functional interface. Option 1 incorrectly uses the parameter name that's already defined in the context. Option 2 is missing a return statement in the lambda body. Option 5 has incorrect syntax (should not include parameter type without parentheses).

Explanation: A lambda expression for a functional interface must match its method signature. The StringProcessor interface has a single abstract method that takes a String parameter and returns a String. The lambda must have one parameter (can be named anything, but not the same as an existing variable in scope) and return a String value.

Question 2: Stream Operations

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

public class Test {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "Dave");

        String result = names.stream()
            .filter(s -> s.length() > 4)
            .map(String::toUpperCase)
            .collect(Collectors.joining(", "));

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

What is the output of this code?

1. `"ALICE, BOB, CHARLIE, DAVE"`
2. `"ALICE, CHARLIE, DAVE"`
3. `"ALICE, CHARLIE"`
4. `"Alice, Charlie"`

Answer: Option 2 - `"ALICE, CHARLIE, DAVE"`

Explanation:

1. The filter operation selects only strings with length > 4, so "Bob" (length 3) is filtered out.
2. The map operation converts each remaining name to uppercase.
3. The collect operation joins the names with a comma and space separator.
4. So the output is "ALICE, CHARLIE, DAVE".

Question 3: Exception Handling

```
class ResourceException extends Exception {}
class NotFoundException extends ResourceException {}
class InvalidResourceException extends ResourceException {}
```

```
public class Test {
    public static void main(String[] args) {
        try {
            processResource();
        } catch (/* Fill in the catches */) {
            System.out.println("Resource exception");
        } catch (Exception e) {
            System.out.println("General exception");
        }
    }

    static void processResource() throws ResourceException {
        double d = Math.random();
        if (d < 0.3) throw new NotFoundException();
        if (d < 0.6) throw new InvalidResourceException();
        if (d < 0.9) throw new ResourceException();
    }
}
```

Which of the following can replace `/* Fill in the catches */` to catch all ResourceException types except NotFoundException?

1. `ResourceException e`
2. `InvalidResourceException | ResourceException e`
3. `ResourceException e if !(e instanceof NotFoundException)`
4. `ResourceException e if (e.getClass() != NotFoundException.class)`
5. `InvalidResourceException e`

Answer: Option 5 is incorrect. The correct answer would be to use two separate catch blocks: one for InvalidResourceException and another for ResourceException, or to catch NotFoundException separately before catching ResourceException.

Explanation: You cannot selectively catch a superclass exception while excluding a specific subclass. The catch blocks are evaluated in order, so you should catch the most specific exceptions first. To catch ResourceException but not NotFoundException, you need to catch NotFoundException separately before catching ResourceException.

Question 4: Class Inheritance and Polymorphism

```
class Animal {
    public void makeSound() {
        System.out.print("Animal sound ");
    }
}

class Dog extends Animal {
    @Override
    public void makeSound() {
        super.makeSound();
```

```
        System.out.print("Woof ");
    }
}

class Labrador extends Dog {
    @Override
    public void makeSound() {
        super.makeSound();
        System.out.print("Loudly ");
    }
}

public class Test {
    public static void main(String[] args) {
        Animal animal = new Labrador();
        animal.makeSound();
    }
}
```

What is the output of this code?

1. `"Animal sound "`
2. `"Animal sound Woof "`
3. `"Animal sound Woof Loudly "`
4. `"Loudly "`

Answer: Option 3 - `"Animal sound Woof Loudly "`

Explanation: The reference type is Animal, but the object is a Labrador. Method calls use virtual method invocation based on the object's actual type, not the reference type. When makeSound() is called:

1. Labrador's makeSound() calls super.makeSound(), which is Dog's makeSound()
2. Dog's makeSound() calls super.makeSound(), which is Animal's makeSound()
3. Animal's makeSound() prints "Animal sound "
4. Then Dog's makeSound() continues and prints "Woof "
5. Then Labrador's makeSound() continues and prints "Loudly "

Question 5: Module System

Given the following module declarations:

```
// module-info.java in module com.app.api
module com.app.api {
    exports com.app.api.service;
    exports com.app.api.model to com.app.client;
    requires java.base;
}

// module-info.java in module com.app.impl
module com.app.impl {
    requires com.app.api;
```

```
    provides com.app.api.service.DataService
        with com.app.impl.DefaultDataService;
}

// module-info.java in module com.app.client
module com.app.client {
    requires com.app.api;
}

// module-info.java in module com.app.admin
module com.app.admin {
    requires com.app.impl;
}
```

Which statements are true? (Choose all that apply)

1. Module com.app.client can access classes in com.app.api.service
2. Module com.app.client can access classes in com.app.api.model
3. Module com.app.admin can access classes in com.app.api.service
4. Module com.app.admin can access classes in com.app.api.model
5. Module com.app.impl can access classes in com.app.api.model

Answer: Options 1, 2, 3, and 5 are true.

Explanation:

1. True - com.app.api exports com.app.api.service to all modules
2. True - com.app.api exports com.app.api.model specifically to com.app.client
3. True - com.app.admin requires com.app.impl, which requires com.app.api, and com.app.api.service is exported to all modules
4. False - com.app.api exports com.app.api.model only to com.app.client, not to com.app.impl or com.app.admin
5. True - com.app.impl requires com.app.api and can access com.app.api.service, but not com.app.api.model

Question 6: Virtual Threads and Concurrency

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

public class Test {
    public static void main(String[] args) throws Exception {
        long start = System.currentTimeMillis();

        Thread vt1 = Thread.startVirtualThread(() -> {
            try {
                Thread.sleep(1000);
                System.out.println("VT1 done");
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
```

```
        });

        Thread vt2 = Thread.startVirtualThread(() -> {
            try {
                Thread.sleep(1000);
                System.out.println("VT2 done");
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });

        vt1.join();
        vt2.join();

        long time = System.currentTimeMillis() - start;
        System.out.println("Time: " + time + "ms");
    }
}
```

Which statement best describes the execution of this code?

1. VT1 and VT2 run sequentially, taking about 2000ms to complete
2. VT1 and VT2 run in parallel, taking about 1000ms to complete
3. The code doesn't compile because Thread.startVirtualThread doesn't exist
4. The code throws IllegalThreadStateException at runtime

Answer: Option 2 - VT1 and VT2 run in parallel, taking about 1000ms to complete

Explanation: Virtual threads in Java 21 are designed for concurrent execution. In this code:

1. Two virtual threads are created and started
2. Both sleep for 1000ms independently
3. The main thread waits for both to complete with join()
4. Since the threads run concurrently, the total execution time will be approximately 1000ms plus a small overhead

Question 7: Record Patterns and Pattern Matching

```
record Point(int x, int y) {}
record Rectangle(Point topLeft, Point bottomRight) {}

public class Test {
    public static void main(String[] args) {
        Object obj = new Rectangle(new Point(1, 1), new Point(5, 5));

        if (obj instanceof Rectangle(Point(int x1, int y1), Point(int x2, int
y2))) {
            System.out.println("Area: " + ((x2 - x1) * (y2 - y1)));
        } else {
            System.out.println("Not a rectangle");
        }
```

```
    }
}
```

What is the output of this code?

1. `"Not a rectangle"`
2. `"Area: 16"`
3. The code doesn't compile
4. The code throws a ClassCastException at runtime

Answer: Option 2 - `"Area: 16"`

Explanation: This code uses record patterns (a preview feature in Java 21) for pattern matching. The pattern extracts the nested record components directly:

1. The object is a Rectangle with Points (1,1) and (5,5)
2. The pattern extracts x1=1, y1=1, x2=5, y2=5
3. The area is calculated as (x2-x1)*(y2-y1) = (5-1)*(5-1) = 4*4 = 16

Question 8: Stream API and Reduction Operations

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

public class Test {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

        Integer result = numbers.stream()
            .filter(n -> n % 2 == 0)
            .reduce(0, Integer::sum);

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

What is the output of this code?

1. `0`
2. `6`
3. `15`
4. The code throws NoSuchElementException

Answer: Option 2 - `6`

Explanation:

1. The filter operation selects only even numbers: 2 and 4
2. The reduce operation starts with the identity value 0 and adds each filtered number

3. 0 + 2 + 4 = 6

Question 9: Sealed Classes

```
public sealed class Shape permits Circle, Rectangle, Triangle {
    public double area() { return 0; }
}

final class Circle extends Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    @Override
    public double area() {
        return Math.PI * radius * radius;
    }
}

sealed class Rectangle extends Shape permits Square {
    private double width, height;

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

    @Override
    public double area() {
        return width * height;
    }
}

final class Square extends Rectangle {
    public Square(double side) {
        super(side, side);
    }
}

class Triangle extends Shape {  // Line A
    private double base, height;

    public Triangle(double base, double height) {
        this.base = base;
        this.height = height;
    }

    @Override
    public double area() {
        return 0.5 * base * height;
```

```
    }
}
```

Which statement about the code is true?

1. The code compiles successfully
2. Line A causes a compilation error because Triangle must be declared final
3. Line A causes a compilation error because Triangle must be declared non-sealed
4. Line A causes a compilation error because Triangle must be in the same package as Shape

Answer: Option 3 - Line A causes a compilation error because Triangle must be declared non-sealed

Explanation: In a sealed class hierarchy, all permitted subclasses must be declared as one of:

1. final - cannot be extended further
2. sealed - can only be extended by its permitted subclasses
3. non-sealed - can be extended by any class

Since Triangle doesn't use any of these modifiers, it causes a compilation error. It should be declared as `non-sealed class Triangle extends Shape`.

Question 10: Switch Expressions

```
enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY }

public class Test {
    public static void main(String[] args) {
        Day day = Day.WEDNESDAY;

        String result = switch (day) {
            case MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY -> "Weekday";
            case SATURDAY, SUNDAY -> "Weekend";
        };

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

What is the output of this code?

1. `"Weekday"`
2. `"Weekend"`
3. The code doesn't compile because switch expressions require a default case
4. The code doesn't compile because -> is not valid in a switch statement

Answer: Option 1 - `"Weekday"`

Explanation: This code uses a switch expression (introduced in Java 14) with the arrow syntax:

1. The day is WEDNESDAY, which matches the first case

2. The expression evaluates to "Weekday"
3. The value is assigned to result and printed

Switch expressions do not require a default case if all possible enum values are covered, as is the case here.

Mock Exam Recommendations

To best prepare for your Java certification exam, practice with a variety of mock exams. Here are some recommended resources:

1. **Official Oracle Practice Exams**
    - Directly from Oracle
    - Most closely resemble the actual exam
    - Available with the exam voucher as part of a bundle
    - https://education.oracle.com/products/trackp_333
2. **Whizlabs Java Certification Practice Tests**
    - Comprehensive question bank
    - Detailed explanations for each answer
    - Performance tracking and assessment
    - https://www.whizlabs.com/ocpjd-java-se-11-developer/
3. **Enthuware JA+ V17 Mock Exams**
    - Highly rated by certification candidates
    - Large pool of questions
    - Timed test mode and practice mode
    - Detailed analytics on your performance
    - https://enthuware.com/java-certification-mock-exams/oracle-certified-professional/ocp-java-17-exam-1z0-829
4. **Java Certification Companion (Book & Mock Exams)**
    - By Scott Selikoff and Jeanne Boyarsky
    - Contains several complete mock exams
    - Questions similar to real exam difficulty
    - Available from major booksellers
5. **Free Online Resources**
    - CodeRanch Java Certification forum
    - Java Revisited blog
    - Baeldung certification articles
    - GitHub repositories with practice questions

Tips for Using Mock Exams

1. **Simulate Real Exam Conditions**
    - Take full-length practice tests (50 questions)

    - Use the same time limit (90 minutes)
    - No breaks, notes, or assistance
    - Take the exam in a quiet environment

2. **Analyze Your Results**

    - Review every question, even those you got right
    - Understand why correct answers are correct
    - Understand why wrong answers are wrong
    - Identify patterns in your mistakes

3. **Track Your Progress**

    - Keep a log of mock exam scores
    - Note which topics need improvement
    - Take multiple mock exams from different providers
    - Aim for consistently scoring 80%+ before the real exam

4. **Use as a Learning Tool**

    - Don't just memorize answers
    - Research topics related to questions you missed
    - Write simple programs to verify your understanding
    - Create flashcards for concepts that trip you up

5. **The Final Week**

    - Take at least 2-3 full mock exams
    - Focus on reviewing weak areas
    - Don't try to learn completely new topics
    - Rest well the day before the exam

By combining thorough study of the Java language features with regular practice using mock exams, you'll be well-prepared to pass your Java certification exam. Remember that understanding the concepts is more important than memorizing answers, as the actual exam questions will test your comprehension of Java principles rather than your recall of specific examples.

Conclusion

Congratulations on working through this comprehensive Java 21 certification guide! You've covered the full spectrum of Java knowledge needed to succeed in your certification journey.

This guide has provided detailed explanations, code examples, and best practices for:

- Setting up your Java development environment
- Core Java language fundamentals
- Object-oriented programming concepts
- The Java standard library
- Modern Java features from Java 8 through 21
- Concurrency and multithreading

- File and network I/O operations
- Database access with JDBC and JPA
- Testing in Java
- Certification-specific preparation strategies

As you prepare for your certification exam, remember these key principles:

1. **Practice regularly** with hands-on coding exercises
2. **Apply concepts** to real-world scenarios to cement your understanding
3. **Review challenging topics** multiple times to strengthen your knowledge
4. **Take mock exams** to familiarize yourself with the testing format
5. **Study efficiently** by focusing on weak areas and official exam objectives

Java is a versatile and powerful language that continues to evolve. Your certification not only validates your current knowledge but also establishes a foundation for keeping pace with future Java innovations.

Good luck with your Java certification exam! With thorough preparation and practice, you're well on your way to becoming a certified Java developer.