

ADVANCED OOP WITH JAVA

Complete Student Course Notes



TABLE OF CONTENTS

CHAPTER 1 - JAVA OBJECT DESIGN 1
What this course builds 1
From source to running code 1
Values and variables 1
Decisions loops and methods 2
Classes objects and references 3
Protecting data 3
Useful advanced building blocks 4
Practical activity 4
Chapter summary 4
Review and exam questions 5

CHAPTER 2 - INHERITANCE 6
Why inheritance exists 6
Building a parent and child 6
Constructors and super 7
Overriding methods correctly 7
Forms of inheritance 7
Inheritance and composition 8
Access across packages 8
Practical activity 8
Chapter summary 8
Review and exam questions 9

CHAPTER 3 - INTERFACE 10
What an interface means 10
Using interfaces as types 10
Interface members 11
Several roles and conflicts 11
Functional interfaces and lambdas 11
Generic interfaces and collections 11
Choosing an interface 12
Practical activity 12
Chapter summary 12
Review and exam questions 12

CHAPTER 4 - ABSTRACTION - I 13
Why some classes stay incomplete 13
What an abstract class can contain 13
Shared state and safe behavior 14
Interface and abstract class comparison 14
A good abstract base 14
Practical activity 14
Chapter summary 15
Review and exam questions 15

CHAPTER 5 - ABSTRACTION - II 16
A method without a body 16
Valid and invalid declarations 16
A fixed process with a changing step 16
Signatures and return values 17
Hooks and required steps 17
When another design fits better 17
Practical activity 18
Chapter summary 18
Review and exam questions 18

CHAPTER 6 - POLYMORPHISM 19
One type many behaviors 19
Tracing a call 19

Overloading and overriding 20
Upcasting and downcasting 20
Collections and equality 20
Benefits and limits 20
Practical activity 21
Chapter summary 21
Review and exam questions 21

CHAPTER 7 - FILE OPERATIONS 22
Why programs use files 22
Paths folders and resources 22
Choosing file classes 22
A simple text format for objects 22
Primitive values in a binary file 24
Appending replacing and random access 24
Common problems and good habits 25
Practical activity 25
Chapter summary 25
Review and exam questions 25

CHAPTER 8 - SERIALIZATION 26
What serialization does 26
Serializable and saved fields 26
A controlled round trip 26
Understanding the filter 27
Graphs identity and repeat writes 27
Choosing a storage format 28
Practical activity 28
Chapter summary 28
Review and exam questions 29

CHAPTER 9 - JAVA GUI 30
From commands to screens 30
Main parts of a Swing screen 30
The event dispatch thread 30
A complete search window 30
Keeping the screen responsive 31
Validation and usable screens 32
Practical activity 32
Chapter summary 33
Review and exam questions 33

CHAPTER 10 - MULTITHREADING 34
Processes threads and shared work 34
Creating and starting a thread 34
Thread states 34
Race conditions and lost updates 35
Executors and returned results 35
Cancellation and interruption 36
Deadlocks and coordination tools 36
Practical activity 36
Chapter summary 37
Review and exam questions 37

CHAPTER 11 - JAVA WEB APPLICATIONS 38
Browser requests and server replies 38
A small local Java web app 38
Servlets and a web container 40
Layers sessions and data 41
Common web failures 42
Practical activity 42
Chapter summary 42

Review and exam questions 42

CHAPTER 12 - EXCEPTION HANDLING 43

Failure is part of normal work 43
The exception family 43
try catch finally throw and throws 43
Catch order and recovery boundaries 44
Custom exceptions for business rules 44
Cleanup and suppressed exceptions 45
State safety and transactions 45
Debugging and meaningful tests 45
Practical activity 45
Chapter summary 46
Review and exam questions 46

CHAPTER 13 - DESIGNING EFFICIENT ALGORITHMS 47

From a task to an algorithm 47
Time and space cost 47
Picking a collection 47
Linear and binary search 48
Sorting choices and stable order 48
Recursion and divide and conquer 49
Trees graphs and traversal 50
Greedy choices and saved subproblems 51
Testing correctness and measuring work 52
Practical activity 52
Chapter summary 52
Review and exam questions 52

CHAPTER 14 - APPLICATIONS OF JAVA 53

Where Java ideas work together 53
The library project requirements 53
A simple project structure 53
A runnable service core 54
Building the full project in stages 56
Project testing and acceptance 56
Maintainable code and delivery 57
Practical project assignment 57
Chapter summary 57
Review and exam questions 57

JAVA PRACTICE SETUP 58

Set up a practice folder 58
Compile run and inspect 58
Common setup problems 58
A useful lab record 58

CHAPTER ANSWER GUIDE 59

Chapter 1 59
Chapter 2 59
Chapter 3 59
Chapter 4 59
Chapter 5 60
Chapter 6 60
Chapter 7 60
Chapter 8 60
Chapter 9 61
Chapter 10 61
Chapter 11 61
Chapter 12 61
Chapter 13 62
Chapter 14 62

PRACTICE EXAM PAPER 63
Instructions and points 63
Part A short answers 63
Part B reasoning problems 63
Part C design and code 63
Exam answer and scoring guide 63

PROJECT ASSESSMENT GUIDE 65
Required submission 65
Scoring rubric 65
Review checklist 65
Short oral defense 65

QUICK REFERENCE TABLES 66
Java keywords and roles 66
Matching primitive file operations 66
Small glossary 67

SOURCES AND FURTHER READING 68
Official language and library sources 68

Advanced OOP with Java

Classroom teaching | Exam preparation | Revision | Practical learning

Course Outline

Java programming language by exploring its intermediate and advanced features in object- oriented programming, such as inheritance, interfaces, abstract classes, abstract methods, and polymorphism. Learn about how to write and read Java primitive types and object classes to and from files, how to serialize objects, and how to implement simple graphical user interfaces, multithread programming and Java web applications. Also, use comprehensive error handling and complex algorithms in programming projects, and investigate real-world applications of the Java programming language.

Learning goals

Model data with classes and protect it with clear rules.

Use inheritance, interfaces, abstract behavior, and polymorphism.

Save and load values and objects with validation and safe resource use.

Build a Swing screen, coordinate tasks, and answer local web requests.

Handle failures, choose algorithms, and deliver a tested project.

Java practice baseline

The examples target Java 21 without preview features. Most run with a standard JDK. The GUI needs a desktop display. The optional servlet exercise needs a compatible Tomcat container. Use invented classroom data.

How to study

Read the chapters in order on your first pass. Run complete programs in separate folders. Predict the result before running a changed input. Complete the activity and questions before reading the answer guide. During revision, explain the rule behind a result rather than only memorizing the output.

Code blocks labeled Fragment need the surrounding objects named in the text. Blocks labeled with a Java filename are complete source files. Command blocks belong in a terminal.

Chapter 1 - Java Object Design

What this course builds

Java is a programming language. A program is a set of instructions that a computer follows. Object-oriented programming, or OOP, groups data and actions into objects. A library book object may hold a title and a loan status. It may also have actions that lend or return the book. This course shows how to connect such objects, save their data, build screens, run tasks together, and answer web requests.

Our shared example is a School Library System. A student can search for a book. A librarian can lend and return it. The system checks rules, saves records, and shows useful results. Each short program is separate unless the notes say to combine files. This prevents a class from one lesson from changing the meaning of a class in another lesson.

The examples use Java 21 language and library features without preview features. Use a JDK that supports Java 21 or later. A JDK, or Java Development Kit, contains tools for building and running Java programs. Java 21 is the teaching baseline, not a claim that it is the newest Java release. Appendix sections explain how to run every complete example.

From source to running code

Source code is the text a programmer writes. Save Java source in a file ending in .java. The compiler, javac, checks the source and creates bytecode in .class files. Bytecode is an instruction format that the Java Virtual Machine, or JVM, runs. The JVM can turn frequently used bytecode into machine instructions while a program runs.

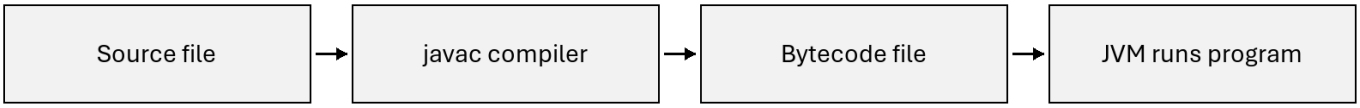


Figure 1 Java build and run path

The same bytecode can run on different systems when they have a suitable JVM and the program's required libraries. A program can still depend on a particular file path, native library, screen, or operating system feature. Test on the systems where people will use it.

Save this complete example as HelloLibrary.java. The public class name and filename must match, including uppercase letters.

HelloLibrary.java

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

Run these commands in the folder that holds the file. A terminal is a window where you type commands.

Terminal commands

```
javac --release 21 HelloLibrary.java
java HelloLibrary
```

The first command creates HelloLibrary.class. The second starts main, the entry method in this example. System.out.println prints text and then starts a new line. A semicolon ends a statement. Braces group statements. The words public, static, and void are explained below.

Values and variables

A variable is a named place for a value. Its type controls what it can hold. Write int copies = 3; to create a whole-number variable. Write copies = 4; to change it. A local variable belongs to a method or block. Give it a value before reading it.

Java has eight primitive types. A primitive variable holds a simple value rather than a reference to an object. A bit is one binary digit, 0 or 1. More bits usually allow a wider range of values.

Type	Simple meaning	Example and important point
byte	8-bit signed whole number	byte level = 5; range is -128 to 127
short	16-bit signed whole number	short copies = 300; range is -32768 to 32767
int	32-bit signed whole number	int id = 101; usual choice for counts
long	64-bit signed whole number	long visits = 3000000000L; L marks a long literal
float	32-bit floating-point number	float weight = 1.5F; F marks a float literal
double	64-bit floating-point number	double average = 3.75; can have rounding error
char	One 16-bit UTF-16 code unit	char grade = 'A'; some symbols need two code units
boolean	A true or false value	boolean available = true; not a numeric type

A floating-point number can store a value with a decimal part, such as 1.5. Its fixed number of bits makes many decimal values approximate. Unicode gives text characters and symbols standard numbers. UTF-16 is one way to encode Unicode text. A code unit is one fixed-size piece of that encoded text.

A literal is a value written directly in code, such as 101 or "Java". String is a class, not a primitive type. A String holds text and cannot be changed after it is created. A statement such as name = name.trim(); makes name refer to the trimmed result. It does not change the old String object.

Use +, -, *, /, and % for addition, subtraction, multiplication, division, and remainder. With two integer values, 7 / 2 gives 3. With a decimal value, 7 / 2.0 gives 3.5. The test copies > 0 gives a boolean. The operator == tests primitive values or whether two references point to the same object. Use equals for the text inside two strings. Use && for AND, || for OR, and ! for NOT.

An int cannot store every long value. A cast such as (int) largeValue asks for a conversion and may lose data. Check the range first, or use Math.toIntExact when overflow must cause an exception. Integer overflow can wrap around without an error. For checked arithmetic, Math.addExact is useful. Do not use double for exact money. Use whole cents or BigDecimal, which stores decimal numbers with controlled rounding.

Decisions loops and methods

An if statement chooses a path. A switch chooses among named cases. A loop repeats work. Use a for loop when you know the range, a while loop when repetition depends on a condition, and an enhanced for loop to visit items in an array or collection. An array is a fixed-length group of values with one element type. Its indexes start at 0.

BasicsDemo.java

```
public class BasicsDemo {
    static int countAvailable(boolean[] states) {
        int total = 0;
        for (boolean state : states) {
            if (state) {
                total++;
            }
        }
        return total;
    }

    public static void main(String[] args) {
        boolean[] states = {true, false, true};
        int count = countAvailable(states);
        System.out.println("Available: " + count);
    }
}
```

The method countAvailable takes a boolean array as a parameter. A parameter is an input name in a method. An argument is the actual value passed by a call. The return type int says that the method sends back a whole number. void means that a method returns no value. This program prints Available: 2.

Break a large task into small methods. A method called lendBook should lend a book and enforce its rules. It should not also draw the entire screen, send email, and write every report. Small methods are easier to check, reuse, and explain.

Classes objects and references

A class describes a kind of object. A field stores part of an object's state, meaning its current data. A method defines behavior, meaning what the object can do. A constructor sets up a new object. It has the class name and no return type. The keyword new creates an object and calls a constructor.

BookDemo.java

```
class Book {
    private final int id;
    private final String title;
    private boolean available = true;

    Book(int id, String title) {
        if (id <= 0 || title == null || title.isBlank()) {
            throw new IllegalArgumentException("Invalid book data");
        }
        this.id = id;
        this.title = title.trim();
    }

    int getId() { return id; }
    String getTitle() { return title; }
    boolean isAvailable() { return available; }

    void lend() {
        if (!available) {
            throw new IllegalStateException("Book is already on loan");
        }
        available = false;
    }

    void returnBook() { available = true; }
}

public class BookDemo {
    public static void main(String[] args) {
        Book first = new Book(101, "Java Basics");
        Book second = first;
        second.lend();
        System.out.println(first.isAvailable());
    }
}
```

The result is false. first and second refer to the same Book. Assigning one object reference to another does not copy the object. null means that a reference points to no object. Calling a method through null causes a NullPointerException. The expression this.id means the current object's id field.

Java always passes arguments by value. For an object, the copied value is a reference. A method can use that reference to change the shared object. Reassigning the parameter itself does not reassign the caller's variable. This distinction matters when a method changes a list or book.

Protecting data

Encapsulation means keeping data and the actions that protect it together. Private fields stop outside code from changing important values directly. The lend method checks availability before changing it. This rule must hold no matter whether a console, GUI, or web page starts the action.

Access level	Where a member can be used
private	Inside its own top-level class and its nested classes
No access word	Inside the same package
protected	Inside the same package and through allowed subclass access elsewhere
public	Wherever the class and member are accessible

A package groups related classes and helps avoid name clashes. A declaration such as package school.library; goes before imports. An import lets you use a class's short name. It does not copy that class's source. A public top-level class is normally stored in its own matching file under the package folder.

An instance member belongs to an object. A static member belongs to a class. main is static so the launcher can call it without making an application object first. A static method cannot use an instance field without an object reference. final stops a variable from being assigned again. For an object reference, final does not make the object itself unchangeable.

Useful advanced building blocks

A collection stores a group of objects. List keeps an ordered sequence and permits duplicates. Set keeps unique values. Map connects a key to a value, such as a book ID to a book. Generics add type information, as in List<Book>. They let the compiler reject a String added to a book list. Generic type arguments must be reference types. Integer is the wrapper class commonly used for int values in collections. Automatic boxing converts int to Integer. Unboxing converts back and fails if the wrapper is null.

An enum gives a fixed set of named values, such as enum Status { AVAILABLE, ON_LOAN }. It is safer than many unrelated strings. A record is a short way to define a data carrier, such as record BookView(int id, String title) {}. It supplies access methods, equals, hashCode, and toString. Its component references cannot be reassigned, but an object held by a component may still change.

The Object class gives common methods. toString makes useful display text. equals states when objects count as equal. hashCode supplies a number used by hash-based collections. When equal objects are possible, they must have equal hash codes. Keep values used in equality stable while an object is a HashSet element or HashMap key.

Practical activity

Purpose: Build and inspect a small object model. Tools: A Java 21-compatible JDK, a text editor or IDE, and a terminal. An IDE is an editor with programming tools, such as IntelliJ IDEA Community Edition, Eclipse, or VS Code with Java support. Any suitable plain editor also works.

Task: Create two independent books for the library and test the availability rule.

1. Run HelloLibrary and BasicsDemo in separate folders.
2. Save and run BookDemo. Record its output.
3. Add a second new Book with ID 102. Lend only the first book.
4. Print both titles and availability values. Explain why the states differ.
5. Call lend twice on the first book. Read the error message. Chapter 12 explains how to handle it.
6. Change the constructor argument to a blank title. Check that the object is rejected.

Expected result: Valid books start as available. Shared references see the same state. A rule violation stops the action with a clear exception. Keep the original source before each experiment.

Practice: Add a positive page count. Decide whether it should change after construction. Explain your choice. Draw one Book class and two Book objects on paper.

Chapter summary

Java source becomes bytecode that a JVM runs. Primitive variables hold simple values. Reference variables refer to objects. Classes join data with behavior. Private fields and checked methods protect rules. Packages, generics, collections, enums, and records help larger programs stay clear.

Review and exam questions

1. Explain the job of javac and the JVM in your own words.
2. Why is String different from int? Give one correct equality test for each.
3. Two variables refer to one Book. One variable calls lend. What does the other variable observe, and why?
4. Why is a private availability field safer than a public field?
5. Predict 9 / 2 and 9 / 2.0. Explain both answers.
6. Write a method that counts all positive values in an int array.

Chapter 2 - Inheritance

Why inheritance exists

Inheritance lets one class build on another class. The existing class is the superclass, also called the parent class. The new class is the subclass, also called the child class. A subclass can use accessible inherited members and add or change behavior. Use extends to name its superclass.

In the library, a StudentMember is a Member. Both have an ID and a name. The student also has a grade level. Inheritance can store shared member behavior once. A Book is not a Member. Giving both an ID does not make one a proper subclass of the other.

The basic test is an is-a relationship. A child object should be usable wherever the parent's promised behavior is needed. Do not use inheritance only to copy a few lines. If the relationship is wrong, future changes become confusing.

Building a parent and child

InheritanceDemo.java

```
class Member {
    private final String name;

    Member(String name) {
        if (name == null || name.isBlank()) {
            throw new IllegalArgumentException("Name is required");
        }
        this.name = name.trim();
    }

    public String getName() { return name; }

    public int loanLimit() { return 2; }

    public String description() {
        return name + " can borrow " + loanLimit() + " books";
    }
}

class StudentMember extends Member {
    private final int grade;

    StudentMember(String name, int grade) {
        super(name);
        if (grade < 1 || grade > 12) {
            throw new IllegalArgumentException("Grade must be 1 to 12");
        }
        this.grade = grade;
    }

    public int getGrade() { return grade; }

    @Override
    public int loanLimit() { return 3; }

    @Override
    public String description() {
        return super.description() + ", grade " + grade;
    }
}

public class InheritanceDemo {
    public static void main(String[] args) {
        StudentMember student = new StudentMember("Maya", 8);
        System.out.println(student.description());
```

```
    }
}
```

The result is Maya can borrow 3 books, grade 8. StudentMember inherits getName. It cannot directly access the private name field. It can use the public getter. An inherited method can still work with the parent's own private data.

Constructors and super

Constructors are not inherited. A child constructor must arrange for a parent constructor to run. In this Java 21 style, super(name) is the first statement. The parent part is set up before the child part. If a constructor does not write a parent call, the compiler tries super(). That fails when the parent has no accessible no-argument constructor.

If a class declares no constructors, the compiler normally supplies a no-argument constructor. Once you declare a constructor yourself, do not assume that an extra no-argument constructor is added. Use this(...) to call another constructor in the same class. In the Java 21 baseline, that call also goes first. Avoid having two constructors call each other forever.

Use super.methodName() to ask for a parent's implementation. In the example, the child description adds a grade to the parent's description. The parent's description calls loanLimit on the current object. Because that object is a StudentMember, the result is 3. Chapter 6 explains this method choice in detail.

Do not call methods that children can override from a parent constructor unless the design carefully supports it. A child's override can run before the child's fields are ready. Constructors should focus on safe setup.

Overriding methods correctly

Overriding means giving a new implementation of an inherited instance method. Use the same method name and parameter types. A return type must be compatible; for reference types, it may be a more specific subtype. Add @Override so the compiler can catch a spelling or parameter mistake.

The child cannot make an inherited public method private. It cannot add a broader checked exception than the parent method allows. A final method cannot be overridden. A private parent method is not inherited, so a same-named child method does not override it.

A static method belongs to a class. A child static method with the same signature hides a parent static method rather than overriding it. Fields can also be hidden. This is hard to read. Prefer different field names and call static methods through their class names.

Forms of inheritance

Form	Example	Java class support
Single	StudentMember extends Member	A class has one direct superclass
Multilevel	SeniorStudent extends StudentMember extends Member	Supported, but keep chains short
Hierarchical	StudentMember and TeacherMember extend Member	Supported
Multiple classes	One class extends two classes	Not supported
Multiple interfaces	A class implements two interfaces	Supported; see Chapter 3

Every ordinary class except Object has a superclass. If none is written, it is Object. Java does not let a class extend two classes because shared state and method rules can become difficult to combine. Interfaces provide a different way to support several roles.

Use final class when the design must not have subclasses. Use final on a method when children may exist but must keep that behavior. Sealed classes are an optional advanced feature that limit which classes may

directly extend a class. They are useful for a fixed family of types. They require explicit permitted subclasses and are not needed for the small library examples.

Inheritance and composition

Composition means building an object from other objects. A Library has a list of books. A Member has a contact record. These are has-a relationships. A class can hold a helper object and call its methods. This is often easier to change than a deep inheritance chain.

Consider a loan reminder service. An EmailSender is not a Member. Put a MessageSender object inside ReminderService instead of making ReminderService extend EmailSender. Then the sender can change without changing what a reminder service is.

Choice	Main strength	Main risk
Inheritance	Reuses a real family relationship and shared behavior	Tight connection to parent changes
Composition	Swaps helper objects and combines independent jobs	Needs some forwarding method calls

A good child preserves the parent's rules. If a parent method promises that returnBook can always return a borrowed item, a child should not reject normal returns just because its author dislikes the feature. Code that expects a Member should not need many special checks for each child type.

Access across packages

protected does not mean public to every object of the parent type. Outside the parent's package, a subclass can access a protected instance member through allowed subclass relationships, not through any arbitrary parent reference. Prefer private fields and clear methods for beginners. They keep access rules easier to reason about.

Keep a parent class small. Expose behavior that children truly need. Avoid a parent with dozens of protected fields. A child that depends on every internal field becomes hard to change safely.

Practical activity

Purpose: Reuse a shared member model. Tools: The JDK, an editor, and a terminal. Task: Add a teacher role while preserving common member behavior.

1. Compile and run InheritanceDemo in its own folder.
2. Add TeacherMember extends Member with loanLimit returning 5.
3. Make a teacher named Luis. Print the inherited description.
4. Add a private subject field and a getter to TeacherMember.
5. Draw Member above its two children. Draw a line from each child toward Member and label it extends.
6. Temporarily misspell loanLimit while leaving @Override in place. Read the compiler error, then fix the name.

Expected result: Maya can borrow 3 books. Luis can borrow 5 books. Both use the same name handling and shared description logic.

Practice: Should a Library extend Book or contain books? Explain. What happens if Member has only a constructor with a name but the child does not call it?

Chapter summary

Inheritance models a real is-a relationship. A class extends one direct parent. Parent setup happens before child setup. Children may override instance methods while following access, return type, and exception rules. Composition is a useful choice when an object has or uses another object.

Review and exam questions

1. Define superclass and subclass using the library example.
2. Why is super(name) needed in StudentMember?
3. Compare overriding, static method hiding, and field hiding.
4. What does final prevent on a class? What does it prevent on a method?
5. A Printer uses an InkCartridge. Choose inheritance or composition and explain.
6. Design TeacherMember and predict its description output.

Chapter 3 - Interface

What an interface means

An interface describes a role that an object can perform. It is a contract: code that uses the interface knows which actions are available. The contract should explain valid inputs, results, and expected failures as well as method names. A class uses implements to accept that contract.

For example, a library reminder can go by email or appear on a screen. Both are message senders. The reminder service needs a send action. It does not need to know how each sender works. A small interface makes that shared action clear.

InterfaceDemo.java

```
interface MessageSender {
    void send(String text);
}

class ConsoleSender implements MessageSender {
    @Override
    public void send(String text) {
        System.out.println("Reminder: " + text);
    }
}

class ReminderService {
    private final MessageSender sender;

    ReminderService(MessageSender sender) {
        this.sender = java.util.Objects.requireNonNull(sender);
    }

    void remind(String member) {
        sender.send(member + ", please return your book.");
    }
}

public class InterfaceDemo {
    public static void main(String[] args) {
        MessageSender sender = new ConsoleSender();
        new ReminderService(sender).remind("Maya");
    }
}
```

The result is Reminder: Maya, please return your book. Objects.requireNonNull rejects a missing sender. It prevents a later surprise when the service tries to send. The plain send declaration is public and abstract even though those words are omitted. The implementation must be public.

Using interfaces as types

MessageSender sender means that the variable may refer to any object that implements MessageSender. It can call send. It cannot call a special email configuration method unless that method is part of the declared type or code safely uses a more specific type.

The service receives its helper through its constructor. This is called dependency injection. A dependency is another object a class needs. Injection simply means supplying that helper from outside. No large framework is required. For a test, supply a sender that records text in a list. For a real app, supply an email sender.

Interfaces reduce the need to rewrite working code when an implementation changes. They also let unrelated classes support the same role. A Book and a LoanReport might both be Printable even though neither should extend the other.

Interface members

Member kind	Simple rule	Common purpose
Abstract instance method	No body; implementing class must supply one unless it stays abstract	Required behavior
default method	Has an inherited instance body	Shared optional behavior
static method	Belongs to the interface; call through its name	Helper tied to the role
private method	Used inside the interface's own method bodies	Avoid repeated internal code
Field	Always public static final	A true shared constant

An interface does not have ordinary instance fields or constructors. A constant reference can point to a changeable object, so do not publish a mutable list as a supposed safe constant. Keep contracts small and focused.

A default method uses default before its return type. For example, default boolean isReady() { return true; } gives a simple inherited answer. A class may override it. Adding a default method can help an interface grow, but a bad default can still change meaning or create method conflicts. Review existing implementations.

Several roles and conflicts

A class can implement several interfaces, separated by commas. An interface can extend several interfaces. A concrete class must satisfy all required methods. When one public method fits two matching abstract declarations, one implementation can satisfy both contracts if their rules agree.

If two unrelated interfaces provide the same default method, the class must resolve the conflict. It can write its own method. It may call a selected direct parent interface default with FirstRole.super.methodName(). A concrete inherited class method takes priority over an interface default. A more specific interface default can take priority over a less specific inherited default.

Do not use several roles whose rules contradict each other. A method that must always save data cannot also promise never to change stored data. Matching names alone do not prove that contracts fit.

Functional interfaces and lambdas

A functional interface has one abstract method after inheritance rules are considered. It may also have default or static methods. @FunctionalInterface asks the compiler to check this rule. A lambda is a short expression that supplies the behavior of that one method.

For MessageSender, the statement MessageSender sender = text -> System.out.println(text); creates a sender without a separate named class. The text before -> is the input. The text after it is the action. A block body can contain several statements. A lambda can use an outside local variable only if that variable is final or effectively final, meaning it is not reassigned.

Common standard functional interfaces include Predicate<T>, a test returning true or false; Function<T, R>, a conversion from T to R; Consumer<T>, an action taking T; and Supplier<T>, a source of T. A method reference such as System.out::println is another short way to supply a matching behavior. Use short forms only when they stay easy to read.

Generic interfaces and collections

A generic interface uses a type placeholder. For example, interface Repository<T> { void add(T item); } can describe storage for different item types. Repository<Book> accepts Book values. This avoids many casts and catches wrong types during compilation.

Java's collection design uses interfaces in this way. Write List<String> titles = new ArrayList<>(); when the code needs List behavior. ArrayList is one implementation. You may later choose another implementation if it meets the same needs. The empty <> asks the compiler to infer the type arguments.

List<Book> is not a subtype of List<Object>. If it were, code could add an unrelated object to a book list. A wildcard such as List<? extends Book> can safely provide Book values to a reader, but that reader generally cannot add arbitrary Book values. List<? super Book> can accept Book values, but reading yields Object without a more specific guarantee. Start with plain generic types and use wildcards when a method needs that flexibility.

Choosing an interface

Use an interface for a role that different implementations can share. Prefer a small MessageSender to an interface called EverythingService with unrelated methods. A class should not have to implement many useless actions.

An interface states what can be done. A class explains how one implementation does it. An abstract class, introduced next, can also hold common object state and construction rules. Interfaces and abstract classes often work together rather than replacing each other.

Practical activity

Purpose: Swap behavior without changing the service. Tools: The JDK and an editor. Task: Test a reminder without sending a real message.

1. Run InterfaceDemo and record its output.
2. Create a RecordingSender that implements MessageSender and stores the last text in a private field.
3. Supply RecordingSender to ReminderService. Call remind with Maya.
4. Use a getter to check the saved message. Do not contact an email service.
5. Replace the sender with a lambda that prints MESSAGE: followed by the text.
6. Explain why ReminderService did not need a new remind method.

Expected result: The same service works with both senders. The recording sender makes the message easy to check.

Practice: Define Searchable with a boolean matches(String query) method. Write a class that implements it. Add a second role only if it has a clear purpose.

Chapter summary

An interface defines a role and its promises. Classes implement the role. Interface types let client code use different implementations. Default, static, and private interface methods have different jobs. Functional interfaces support lambdas. Generic interfaces make shared designs safer for different data types.

Review and exam questions

1. Explain an interface contract without using the word contract.
2. Why must ConsoleSender.send be public?
3. What makes an interface a functional interface?
4. How should a class resolve two unrelated matching default methods?
5. Why can a test sender be useful even when it sends nothing?
6. Explain why List<Book> cannot simply be treated as List<Object>.

Chapter 4 - Abstraction - I

Why some classes stay incomplete

An abstract class is a class that cannot be created directly with new. It represents a shared base whose full behavior belongs to more specific children. It can store fields, run a constructor, and contain ordinary methods. It may also declare abstract methods, which have no body.

In a library, both a PrintedBook and a DigitalBook have a title. They may produce different access instructions. A general LibraryItem can store the title and require an access instruction without pretending that every item works in the same way.

AbstractClassDemo.java

```
abstract class LibraryItem {
    private final String title;

    protected LibraryItem(String title) {
        if (title == null || title.isBlank()) {
            throw new IllegalArgumentException("Title is required");
        }
        this.title = title.trim();
    }

    public final String getTitle() { return title; }

    public abstract String accessInfo();

    public String label() {
        return getTitle() + " - " + accessInfo();
    }
}

class PrintedBook extends LibraryItem {
    private final String shelf;

    PrintedBook(String title, String shelf) {
        super(title);
        this.shelf = shelf;
    }

    @Override
    public String accessInfo() { return "Shelf " + shelf; }
}

public class AbstractClassDemo {
    public static void main(String[] args) {
        LibraryItem item = new PrintedBook("Java Basics", "A2");
        System.out.println(item.label());
    }
}
```

The output is Java Basics - Shelf A2. The reference type LibraryItem is legal. The expression new LibraryItem(...) would not be legal. The object actually created is a PrintedBook. Its construction still calls the abstract parent's constructor to set the title.

What an abstract class can contain

An abstract class can have private state and methods that protect it. It can have concrete methods, meaning methods with bodies. It can have static methods and final methods. It can implement an interface and leave some required methods for its children.

An abstract class does not have to declare an abstract method. Sometimes a designer only wants to stop direct construction of a shared base. However, abstract and final cannot be combined on the same class: one asks for children to complete the design, while the other forbids children.

The parent constructor should establish valid shared state. The child constructor adds its own valid state. Avoid leaving a parent object partly usable with many null fields. For a real PrintedBook, also validate the shelf. Validation is kept small in the example so the main inheritance idea is easy to see.

Shared state and safe behavior

The title field is private and final. Every child uses the same title rule. The final getter prevents a child from changing what getTitle means. The label method is concrete but calls accessInfo, which each child supplies. Shared code and changing code are kept in clear places.

This is useful when several types follow a common process but differ in one step. For example, a report may always load data, build rows, and save output. A child can supply only the output step. Chapter 5 develops this pattern at the method level.

Do not make every method abstract without a reason. That can remove the benefit of sharing a base. Do not place every small difference into a long parent if-else chain either. Use the common behavior only where it is truly common.

Interface and abstract class comparison

Question	Interface	Abstract class
Main goal	Describe a role or capability	Share a family base and some implementation
Ordinary instance fields	No	Yes
Constructors	No	Yes
Several direct parents of same kind	May extend several interfaces	May extend only one class
Methods with bodies	default, static, and private methods	Ordinary, static, final, and private methods
Direct object creation	No	No
Typical library example	Searchable or MessageSender	LibraryItem with a shared title

A concrete class can extend one abstract class and implement several interfaces. This is common. For example, PrintedBook can extend LibraryItem and implement Searchable. The parent holds the shared title. The interface describes matching a search query.

A good abstract base

A useful abstract base has a small purpose and a clear rule for children. It should explain which methods must be supplied and which rules must stay true. Its ordinary methods should not depend on undocumented child behavior.

For example, say that accessInfo must return useful non-null text. If a child returns null, label may print confusing output. Java can require the method signature, but it cannot prove every written rule. Tests and documentation are still needed.

Use composition when children need very different data or steps. If every child ignores most of a base class, the family may be wrong. Interfaces plus separate helper objects may fit better. Abstract classes are a design tool, not a requirement for every project.

Practical activity

Purpose: Share title handling while adding a new item type. Tools: The JDK and an editor. Task: Add a DigitalBook with download instructions.

1. Compile and run AbstractClassDemo in its own folder.

2. Add DigitalBook extends LibraryItem with a private format field, such as PDF.

3. Write its constructor and call super(title).

4. Override accessInfo to return Open the PDF file for a PDF item.

5. Create one printed item and one digital item. Call label on each.

6. Temporarily try new LibraryItem("Test"). Read the compiler error, then remove the line.

Expected result: Both items use one title rule and one label format. Their access instructions differ. The abstract base cannot be created directly.

Practice: Add a blank-format check to DigitalBook. Decide whether title should have a setter. Explain how a setter would affect the design.

Chapter summary

An abstract class is a shared base that cannot be created directly. It can hold state, constructors, and completed methods. Children finish missing behavior. Interfaces describe roles, while abstract classes can share family state and code. A small, clear base is easier to maintain than a large, rigid one.

Review and exam questions

1. Why can an abstract class have a constructor if it cannot be created directly?

2. Must an abstract class declare an abstract method? Explain.

3. Why is final abstract class invalid?

4. Choose an interface or abstract class for a shared reminder role. Explain your choice.

5. Create DigitalBook and predict the result of label.

6. Give one sign that a proposed abstract base is too broad.

Chapter 5 - Abstraction - II

A method without a body

An abstract method states an action that a child must complete. It gives a name, parameter list, return type, and any declared checked exceptions. It ends with a semicolon instead of a method body. A checked exception is a failure type that the compiler requires code to handle or declare. Chapter 12 explains exceptions fully.

The declaration public abstract int loanDays(); says that an item must report its loan period as a whole number. It does not choose the period. A child can return 14 for a printed book and 7 for a digital item. The parent can then use that answer in shared work.

Abstract classes and abstract methods are connected but different. The class says that the whole type is not directly complete. The method identifies an exact piece of required behavior. A non-abstract child must supply every abstract method it inherits. An abstract child may leave some unfinished.

Valid and invalid declarations

Declaration	Validity and reason
public abstract int loanDays();	Valid inside an abstract class
protected abstract String format();	Valid; children can complete it
private abstract void check();	Invalid; a child cannot override a private method
static abstract void run();	Invalid; static methods are not completed by instance overriding
final abstract void save();	Invalid; final prevents the required override
abstract void open() {}	Invalid; an abstract method has no body

An ordinary class cannot declare an abstract method. Add abstract to the class or give the method a body. A constructor cannot be abstract because it performs setup rather than an inherited action that a child overrides.

In an interface, a plain instance method declaration is normally already public and abstract. The method rules still matter. An implementation must be public, use matching parameters, and have a compatible return type. A child cannot add a broader checked exception to an override. Use @Override to catch errors early.

A fixed process with a changing step

The template method pattern puts a shared sequence into one method and lets children supply selected steps. A pattern is a common way to solve a recurring design problem. Here, all loan labels follow the same sequence, but the loan period changes by item type.

AbstractMethodDemo.java

```
abstract class LoanRule {
    protected abstract int loanDays();

    public final String makeLabel(String title) {
        if (title == null || title.isBlank()) {
            throw new IllegalArgumentException("Title is required");
        }
        int days = loanDays();
        if (days <= 0) {
            throw new IllegalStateException("Loan days must be positive");
        }
        return title.trim() + ": return within " + days + " days";
    }
}
```

```
class PrintedLoanRule extends LoanRule {
    @Override
    protected int loanDays() { return 14; }
}

class DigitalLoanRule extends LoanRule {
    @Override
    protected int loanDays() { return 7; }
}

public class AbstractMethodDemo {
    public static void main(String[] args) {
        LoanRule printed = new PrintedLoanRule();
        LoanRule digital = new DigitalLoanRule();
        System.out.println(printed.makeLabel("Java Basics"));
        System.out.println(digital.makeLabel("Web Guide"));
    }
}
```

The two lines end with 14 days and 7 days. makeLabel is final, so children keep its title check, day check, and output structure. loanDays is abstract, so each concrete rule must choose a period. A bad child returning 0 is rejected by the shared method.

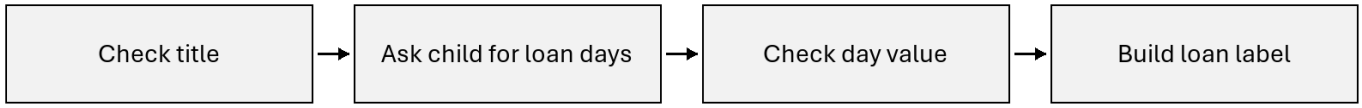


Figure 2 Shared loan label process

The parent's call to loanDays uses the current object's implementation. It is not a call to an empty body. At run time, the object is a PrintedLoanRule or DigitalLoanRule. This behavior leads directly to polymorphism in Chapter 6.

Signatures and return values

A method signature includes its name and parameter types. Return type alone cannot distinguish two overloads. Methods int read() and String read() with no parameters cannot both be declared in one class. They have the same signature.

Parameters must match when overriding. loanDays(int age) does not override loanDays(). It is a different method. A child that writes only the first one has not completed the parent's no-argument method. Parameter names may differ because their names are not what selects the method.

A reference return type can become more specific in an override. If a parent returns LibraryItem, a child may return PrintedBook when PrintedBook is a LibraryItem. A primitive return type must match the required type. Do not replace an int return with long and call it an override.

Write down what a returned value means. Does loanDays return a positive number? Does search return an empty list or null when nothing matches? Clear meaning prevents children and callers from making different guesses. Prefer an empty collection for no collection results when that fits the contract.

Hooks and required steps

A required step belongs in an abstract method. An optional step can be a concrete method with a safe default, sometimes called a hook. For example, protected void afterSave() {} lets a child add a follow-up action without forcing every child to do so. The empty body is a completed no-action behavior, not an abstract method.

Keep the shared process easy to follow. Too many hooks make its order difficult to understand. Do not call overridable hooks from a constructor without careful design, because child fields may not be ready. Use ordinary methods after construction.

When another design fits better

Abstract methods tie changing behavior to subclasses. That is useful for a stable family. If the loan period must change while a program runs, a separate LoanPolicy interface passed into a service may be easier. This is the

strategy pattern: a helper object holds a choice of behavior. The service can receive a different strategy without changing its own class.

For example, a holiday loan policy may temporarily replace a normal policy. Making a new subclass of every book type for every holiday would create too many classes. One policy object can serve many books. Choose the design that expresses the change simply.

Practical activity

Purpose: Complete a required method and test the shared process. Tools: The JDK and an editor. Task: Add a short loan rule for a magazine.

1. Run AbstractMethodDemo and record both output lines.
2. Add MagazineLoanRule extends LoanRule.
3. Implement loanDays to return 3. Call makeLabel for School News.
4. Temporarily remove loanDays from the concrete magazine class. Read the compiler error and restore it.
5. Return 0 once. Check that makeLabel rejects the result, then restore 3.
6. Try to override makeLabel. Observe the final-method error, then remove the attempted override.

Expected result: The new rule uses the common process and produces a three-day label. Missing implementations fail at compile time. A broken day rule fails during execution.

Practice: Design an abstract ReportExporter with a final export method and one required writeRows step. State which errors its contract should allow.

Chapter summary

An abstract method requires a child implementation. Its name, parameters, return type, access level, and exception rules must fit the parent declaration. A template method shares a process and calls child-supplied steps. A separate strategy object may fit behavior that must change independently.

Review and exam questions

1. Compare an abstract method with a concrete method whose body is empty.
2. Why is a private abstract method invalid?
3. Does loanDays(int age) complete loanDays()? Explain.
4. Why does makeLabel validate the value supplied by a child?
5. Design a new rule for a two-day reference loan.
6. Choose a template method or strategy object for a changeable holiday policy. Explain.

Chapter 6 - Polymorphism

One type many behaviors

Polymorphism means that one shared type can refer to objects with different implementations. The same instance method call can run different code based on the actual object. A library service can process student and teacher members through the Member type without separate versions of every method.

There are two types to keep in mind. The declared type is written on the variable, parameter, or field. It controls which members the compiler lets you call. The actual object type is created with new. It selects an overridden instance method during execution. This run-time choice is called dynamic dispatch.

PolymorphismDemo.java

```
import java.util.List;

abstract class MemberRole {
    private final String name;

    MemberRole(String name) { this.name = name; }
    public String getName() { return name; }
    public abstract int loanLimit();
}

class StudentRole extends MemberRole {
    StudentRole(String name) { super(name); }
    @Override
    public int loanLimit() { return 3; }
}

class TeacherRole extends MemberRole {
    TeacherRole(String name) { super(name); }
    @Override
    public int loanLimit() { return 5; }
}

public class PolymorphismDemo {
    static void printLimit(MemberRole member) {
        System.out.println(member.getName() + ": " + member.loanLimit());
    }

    public static void main(String[] args) {
        List<MemberRole> members = List.of(
            new StudentRole("Maya"), new TeacherRole("Luis"));
        for (MemberRole member : members) {
            printLimit(member);
        }
    }
}
```

The program prints Maya: 3 and Luis: 5. printLimit has one parameter type and one body. A new role can be added without changing that body if it keeps the MemberRole promises. This is one reason to separate a contract from its implementations.

Tracing a call

For the first loop step, member has declared type MemberRole and refers to a StudentRole object. The compiler accepts loanLimit because MemberRole declares it. When the method runs, Java chooses StudentRole.loanLimit. In the next step, the reference points to a TeacherRole, so Java chooses TeacherRole.loanLimit.

This choice applies to overridden instance methods. It does not make fields polymorphic. Static methods are selected from the compile-time type rules rather than by the same run-time override process. Constructors do not use overriding. Private methods are not overridden, and final methods cannot be overridden.

Overloading and overriding

Overloading means having the same method name with different parameter lists. Overload choice happens at compile time using the available declared types and conversion rules. Overriding replaces an inherited instance method's implementation in a child. Its run-time choice uses the actual object.

Feature	Overloading	Overriding
Main question	Which parameter list matches this call?	Which object implementation runs?
Where it appears	Same class or inherited overload set	Parent and child instance methods
Parameters	Must differ	Must match
Selection	Compile time	Run time for the chosen instance method
Typical purpose	Convenient inputs for one kind of task	Different behavior through one shared type

Suppose methods print(MemberRole x) and print(StudentRole x) both exist. A variable declared MemberRole still selects print(MemberRole), even when it refers to a StudentRole. After an overload is chosen, any overridden instance behavior inside that chosen method can still use dynamic dispatch. Keep overload sets simple so the choice is clear.

Upcasting and downcasting

Upcasting means using a child object through a parent or interface type. MemberRole m = new StudentRole("Maya"); is safe without an explicit cast. It does not change the object or remove its child fields. It changes the type through which code views the object.

Downcasting asks for a more specific view. An incorrect cast can throw ClassCastException. Check first with instanceof when a specific child feature is truly needed. In Java 21, if (m instanceof StudentRole student) { ... } both tests the type and supplies a checked local reference named student. The test is false for null.

Frequent downcasts can be a design warning. If a report checks every possible role with a long instanceof chain, a shared report method or strategy may be clearer. A cast is useful at some system boundaries, but it should not replace a good common contract.

Collections and equality

Polymorphic collections hold different objects through one element type. List<MemberRole> can hold both roles. List.of creates an unmodifiable list: add and remove are not allowed. Its elements may still be changeable objects. Use new ArrayList<>() when the list itself must grow.

HashSet and HashMap depend on equals and hashCode. If books are equal by ID, both methods must use that rule. Never override equals without a matching hashCode rule. A record can supply suitable equality for all its components when that is the equality you need. Do not use a record automatically if the real rule is based on only one component.

Sorting can also use a shared behavior. Comparable<T> lets a type define its natural order with compareTo. Comparator<T> defines an outside order, such as title order today and ID order tomorrow. Return a negative value, zero, or a positive value for less than, equal order, or greater than. Use Integer.compare(a, b), not a - b, because subtraction may overflow.

Benefits and limits

Polymorphism reduces repeated type-specific code. It helps a service accept a real sender, a test sender, or a different storage implementation. It works best when all implementations keep the same promises about input, output, and failure.

It does not remove the need for clear data design. If one implementation silently loses records while another saves them, sharing a save method name is not enough. State rules, validate results, and test each implementation through the shared contract.

Practical activity

Purpose: Add a role without changing shared processing. Tools: The JDK and an editor. Task: Add a GuestRole with a one-book loan limit.

1. Run PolymorphismDemo.
2. Add GuestRole extends MemberRole and implement loanLimit.
3. Add a GuestRole object to the List.of arguments.
4. Run the same loop and printLimit method without changing their bodies.
5. Make a trace table with declared type, actual type, selected method, and result for each object.
6. Explain why no cast was needed.

Expected result: The three roles print limits of 3, 5, and 1. The shared method handles all three.

Practice: Write two overloads, inspect a parent-typed variable, and predict the chosen overload before running it. Keep this experiment separate from the main example.

Chapter summary

Declared types control available calls. Actual object types select overridden instance behavior. Overloading uses parameter lists at compile time. Upcasting is a safe move to a shared view. Downcasting needs care. Polymorphism makes services and collections work with several implementations through one clear contract.

Review and exam questions

1. Trace member.loanLimit for the student and teacher in the example.
2. Compare declared type and actual object type.
3. Why does changing a reference type not change the object itself?
4. Explain one difference between overloading and overriding with an example.
5. When could a downcast fail? Show a safe type check.
6. Why must equal objects have equal hash codes?

Chapter 7 - File Operations

Why programs use files

Objects in memory normally disappear when a program ends. A file keeps data outside the running program. Input means reading data into a program. Output means writing data out. The library can save book records in a file and load them when it starts again.

Text files store readable characters using a character encoding. An encoding is a rule that maps characters to bytes. Use explicit UTF-8 for shared text. Binary files store bytes in a format chosen by the program. A normal text editor may not show them meaningfully. Neither format is automatically a database.

Paths folders and resources

Path describes a file or folder location. Files provides operations such as reading, writing, copying, and creating folders. Path.of("data", "books.txt") builds a path from parts. A relative path starts from the program's working folder. An absolute path starts from a full system location. The working folder is not always the source folder, especially in an IDE.

Create needed folders with Files.createDirectories. Do not assume that a file exists or can be opened just because a previous check succeeded. Another program may change it between the check and use. Handle the actual operation's exception. Avoid building a server file path directly from unchecked user text.

An open stream or reader is a resource that should be closed. try-with-resources closes its declared resources whether the work succeeds or throws an exception. Closing an outer buffered wrapper also closes its wrapped stream. A buffer is a temporary memory area that reduces many small physical reads or writes.

Choosing file classes

Need	Common choice	Important detail
Small whole text file	Files.readString and Files.writeString	Holds the whole text in memory
Large line-based text	BufferedReader and BufferedWriter	Read and write one line at a time
Raw bytes	InputStream and OutputStream	Good for images or custom byte formats
Primitive binary values	DataInputStream and DataOutputStream	Reader must follow the writer's exact order
Saved Java objects	ObjectInputStream and ObjectOutputStream	Native serialization; see Chapter 8
Move within a binary file	RandomAccessFile	Requires a known record layout and offsets

Reader and Writer work with characters. InputStream and OutputStream work with bytes. InputStreamReader bridges bytes to characters with an encoding. OutputStreamWriter does the reverse. Do not copy an image through a text reader, because character conversion can change its bytes.

A simple text format for objects

An object is not automatically written to a text file just by printing its reference. Choose a format that can be read back. The next example stores one ID and one title per line, separated by a tab. A tab is written as \t in a Java string. This teaching format rejects tabs and line breaks inside a title. Real CSV files need proper rules for commas, quotes, and line breaks; splitting on commas alone is not a general CSV reader.

TextFileDemo.java

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
```

```
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.List;

public class TextFileDemo {
    record BookRow(int id, String title) {
        BookRow {
            if (id <= 0 || title == null || title.isBlank()
                    || title.contains("\t") || title.contains("\n")
                    || title.contains("\r")) {
                throw new IllegalArgumentException("Invalid book row");
            }
        }
    }

    static void save(Path file, List<BookRow> books) throws IOException {
        try (BufferedWriter out = Files.newBufferedWriter(
                file, StandardCharsets.UTF_8)) {
            for (BookRow book : books) {
                out.write(book.id() + "\t" + book.title());
                out.newLine();
            }
        }
    }

    static List<BookRow> load(Path file) throws IOException {
        List<BookRow> books = new ArrayList<>();
        try (BufferedReader in = Files.newBufferedReader(
                file, StandardCharsets.UTF_8)) {
            String line;
            int lineNumber = 0;
            while ((line = in.readLine()) != null) {
                lineNumber++;
                String[] fields = line.split("\t", -1);
                if (fields.length != 2) {
                    throw new IOException("Bad row at line " + lineNumber);
                }
                try {
                    books.add(new BookRow(
                        Integer.parseInt(fields[0]), fields[1]));
                } catch (IllegalArgumentException ex) {
                    throw new IOException(
                        "Bad value at line " + lineNumber, ex);
                }
            }
        }
        return books;
    }

    public static void main(String[] args) throws IOException {
        Path folder = Path.of("data");
        Files.createDirectories(folder);
        Path file = folder.resolve("books.tsv");
        save(file, List.of(new BookRow(101, "Java Basics"),
                           new BookRow(102, "Web Guide")));
        for (BookRow book : load(file)) {
            System.out.println(book.id() + ": " + book.title());
        }
    }
}
```

The output is 101: Java Basics and 102: Web Guide on separate lines. readLine returns null at the end of the file. It returns an empty string for an empty line, so an empty line and end of file are different. split with -1 keeps an empty final field so validation can detect a missing title.

The program builds a temporary list while loading. If one line is bad, it reports failure rather than returning a partly loaded list as if it were complete. A real import tool may instead collect bad rows and continue, but that must be a clear, reported policy. Also check duplicate IDs when the system requires unique IDs.

Primitive values in a binary file

DataOutputStream writes primitive values in a defined binary form. DataInputStream reads matching values. The next example writes a format version, one count, and one title. Both streams are wrapped in buffers.

BinaryFileDemo.java

```
import java.io.*;
import java.nio.file.*;

public class BinaryFileDemo {
    public static void main(String[] args) throws IOException {
        Path file = Path.of("stock.bin");
        try (DataOutputStream out = new DataOutputStream(
                new BufferedOutputStream(Files.newOutputStream(file)))) {
            out.writeInt(1);
            out.writeInt(12);
            out.writeUTF("Java Basics");
        }
        try (DataInputStream in = new DataInputStream(
                new BufferedInputStream(Files.newInputStream(file)))) {
            int version = in.readInt();
            if (version != 1) {
                throw new IOException("Unsupported format version");
            }
            int copies = in.readInt();
            String title = in.readUTF();
            if (copies < 0) {
                throw new IOException("Negative copy count");
            }
            System.out.println(title + ": " + copies);
        }
    }
}
```

Use writeBoolean/readBoolean, writeLong/readLong, and writeDouble/readDouble for their matching types. An int uses four bytes. In this example, the two ints use 8 bytes. The ASCII title Java Basics uses 11 modified UTF-8 bytes. writeUTF adds a two-byte length, so the file is 8 + 2 + 11 = 21 bytes. This calculation applies to this title, not to every possible string.

writeUTF uses modified UTF-8 and limits one encoded string to 65,535 bytes. It is not a general replacement for standard UTF-8 text files. A binary reader must follow the writer's type order. Reading a double where an int was written does not convert the stored field correctly. A partly written value causes EOFException when a reader cannot obtain all required bytes. EOF is only normal where the file format says it is allowed. [Java DataInput reference]

Appending replacing and random access

Many Files write methods create or replace a file by default. For a log, explicit CREATE and APPEND options can add text. Appending a new record is suitable only if the format supports it. Updating one record by appending may create two conflicting copies.

A safer replacement process writes a complete new file in the same folder, closes it, and then moves it over the target. Request an atomic move when supported. Atomic means other code sees the old file or the new file rather than a half-replaced file. Atomic moves are not supported everywhere. Handle AtomicMoveNotSupportedException and choose a documented fallback with a backup when needed. A successful move is not a promise that data will survive every power failure.

RandomAccessFile can seek to a byte offset. If each record is exactly 12 bytes, record index 3 starts at 3 x 12 = 36 bytes after the data start. Indexes here begin at 0. Variable-length titles make such offsets harder. Keep an index of positions or use a database when records need many searches and updates.

Common problems and good habits

Use clear file formats with a version, field order, encoding, and limits. Reject absurd record counts before allocating large arrays. Do not use available() to decide that a stream has reached its end; it does not mean total remaining file data. Do not use ready() as a general end-of-file test for text. Read the data and follow the return value or documented EOF rule.

Handle missing files, permission failures, full disks, bad numbers, and corrupted rows. Show a helpful message at the user boundary. Keep technical details in a log when appropriate. Do not report Saved until writing and closing succeed. Use a copy of classroom data when trying failure cases.

Practical activity

Purpose: Save and rebuild values and objects. Tools: The JDK, an editor, and a normal text editor for inspecting the TSV file. Task: Save three library records, then test one bad record.

1. Run TextFileDemo. Open data/books.tsv and inspect the two rows.
2. Add a third BookRow and run again. Confirm the new row is loaded.
3. To test a damaged file, temporarily skip the save call so the next run will not overwrite your edit. Change one ID to abc. Run load and record the line-specific error.
4. Restore the valid file and original save call.
5. Run BinaryFileDemo. Check the file size. Explain the 21-byte result.
6. Change copies to 0 and confirm that zero is accepted. Test a negative count by changing the written value and observe the read check.

Expected result: Valid text objects are rebuilt correctly. Invalid text produces a clear error. The binary values are recovered in the same order in which they were written.

Practice: Design a format for ID, title, and available. State how it represents tabs or rejects them. Explain how you would avoid losing the last good file during a failed save.

Chapter summary

Files preserve data after a program ends. Choose text for readable exchange and binary for a defined byte layout. Use explicit encoding, matching read/write rules, validation, and try-with-resources. Object text storage needs a clear mapping between fields and file values. A safe save process protects the last good data.

Review and exam questions

1. Compare a byte stream with a character reader.
2. Why must a binary reader follow the writer's type order?
3. Explain the difference between an empty line and end of file.
4. A binary file holds one int and one long. How many data bytes are needed, excluding any header?
5. Why is split(",") not enough for all CSV files?
6. Describe how to test a failed save without risking original student data.

Chapter 8 - Serialization

What serialization does

Serialization converts an object's saved state into bytes. Deserialization rebuilds objects from those bytes. Java's native object serialization uses ObjectOutputStream and ObjectInputStream. It can follow references to other serializable objects and preserve shared references within an object graph. An object graph is the set of connected objects reached through those references.

This differs from Chapter 7's text mapping. In text mapping, we chose each field and rebuilt each object ourselves. Native serialization follows Java serialization rules and records Java class information. It is convenient for a controlled lesson, but it tightly connects stored data to Java classes.

Serializable and saved fields

A class uses implements Serializable to allow normal native serialization. Serializable is a marker interface: it has no required methods. Its presence signals permission for this mechanism. A referenced object must also be serializable unless the reference is excluded or handled by custom code. Otherwise writing may throw NotSerializableException.

Default serialization saves non-static, non-transient instance fields. static fields belong to the class, so their current values are not normal object state in the file. transient marks an instance field that should be skipped by default. A skipped int normally becomes 0 after reading, a boolean false, and an object reference null. A field initializer in an ordinary serializable class does not automatically run again during deserialization.

serialVersionUID is a long class version identifier used in compatibility checks. Declare it explicitly. A matching UID does not prove that two versions have the same meaning or that every change is safe. Class changes can still need migration and validation.

A controlled round trip

Use this example only with the local file that the same program creates. Do not load a file received from a stranger, a web upload, or an unknown download. Deserializing untrusted bytes is dangerous. A filter narrows what can be read, but it does not make all untrusted native serialization safe. [Java ObjectInputFilter reference]

SerializationDemo.java

```
import java.io.*;
import java.nio.file.*;

public class SerializationDemo {
    static final class SavedBook implements Serializable {
        @Serial
        private static final long serialVersionUID = 1L;
        private final int id;
        private final String title;
        private transient int searchCount = 99;

        SavedBook(int id, String title) {
            if (id <= 0 || title == null || title.isBlank()) {
                throw new IllegalArgumentException("Invalid book");
            }
            this.id = id;
            this.title = title;
        }

        @Serial
        private void readObject(ObjectInputStream in)
                throws IOException, ClassNotFoundException {
            in.defaultReadObject();
            if (id <= 0 || title == null || title.isBlank()
                    || title.length() > 200) {
                throw new InvalidObjectException("Invalid saved book");
            }
        }
```

```
        @Override
        public String toString() {
            return id + ": " + title + ", searches=" + searchCount;
        }
    }

    public static void main(String[] args)
            throws IOException, ClassNotFoundException {
        Path file = Path.of("trusted-book.ser");
        SavedBook original = new SavedBook(101, "Java Basics");
        try (ObjectOutputStream out = new ObjectOutputStream(
                Files.newOutputStream(file))) {
            out.writeObject(original);
        }

        try (ObjectInputStream in = new ObjectInputStream(
                Files.newInputStream(file))) {
            ObjectInputFilter filter = ObjectInputFilter.Config.createFilter(
                "maxdepth=5;maxrefs=20;maxbytes=4096;"
                + "maxarray=100;SerializationDemo$SavedBook;!*"
            );
            in.setObjectInputFilter(filter);
            Object value = in.readObject();
            if (!(value instanceof SavedBook restored)) {
                throw new InvalidObjectException("Unexpected object type");
            }
            System.out.println(restored);
        }
    }
}
```

The output is 101: Java Basics, searches=0. The transient searchCount was skipped. Its initializer value of 99 was not applied again while reading this ordinary serializable class. readObject calls defaultReadObject, then checks the restored values. This protects the class rule because normal serializable class constructors do not perform the usual setup on the read path.

SavedBook is a nested class, meaning a class declared inside another class. Its static form does not need an enclosing SerializationDemo object. @Serial marks fields and methods intended for Java's serialization mechanism. It helps tools check their intended use; it does not make data safe by itself.

The first non-serializable superclass must have an accessible no-argument constructor, and that constructor is called. Serialization rules for records and other special types differ. Do not assume that every Java type follows the ordinary class construction path.

Understanding the filter

The filter allows this one SavedBook class and limits depth, references, bytes checked by the mechanism, and array size. The $ is part of the binary name of a nested class. The final !* rejects other class names. Filters do not inspect every business value and do not apply to every kind of data in the same way. For example, strings written directly in the stream do not receive ordinary class filtering callbacks. The readObject title check is still needed.

Set the filter before calling readObject. A type check after reading is useful but happens after deserialization has already done work. It cannot undo unsafe actions caused while an unknown object was being rebuilt. Prefer a simple documented data format and strict parser for outside data.

Graphs identity and repeat writes

Suppose two Loan objects refer to the same Member. When one stream writes that graph, it normally records the shared identity. After reading, the restored loans can refer to the same restored member. This avoids accidentally making two different copies of a shared object. Cycles can also be represented when the connected objects meet the rules.

ObjectOutputStream remembers objects it has already written. Writing the same object again after changing it may write a reference to its earlier saved form rather than a fresh independent state. reset can clear the

stream's remembered object state, but it is not a general data-update system. Define whether a stream holds a snapshot or a sequence of independent records.

Do not open a new ObjectOutputStream in append mode each time you add one object to an existing serialization file. Each new stream writes a header. Extra headers can break a normal reader. A beginner-friendly design writes one complete snapshot with one stream and replaces the old file safely.

Choosing a storage format

Format	Strength	Limit
Plain text or TSV	Easy to inspect and edit	Needs escaping or clear restrictions
JSON with a suitable library	Common exchange format across languages	Needs parsing, validation, and a library in many Java projects
Custom primitive binary	Explicit compact layout	Must define version and read/write order
Native Java serialization	Saves supported Java object graphs	Class coupling and serious untrusted-input risks
Database	Searches, rules, and multiuser updates	Requires setup and transaction design

transient is not encryption. It only skips a field in default serialization. Other fields may still reveal private information, and custom code may save a skipped value in another way. Keep passwords and secret tokens out of saved classroom objects. Serialization is also not a backup plan by itself: copies, version changes, and restore tests still matter.

Externalizable is a more manual interface with writeExternal and readExternal. It gives the class responsibility for its format and requires a public no-argument constructor for normal reconstruction. It adds complexity and does not remove the risks of reading untrusted object streams. Learn ordinary data formats before choosing it.

Practical activity

Purpose: Observe object state across a controlled save and load. Tools: The JDK and an editor. Task: Save one known local book and explain the skipped field.

1. Run SerializationDemo in an empty practice folder.
2. Record the output and the created filename.
3. Change the original title to Web Guide, run again, and confirm the output.
4. Explain why searchCount is 0 rather than 99 after loading.
5. Keep the original version, then temporarily remove the allowed SavedBook class name from the filter. Confirm that the read is rejected. Restore the filter.
6. Write a short storage choice note for a public web upload. Choose a validated data format rather than native object deserialization.

Expected result: The known local object is rebuilt, the transient field is reset, and the restricted filter rejects an unapproved class.

Practice: Explain why a matching serialVersionUID is not enough to trust a file. Design a versioned text alternative for the same two book fields.

Chapter summary

Native serialization saves supported object graphs. Serializable permits the mechanism; transient skips selected normal instance fields; serialVersionUID helps with compatibility checks. Restored values still need validation. Use filters for controlled streams, avoid unknown serialized input, and choose a clear data format for outside exchange.

Review and exam questions

1. Compare serialization with manually writing book fields to a text file.
2. Which normal fields are skipped by default serialization?
3. Why does searchCount become 0 in the example?
4. Why is checking the type after readObject not a complete safety measure?
5. What can go wrong when a new object stream is repeatedly appended to a file?
6. Choose a storage format for sharing records with a non-Java system and explain.

Chapter 9 - Java GUI

From commands to screens

A graphical user interface, or GUI, lets a person use windows, buttons, text fields, menus, and lists. A console program often asks for input in a fixed sequence. A GUI usually waits for events. An event is something that happens, such as a button click, a key press, or a window closing.

Java Swing provides desktop GUI classes in the javax.swing package. Swing is included in the Java desktop module. The package name remains javax.swing; it has not changed to jakarta.swing. Swing is useful for this course because the simple examples do not need an extra GUI framework.

AWT, the Abstract Window Toolkit, supplies lower-level desktop support such as events, colors, and layout classes. Swing builds on that support. JavaFX is another GUI option that normally needs separate dependencies. This chapter teaches Swing so the focus stays on events, layout, validation, and object design.

Main parts of a Swing screen

Part	Swing class	Job in the library
Window	JFrame	Holds the main screen
Container	JPanel	Groups related controls
Label	JLabel	Shows a prompt or short result
Text input	JTextField	Accepts a search term
Button	JButton	Starts a search
List	JList	Shows matching titles
Scroll area	JScrollPane	Lets a list or text area scroll
Table	JTable	Shows records in rows and columns
Message dialog	JOptionPane	Shows a short error or message

A component is one visible control or container. Add components to containers. A layout manager decides their size and position. BorderLayout has north, south, east, west, and center areas. FlowLayout places controls in a row and wraps when needed. GridLayout makes equal-size cells. BoxLayout arranges a line of components. GridBagLayout supports more flexible grids but needs more setup.

Use layout managers instead of fixed pixel positions. Fixed positions can fail when a font, screen size, or language changes. Use pack to size a window from its components, then allow resizing. Put large lists and tables inside scroll panes.

The event dispatch thread

Swing normally expects component creation and access on the event dispatch thread, or EDT. The EDT is the thread that handles screen events. Use SwingUtilities.invokeLater to schedule the first screen setup there. Button listeners also run there. Keep listener work short so the window can still repaint and answer clicks. [Swing package reference]

A listener is an object that receives an event. An ActionListener has an actionPerformed method. A lambda can supply it because the interface has one abstract method. A listener should pass work to the model or service, then show the result. It should not become the entire program.

A complete search window

LibraryGui.java

```
import java.awt.BorderLayout;
import java.util.List;
import java.util.Locale;
import javax.swing.*;
```

```
public class LibraryGui {
    private final List<String> titles = List.of(
        "Java Basics", "Web Guide", "Data Skills");
    private final JTextField query = new JTextField(18);
    private final DefaultListModel<String> results =
        new DefaultListModel<>();
    private final JLabel status = new JLabel("Enter a title word.");

    private void search() {
        String text = query.getText().trim().toLowerCase(Locale.ROOT);
        if (text.isEmpty()) {
            status.setText("Enter at least one letter or word.");
            query.requestFocusInWindow();
            return;
        }
        results.clear();
        for (String title : titles) {
            if (title.toLowerCase(Locale.ROOT).contains(text)) {
                results.addElement(title);
            }
        }
        status.setText("Matches: " + results.size());
    }

    private void showWindow() {
        JFrame frame = new JFrame("School Library Search");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JPanel controls = new JPanel();
        JLabel prompt = new JLabel("Title word:");
        prompt.setLabelFor(query);
        JButton searchButton = new JButton("Search");
        controls.add(prompt);
        controls.add(query);
        controls.add(searchButton);
        searchButton.addActionListener(event -> search());
        query.addActionListener(event -> search());
        JList<String> list = new JList<>(results);
        list.setVisibleRowCount(6);
        list.getAccessibleContext().setAccessibleName("Matching titles");
        frame.add(controls, BorderLayout.NORTH);
        frame.add(new JScrollPane(list), BorderLayout.CENTER);
        frame.add(status, BorderLayout.SOUTH);
        frame.getRootPane().setDefaultButton(searchButton);
        frame.pack();
        frame.setLocationRelativeTo(null);
        frame.setVisible(true);
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> new LibraryGui().showWindow());
    }
}
```

Run javac --release 21 LibraryGui.java, then java LibraryGui on a computer with a desktop display. Type java and click Search. The list shows Java Basics and the status says Matches: 1. Type a word that is absent and the list becomes empty with Matches: 0. Pressing Enter in the text field also searches.

The title list is the small model in this example. The controls are the view. The listener passes the request to search, which joins input handling and a short in-memory search. A larger program should move the search into a separate service so the same logic can serve the console and the web app.

Keeping the screen responsive

Do not load a large file or wait for a network reply inside a normal button listener. That blocks the EDT and makes the screen look frozen. SwingWorker supports a background task. Put slow work in doInBackground.

Update Swing components in done or process, which run on the EDT. Do not update the label directly from doInBackground.

The following fragment belongs inside a GUI method. It assumes a JButton named button, a JLabel named label, and a background service method named loadReport that returns String. It is a pattern to adapt, not a separate complete program.

Fragment

```
button.setEnabled(false);
new SwingWorker<String, Void>() {
    @Override
    protected String doInBackground() throws Exception {
        return loadReport();
    }

    @Override
    protected void done() {
        try {
            label.setText(get());
        } catch (InterruptedException ex) {
            Thread.currentThread().interrupt();
            label.setText("Report was interrupted.");
        } catch (java.util.concurrent.CancellationException ex) {
            label.setText("Report was canceled.");
        } catch (java.util.concurrent.ExecutionException ex) {
            label.setText("Report could not be loaded.");
        } finally {
            button.setEnabled(true);
        }
    }
}.execute();
```

get obtains the task's result and reports a task failure through ExecutionException. In done, the task has already finished, so get does not wait for normal completion. A real app should record the technical cause in a log. Chapter 10 explains background task coordination.

Validation and usable screens

Validate at two places for different reasons. The GUI gives a quick helpful message near the input. The model or service enforces the real rule even if another interface calls it. A screen check alone cannot protect a web request or a direct service call.

For an ID field, trim text, try Integer.parseInt, check that the value is positive, and then ask the service whether the ID exists. Keep the user's valid input when another field fails. Disable a save button during a save when double submission would create duplicates. Reenable it whether the save succeeds or fails.

Use clear labels, a sensible tab order, keyboard actions, and accessible names. Do not use color alone to show success or failure. A text message works for grayscale printing and for users who do not see some colors. Ask before closing a document with unsaved changes when the app supports editing.

For a JTable, use a table model that describes rows and columns. Changing a normal ArrayList alone does not automatically tell every GUI component to redraw. Swing's list and table models send update events so the view can refresh.

Practical activity

Purpose: Build and test a simple event-driven screen. Tools: The JDK, an editor, and a desktop display. Task: Improve the library search window.

1. Run LibraryGui and search for java, WEB, and a missing word.
2. Click Search with an empty field. Check the message and focus.
3. Search by pressing Enter. Resize the window and confirm that controls stay usable.
4. Add two more titles to the model and test the results.
5. Add a Clear button that clears the query, results, and status message.

6. Write a test checklist for mouse use, keyboard use, empty input, no match, and resize.

Expected result: The window shows correct matches without restarting. Empty input gives a clear message. Resizing does not cut off a fixed drawing of controls.

Practice: Sketch a loan-entry screen with member ID, book ID, Lend, and a result area. Mark which checks belong in the screen and which belong in the service.

Chapter summary

A GUI responds to events. Swing components live in containers with layout managers. Create and use Swing controls on the EDT. Keep slow work off that thread and return results safely. A usable screen has clear labels, validation, keyboard support, and helpful messages.

Review and exam questions

1. Compare a console input sequence with event-driven input.
2. Why are layout managers safer than fixed positions?
3. What happens if a listener spends ten seconds reading a file on the EDT?
4. Which SwingWorker method should do slow work? Which should update the finished result?
5. Why must a service repeat important checks already made by the GUI?
6. Design three useful tests for the search window.

Chapter 10 - Multithreading

Processes threads and shared work

A process is a running program with its own main memory space. A thread is a path of execution inside a process. Threads in one Java process can share objects. Concurrency means several tasks make progress during overlapping periods. Parallelism means tasks actually run at the same time on different processing resources. Concurrent code may still run on one processor by taking turns.

In the library, one task may save a report while the GUI remains responsive. A server may answer several people. Threads can improve responsiveness and overlap waiting, but they add coordination work. More threads do not always make a CPU-heavy job faster.

Creating and starting a thread

Runnable represents a task that returns no result. Thread holds execution machinery. A call to start schedules a new thread, which then runs the task. Calling run directly is just an ordinary call on the current thread. Start a Thread object only once.

The following complete program waits for both tasks before printing its final message.

ThreadDemo.java

```
public class ThreadDemo {
    public static void main(String[] args) throws InterruptedException {
        Runnable scan = () -> {
            for (int i = 1; i <= 3; i++) {
                System.out.println(Thread.currentThread().getName()
                    + " checked shelf " + i);
            }
        };
        Thread first = new Thread(scan, "Worker-A");
        Thread second = new Thread(scan, "Worker-B");
        first.start();
        second.start();
        first.join();
        second.join();
        System.out.println("All shelves checked");
    }
}
```

The worker lines may appear in different orders on different runs. Do not write a test that expects one exact order unless the program enforces that order. join waits for a thread to end. It also provides memory visibility guarantees for the completed thread's work. sleep asks the current thread to pause for a requested time. Scheduling can delay its return, and interruption can end the wait early. It is not a safe substitute for coordination and does not release a held monitor lock.

Thread states

State	Meaning
NEW	Created but not started
RUNNABLE	Eligible to run or running in the JVM
BLOCKED	Waiting to enter a synchronized monitor
WAITING	Waiting without a timed limit for another action
TIMED_WAITING	Waiting with a time limit, such as sleep
TERMINATED	Finished execution

The exact state is a momentary observation. It may change immediately. Do not control correctness by repeatedly checking a thread state. Use locks, task results, queues, or other coordination tools with clear guarantees.

Race conditions and lost updates

A race condition occurs when a result depends on uncontrolled timing between tasks. Consider count++. It looks like one line, but it reads count, adds one, and writes the result. Two threads can both read 4, both compute 5, and both write 5. Two intended increments then produce only one visible increase.

Three related needs are atomicity, visibility, and ordering. Atomicity means an operation happens as one indivisible action from the other threads' view. Visibility means another thread can see a completed change under the memory rules. Ordering means changes happen in a relationship other threads can rely on. Correct synchronization provides these relationships.

SafeCounterDemo.java

```
public class SafeCounterDemo {
    static class Counter {
        private int value;

        synchronized void increment() { value++; }
        synchronized int get() { return value; }
    }

    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();
        Runnable task = () -> {
            for (int i = 0; i < 10000; i++) {
                counter.increment();
            }
        };
        Thread a = new Thread(task);
        Thread b = new Thread(task);
        a.start();
        b.start();
        a.join();
        b.join();
        System.out.println(counter.get());
    }
}
```

The expected output is 20000. A synchronized instance method locks its current object. Only one thread at a time can hold that monitor lock. A different Counter has a different lock. A static synchronized method locks the Class object. Code must use the same lock to protect the same shared rule.

volatile makes reads and writes of a field follow visibility and ordering rules, but it does not turn count++ into one atomic operation. AtomicInteger provides atomic single-value actions such as incrementAndGet. If a rule involves two fields, such as checking copies then reducing copies, protecting only one individual increment is not enough. Protect the whole check-and-change action with a suitable lock or transaction.

Executors and returned results

An executor runs submitted tasks and separates task logic from thread creation. A fixed thread pool uses a limited number of worker threads. Callable<T> returns a result and may throw an exception. Future<T> represents the pending result. get waits if needed and reports task failure through ExecutionException.

ExecutorDemo.java

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

public class ExecutorDemo {
    public static void main(String[] args)
            throws InterruptedException, ExecutionException {
        try (ExecutorService pool = Executors.newFixedThreadPool(2)) {
            Future<Integer> first = pool.submit(() -> 12 + 8);
            Future<Integer> second = pool.submit(() -> 5 + 10);
            int total = first.get() + second.get();
            System.out.println("Checked books: " + total);
        }
    }
}
```

The result is Checked books: 35. In this Java 21 example, ExecutorService supports try-with-resources. Closing waits for tasks to finish. For a long-running app, design explicit shutdown, timeout, and cancellation rules. Do not call blocking get on the EDT while a task is still running.

A fixed pool limits active workers, but the common factory uses a queue that can grow. A busy public server needs bounded queues or another clear admission policy. Admission policy means deciding how much work can enter and what happens when capacity is full.

Virtual threads are lightweight Java threads suited to many tasks that spend time waiting for I/O. Java 21 supplies Executors.newVirtualThreadPerTaskExecutor. Virtual threads do not make a CPU calculation itself faster and do not remove shared-data races. Do not create a small fixed pool of virtual threads just to imitate platform-thread limits. Limit scarce resources such as database connections separately. [Java Executors reference]

Cancellation and interruption

Interruption is a request for a thread to stop or change its waiting work. It is not forced safe termination. Some blocking methods throw InterruptedException. Code should normally stop the task or pass the exception upward. If it catches the exception and cannot pass it on, it often restores the flag with Thread.currentThread().interrupt() before returning.

For a CPU loop, check Thread.currentThread().isInterrupted() at sensible points and return when cancellation is requested. Future.cancel(true) attempts to interrupt a running task. It does not guarantee that all I/O or badly written work will stop immediately. Avoid the unsafe deprecated stop, suspend, and resume thread controls.

Deadlocks and coordination tools

A deadlock occurs when tasks wait forever for locks held by each other. For example, Task A holds the member lock and waits for the book lock. Task B holds the book lock and waits for the member lock. Prevent this by acquiring multiple locks in one fixed order, keeping locked sections short, and avoiding slow outside calls while holding locks.

A BlockingQueue safely passes data from producers to consumers. A producer adds a job; a consumer takes one. A bounded queue applies backpressure, which makes producers slow down when consumers cannot keep up. Use clear shutdown signals or interruption. This is usually easier than writing wait and notify code by hand.

If using wait and notifyAll, hold the correct monitor and wait inside a while loop that checks the condition. A wake-up does not prove that the condition is now true. Higher-level tools such as queues, latches, and futures are often easier to use correctly. ConcurrentHashMap supports safe concurrent map operations, but a larger rule across several calls may still need an atomic map method or a separate lock.

Practical activity

Purpose: Observe scheduling and protect shared state. Tools: The JDK and an editor. Task: Run a safe counter and compare a deliberately unsafe copy.

1. Run ThreadDemo several times. Record two different possible worker orders.
2. Run SafeCounterDemo and confirm 20000.
3. In a separate copy, remove synchronized from both methods. Increase the loop count and run several times.
4. Record any lost updates. If the result is correct on one run, explain why that does not prove safety.
5. Restore the safe version. Replace Counter with AtomicInteger as a separate exercise.
6. Run ExecutorDemo and explain where each result is obtained and where workers are closed.

Expected result: Scheduling order can vary. The correctly locked counter always meets its intended total under successful completion. The unsafe counter has no correctness guarantee even if a test happens to pass.

Practice: Two people try to borrow the last copy. State the complete action that must be protected. Draw a two-lock deadlock and then choose a lock order that avoids it.

Chapter summary

Threads share process objects and may run in unpredictable orders. Use start to begin a thread and join or futures to wait for work. Protect shared rules, not just individual lines. Executors manage tasks, interruption supports cooperative cancellation, and careful lock order helps avoid deadlocks. Background work must respect Swing's EDT rules.

Review and exam questions

1. Compare concurrency with parallelism.
2. Why does calling run differ from calling start?
3. Trace one lost update when two threads use count++.
4. Why is volatile not enough for count++?
5. What does Future.get do when a task fails?
6. Describe a deadlock and two ways to reduce its risk.

Chapter 11 - Java Web Applications

Browser requests and server replies

A web application runs server-side code to answer requests from a browser or another client. The browser sends an HTTP request. HTTP is a set of rules for web messages. The server handles the request and sends an HTTP response. A Java web app may return HTML for a page or JSON for another program.

A URL identifies a resource. In http://127.0.0.1:8080/search?q=java, http is the scheme, 127.0.0.1 is the local host address, 8080 is the port, /search is the path, and q=java is the query string. A port identifies a service endpoint on a computer. Localhost addresses refer to the same computer, not the public internet.

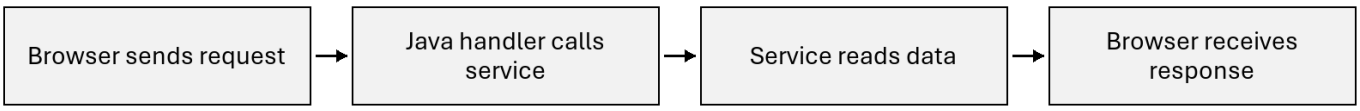


Figure 3 Web request and response path

A request has a method, path, headers, and sometimes a body. A response has a status code, headers, and usually a body. A header is a named piece of message information. Content-Type states the data type and may name a text encoding. For UTF-8 text, calculate response length in encoded bytes, not Java character count.

Method or code	Simple purpose	Library example
GET	Read a resource without requesting a state change	Search books
POST	Submit data for processing	Create a loan
PUT	Replace a resource at a chosen address	Replace a book record
DELETE	Remove a resource	Remove a withdrawn book
200	Successful response	Matching titles returned
201	Resource created	New loan created
400	Bad request data	Invalid book ID
404	Resource not found	Unknown path or book
405	Method not allowed	POST sent to a GET-only search
409	Conflict with current state	Last copy already borrowed
500	Unexpected server failure	Internal save failure

A small local Java web app

The JDK's jdk.httpserver module provides a small HTTP server API. It is enough to learn requests, routes, query parameters, output encoding, and responses without adding a framework. The example is a local teaching app with read-only sample titles. It is not a complete public deployment system. [JDK HTTP server module reference]

LibraryWeb.java

```
import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpServer;
import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.URLDecoder;
import java.nio.charset.StandardCharsets;
import java.util.List;
import java.util.Locale;
```

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class LibraryWeb {
    private static final List<String> TITLES = List.of(
        "Java Basics", "Web Guide", "Data Skills");

    private static String escape(String text) {
        return text.replace("&", "&amp;")
            .replace("<", "&lt;").replace(">", "&gt;")
            .replace("\"", "&quot;").replace("'", "&#39;");
    }

    private static String queryValue(String raw) {
        if (raw == null) return "";
        if (raw.length() > 1000) {
            throw new IllegalArgumentException("Query is too long");
        }
        String found = "";
        boolean seen = false;
        for (String part : raw.split("&")) {
            String[] pair = part.split("=", 2);
            String key = URLDecoder.decode(pair[0], StandardCharsets.UTF_8);
            if (key.equals("q")) {
                if (seen) {
                    throw new IllegalArgumentException("Repeated q");
                }
                seen = true;
                found = pair.length == 2 ? URLDecoder.decode(
                    pair[1], StandardCharsets.UTF_8) : "";
            }
        }
        if (found.length() > 100) {
            throw new IllegalArgumentException("Search is too long");
        }
        return found.trim();
    }

    private static void send(HttpExchange exchange, int status,
                             String body) throws IOException {
        byte[] bytes = body.getBytes(StandardCharsets.UTF_8);
        exchange.getResponseHeaders().set(
            "Content-Type", "text/html; charset=UTF-8");
        exchange.getResponseHeaders().set("X-Content-Type-Options", "nosniff");
        exchange.sendResponseHeaders(status, bytes.length);
        try (var out = exchange.getResponseBody()) {
            out.write(bytes);
        }
    }

    private static void handle(HttpExchange exchange) throws IOException {
        try {
            String path = exchange.getRequestURI().getPath();
            if (!path.equals("/") && !path.equals("/search")) {
                send(exchange, 404, "<p>Page not found.</p>");
                return;
            }
            if (!exchange.getRequestMethod().equals("GET")) {
                exchange.getResponseHeaders().set("Allow", "GET");
                send(exchange, 405, "<p>Use GET for this page.</p>");
                return;
            }
            String query;
            try {
                query = queryValue(exchange.getRequestURI().getRawQuery());
```

```
            } catch (IllegalArgumentException ex) {
                send(exchange, 400, "<p>Invalid search text.</p>");
                return;
            }
            StringBuilder html = new StringBuilder(
                "<!doctype html><html lang='en'><meta charset='UTF-8'>"
                + "<title>Library Search</title><h1>Library Search</h1>"
                + "<form action='/search' method='get'>"
                + "<label>Title word: <input name='q' maxlength='100'></label>"
                + "<button>Search</button></form>");
            html.append("<p>Search: ").append(escape(query)).append("</p><ul>");
            int count = 0;
            String normalized = query.toLowerCase(Locale.ROOT);
            for (String title : TITLES) {
                if (title.toLowerCase(Locale.ROOT).contains(normalized)) {
                    html.append("<li>").append(escape(title)).append("</li>");
                    count++;
                }
            }
            html.append("</ul><p>Matches: ").append(count)
                .append("</p></html>");
            send(exchange, 200, html.toString());
        } finally {
            exchange.close();
        }
    }

    public static void main(String[] args) throws IOException {
        HttpServer server = HttpServer.create(
            new InetSocketAddress("127.0.0.1", 8080), 0);
        ExecutorService pool = Executors.newFixedThreadPool(4);
        server.createContext("/", LibraryWeb::handle);
        server.setExecutor(pool);
        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
            server.stop(1);
            pool.shutdown();
        }));
        server.start();
        System.out.println("Open http://127.0.0.1:8080/");
    }
}
```

Compile and run with these commands. Keep the terminal open while browsing. Press Ctrl+C in the terminal to stop the server.

Terminal commands

```
javac --release 21 --add-modules jdk.httpserver LibraryWeb.java
java --add-modules jdk.httpserver LibraryWeb
```

The root path shows all three titles because every title contains the empty string. A search for java shows one title. This is a deliberate empty-search rule and differs from the GUI lesson's required-search rule. A real shared service should choose one common rule and use it across all interfaces.

URL decoding changes encoded query text into characters. HTML escaping changes special characters into safe displayed text. These are different steps. The server escapes both the query and the titles before placing them in HTML text. A search such as <script> must appear as text, never run as markup. Escaping rules depend on where data goes; this helper is for HTML text and quoted attribute contexts, not JavaScript or SQL.

Servlets and a web container

A servlet is a Java server component managed by a web container. A container starts it, maps requests to it, calls its methods, and later destroys it. Tomcat is one common container. Java SE alone does not include the Jakarta Servlet API. Tomcat 10.1 implements Servlet 6.0 and requires Java 11 or later, so the Java 21 baseline fits. Use jakarta.servlet imports for this pairing. Older javax.servlet applications need compatible older containers or migration. [Tomcat version guide]

The servlet life cycle normally calls init once for an instance, service for requests, and destroy when the instance is removed. HttpServlet routes service calls to methods such as doGet and doPost. Requests may run concurrently on the same servlet object. Never store one user's current request, member ID, or response in an instance field.

This complete optional servlet gives a fixed response. It needs Tomcat's servlet-api.jar at compile time and the Tomcat container at run time.

HelloServlet.java

```
import java.io.IOException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

@WebServlet("/hello")
public class HelloServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response) throws IOException {
        response.setContentType("text/plain;charset=UTF-8");
        response.getWriter().println("School Library Service");
    }
}
```

To try it, obtain a current supported Tomcat 10.1 release from the official site. In a practice folder, create library/WEB-INF/classes. Replace the example Tomcat path below with your actual installation path. These commands assume that library is inside your current terminal folder.

Terminal commands

```
javac --release 21 -cp "C:/tools/tomcat/lib/servlet-api.jar" -d library/WEB-INF/classes
HelloServlet.java
jar --create --file library.war -C library .
```

Copy library.war into Tomcat's webapps folder, start Tomcat using its documented startup command, and open http://localhost:8080/library/hello. Stop the earlier LibraryWeb first because both examples use port 8080. The servlet container supplies the API at run time, so do not add servlet-api.jar inside this WAR. WAR means Web Application Archive. The annotation supplies the route, so this small example needs no web.xml.

For form input, request.getParameter("q") reads a decoded parameter. Treat a missing result as null and validate it. doPost handles form submissions that request changes. With appropriate form handling, redirect after a successful POST so a browser refresh does not casually repeat the action. A redirect alone does not guarantee duplicate prevention; important actions also need server-side rules or request identifiers.

Layers sessions and data

Keep three basic layers. The web layer handles HTTP and output. The service layer applies business rules. The data layer stores and finds records. A shared LibraryService can serve a GUI, a console, and a servlet. A repository interface can allow file storage during class and database storage later.

HTTP requests do not automatically remember a user. A cookie is a small value the browser sends back under defined rules. A session is server-side state usually linked by a session cookie. Authentication checks who the user is. Authorization checks what that user may do. A session ID should not be placed in logs or shared URLs. Logging out should invalidate the relevant session.

JDBC is Java's standard database access API. A Connection represents a database connection. PreparedStatement sends SQL with separate parameter values. ResultSet reads returned rows. Use placeholders such as WHERE id = ? and bind the ID instead of joining user text into SQL. A transaction groups changes that must succeed or fail together, such as creating a loan and reducing available stock. The database driver and database are additional dependencies.

Frameworks such as Spring can supply routing, dependency injection, validation, and database support. They do not remove the need to understand HTTP, thread safety, or service rules. Learn the small request-response cycle before adding a large framework.

Common web failures

Validate all outside input on the server. Browser maxlength is a convenience, not a server rule. Use parameterized database statements to avoid SQL injection. Escape displayed data to reduce cross-site scripting, which is unwanted script execution in a user's page. State-changing authenticated requests need suitable protection against cross-site request forgery, where another site tries to trigger a user's action.

For deployment, use HTTPS, a suitable application server or reverse proxy, timeouts, request-size limits, access control, logs without secrets, backups, and controlled updates. A reverse proxy is a front server that passes approved requests to the app. The teaching server's fixed pool has a growing task queue, so it does not provide a full overload policy. Keep it local for this activity. Do not return stack traces or passwords to a browser.

Practical activity

Purpose: Follow a complete request-response cycle. Tools: The JDK, a browser, an editor, and optionally the browser's network inspection panel. Task: Test the local library web search.

1. Compile and start LibraryWeb. Open the printed local address.
2. Search for java, WEB, and a missing word. Record match counts.
3. Visit /missing and confirm a 404 response. A network panel can show the code.
4. Search for <b> and confirm that it appears as text rather than bold markup.
5. Try /search?q=java&q=web. Confirm rejection with 400.
6. Stop the server. Write a trace from browser input to query decoding, service work, escaping, and response bytes.

Expected result: Valid searches return HTML with correct counts. Unknown routes and invalid parameters get meaningful error codes. Special text stays text. The app stops without leaving its listener running.

Practice: Optionally deploy HelloServlet in Tomcat. Then sketch a POST /loans service that checks member access, validates IDs, and handles the last-copy conflict. State which operation needs one database transaction.

Chapter summary

A Java web app receives HTTP requests and returns responses. Routes, methods, status codes, encoding, and output escaping have different jobs. A small JDK server teaches the basics. Servlets run inside a compatible container. Separate HTTP handling, business rules, and storage, and protect shared data when requests run together.

Review and exam questions

1. Identify the parts of http://127.0.0.1:8080/search?q=java.
2. Why should GET not create a loan?
3. Compare URL decoding with HTML escaping.
4. Why are servlet instance fields unsafe for per-request user data?
5. Which code fits an unknown book, invalid ID text, and last-copy conflict?
6. Explain why a database transaction matters when two users borrow the last copy.

Chapter 12 - Exception Handling

Failure is part of normal work

A program can receive a blank name, read a damaged file, lose a connection, or try to lend a book with no copies left. Error handling means detecting such situations, protecting valid state, and reporting what happened clearly. It is not enough to surround the entire program with one catch block that ignores every problem.

Three broad problem groups help during testing. A compile-time problem stops source from compiling, such as a missing brace or wrong argument type. A run-time failure occurs while the program runs, such as a missing file. A logic error gives a wrong result even though execution continues, such as charging one extra late day. Exceptions help with run-time failures, but tests and reasoning are needed for logic errors too.

The exception family

An exception is an object that describes an unusual failure and changes normal control flow. Java's Throwable family includes Error and Exception. Most application recovery uses suitable Exception types. Error often describes serious JVM or environment problems that ordinary application code should not try to recover from broadly.

Group	Examples	Compiler rule and normal use
Checked exception	IOException, ClassNotFoundException	Handle with catch or declare with throws
Unchecked exception	IllegalArgumentException, NullPointerException, IllegalStateException	No compiler requirement to handle; often invalid input, state, or a bug
Error	OutOfMemoryError, LinkageError	Usually not a normal business recovery path

RuntimeException and its subclasses are unchecked. Error and its subclasses are also unchecked. The checked/unchecked distinction is about compiler rules, not whether a problem is serious. An unchecked failure can still require careful handling at a user boundary.

try catch finally throw and throws

Put an operation that may fail inside try. catch handles a matching exception. finally performs cleanup after the try/catch flow in normal Java control transfer. It is not guaranteed after forced process termination or a JVM crash. Prefer try-with-resources for closeable resources because it handles cleanup clearly.

throw actually raises one exception object. throws lists exception types that a method may pass to its caller. A declaration does not itself cause an exception. When an exception is thrown, Java searches for a matching handler up the call stack. The call stack is the chain of active method calls. Statements after the failing operation are skipped until handling takes control.

ExceptionDemo.java

```
public class ExceptionDemo {
    static int positiveId(String text) {
        int id;
        try {
            id = Integer.parseInt(text.trim());
        } catch (NumberFormatException ex) {
            throw new IllegalArgumentException("Enter a whole-number ID", ex);
        }
        if (id <= 0) {
            throw new IllegalArgumentException("ID must be positive");
        }
        return id;
    }

    public static void main(String[] args) {
        String input = args.length == 0 ? "101" : args[0];
```

```
        try {
            System.out.println("Accepted ID: " + positiveId(input));
        } catch (IllegalArgumentException ex) {
            System.out.println("Input problem: " + ex.getMessage());
        }
    }
}
```

This example accepts one command-line string. java ExceptionDemo abc prints a helpful number message instead of a raw stack trace. java ExceptionDemo -3 gives the positive-ID message. A caller that can pass null would need a documented null rule and check; this particular main always supplies a non-null string.

The cause passed to IllegalArgumentException keeps the original parsing failure. An outside user needs a short useful message. A developer may need the cause and stack trace in a log. Keep both needs without exposing sensitive data.

Catch order and recovery boundaries

Catch more specific exceptions before their parents. A catch for Exception before IOException would make the later IOException handler unreachable. A multi-catch such as catch (IOException | SecurityException ex) handles unrelated types in one block. It cannot list a type and its own subclass as separate alternatives.

Handle a failure where the code can make a useful decision. A file repository may add the filename context and pass an IOException upward. A GUI boundary may show Save failed. A web boundary may return a safe status message. Do not show a success message merely because the exception was caught.

Avoid catch (Exception ex) with an empty body. Avoid returning 0 or an empty list after an unexpected storage failure when that would falsely mean a real zero or no records. If an operation failed, communicate failure clearly. A recovery value is suitable only when it has an agreed meaning, such as a missing optional settings file using defaults.

Custom exceptions for business rules

A custom exception names a problem in the application's language. For example, LoanRejectedException can mean that the requested loan cannot be made. Keep useful details such as a safe reason code or message. Do not create a new type for every small formatting mistake unless callers need different handling.

LoanFailureDemo.java

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

class LoanStock {
    private int available = 1;

    synchronized void lend() throws LoanRejectedException {
        if (available == 0) {
            throw new LoanRejectedException("No copy is available");
        }
        available--;
    }

    synchronized int getAvailable() { return available; }
}

public class LoanFailureDemo {
    public static void main(String[] args) {
        LoanStock stock = new LoanStock();
        try {
            stock.lend();
            stock.lend();
        } catch (LoanRejectedException ex) {
            System.out.println("Loan rejected: " + ex.getMessage());
        }
        System.out.println("Copies left: " + stock.getAvailable());
```

```
    }
}
```

The second loan is rejected, and the stock stays at 0 rather than becoming negative. The synchronized method protects the check and change together inside this one JVM. It does not protect stock in another process or a shared database by itself.

Choosing checked or unchecked for a business failure is a design decision. Checked exceptions force callers to address a declared outcome. Unchecked domain exceptions can reduce declaration noise but still need clear boundary handling. Use a consistent policy rather than choosing only to silence the compiler.

Cleanup and suppressed exceptions

try-with-resources closes resources in reverse declaration order. If the main work throws and closing also throws, the main exception remains primary and the close failure is normally attached as a suppressed exception. This avoids losing the original problem. Use getSuppressed when investigating such failures. [Java exception tutorial]

Do not return from finally. A return there can hide a return value or exception from the try block. Do not throw a new unrelated exception during cleanup without preserving context. A log message should state the failed operation, relevant safe ID, and cause, but should omit passwords, session IDs, and private record contents unless there is a justified protected logging rule.

State safety and transactions

Validate before changing state where possible. If loading many records, build a new collection and replace the active collection only after every required check succeeds. If saving a file, use a temporary file and deliberate replacement. If updating related database rows, use a transaction with commit on success and rollback on failure.

Some operations cannot simply be undone. An email may already be sent before a later file save fails. Plan the order and track what happened. A retry should not create duplicate loans or duplicate charges. An idempotent operation has the same intended effect when the same request is repeated. A request ID can help the service recognize a repeated action.

Retry only a failure that may be temporary, use a small limit and delay, and keep a clear final failure path. Invalid IDs do not become valid by retrying. Repeatedly catching every failure and immediately retrying can overload a service or hide a real bug.

Debugging and meaningful tests

Read a stack trace from the exception type and message to the relevant application line. Follow any Caused by section. Reproduce the failure with a small input. A debugger can pause at a breakpoint, step through calls, and inspect variables. Do not change several unrelated parts at once while guessing.

Test normal values, boundary values, and invalid values. For a positive ID, include 1, 0, -1, blank text, letters, and a number larger than int can hold. For a loan, test the first copy, last copy, no copies, duplicate request, and storage failure. A test should check the result and the state after failure.

Assertions with assert can check internal assumptions during development, but they may be disabled. Do not use assert as the only check for user input or a required business rule. Throw an appropriate exception or return an explicit failure result for those rules.

Practical activity

Purpose: Handle failures without damaging state. Tools: The JDK and an editor. Task: Test input conversion and a last-copy rule.

1. Run ExceptionDemo with 101, 1, 0, -3, abc, and 999999999999.
2. Record the accepted or rejected result for each input.
3. Run LoanFailureDemo. Confirm that copies never become negative.
4. Add a third lend attempt in a separate try/catch and confirm another clear rejection.
5. In a file-loading copy from Chapter 7, add a malformed line and check that the error keeps its original cause.
6. Write a safe user message and a separate technical log message for a permission-denied save.

Expected result: Valid values work, invalid values are explained, and failed actions preserve the important state rules.

Practice: Explain which failures can be retried: invalid title, brief connection timeout, unsupported file version, and full disk. State what must change before retrying each one.

Chapter summary

Good error handling detects a problem, protects state, and explains the outcome. Checked exceptions must be handled or declared. throw raises a failure; throws declares possible failures. Preserve causes, close resources, catch at useful boundaries, and test what remains after an operation fails.

Review and exam questions

1. Compare compile-time, run-time, and logic problems.
2. Explain throw and throws with one example each.
3. Why must a specific catch appear before a general matching catch?
4. Why is an empty catch block risky in a save method?
5. Explain primary and suppressed exceptions in try-with-resources.
6. A save fails after memory changed. Describe two design choices that can keep the app honest about its state.

Chapter 13 - Designing Efficient Algorithms

From a task to an algorithm

An algorithm is a clear set of steps for solving a problem. A complex program often combines several simple algorithms. The library may search for an ID, sort titles, find overdue loans, choose jobs, and walk a category tree. Choose a data structure and algorithm based on the operations needed, the amount of data, and the memory limit.

Begin by stating inputs, outputs, rules, and failure cases. For searching, the input is a list and a target. The result may be an index or -1 for no match. State whether IDs are unique and whether the input is sorted. These conditions are part of correctness, not optional details.

Time and space cost

Time complexity describes how the amount of work grows as input grows. Space complexity describes how much extra memory an algorithm needs. Big O gives a growth upper bound, usually used here for the main worst-case trend. It does not give exact seconds. A small input, a slow disk, or expensive comparisons can change which version is fastest in practice.

Growth	Common example	Work when n becomes twice as large
O(1)	Array access by index	About the same number of basic operations
O(log n)	Binary search on a sorted array	A small number of extra steps
O(n)	One full scan	About twice the work
O(n log n)	Merge sort	A little more than twice the work
O(n squared)	Two full nested scans	About four times the work
O(2 to the n)	Trying every subset	Can grow very quickly

n means the number of input items. For a single loop over n items, the scan grows with n. Two nested loops each covering n items can perform n x n checks. If a loop halves the remaining range each time, the number of steps grows with log base 2 of n. Ignore fixed factors only when describing growth; they can still matter when measuring a real program.

For 1,024 sorted IDs, binary search halves the range roughly ten times. A standard equality-checking binary search can need up to floor(log2(n)) + 1 comparisons, which is 11 for n = 1,024. Linear search may inspect all 1,024 IDs. Sorting first also costs time, so binary search is most useful when sorted data is already available or many searches reuse it.

Picking a collection

Structure	Useful operations	Important cost or limit
Array	Fixed-size indexed data	O(1) indexing; size cannot grow
ArrayList	Growing sequence, frequent indexed reads	O(1) indexing; middle insert/remove shifts items
LinkedList	Deque operations at ends	O(1) end operations; indexed access requires traversal
HashMap	Find a value by key	Average O(1) basic lookup with good hashing; no key-order promise
TreeMap	Sorted keys and range queries	O(log n) basic lookup and update
HashSet	Unique membership checks	Depends on consistent equals and hashCode

Structure	Useful operations	Important cost or limit
ArrayDeque	Stack or queue	Efficient operations at both ends; no null elements
PriorityQueue	Repeatedly take the smallest-priority item	O(log n) insert/remove; iteration is not sorted order

Do not choose LinkedList just because you expect many insertions. Finding the insertion point may require a long traversal, and linked nodes use extra memory. Consider the complete operation. Use Map<Integer, Book> when exact book-ID lookup is common, and keep a separate title index if the project needs it.

Linear and binary search

Linear search checks each value until it finds the target. It works without sorted input. Binary search compares the middle value of a sorted range. It discards the half that cannot contain the target. Its comparison order must match the order used to sort the input.

SearchDemo.java

```
public class SearchDemo {
    static int linearSearch(int[] values, int target) {
        for (int i = 0; i < values.length; i++) {
            if (values[i] == target) return i;
        }
        return -1;
    }

    static int binarySearch(int[] values, int target) {
        int low = 0;
        int high = values.length - 1;
        while (low <= high) {
            int mid = low + (high - low) / 2;
            if (values[mid] == target) return mid;
            if (values[mid] < target) {
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }
        return -1;
    }

    public static void main(String[] args) {
        int[] ids = {101, 104, 108, 112, 120, 125, 130};
        System.out.println(linearSearch(ids, 120));
        System.out.println(binarySearch(ids, 120));
        System.out.println(binarySearch(ids, 999));
    }
}
```

The output is 4, 4, and -1 on separate lines. For target 120, low = 0 and high = 6, so mid = 3 and the value is 112. Since 120 is larger, low becomes 4. The next middle is 5 with value 125, so high becomes 4. The next middle is 4 with value 120, so the result is 4. The mid formula avoids adding two potentially large indexes directly.

With duplicates, this version returns some matching index, not necessarily the first. To find the first, save a matching index and continue to the left. Java's Arrays.binarySearch is available for standard use. Its negative not-found result encodes an insertion position; it is not simply always -1. Read that contract before using the result as an index.

Sorting choices and stable order

Sorting arranges values in a chosen order. Insertion sort grows a sorted left part by moving larger values right. It is simple and can work well for small or nearly sorted input. Merge sort divides the data, sorts the halves, and merges them. Quicksort partitions around a pivot and sorts the partitions. Heapsort uses a heap, a structure that keeps a largest or smallest item easy to reach.

Algorithm	Typical or average time	Worst time	Main extra-space idea
Insertion sort	O(n squared); O(n) on already sorted input with the normal early stop	O(n squared)	O(1)
Merge sort on arrays	O(n log n)	O(n log n)	Usually O(n) helper array
Quicksort	O(n log n) with good partitions	O(n squared)	Recursion stack depends on partition balance
Heapsort	O(n log n)	O(n log n)	O(1) array work space in a standard iterative version

A stable sort keeps the input order of elements that compare equal. Suppose two loans have the same due date. A stable due-date sort keeps their earlier order, which may already be by member name. Stability depends on the specific algorithm and implementation. A usual insertion sort is stable when it shifts only strictly larger values.

SortDemo.java

```
import java.util.Arrays;

public class SortDemo {
    static void insertionSort(int[] values) {
        for (int i = 1; i < values.length; i++) {
            int key = values[i];
            int j = i - 1;
            while (j >= 0 && values[j] > key) {
                values[j + 1] = values[j];
                j--;
            }
            values[j + 1] = key;
        }
    }

    public static void main(String[] args) {
        int[] ids = {108, 101, 112, 104};
        insertionSort(ids);
        System.out.println(Arrays.toString(ids));
    }
}
```

Start with [108, 101, 112, 104]. Insert 101 into the left part to get [101, 108, 112, 104]. The 112 stays where it is. For 104, move 112 and 108 right, then insert 104: [101, 104, 108, 112]. The outer loop's rule is that the part before i is sorted before the next insertion. Such a rule is called a loop invariant. It helps explain why the algorithm is correct.

For ordinary project code, prefer tested library sorts. For books, books.sort(Comparator.comparing(Book::getTitle).thenComparingInt(Book::getId)) sorts by title and then ID when titles match. This is a fragment that assumes a Book class with those getters and a changeable List<Book>. Compare strings with a stated case and language policy. A default string comparison is not automatically a human dictionary order for every language.

Recursion and divide and conquer

Recursion means that a method calls itself on a smaller problem. It needs a base case that stops and progress toward that case. Each active call uses stack space. Very deep recursion can cause StackOverflowError. A loop is often safer for a long simple count.

Divide and conquer splits a problem, solves smaller pieces, and combines results. Merge sort follows this pattern. The next complete example uses a shared temporary array and half-open ranges. A half-open range includes its start and excludes its end.

MergeSortDemo.java

```
import java.util.Arrays;

public class MergeSortDemo {
    static void sort(int[] values) {
        split(values, new int[values.length], 0, values.length);
    }

    private static void split(int[] a, int[] temp, int from, int to) {
        if (to - from <= 1) return;
        int mid = from + (to - from) / 2;
        split(a, temp, from, mid);
        split(a, temp, mid, to);
        int left = from;
        int right = mid;
        int out = from;
        while (left < mid && right < to) {
            if (a[left] <= a[right]) {
                temp[out++] = a[left++];
            } else {
                temp[out++] = a[right++];
            }
        }
        while (left < mid) temp[out++] = a[left++];
        while (right < to) temp[out++] = a[right++];
        for (int i = from; i < to; i++) a[i] = temp[i];
    }

    public static void main(String[] args) {
        int[] values = {8, 3, 7, 1, 3};
        sort(values);
        System.out.println(Arrays.toString(values));
    }
}
```

The output is [1, 3, 3, 7, 8]. At each level, merging covers about n values. Balanced splitting creates about log2(n) levels. Therefore the total work grows as O(n log n). The helper array uses O(n) space, and the call stack adds O(log n). Taking the left value on equality preserves stable order when the algorithm is adapted to records.

Trees graphs and traversal

A tree is a hierarchy with a root and child branches. A folder system is a familiar example. A graph is a set of vertices, also called nodes, joined by edges. Edges may have directions or weights. A weight is a cost such as travel time. A graph can describe course prerequisites or links between library rooms.

Breadth-first search, or BFS, visits nearby nodes before more distant nodes. It uses a queue and a visited set. Mark a node visited when adding it to the queue, so cycles do not add it repeatedly. With an adjacency-list graph, BFS takes O(V + E) time, where V is the number of vertices and E is the number of edges represented. It finds a shortest path by edge count in an unweighted graph, not the cheapest route for arbitrary weights.

GraphDemo.java

```
import java.util.*;

public class GraphDemo {
    static List<String> breadthFirst(Map<String, List<String>> graph,
                                     String start) {
        Set<String> seen = new HashSet<>();
        Queue<String> queue = new ArrayDeque<>();
        List<String> order = new ArrayList<>();
        seen.add(start);
```

```
        queue.add(start);
        while (!queue.isEmpty()) {
            String node = queue.remove();
            order.add(node);
            for (String next : graph.getOrDefault(node, List.of())) {
                if (seen.add(next)) queue.add(next);
            }
        }
        return order;
    }

    public static void main(String[] args) {
        Map<String, List<String>> rooms = Map.of(
            "Desk", List.of("A", "B"),
            "A", List.of("Desk", "C"),
            "B", List.of("Desk"),
            "C", List.of("A"));
        System.out.println(breadthFirst(rooms, "Desk"));
    }
}
```

The result is [Desk, A, B, C] because each neighbor list has that stated order. BFS only visits the component reachable from Desk. Depth-first search, or DFS, follows a branch deeply before going back. It can use recursion or an explicit stack. Both need visited tracking for general graphs. Dijkstra's algorithm finds shortest weighted paths when edge weights are nonnegative. Do not use plain BFS for unequal travel costs.

Greedy choices and saved subproblems

A greedy algorithm chooses what looks best at the current step. It is correct only when the problem supports that choice. For selecting the maximum number of non-overlapping room bookings, sort by end time and repeatedly choose the next booking that starts after the previous chosen one ends. For a different goal, such as maximum income, the same rule may fail.

Dynamic programming saves answers to repeated smaller problems. Consider the fewest tokens needed to pay a total using token values 1, 3, and 4. This is a teaching problem, not a claim about real currency. For total 6, a greedy largest-first choice gives 4 + 1 + 1, or three tokens. The best result is 3 + 3, or two tokens.

Let dp[a] mean the fewest tokens for amount a. Set dp[0] = 0. For each positive amount, try every token value c that is not larger than a. Use the smallest value of dp[a - c] + 1. The +1 counts the current token. The earlier dp value counts the remaining amount.

Amount a	0	1	2	3	4	5	6
Fewest tokens	0	1	2	1	1	2	2

DynamicDemo.java

```
public class DynamicDemo {
    static int fewestTokens(int amount, int[] tokens) {
        if (amount < 0 || amount > 100000) {
            throw new IllegalArgumentException("Amount out of lesson range");
        }
        for (int token : tokens) {
            if (token <= 0) {
                throw new IllegalArgumentException("Tokens must be positive");
            }
        }
        int unreachable = amount + 1;
        int[] dp = new int[amount + 1];
        java.util.Arrays.fill(dp, unreachable);
        dp[0] = 0;
        for (int a = 1; a <= amount; a++) {
            for (int token : tokens) {
                if (token <= a) {
                    dp[a] = Math.min(dp[a], dp[a - token] + 1);
                }
            }
```

```
        }
        return dp[amount] == unreachable ? -1 : dp[amount];
    }

    public static void main(String[] args) {
        System.out.println(fewestTokens(6, new int[] {1, 3, 4}));
    }
}
```

The result is 2. If A is the amount and C is the number of token values, time is O(A x C) and extra space is O(A). The lesson's range check keeps the array size controlled. Backtracking is another approach that tries a choice, explores it, and undoes it before trying another. It is useful for constraint puzzles and small combinations, but may explore exponentially many possibilities. Limit input and prune choices that already break a rule.

Testing correctness and measuring work

Test empty input, one item, duplicates, already sorted values, reverse order, absent targets, and extreme integer values. A sorting result must be ordered and contain exactly the original values with their counts. A search result must point to the target when found and obey its not-found contract otherwise.

For small random arrays, compare your sort with Arrays.sort on a copy. Compare binary-search results by validity rather than requiring the same duplicate index as another correct implementation. For timing, keep input data and task the same, warm up the JVM, run multiple trials, and avoid printing inside the timed loop. System.nanoTime measures elapsed intervals; it is not a wall clock date. For careful benchmarks, a dedicated tool such as JMH handles many JVM measurement traps.

Practical activity

Purpose: Choose and verify algorithms rather than only memorize their names. Tools: The JDK, an editor, paper, and optionally a spreadsheet. Task: Search and sort library IDs, then solve a graph and token problem.

1. Run SearchDemo and trace each binary-search range for target 120.
2. Run SortDemo. Show the sorted left part after each outer loop step.
3. Run MergeSortDemo. Draw its first two splits and the final merge.
4. Test both sorts with empty, single, duplicate, sorted, and reverse-sorted arrays.
5. Run GraphDemo. Add an isolated node D and explain why a search from Desk does not visit it.
6. Run DynamicDemo with total 6. Calculate the table by hand. Then test amount 3 with only token value 2 and confirm -1.

Expected result: Correct searches and sorts match their stated contracts. BFS handles cycles without repeated visits. Dynamic programming beats the shown greedy choice for total 6.

Practice: Add comparison counters to linear and binary search. Compare 1,000 IDs for first, middle, last, and absent targets. Explain why sorting cost must be included when the data starts unsorted.

Chapter summary

An algorithm must be correct for its stated input conditions. Complexity describes growth, not exact seconds. Data structures affect total cost. Linear search, binary search, sorting, recursion, graph traversal, greedy choices, and dynamic programming solve different kinds of problems. Test edge cases and use trusted library algorithms in normal application code when they meet the need.

Review and exam questions

1. Explain O(n) and O(n squared) with simple loops.
2. Trace binary search for 104 in [101, 104, 108, 112, 120, 125, 130].
3. Why does binary search need a matching sorted order?
4. Show every insertion-sort step for [5, 2, 4, 1].
5. Why does graph traversal need a visited set? When does BFS give a shortest path?
6. Explain why greedy largest-first fails for total 6 with token values 1, 3, and 4.

Chapter 14 - Applications of Java

Where Java ideas work together

Real applications combine data models, algorithms, input/output, interfaces, failure handling, and tests. Java can be used for business services, internal desktop tools, education systems, data-processing jobs, and many other JVM applications. Android development uses many Java language and library ideas, but Android has its own runtime and APIs. A Swing desktop window is not an Android screen.

Do not choose a technology from its name alone. A small offline office tool may suit Swing and a local database. A multiuser portal may need a web service, an application server, and a shared database. A large text import may be a command-line batch job. A batch job processes a group of inputs without interactive steps for each record.

Situation	Useful course ideas	Additional real-project needs
School library	OOP model, search, loans, GUI or web	Access control, backups, reliable shared storage
Store inventory	Collections, validation, transactions	Stock audit, concurrent sales, reporting
Office file importer	File parsing, algorithms, error reports	Format versions, large-file limits, restart rules
Booking service	Interfaces, web requests, thread safety	Conflict prevention, identity checks, durable storage
Report tool	Sorting, background tasks, templates	Clear data rules, export checks, cancellation

The library project requirements

Build a small Library System in stages. A requirement states what the system must do. Keep each requirement observable so it can be tested. Start with sample data rather than private student records.

1. Store books with a unique positive ID, a nonblank title, and a nonnegative available-copy count.
2. Find a book by exact ID and search titles without letter-case differences.
3. Lend one copy only when the member is allowed and a copy is available.
4. Record and close loans so one loan cannot be returned twice.
5. Save and load validated records in a documented format.
6. Offer a clear console interface first, then a GUI or local web view.
7. Report rejected actions clearly and preserve valid state after failure.
8. Run a report in the background without breaking shared rules or freezing the GUI.

The first build does not need real email, payments, public hosting, or many servers. Add one feature at a time and keep a working version. Once features work separately, test how they interact. A correct search does not prove that save-and-reload preserves the searched data.

A simple project structure

Use packages such as school.library.model, school.library.service, school.library.storage, and school.library.ui. Model classes represent Book, Member, and Loan. LibraryService applies the rules. A repository handles storage. A view collects input and shows results. A test folder holds checks. Avoid one class with every field and every action.

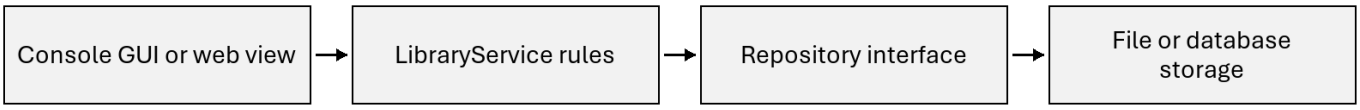


Figure 4 Application layers

The view calls the service rather than editing a Book field directly. The service uses a repository interface rather than hard-coding every file detail. A file repository and a test repository can satisfy the same contract. This connects encapsulation, interfaces, composition, and polymorphism.

Store important dates with LocalDate when a date without a time zone is enough. Define the school's due-date rule. If due on October 10 and returned on October 12 under a whole-day rule, there are 2 late days. A simple formula is lateDays = max(0, daysBetween(dueDate, returnDate)). If the teaching fine is 25 cents a day, fineCents = lateDays x 25 = 2 x 25 = 50 cents. State holiday, grace-period, and maximum-fine rules before adding them. These amounts are classroom examples.

A runnable service core

The next complete example keeps stock and loan identity in memory. It uses a synchronized service boundary so two threads in this JVM cannot both take the last available copy. It does not claim to be a durable database. Its repository is represented by a Map to keep the core rule visible.

LibraryCoreDemo.java

```
import java.util.*;

public class LibraryCoreDemo {
    record BookView(int id, String title, int available) {}
    record Loan(int loanId, int bookId, String member) {}

    static class BookStock {
        final int id;
        final String title;
        int available;

        BookStock(int id, String title, int available) {
            if (id <= 0 || title == null || title.isBlank() || available < 0) {
                throw new IllegalArgumentException("Invalid book data");
            }
            this.id = id;
            this.title = title.trim();
            this.available = available;
        }

        BookView view() { return new BookView(id, title, available); }
    }

    static class LibraryService {
        private final Map<Integer, BookStock> books = new HashMap<>();
        private final Map<Integer, Loan> activeLoans = new HashMap<>();
        private int nextLoanId = 1;

        synchronized void addBook(int id, String title, int copies) {
            BookStock book = new BookStock(id, title, copies);
            if (books.containsKey(id)) {
                throw new IllegalArgumentException("Duplicate book ID");
            }
            books.put(id, book);
        }

        synchronized BookView find(int id) {
            BookStock book = books.get(id);
            if (book == null) throw new IllegalArgumentException("Unknown book");
            return book.view();
        }
```

```
        synchronized Loan lend(int bookId, String member) {
            if (member == null || member.isBlank()) {
                throw new IllegalArgumentException("Member is required");
            }
            BookStock book = books.get(bookId);
            if (book == null) throw new IllegalArgumentException("Unknown book");
            if (book.available == 0) {
                throw new IllegalStateException("No copy is available");
            }
            int followingId = Math.incrementExact(nextLoanId);
            Loan loan = new Loan(nextLoanId, bookId, member.trim());
            activeLoans.put(loan.loanId(), loan);
            book.available--;
            nextLoanId = followingId;
            return loan;
        }

        synchronized void returnLoan(int loanId) {
            Loan loan = activeLoans.get(loanId);
            if (loan == null) {
                throw new IllegalArgumentException("No active loan with that ID");
            }
            BookStock book = books.get(loan.bookId());
            int restored = Math.incrementExact(book.available);
            activeLoans.remove(loanId);
            book.available = restored;
        }

        synchronized List<BookView> search(String text) {
            if (text == null) throw new IllegalArgumentException("Search required");
            String query = text.trim().toLowerCase(Locale.ROOT);
            List<BookView> found = new ArrayList<>();
            for (BookStock book : books.values()) {
                if (book.title.toLowerCase(Locale.ROOT).contains(query)) {
                    found.add(book.view());
                }
            }
            found.sort(Comparator.comparing(BookView::title)
                .thenComparingInt(BookView::id));
            return List.copyOf(found);
        }
    }

    public static void main(String[] args) {
        LibraryService service = new LibraryService();
        service.addBook(101, "Java Basics", 1);
        service.addBook(102, "Web Guide", 2);
        Loan loan = service.lend(101, "Maya");
        try {
            service.lend(101, "Luis");
        } catch (IllegalStateException ex) {
            System.out.println("Rejected: " + ex.getMessage());
        }
        service.returnLoan(loan.loanId());
        System.out.println(service.find(101));
        System.out.println(service.search("guide"));
    }
}
```

The first message rejects a second loan of book 101. After the valid return, the BookView for 101 has available=1. The final search contains Web Guide with available=2. A second return of the same loan is rejected because it is no longer active.

BookView is an immutable snapshot with only primitive and String components. It does not expose the changeable BookStock. List.copyOf makes the returned list unmodifiable. This keeps callers from changing the

service's internal stock behind its checks. The synchronized methods protect all shared maps and counts through the same service object.

The core identifies a member by a nonblank name only. That is enough for this small runnable rule example, but it does not enforce real membership, role limits, or due dates. Those are explicit next-stage tasks below. Treating a name as proof of authorization would be wrong in a real system.

Building the full project in stages

Stage 1 is the service core. Check IDs, title rules, last-copy behavior, and duplicate returns. Stage 2 adds Member objects and role-based loan limits using the earlier parent/interface ideas. Count a member's active loans before accepting a new one. Give members stable IDs; names may repeat or change.

Stage 3 adds due dates and return dates. Store a Loan with an ID, book ID, member ID, issue date, due date, and status. A loan is not just a decrease in stock. It needs identity so the same loan cannot be returned twice. Do not erase history if the project requires an audit trail. An audit trail records who did what and when.

Stage 4 adds storage through a repository interface. Save books and loans together as one consistent snapshot, including enough information to choose the next unused loan ID after loading. Validate every ID reference and count. Reject a loan pointing to a missing book or member. Do not independently replace book and loan files without planning what happens if only one replacement succeeds.

Stage 5 adds a GUI or web interface. The interface calls the same service methods. Keep services free of JOptionPane and HttpExchange so they can be tested without a screen or server. Stage 6 adds a background overdue report and checks a consistent snapshot, rather than iterating over changing maps without a rule.

For several processes or servers, replace the single-JVM lock assumption with database transactions and suitable constraints. A unique key protects IDs. A conditional stock update or locked row can prevent two successful loans of the last copy. Recheck service behavior under real concurrent requests.

Project testing and acceptance

Test	Starting condition and action	Required result
Valid loan	One copy; authorized member borrows	One active loan; zero copies
No stock	Zero copies; member borrows	Rejected; no new loan
Duplicate return	Return the same loan twice	First succeeds; second rejected; no extra copy
Duplicate ID	Add two books with ID 101	Second rejected; original stays
Missing reference	Load a loan for absent book	Load rejected or reported under a clear import policy
Save and reload	Save a valid snapshot and reload	Same logical books, loans, and next-ID safety
Simultaneous last copy	Two valid requests for one copy	Exactly one succeeds
Interface failure	Bad numeric text in GUI or web	Helpful message; no state change

Use unit tests for one method or small class, integration tests for components working together, and acceptance tests for complete user goals. A test double is a simple replacement for a real helper, such as a repository that deliberately fails a save. It allows a repeatable failure test without damaging real files.

For the last-copy test, use two tasks released from a shared latch, collect both outcomes, and check that exactly one succeeds. A latch is a synchronization gate that opens after a count reaches zero. Do not use sleep alone to prove that requests overlapped. Repeat tests, but also inspect the locking or transaction rule: stress tests cannot prove every possible schedule is safe.

Maintainable code and delivery

Give names a clear job, keep methods small, validate at boundaries, and document surprising decisions. Favor high cohesion, meaning each class has one closely related purpose. Reduce coupling, meaning fewer hidden dependencies between unrelated parts. Use composition and interfaces where they make changes simpler. Do not add many layers to a tiny program just to display design terms.

Build with the same Java baseline used in class. A JAR, or Java Archive, packages compiled classes and resources. For a simple application, compile to a classes folder and use jar --create --file library.jar --main-class LibraryCoreDemo -C classes . after compiling the core there. Run it with java -jar library.jar. A plain JAR does not include every outside dependency automatically.

Keep source in version control so changes can be compared and restored. A README should state the required JDK, build command, run command, sample data location, storage format, known limits, and test steps. A deployment plan should include backup and restore, configuration outside source, suitable access controls, and a way to detect failures. Test restoring a backup instead of only checking that a backup file exists.

Practical project assignment

Purpose: Apply the course as one working system. Tools: A Java 21-compatible JDK, an editor or IDE, a terminal, sample data, and optionally a browser. Task: Build the staged Library System above.

1. Run LibraryCoreDemo and add a deliberate duplicate-return test.
2. Add member IDs, student/teacher roles, and checked loan limits.
3. Add LocalDate issue and due dates and an overdue calculation with a stated daily fine in whole cents.
4. Define a repository interface. Implement one validated text snapshot format and a failing test repository.
5. Connect either a console menu plus the GUI, or a console menu plus the local web interface.
6. Run a background report over a safe snapshot and support cancellation if it takes long.
7. Complete the acceptance table and record actual results. Fix failed rules before adding extra features.
8. Package the program with a README, sample input, sample output, and a short explanation of class relationships.

Expected result: A small working application that preserves book and loan rules across input, output, failures, and repeated use. The same service behavior works through more than one interface.

Practice: Prepare a five-minute explanation showing one valid loan, one rejected loan, one save/reload, and one algorithm choice. Explain one current limit honestly and how a larger project would address it.

Chapter summary

Real Java work connects models, services, storage, interfaces, algorithms, and tests. Begin with explicit requirements and a small working service. Add member rules, dates, persistence, interfaces, and concurrency in controlled stages. Use testable boundaries and acceptance checks. Package the program so another student can build, run, and understand it.

Review and exam questions

1. Why should a GUI call LibraryService instead of changing stock fields directly?
2. Why does a loan need its own ID?
3. Explain how the core prevents a duplicate return from increasing stock twice.
4. What extra data must be saved to restart the service without reusing an active loan ID?
5. Compare a synchronized single-JVM service with a multi-server database transaction design.
6. A backup file exists but has never been restored. What remains untested?

Java Practice Setup

Set up a practice folder

Install a JDK that supports Java 21. A runtime-only installation may not include javac. Open a new terminal. Run java -version and javac -version. Both should work. The compiler must support --release 21. If the commands use different installations, correct PATH or your IDE's JDK setting.

Create JavaPractice with one folder per example, such as BookDemo. Copy each complete code block into its matching .java file. Leave out block labels and terminal commands. Fragments need the surrounding code named in their explanations.

Chapters reuse names such as Member and Book. Separate folders prevent duplicate classes and accidental links to another lesson's class. Do not compile all examples together in one default package.

Compile run and inspect

For most examples, open their folder. Run javac --release 21 FileName.java, then java FileName. Replace FileName with the class name. Compilation produces .class files, including files for helper and nested classes. Keep them together while running.

Run java ExceptionDemo abc to pass abc as an argument. Quote an argument containing spaces. Arguments are separate from keyboard input during execution. Programs with fixed sample values do not wait for keyboard input.

LibraryWeb needs the additional jdk.httpserver module option shown in Chapter 11. HelloServlet needs the external Jakarta Servlet API and a compatible container. LibraryGui needs a desktop display. The remaining complete examples use the standard JDK directly. The SwingWorker block is a fragment and is labeled that way.

Common setup problems

Symptom	Likely cause	Useful action
javac is not recognized	JDK tool folder not on PATH	Select or install a JDK and reopen the terminal
Public class filename error	Source filename differs from public class	Match spelling and uppercase letters
Could not find main class	Wrong folder, package name, or classpath	Run from the compiled output root and use the right full name
UnsupportedClassVersionError	Runtime is older than the compiled bytecode	Use a compatible runtime or rebuild for the baseline
Duplicate class	Examples with repeated helper names were mixed	Put each example in its own folder
Address already in use	Another app uses port 8080	Stop the earlier server or choose a free practice port
HeadlessException	No desktop display is available	Run the GUI on a desktop system
File appears in unexpected folder	Working folder differs from source folder	Inspect Path.of(".").toAbsolutePath()

A useful lab record

For each activity, record the goal, source filename, command, test input, expected result, actual result, and explanation. Keep one normal case and at least one failure case. A screenshot alone does not prove the underlying rule is correct. Include the result and the state, such as the copy count after a rejected loan.

When code fails, read the first useful compiler message or the main exception and cause. Fix one issue at a time. Run again after a small change. Keep a working copy so an experiment does not erase your last correct version.

Chapter Answer Guide

Chapter 1

1. javac checks source and creates bytecode. The JVM runs that bytecode and manages execution. A suitable JVM can run compatible bytecode on different systems.
2. int is a primitive whole-number type. String is an object type for text. Use a == b for two int values. Use first.equals(second) for non-null strings, or Objects.equals(first, second) when either may be null.
3. The other reference sees the book as unavailable. Both references point to the same object, so the state change is shared.
4. A private field prevents outside assignments that skip loan rules. A method can check the rule and then update the field.
5. 9 / 2 is 4 because both operands are integers. 9 / 2.0 is 4.5 because the operation uses floating-point division.
6. Start count at 0. Visit each array value. Increase count when value > 0. Return count. The empty array returns 0.

Chapter 2

1. Member is the superclass with shared name behavior. StudentMember is the subclass that adds grade and its own loan limit.
2. Member requires a name for setup and has no accessible no-argument constructor in the example. super(name) supplies that required value.
3. An override replaces inherited instance behavior for a child object. Static hiding selects a class method by compile-time rules. Field hiding gives separate fields with the same name and can confuse readers.
4. A final class cannot have subclasses. A final method cannot be overridden by a subclass.
5. Use composition: a Printer has or uses an InkCartridge. A Printer is not a kind of InkCartridge.
6. TeacherMember extends Member, calls super(name), and overrides public int loanLimit() to return 5. For Luis, the inherited description says Luis can borrow 5 books.

Chapter 3

1. An interface tells callers which actions an object supports and what those actions promise. Several classes can provide those actions differently.
2. The interface's send declaration is public. Its implementation cannot reduce that access level.
3. It has one abstract method after Java's inheritance and Object-method rules are considered. It can still contain default and static methods.
4. Write an override that chooses clear behavior. Where allowed, call a chosen direct interface default with InterfaceName.super.methodName().
5. A test sender records the intended message without a real network service. A test can check exactly what the service asked it to send.
6. If a List<Book> were accepted as List<Object>, a caller could put a non-Book into it. Generic invariance prevents that unsafe change.

Chapter 4

1. A concrete child's construction calls the abstract parent's constructor to set up the parent part of the child object.
2. No. A class can be abstract simply to prevent direct construction while sharing completed behavior.
3. abstract requires the possibility of children completing the design, while final forbids subclasses. The two class rules conflict.
4. A reminder role shared by unrelated implementations usually fits an interface such as MessageSender. Shared state and setup for one close family may justify an abstract base too.
5. DigitalBook calls super(title), stores a format, and returns a format-specific access instruction. label combines the common title with that instruction.

6. Warning signs include children ignoring many fields, overriding many methods only to disable them, or depending on a large amount of protected internal state.

Chapter 5

1. An abstract method has no implementation and must be completed by a concrete child. An empty concrete method already defines a do-nothing implementation.
2. A private method cannot be overridden by a child, while an abstract method requires such completion.
3. No. Adding an int parameter creates another signature. The no-argument method remains unfinished.
4. The shared process relies on positive days. Validating the child result preserves that rule and gives a clear failure for a broken implementation.
5. A ReferenceLoanRule child overrides protected int loanDays() to return 2. The shared label then states a two-day period.
6. A strategy object usually fits a policy that changes independently while the program runs. A stable family with a fixed process may suit a template method.

Chapter 6

1. The student call runs StudentRole.loanLimit and returns 3. The teacher call runs TeacherRole.loanLimit and returns 5. Each selection uses the actual object.
2. The declared type controls available members at compile time. The actual type identifies the object and selects overridden instance behavior at run time.
3. A reference assignment changes a view or alias. It does not rebuild or alter the object that already exists.
4. print(int) and print(String) are overloads chosen by parameter types. A child loanLimit() replacing a parent declaration is an override selected for the current object.
5. A cast to StudentRole fails if the object is a TeacherRole. Use if (member instanceof StudentRole student) before using student-specific behavior.
6. Hash-based collections use the hash code to help locate equal keys. Equal objects with unequal hashes can break expected lookup and uniqueness behavior.

Chapter 7

1. A byte stream reads or writes raw bytes. A reader interprets bytes as characters using an encoding and is suited to text.
2. Binary bytes do not carry a simple field label for every read call. The chosen types and order define how to interpret them.
3. readLine returns an empty string for a line with no characters. It returns null when no next line remains.
4. An int uses 4 bytes and a long uses 8 bytes. Total data size is 4 + 8 = 12 bytes.
5. CSV can contain quoted commas, quotes, and line breaks. A plain comma split does not follow those rules.
6. Use disposable copies or a failing storage test double. Check that failure is reported and the last valid data remains available under the save policy.

Chapter 8

1. Manual text storage chooses fields and parsing rules directly. Native serialization uses Java class information and object graph rules to save supported objects.
2. Default native serialization skips static fields and transient instance fields. Custom serialization can define additional behavior.
3. The transient field is not restored. Its normal instance initializer is not rerun for this ordinary serializable class, so it has the default int value 0.
4. Dangerous reconstruction work can happen before readObject returns. A later type check cannot undo it. Avoid unknown streams and restrict controlled streams before reading.
5. A new stream writes another header. A normal reader expecting one stream may reject the extra header or read corrupted structure.

6. A documented text or JSON format usually fits exchange with a non-Java system. Use proper parsing, version rules, and value checks.

Chapter 9

1. A console often follows a fixed input sequence. A GUI waits for user events and reacts through listeners.
2. Layout managers adapt to component sizes, fonts, and resizing. Fixed coordinates often leave controls clipped or badly spaced.
3. The EDT cannot process other screen work during that blocking operation. The interface appears frozen.
4. doInBackground does slow work. done handles the completed result on the EDT; process can handle published intermediate updates there.
5. Other callers can skip the GUI. The service must protect the real business rule no matter how an action begins.
6. Suitable tests include case-insensitive matching, blank-input handling, no-match output, Enter-key behavior, and window resizing. State expected results for each.

Chapter 10

1. Concurrency overlaps task progress. Parallelism runs tasks at the same time on separate processing resources.
2. run is an ordinary call on the current thread. start schedules a new thread to call the task.
3. Both threads read 4, both compute 5, and both write 5. One intended increment is lost.
4. volatile supplies visibility and ordering rules for the field but does not combine read, add, and write into one action.
5. get throws ExecutionException for a task's failure. Its cause holds the underlying problem. Waiting can also be interrupted.
6. Each of two tasks holds a lock the other needs. Use one consistent lock order and keep locked work short; avoid slow outside calls while holding locks.

Chapter 11

1. Scheme: http. Host: 127.0.0.1. Port: 8080. Path: /search. Query: q=java.
2. GET is for retrieval without requesting a state change. Browsers, crawlers, and caches may repeat it. Use an appropriate state-changing method with service checks.
3. URL decoding interprets query escapes such as %20. HTML escaping makes special text display safely in markup. They solve different problems.
4. A servlet instance can handle concurrent requests. Fields holding one request's user data may be overwritten or exposed to another request.
5. An unknown book can use 404. Invalid ID text fits 400. A last-copy state conflict fits 409 under this API design.
6. A transaction can make the stock check/update and loan creation succeed together. Two independent actions can otherwise oversell stock or leave incomplete records.

Chapter 12

1. A missing brace is a compile-time problem. A missing input file is a run-time failure. A wrong fine formula is a logic error that can run without an exception.
2. throw new IOException("Read failed"); raises a failure. A method declaration ending with throws IOException states a possible checked failure for callers.
3. Java selects the first matching handler. A general parent handler first would make a later child handler unreachable.
4. The caller may believe saving succeeded. Data can be lost while no useful message or cause remains.
5. If work and close both fail, the work failure normally stays primary and the close failure is suppressed. Both can be inspected during debugging.

6. One choice changes active memory only after the save succeeds. Another keeps the memory edit but marks it unsaved and reports failure with a retry or revert path. Choose and document the policy.

Chapter 13

1. One scan over n values is O(n). Two full nested n-item scans are O(n squared), because each outer step does n inner steps.
2. For 104, first inspect index 3, value 112; set high to 2. Next inspect index 1, value 104; return 1.
3. Discarding half relies on knowing which side contains smaller or larger values. A mismatched or unsorted order removes that guarantee.
4. Start [5, 2, 4, 1]. Insert 2: [2, 5, 4, 1]. Insert 4: [2, 4, 5, 1]. Insert 1: [1, 2, 4, 5].
5. A visited set prevents repeated work and endless cycles. BFS finds shortest paths by number of edges in an unweighted graph.
6. Greedy picks 4, then 1, then 1, using three tokens. The two 3-value tokens reach 6 with only two. The locally largest choice is not always globally best.

Chapter 14

1. The service enforces rules for every interface and test. Direct field edits can bypass stock and membership checks.
2. A loan ID identifies one borrowing event, links its return, and helps prevent duplicate returns. Book ID alone cannot identify which loan is being returned.
3. The first return removes the active loan. A later return cannot find it, so the service rejects the action before increasing stock.
4. Save books, active loans, member references, and next-ID information or a safe rule that derives it from all relevant saved IDs. Preserve history if its IDs must never be reused.
5. synchronized protects access through the same locked object in one JVM. Several servers need shared database coordination and transactions; their separate Java locks do not lock each other.
6. The ability to restore usable, consistent records remains untested. A file can exist but be incomplete, corrupted, or missing a required key or version.

Practice Exam Paper

Instructions and points

Suggested time: 2 hours. Total: 100 points. Answer all questions. Explain reasons and show calculation steps. Java code may use the Java 21 baseline. Clear equivalent designs can earn full credit when they keep the stated rules.

Part A short answers

1. Explain class, object, and reference with one book example. Then explain why private fields help protect a loan rule. 6 points.
2. Compare an interface, an abstract class, and an abstract method. Include one use for each. 6 points.
3. Compare overriding with overloading and explain how an actual object affects an overridden call. 6 points.
4. Compare checked and unchecked exceptions. Explain why returning an empty list after a failed file read may be misleading. 6 points.
5. Explain text encoding, binary type order, and try-with-resources. Give one risk of unknown native serialized data. 6 points.

Part B reasoning problems

6. Member has public int limit() returning 2. Student extends Member and overrides limit to return 3. For Member m = new Student(); predict m.limit(). Explain the declared type, actual type, and method selection. Then state whether Student-specific methods are directly available through m. 10 points.
7. A binary record contains an int ID, a long visit count, and a boolean flag. Give the primitive data byte count. Then explain what happens if a reader expects the long first and why a successful open does not prove a valid format. 10 points.
8. Two tasks try to lend the last copy. Each reads available = 1 before either writes. Show a bad schedule for separate check and update operations. Give a safe single-JVM design and state what changes for a multi-server database app. 10 points.

Part C design and code

9. Search for 18 in [3, 7, 11, 18, 22, 29, 35] using binary search. Show low, high, mid, and decision at every step. State its required input order, time complexity, and extra-space cost for an iterative implementation. Then explain why sorting before one search may be unnecessary. 10 points.
10. Design a library loan feature with a service, an interface for storage, and either a GUI or web view. Include valid/invalid input checks, a last-copy rule, duplicate-return protection, safe failure reporting, a save/reload plan, and four acceptance tests. Add a short Java method or clear pseudocode for the protected lend operation. 30 points.

Exam answer and scoring guide

Question 1: Award 1 point each for a class description, an object example, a reference explanation, and a shared-reference distinction. Award 2 points for explaining how methods enforce a rule around private state. Total 6.

Question 2: Award 2 points for an interface role with an example, 2 for an abstract shared base with an example, and 2 for a required no-body method completed by a concrete child. Total 6.

Question 3: Award 2 points for parameter-list overload selection at compile time, 2 for matching inherited instance behavior in an override, and 2 for actual-object run-time selection. Total 6.

Question 4: Award 2 points for the checked handle-or-declare rule, 2 for the unchecked distinction, and 2 for explaining how an empty list can falsely mean a successful read with no data. Total 6.

Question 5: Award 2 points for character-to-byte encoding, 1 for matching binary order, 1 for guaranteed normal resource closing through try-with-resources, and 2 for explaining that deserialization can execute dangerous reconstruction work before a later type check. Total 6.

Question 6: The result is 3. m is declared Member but holds a Student. The compiler sees limit in Member; run time chooses Student.limit. A Student-only method is unavailable directly through m. Award 2 points for each of those five ideas. Total 10.

Question 7: int uses 4 bytes, long uses 8 bytes, and this DataOutput boolean uses 1 byte. Total is 4 + 8 + 1 = 13 primitive data bytes, excluding headers. Reading long first consumes bytes from the wrong field boundaries and may produce meaningless values or later EOF. Opening proves file access, not schema, version, or value validity. Award 4 points for the calculation, 3 for type-order reasoning, and 3 for format validation. Total 10.

Question 8: A reads 1; B reads 1; both approve; both create a loan; both write 0. Two loans now claim one copy. Lock the whole check-and-change plus loan registration through the same service lock. For multiple servers, use a database transaction and suitable row/conditional update rules. Award 4 points for a valid bad schedule, 3 for one complete local atomic action, and 3 for database coordination. Total 10.

Question 9: low = 0, high = 6, mid = 3, value = 18. It is found immediately at index 3. Required order: ascending according to the same comparison. Worst-case time is O(log n), and iterative extra space is O(1). For one lookup in unsorted data, O(n) linear search may be cheaper than sorting first. Award 4 points for the trace, 2 for the order condition, 2 for the complexity pair, and 2 for the sorting tradeoff. Total 10.

Question 10: Award 5 points for clear layers and a storage interface; 5 for ID, member, and stock checks; 5 for one atomic lend action; 4 for loan IDs and duplicate-return protection; 4 for honest errors and preserved state; 3 for consistent save/reload and next-ID safety; and 4 for observable acceptance tests. Total 30. Equivalent designs can earn credit. A GUI-only input check, an empty catch, or two independent stock changes do not satisfy the rules.

Project Assessment Guide

Required submission

Submit source code, sample data, a README, the completed acceptance table, and a short class diagram. Include build and run commands that another student can follow. State whether the interface is console, GUI, web, or a combination. Clearly name optional external dependencies and their compatible versions.

Use a small invented data set: at least six books, three members, two roles, and several loans. Include an unavailable book and an overdue loan. Do not use real student names or private records. Save an example of a rejected input and show that it leaves stock unchanged.

Scoring rubric

Area	Points	Evidence
OOP design	20	Private state, clear service, useful interface, suitable inheritance or composition
Loan rules	20	Unique IDs, member limits, last-copy safety, correct returns
Storage	15	Documented format, validated load, save/reload, failure policy
User interface	10	Clear labels or routes, input messages, usable success and failure paths
Concurrency	10	Safe shared state, background report, clear completion/cancellation rule
Algorithms	10	Correct search/sort choice, complexity explanation, edge cases
Testing	10	Repeatable normal, boundary, failure, and integration checks
Delivery	5	Reproducible build/run instructions and stated limits
Total	100	A working project supported by evidence

Review checklist

1. Can another student compile and run the program using only the README?
2. Does every input path call the same service rules?
3. Can stock ever become negative or increase twice for one returned loan?
4. Does a failed load avoid replacing valid active data with partial records?
5. Does a failed save clearly remain failed to the user?
6. Can the saved state restart without duplicate loan IDs?
7. Does background work use a safe snapshot or a documented lock rule?
8. Are every external file, request parameter, and restored record checked?
9. Are technical failures logged without secrets?
10. Do tests check both output and state after rejection?

Short oral defense

Explain why you chose your main collection, which behavior is polymorphic, where a required rule is enforced, and how a bad file is handled. Trace one complete request from input to service to storage and back. Explain what synchronized protects in your version and what it does not protect. Show one tradeoff rather than claiming the project has no limits.

Quick Reference Tables

Java keywords and roles

Term	Quick reminder
extends	Gives a class one direct superclass or an interface its parent interfaces
implements	Makes a class support an interface
abstract	Marks an incomplete class or required no-body method
final	Stops reassignment, overriding, or subclassing depending on use
static	Makes a member belong to the class
this	Refers to the current object or another constructor in the same class
super	Selects parent construction or parent member behavior
throw	Raises an exception object now
throws	Declares possible propagated exception types
transient	Skips a normal instance field in default native serialization
synchronized	Uses a monitor lock for coordinated access
volatile	Adds field visibility and ordering guarantees; not compound atomicity

Matching primitive file operations

These are DataOutputStream/DataInputStream pairs. The file must define which pair comes next. Sizes below refer to their binary data representation, not ordinary object memory use.

Value type	Write operation	Read operation	Data bytes
boolean	writeBoolean	readBoolean	1
byte	writeByte	readByte	1
short	writeShort	readShort	2
char	writeChar	readChar	2
int	writeInt	readInt	4
long	writeLong	readLong	8
float	writeFloat	readFloat	4
double	writeDouble	readDouble	8
String in modified UTF-8	writeUTF	readUTF	2-byte length plus encoded bytes, at most 65,535 encoded bytes

readUnsignedByte and readUnsignedShort return int values when the format needs nonnegative byte or short interpretation. They do not create new unsigned primitive types. Object streams also offer primitive operations, but their framing means total object-stream file size is not just the simple data sizes in this table.

Small glossary

Term	Easy meaning
API	Named classes and methods that other code can use
Bytecode	Compiled instructions for a JVM
Constructor	Setup code called when a new object is created normally
Dynamic dispatch	Choosing an override from the actual object at run time
Encapsulation	Keeping state with methods that protect its rules
Immutable	Unable to change after construction under its design
Invariant	A rule that should stay true at a stated point or boundary
Interface	A type that describes a supported role
Lambda	A short behavior expression for a functional interface
Serialization	Converting supported state into a saved byte form
Event	Something that triggers a response, such as a click
Race condition	A result that wrongly depends on uncontrolled task timing
Deadlock	Tasks waiting forever for resources held by each other
Transaction	Related changes handled as one success or failure unit
Repository	A component responsible for storing and finding data
Complexity	How work or memory needs grow with input size

Sources and Further Reading

Official language and library sources

The explanations and classroom programs in these notes are written for teaching. The official sources below provide detailed contracts and further study. The named references in chapters point to the matching entries here. Use the Java 21 documentation for the baseline examples. Check newer versions separately before changing a project baseline.

1. Dev.java inheritance guide. Shared classes, overrides, polymorphism, and abstract behavior. Open official source
2. Oracle Java SE 21 API. The complete baseline class and method reference. Open official source
3. Java Files reference. Text, binary, path-based, and file replacement operations. Open official source
4. Java DataInput reference. Binary primitive input and modified UTF-8 rules. Open official source
5. Java Serializable reference. Class support and reconstruction rules. Open official source
6. Java ObjectInputFilter reference. Filters, limits, and native deserialization cautions. Open official source
7. Swing package reference. Components and the EDT access policy. Open official source
8. Java Executors reference. Task executors, pools, and virtual-thread factories. Open official source
9. JDK HTTP server module reference. The small HTTP/HTTPS server API used in the local lab. Open official source
10. Tomcat version guide. Compatibility between Tomcat, Java, and servlet specifications. Open official source
11. Jakarta Servlet 6.0. The servlet specification and API for the optional Tomcat 10.1 lab. Open official source
12. Java exception tutorial. Resource closing and suppressed exceptions; the tutorial's JDK 8 examples remain useful for these established rules. Open official source
13. Java Collections reference. Standard collection algorithms and contracts. Open official source
14. Java Comparator reference. Custom comparison and ordering rules. Open official source
15. JDBC Connection reference. Database connections, commits, and transaction controls. Open official source