Java Spring Series

Java Programming: Intermediate to Advanced

Master Advanced Java Programming
From Confident Coder to Professional Developer

Gerry Byrne
David Wilson

Java Programming

Intermediate to Advanced

Step by step instructions
for practical hands-on programming

Gerry Byrne and David Wilson

ABOUT THE AUTHORS

Gerry Byrne and David Wilson are Senior Technical Trainers for a Forbes 100 company. They work to upskill and reskill software engineers who develop business-critical software applications. They also help refine the programming skills of returners to the workforce and introduce new graduates to the application of software development within the commercial environment.

Their subject expertise has been developed over multi-decade careers as teachers, lecturers, and technical trainers in a corporate technology environment. They have delivered a range of courses across computer languages and frameworks and understand how to teach skills and impart knowledge to a range of learners. They have taught software engineers in the use of modern technologies and frameworks such as C#, Java, Spring, Android, JavaScript, Node, HTML, CSS, Bootstrap, React, Python, and Test-Driven Development, not to mention legacy technologies such as COBOL and JCL.

Gerry and David have mastered how to teach difficult concepts in a simple way that makes learning accessible and enjoyable. Their delivery and content follow the same philosophy of keeping it simple, while making the instructions detailed and applying concepts to real-world scenarios. They are passionate about software development and believe we can all learn to write code if we are patient, grasp the basic coding concepts, get plenty of hands-on coding, and most of all, persevere through 'thick and thin'.

DEDICATION

Writing a book is a rewarding undertaking, but it requires time, effort and patience. It requires patience from those who help you write the book and those around you in your life.

So, we start by thanking our families for 'facilitating' us as we worked over many hours, days, weeks and months to write this programming book.

We also wish to thank each other, we have learnt so much in writing this book and in delivering many enjoyable programming courses. When we need coding inspiration, we work together, we try things, and we persevere in pursuit of excellence.

ACKNOWLEDGMENTS

We would like to express our deepest gratitude to everyone who contributed to the creation of this book. We are especially grateful to those who offered invaluable guidance and insightful feedback during the writing process.

Special thanks to the open-source community and all those who have contributed to the Java platform and its resources, which have been fundamental in shaping the examples and content presented here.

Finally, to all the readers, thank you for your interest and enthusiasm. We hope this book serves as a valuable resource on your journey with Java programming, and programming in general.

15 Understanding Inheritance

Inheritance is a mechanism that allows one class, the subclass, to inherit the properties and behaviors of another class, the superclass. Inheritance is used to share structure and behavior between classes This promotes code reuse and reduces redundancy.

In our insurance scenario, inheritance could work by having the base class where we define the common offerings and then we can extend the base class into tailored products without duplicating logic. It is efficient and keeps policies consistent across product lines.

Think about the Policy class we mentioned in the encapsulation pillar in Chapter 14. The policy could be used as a base class and has shared features like policyHolder, startDate, and premiumAmount. From this base class we could derive specialized policies like LifeInsurancePolicy or AutoInsurancePolicy, each adding its own traits.

In our insurance example, we can create a base class **InsurancePolicy** and derive specialized policies from it. We will create three new classes inside a package, one called InsurancePolicy which will be the superclass, one called LifeInsurancePolicy which will be a sub class of the superclass, and the third will be called AutoInsurancePolicy which will also be a sub class of the superclass. The hierarchy is shown below:

```
InsurancePolicy
├── LifeInsurancePolicy
└── AutoInsurancePolicy
```

We will also create an **InsuranceApplication** class with a main() method that will demonstrate the use of the super and sub classes.

1. Right click on the **code** folder.
2. Choose **New**.
3. Choose **Package**.
4. Name the package **chapter15**.
5. Press the **Enter** key.
6. Right click on the **chapter15** package.
7. Choose **New**.
8. Choose **package**.

9. Name the package **inheritance**.
10. Right click on the **inheritance** package.
11. Choose **New**.
12. Choose **Java Class**.
13. Name the class **InsurancePolicy**.

This will be our base class, the superclass. The class contains:

- Five private properties, policyHolderName, premiumAmount, coverageType, startDate and endDate
- A parameterized constructor that has five parameters and is used to initialize all the properties.
- A method that is used to display the details of the insurance policy.
- Getters and setters for the private properties (encapsulation).
- A toString() method.

14. Amend the code as Listing 15-1.

Listing 15-1. Creating the superclass

```
package code.chapter15.inheritance;

public class InsurancePolicy
{
    // Declare the class properties, these are the attributes of the class
    private String policyHolderName;
    private double premiumAmount;
    private String coverageType;
    private String startDate;
    private String endDate;

    // Declare the parameterized constructor of the class
    public InsurancePolicy(String policyHolderName, double premiumAmount,
                           String coverageType, String startDate, String endDate)
    {
        this.policyHolderName = policyHolderName;
        this.premiumAmount = premiumAmount;
        this.coverageType = coverageType;
        this.startDate = startDate;
        this.endDate = endDate;
    } // End of parameterized constructor

    // Declare the displayPolicyDetails() method, this is a behavior of the class
    public void displayPolicyDetails() {
        System.out.println("Policy Details");
        System.out.printf("%-20s %s\n", "Policy Holder Name:", policyHolderName);
        System.out.printf("%-20s $%.2f\n", "Premium Amount:", premiumAmount);
        System.out.printf("%-20s %s\n", "Coverage Type:", coverageType);
```

```
        System.out.printf("%-20s %s\n", "Start Date:", startDate);
        System.out.printf("%-20s %s\n", "End Date:", endDate);
    } // End of displayPolicyDetails() method

    // Getters and Setters for encapsulation
    public String getPolicyHolderName() {
        return policyHolderName;
    } // End of getPolicyHolderName()

    public void setPolicyHolderName(String policyHolderName) {
        this.policyHolderName = policyHolderName;
    } // End of setPolicyHolderName()

    public double getPremiumAmount() {
        return premiumAmount;
    } // End of getPremiumAmount()

    public void setPremiumAmount(double premiumAmount) {
        this.premiumAmount = premiumAmount;
    } // End of setPremiumAmount()

    public String getCoverageType() {
        return coverageType;
    } // End of getCoverageType()

    public void setCoverageType(String coverageType) {
        this.coverageType = coverageType;
    } // End of setCoverageType()

    public String getStartDate() {
        return startDate;
    } // End of getStartDate()

    public void setStartDate(String startDate) {
        this.startDate = startDate;
    } // End of setStartDate()

    public String getEndDate() {
        return endDate;
    } // End of getEndDate()

    public void setEndDate(String endDate) {
        this.endDate = endDate;
    } // End of setEndDate()

    // Override the toString() method to provide a string
    // representation of the object
    @Override
    public String toString() {
        return "InsurancePolicy{" +
                "policyHolderName='" + policyHolderName + '\'' +
                ", premiumAmount=" + premiumAmount +
                ", coverageType='" + coverageType + '\'' +
                ", startDate='" + startDate + '\'' +
                ", endDate='" + endDate + '\'' +
                '}';
    } // End of toString() method

} // End of InsurancePolicy class
```

Polymorphism in Java

Imagine we are building a software system for an insurance company that offers various types of policies, health, auto, and life insurance. While each policy type calculates its premium differently, they all share common behaviors, such as displaying the policyholder's information. To design a system that's flexible, maintainable, and scalable, we need a way to handle these variations without duplicating code or creating rigid structures. This is where polymorphism becomes essential.

To see this, follow the instructions below:

15. Right click on the **chapter15** package.
16. Choose **New**.
17. Choose **Package**.
18. Name the package **polymorphism.overloading**.
19. Right click on the **overloading** package.
20. Choose **New**.
21. Choose **Java Class**.
22. Name the class **PremiumCalculator**.
23. Amend the code as Listing 15-7.

Listing 15-7. Create the PremiumCalculator class

```
package code.chapter15.polymorphism.overloading;

public class PremiumCalculator {

    /*
    Method to calculate premium for Health Insurance. It takes age as a parameter
    of type int and returns the premium amount based on the age criteria.
    */
    public double calculatePremium(int age) {
        return age < 40 ? 5000 : 8000;
    } // End of calculatePremium() method with int parameter

    /*
    Overloaded method to calculate premium for Auto Insurance. It is overloaded
    because of the different parameter type (String instead of int or double).
    */
    public double calculatePremium(String vehicleModel) {
        return vehicleModel.equalsIgnoreCase("SUV") ? 10000 : 7000;
    } // End of calculatePremium() method with String parameter

    /*
    Overloaded method to calculate premium for Life Insurance. It is overloaded
    because of the different parameter type (double instead of int or String).
```

```
    */
    public double calculatePremium(double coverageAmount) {
        return coverageAmount * 0.02;
    } // End of calculatePremium() method with double parameter
} // End of PremiumCalculator class
```

16 Abstraction and Interfaces

Abstraction is about defining **what** an object should do, without specifying **how** it does it. It is the art of hiding complexity and exposing only the essential features. In other words, abstraction lets us design systems that are **clean**, **extensible**, and **easy to reason about**.

Imagine we are building an insurance platform. We do not want to worry about the internal details of every policy type, we just want to know that each one can calculate a premium, validate coverage, or generate a summary. That is what abstraction is, we define the required behaviours and let each policy type handle the specifics. In Java, abstraction is implemented through:

- **Abstract classes** which can define both concrete and abstract methods.
- **Interfaces** which declare method signatures that implementing classes must fulfill and, since Java 8, can also include default and static methods with concrete implementations.

An **abstract class** in Java lets us define a **common structure and expected behaviors** for a group of related classes, while leaving the specific implementation to each subclass. If we think about our insurance platform, we may have an insurance policy template which states that every policy must calculate a premium and print its details. When we use the template to design a real insurance policy, we will need to calculate the premium and print the details specific to the policy type e.g., CarPolicy, HealthPolicy, HomePolicy.

1. Right click on the code folder.
2. Choose **New**.
3. Choose **Package**.
4. Name the package **chapter16**.
5. Press the **Enter** key.
6. Right click on the **chapter16** package.
7. Choose **New**.
8. Choose **package**.
9. Name the package **abstraction**.
10. Right click on the **abstraction** package.
11. Choose **New**.

12. Choose **Java Class**.
13. Name the class **InsurancePolicy**.

We will now create an abstract class called InsurancePolicy to serve as a blueprint for different types of insurance policies. In this class, we will add two protected fields, policyHolder and basePremium. These fields will store information common to all insurance policies and will be accessible to subclasses.

Next, we will create a parameterized constructor to initialize these fields when an object is created. Then, we will declare an abstract method named calculatePremium(), which will require all subclasses to provide their own implementation for calculating the premium.

Finally, we will implement two concrete methods, printPolicyHolder() and printBasePremium(). These methods will allow us to display the policy holder's name and the base premium, and they can be used by any subclass, but this is not mandatory. Our class design helps to establish a flexible and extensible foundation for various insurance policy types.

14. Amend the code as Listing 16-5.

Listing 16-5. Abstract class

```
package code.chapter16.abstraction;

// Create an abstract class- InsurancePolicy by adding the keyword abstract
public abstract class InsurancePolicy
{
    /*
    Create the fields common to all insurance policies. These are protected
    so that subclasses can access them directly.
    */
    protected String policyHolder;
    protected double basePremium;

    // Parameterized constructor
    public InsurancePolicy(String policyHolder, double basePremium) {
        this.policyHolder = policyHolder;
        this.basePremium = basePremium;
    }

    // Abstract method which must be implemented by subclasses
    public abstract double calculatePremium();

    // Concrete method which is shared by all policies
    public void printPolicyHolder()
    {
        System.out.println("Policy Holder: " + policyHolder);
    } // End of printPolicyHolder() method
```

```
    // A method to print the base premium and is optional for subclasses to use
    public void printBasePremium() {
        System.out.println("Base Premium: $" + basePremium);
    } // End of printBasePremium() method
    }// End InsurancePolicy class
```

The class can be thought of as a contract that all policies must abide by. It says, "Every insurance policy must have a policyholder, a base premium, and a way to calculate the final premium."

To setup this contract any subclass must extend the abstract InsurancePolicy class, thereby agreeing to fulfill certain responsibilities defined by the base class.

15. Right click on the **abstraction** package.
16. Choose **New**.
17. Choose **Java Class**.
18. Name the class **AutoInsurancePolicy**.

Interfaces

Imagine we are building a software system for an insurance company that offers various types of insurance, such as auto, home, and life. Each type of insurance has some common tasks, like calculating premiums, issuing policies, and processing claims, but they each perform these tasks differently.

Our challenge, therefore, is to ensure that every type of insurance follows the same basic rules while still allowing them to operate in their unique ways.

This is where interfaces come to our assistance.

Think of an interface like a contract. It says: "If you want to be considered an Insurance type, you must promise to do these things." But it does not care how we do them, that is up to us. In Java, an interface is a way to define what methods a class should have, without writing the actual code for those methods. In Java, an interface is a special type that:

- Defines method names, but not how they work.
- Says, "Any class that implements me must provide these methods."

- Does not contain properties, except constants.
- Can be implemented by any class, even if they are unrelated.
- Lets us build flexible systems where different classes can share behavior without sharing structure.

From Java 8 onwards, interfaces can also contain **default methods** and **static methods**. Default methods provide a way to specify a default implementation for a method, which can be overridden by implementing classes if needed. Static methods in interfaces can be called independently of any object instance, providing utility methods related to the interface.

Listing 16-13 shows a simple interface.

Listing 16-13. Interface example

```
public interface InsurancePolicy {
    // Every insurance policy must be able to calculate its premium
    double calculatePremium();
}
```

Notice the use of the name **interface** rather than the name **class**. To use an interface, a class must say it **implements** that interface as shown in Listing 16-14. This means the class agrees to provide concrete versions of all the methods listed in the interface.

17 Records, Sealed classes and interfaces

Records

Records were introduced in Java 14 as a preview feature and became a standard feature in Java 16. Records are a special kind of class in Java, designed to simplify the creation of classes that are primarily used to store data. In programming we can refer to boilerplate code as including things like getters and setters, constructors, and other common methods that do not add much value but are necessary for the program to function correctly. Using a record can help reduce boilerplate code by automatically providing a constructor to initialize the fields and adding implementations for methods such as **equals(), hashCode()**, and **toString()**, as well as providing a compact syntax for defining immutable fields.

Traditional class to hold data

We will look at an example to see how we would create a 'traditional' class to hold data and have the constructor and methods, **equals(), hashCode()**, and **toString()**, Then we will see the new way to do things using a record, and therefore there will be less boilerplate code.

1. Right click on the **code** package.
2. Choose **New**.
3. Choose **Package**.
4. Name the package **chapter17**.
5. Press the **Enter** key.
6. Right click on the **chapter17** package.
7. Choose **New**.
8. Choose **Java Class**.
9. Name the class **PolicyHolder**.
10. Press the **Enter** key.

The PolicyHolder class will be used to represent a PolicyHolder with:

- Three private fields: name, address, and email.
- A constructor to initialize all fields.
- Getter methods for each field, getName(), getAddress(), getEmail().
- Override methods for:
    - The equals() method to compare PolicyHolder objects based on their fields.

- The hashCode() method to generate a hash code based on the fields.
- The toString() method to return a formatted string representation of the object.

11. Amend the code as shown in Listing 17-1, and add any required imports.

Listing 17-1. InsurancePolicy traditional class

```
package code.chapter17;

import java.util.Objects;

// A traditional class that represents a PolicyHolder
public class PolicyHolder {
    // Private fields for the PolicyHolder class
    private final String name;
    private final String address;
    private final String email;

    // Parameterized constructor with all fields
    public PolicyHolder(String name, String address, String email) {
        this.name = name;
        this.address = address;
        this.email = email;
    }

    // Getters
    public String getName() {
        return name;
    }
    public String getAddress() {
        return address;
    }

    public String getEmail() {
        return email;
    }

    // Methods
    // The equals method is used to compare the
    // object with another object
    @Override
    public boolean equals(Object myObject) {
        if (this == myObject) {
            return true;
        }
        if (myObject == null || getClass() != myObject.getClass()) {
            return false;
        }
        // Cast the object to a PolicyHolder
        PolicyHolder otherPolicyHolder = (PolicyHolder) myObject;
        return Objects.equals(name, otherPolicyHolder.name) &&
                Objects.equals(address, otherPolicyHolder.address) &&
                Objects.equals(email, otherPolicyHolder.email);
    } // End of equals() method

    // The hashCode method is used to determine the
```

```
    // location of the object in the memory
    @Override
    public int hashCode() {
        return Objects.hash(name, address, email);
    } // End of hashCode() method

    // The toString method is used to return the string representation of the object
    @Override
    public String toString() {
       return String.format("PolicyHolder{name='%s', address='%s', email='%s'}",
                                name, address, email);
    } // End of toString() method
} // End of the PolicyHolder class
```

Sealed class example

In this example, we will use a sealed class to define different insurance policy types, AutoInsurancePolicy, HomeInsurancePolicy and HealthInsurancePolicy. These classes will be declared final. We will also create a PolicyHolder record, InsurancePolicy sealed class and an InsuranceApplication class, which contains the main method and a method to generate an insurance policy document using the sealed class.

1. Right click on the **chapter17** package.
2. Choose **New**.
3. Choose **Package**.
4. Name the package **sealedclasses**.
5. Press the **Enter** key.

We will create the PolicyHolder record.

18 Exception Handling

What is an Exception?

We should think of an exception as an exceptional event that occurs when an application is being executed. If we think about a time when we have seen an unhandled exception in our personal life, we will understand the consequences of such exceptions. An example of an unhandled exception in our personal life could be forgetting an important appointment, such as a doctor's visit. Imagine we have scheduled the appointment weeks in advance but did not set a reminder. When the appointment day arrives, we carry on with our life as usual, oblivious to the missed appointment we needed so much. The consequences might range from simply having to reschedule the appointment, to more serious outcomes, like missing a critical medical test.

exceptions that we might encounter are:

File not found exception

When code is required to read a text file, as Listing 18-1, it is possible that the text file could be corrupted or not at the location. If the application runs, we would get a **file not found exception**.

Listing 18-1. Code will cause an exception

```
import java.io.FileReader;

public class FileNotFoundExample {
    public static void main(String[] args) { FileReader reader = new
FileReader("policydetail.txt") ;
            int character;

            // Read characters until the end of the file is reached
            while ((character = reader.read()) != -1) {
                System.out.print((char) character);
            }
    } // End of main() method
} // End of FileNotFoundExample class
```

In Java we will not be allowed to compile the code without having code in place that handles a possible exception. Figure 18-1 shows the exception handling being requested by the compiler.

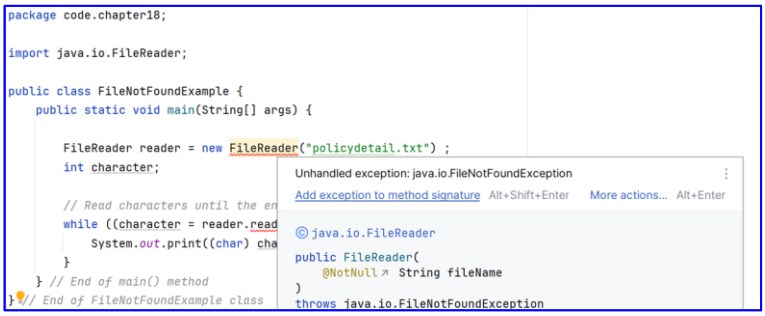

Figure 18-1. Compile Error as FileReader could cause an error

19 File Handling

An Overview of File Class

It is great to be able to read and write files, but we should understand that files need to be in a directory, which needs to be created, and this is one role for the File class. Equally we may need to rename, delete, or move a file or directory, and these are other roles for the File class methods. We might even need to check if a file exists before we move, rename or delete it, and once again the File class can help us. The File class exists within the java.io package and we can use the pathname java.io.File.

The **FileMakeDirectory** class code with the main() method added should be like the code shown in Listing 19-1.

Listing 19-1. FileMakeDirectory class with main() method

```
package code.chapter19;

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

    } // End of main method
} // End of FileMakeDirectory class
```

The way forward

Now that we've covered the core and advanced concepts in Java, we are in a great position to develop and understand enterprise application code. As with many things in life, there is always more to learn. Java programming is no different, there is still plenty to explore and master.

This is a great beginning, and now it's time to think about real-world applications that use Java integrated with Spring Boot. Spring Boot is a powerful framework that makes it much easier to build robust, production-ready applications and APIs with Java.

APIs (Application Programming Interfaces) are now commonplace in the technology world. Many people use APIs every day without even realizing it. For example, when we listen to music on Spotify, get directions from Google Maps, or check the weather on our phone, we are interacting with APIs developed by others, which provide access to specific features or data.

In the next book in the Java Spring Series, **Spring Boot API – Insurance Quote Application,** we build on the extensive Java knowledge we have gained from the Java Programming: Beginners to Intermediate book and the Java Programming: Intermediate to Advanced book. We will learn how to create APIs and microservices using our existing Java skills, while extending our expertise with frameworks and libraries that help us build scalable, enterprise-style applications. We will cover the layers of a typical Spring Boot API (Controller, Service, Repository, Exception, and Model), use Agile user stories and the Gherkin format with Test Driven Development (TDD), and work with Spring Boot features such as JPA, database connectivity, database queries, and derived queries.

There is a lot to learn, so let's get started with **Spring Boot API – Insurance Quote Application**.