DEITEL® DEVELOPER SERIES

FIFTH EDITION

Java for Programmers *with* Generative AI

- GenAI Prompt Engineering, API Calls, 600 GenAI Exercises
- ChatGPT, Gemini, Claude, Perplexity
- Multimodal: Text, Code, Images, Audio, Speech-to-Text, Text-to-Speech, Video
- Generics: Collections, Classes, Methods
- Functional Programming: Lambdas & Streams
- JavaFX: GUI, Graphics, Multimedia
- Concurrency: Parallel Streams, Virtual Threads, Structured Concurrency, Scoped Values, Concurrent Collections, Multi-Core
- Database: JDBC, SQL, SQLite
- Java Platform Module System (JPMS)
- Objects Natural: Java API, String, BigInteger, BigDecimal, Date/Time, Cryptography, ArrayList, Regex, JSON, CSV, Web Services
- JShell for Python-Like Interactivity

PAUL DEITEL • HARVEY DEITEL

Generative AI Innovations in *Java for Programmers, 5e*

Fully Coded GenAI Case Studies

Chapter 19 presents the following **code examples that interact with OpenAI's APIs**: **Text Summarization**, **Sentiment Analysis**, **Accessible Image Descriptions**, **Language Detection & Translation**, **Java Code Generation**, **Named-Entity Recognition & Structured Outputs**, **Speech-to-Text**, **Text-to-Speech**, **Image Generation**, **Creating Closed Captions for a Video**, **Moderation**.

GenAI Prompt Exercises

We fed the complete list of all the book's **approximately 600 genAI exercises** (a 100+ page PDF) to **ChatGPT**, **Gemini**, **Claude** and **Perplexity**, asking them to categorize the kinds of things we do in those exercises. Next, we fed their categorized lists to the four genAIs, asking them to summarize the summaries, and we chose the best one—**Claude** in this case:

- **Code Generation and Implementation**—Writing new Java programs from specifications. Implementing specific features, algorithms and APIs. Creating test programs and practical applications. Generating solutions for basic and advanced tasks.
- **Code Refactoring and Enhancement**—Modernizing code. Improving code structure, readability, and maintainability. Converting between different approaches while maintaining functionality. Improving performance.
- **Educational Content**—Creating tutorials, exercises, and learning materials. Further exploring complex concepts. Developing programming exercises. Writing comprehensive documentation and guides.
- **Technical Analysis**—Analyzing code behavior and feature implementations. Comparing different approaches, tools, and frameworks. Evaluating trade-offs in design decisions. Breaking down complex technical concepts.
- **Best Practices and Standards**—Implementing coding standards and design patterns. Addressing security considerations. Optimizing performance. Following Java development best practices.
- **Technology Evaluation**—Comparing libraries, tools, and frameworks. Assessing the pros and cons of different approaches. Making informed technology choices. Exploring new features and updates.
- **Debugging and Error Handling**—Finding and fixing syntax and logical errors. Implementing exception handling. Improving fault tolerance. Preventing common pitfalls.
- **API and Library Integration**—Working with Java APIs and external libraries. Understanding API features and capabilities. Implementing integration techniques. Creating API documentation and tutorials.
- **Real-world Applications**—Developing practical use cases and industry applications. Creating interactive applications (GUIs, games, multimedia). Implementing real-world scenarios. Building sample projects.
- **Performance Optimization**—Analyzing and improving performance. Optimizing resource usage. Conducting benchmarks. Implementing efficiency improvements.
- **Creative Development**—Building multimodal applications. Creating visualizations. Generating test scenarios and sample data. Developing unique use cases.

GenAI API-Based Java Programming Exercises

Chapter 19, Building API-Based Java Generative AI Applications, suggests challenging project exercises like creating genAI multimedia apps that can debate one another and using genAIs to build and solve crossword puzzles. We fed the 94 exercises into the genAIs, asking for a categorized summary of them, then summarized the summaries. Here's what **Claude** produced:

- **Multimodal Applications**— Building comprehensive tools that combine text, image, audio, speech and video capabilities. Creating integrated experiences like interactive books. Developing multimedia educational content.
- **Text-Based Applications**—Document processing (indexing, summarization, exploration). Creative writing (stories, poetry, debates). Language tools (translation, tone rewriting). Professional document creation (resumes, presentations). Structured outputs.
- **Image Processing Applications**—Generative art and design (logos, fashion, floor plans). Technical visualization (UML diagrams). Image analysis and recognition.
- **Audio and Music Applications**—Speech processing (transcription, voice cloning). Music generation (MIDI, Magenta AI). Multilingual audio applications. Podcast and audio content analysis.
- **Educational Tools**—Programming tutors (Java, coding exercises). Subject-specific learning aids (math). Course content creation. Interactive educational experiences.
- **Gaming and Puzzle Applications**—Puzzle generators and solvers. Interactive game development.
- **Video**—Investigating and experimenting with generative AI video creation tools.
- **Chatbot Development**—Character-based chat experiences. Specialized domain experts.
- **Research and Analysis Tools**—Medical applications (researching drug discovery and personalized medicine). AI capability exploration. Text detection and analysis. Educational research.
- **Creative Applications**—Children's book creation. Interactive storytelling. Artistic content generation. Creative writing.
- **Practical Tools and Utilities**—Document generators. Translation services. Content summarizers. Professional tools (resume filters, presentation creators).

"Wei Li" to the constructor to initialize that object's name. Line 9 repeats this process, passing the argument "Logan Brown" to initialize account2's name. Lines 12–13 use each object's getName method to obtain its name. The output shows different names, confirming that each Account maintains its own copy of the instance variable name.

```
1  // Fig. 8.4: AccountTest.java
2  // Using the Account constructor to initialize the name
3  // instance variable of each new Account object.
4
5  public class AccountTest {
6     public static void main(String[] args) {
7        // create two Account objects
8        var account1 = new Account("Wei Li");
9        var account2 = new Account("Logan Brown");
10
11       // display initial value of name for each Account
12       System.out.printf("account1 name is: %s%n", account1.getName());
13       System.out.printf("account2 name is: %s%n", account2.getName());
14    }
15 }
```

```
account1 name is: Wei Li
account2 name is: Logan Brown
```

Fig. 8.4 | Using the Account constructor to initialize the name instance variable of each new Account object.

Constructors Cannot Return Values

A difference between constructors and methods is that constructors cannot return values, so they do not specify a return type (not even void). Usually, constructors are public so other classes can call them—in Chapter 9, we'll discuss when to use private constructors.

Default Constructor

Recall that line 11 of Fig. 8.2

```
var myAccount = new Account();
```

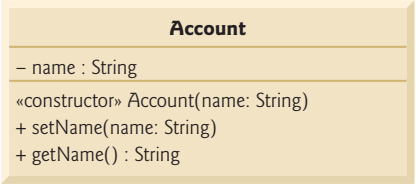
used new to create an Account object. The empty parentheses after "new Account" indicate a call to the **class's default constructor**. The compiler provides a default constructor with no parameters in any class that does not explicitly declare a constructor. When a class has only the default constructor, its instance variables are initialized to their default values. In Section 8.10, you'll see that classes can have multiple constructors.

There's No Default Constructor in a Class That Declares a Constructor

If you declare a constructor with one or more parameters, you will not be able to create an Account with the expression new Account() as in Fig. 8.2, unless you also declare a custom constructor with no parameters. If default initialization of your class's instance variables is not acceptable, provide a custom constructor to ensure each new object's instance variables are initialized with meaningful values.

Adding the Constructor to Class Account's UML Class Diagram

The following is the UML class diagram for class `Account` (Fig. 8.3), which has a constructor with a `String name` parameter:



The UML models a constructor with other operations in a class diagram's third compartment and distinguishes it from the class's operations by enclosing the word "constructor" in guillemets (« and ») before its name. It's customary to list constructors before other operations in the third compartment and before other operations in the class declaration.

Generative AI

1 Prompt genAIs to provide insights as to why Java constructors cannot return values.

8.5 Account Class with a Balance

We now declare an `Account` class that maintains a bank account's name and balance. Class `Account` represents the account balance as a `BigDecimal`.

8.5.1 Account Class with a BigDecimal balance Instance Variable

Our next version of class `Account` (Fig. 8.5) maintains `name` and `balance` instance variables. A typical bank services many accounts, each with its own balance, so line 8 declares an instance variable `balance` of type `BigDecimal`.

```
1  // Fig. 8.5: Account.java
2  // Account class with a BigDecimal instance variable balance, and
3  // a constructor and deposit method that perform validation.
4  import java.math.BigDecimal;
5
6  public class Account {
7     private String name; // instance variable
8     private BigDecimal balance = BigDecimal.ZERO; // instance variable
9
10    // Account constructor that receives two parameters
11    public Account(String name, BigDecimal balance) {
12       this.name = name; // assign name to instance variable name
13
14       // validate that the balance is greater than 0; if it's not,
15       // instance variable balance keeps its default initial value of 0
16       if (balance.compareTo(BigDecimal.ZERO) > 0) { // is balance valid?
17          this.balance = balance; // assign it to instance variable balance
18       }
19    }
```

Fig. 8.5 | `Account` class with a `BigDecimal` instance variable `balance` and a constructor and `deposit` method that perform validation. (Part 1 of 2.)

```
20
21      // method that deposits (adds) only a valid amount to the balance
22      public void deposit(BigDecimal amount) {
23         if (amount.compareTo(BigDecimal.ZERO) > 0) { // is amount valid?
24            balance = balance.add(amount); // add it to the balance
25         }
26      }
27
28      // method returns the account balance
29      public BigDecimal getBalance() {
30         return balance;
31      }
32
33      // method that sets the name
34      public void setName(String name) {
35         this.name = name;
36      }
37
38      // method that returns the name
39      public String getName() {
40         return name;
41      }
42   }
```

Fig. 8.5 | `Account` class with a `BigDecimal` instance variable `balance` and a constructor and `deposit` method that perform validation. (Part 2 of 2.)

Account Class Two-Parameter Constructor

The class has a constructor and four methods. It's common for someone opening an account to deposit money immediately, so the constructor (lines 11–19) now receives a second parameter—`BigDecimal balance`—representing the starting balance. Lines 16–18 ensure that the `balance` parameter's value is greater than `BigDecimal.ZERO`. If so, line 17 assigns it to the instance variable `balance`. Otherwise, the instance variable `balance` remains at `BigDecimal.ZERO`—its default initial value in line 8. Later in this chapter, we'll throw exceptions for invalid initial values.

Account Class deposit Method

Method `deposit` (lines 22–26) does not return any data when it completes execution, so its return type is `void`. The method receives a `BigDecimal` parameter named `amount` that we add to the instance variable `balance` only if the parameter's value is greater than the value `BigDecimal.ZERO`; otherwise, `balance` remains unchanged.

Account Class getBalance Method

Method `getBalance` (lines 29–31) specifies no parameters and returns a `BigDecimal`, enabling the class's clients to obtain a particular `Account` object's `balance`.

Generative AI

1 Prompt genAIs asking why Java does not allow you to use convenient operators like `+` and `-` when manipulating `BigDecimal` monetary amounts, as you can with primitive types like `int` and `double`.

8.5.2 AccountTest Class

Class `AccountTest` (Fig. 8.6) creates two `Account` objects (lines 9–10) and initializes them with a valid balance of `50.00` and an invalid balance of `-7.53`, respectively—for our examples, we assume that balances must be greater than or equal to zero. The calls to method `printf` in lines 13–16 display the account names and balances, obtained by calling each `Account`'s `getName` and `getBalance` methods. For the balances, we call each `BigDecimal` object's `setScale` method to format the balance with two digits to the right of the decimal point using `RoundingMode.HALF_EVEN`, as introduced in Section 4.11.

```
1  // Fig. 8.6: AccountTest.java
2  // Inputting and outputting floating-point numbers with Account objects.
3  import java.math.BigDecimal;
4  import java.math.RoundingMode;
5  import java.util.Scanner;
6
7  public class AccountTest {
8     public static void main(String[] args) {
9        var account1 = new Account("Wei Li", new BigDecimal("50.00"));
10       var account2 = new Account("Logan Brown", new BigDecimal("-7.53"));
11
12       // display initial balance of each object
13       System.out.printf("%s balance: $%s%n", account1.getName(),
14          account1.getBalance().setScale(2, RoundingMode.HALF_EVEN));
15       System.out.printf("%s balance: $%s%n%n", account2.getName(),
16          account2.getBalance().setScale(2, RoundingMode.HALF_EVEN));
17
18       // create a Scanner to obtain input from the user
19       var input = new Scanner(System.in);
20
21       System.out.print("Enter deposit amount for account1: "); // prompt
22       var depositAmount = input.nextBigDecimal(); // get input
23       System.out.printf("%nadding %s to account1 balance%n%n",
24          depositAmount.setScale(2, RoundingMode.HALF_EVEN));
25       account1.deposit(depositAmount); // add to account1 balance
26
27       // display balances
28       System.out.printf("%s balance: $%s%n", account1.getName(),
29          account1.getBalance().setScale(2, RoundingMode.HALF_EVEN));
30       System.out.printf("%s balance: $%s%n%n", account2.getName(),
31          account2.getBalance().setScale(2, RoundingMode.HALF_EVEN));
32
33       System.out.print("Enter deposit amount for account2: "); // prompt
34       depositAmount = input.nextBigDecimal(); // get input
35       System.out.printf("%nadding %s to account2 balance%n%n",
36          depositAmount.setScale(2, RoundingMode.HALF_EVEN));
37       account2.deposit(depositAmount); // add to account2 balance
38
39       // display balances
40       System.out.printf("%s balance: $%s%n", account1.getName(),
41          account1.getBalance().setScale(2, RoundingMode.HALF_EVEN));
```

Fig. 8.6 | Inputting and outputting floating-point numbers with `Account` objects. (Part 1 of 2.)

```
42         System.out.printf("%s balance: $%s%n%n", account2.getName(),
43            account2.getBalance().setScale(2, RoundingMode.HALF_EVEN));
44      }
45   }
```

```
Wei Li balance: $50.00
Logan Brown balance: $0.00

Enter deposit amount for account1: 25.53

adding 25.53 to account1 balance

Wei Li balance: $75.53
Logan Brown balance: $0.00

Enter deposit amount for account2: 123.45

adding 123.45 to account2 balance

Wei Li balance: $75.53
Logan Brown balance: $123.45
```

Fig. 8.6 | Inputting and outputting floating-point numbers with `Account` objects. (Part 2 of 2.)

Displaying the `Account` Objects' Initial Balances

When we display the initial balances in lines 13–16, `account2`'s `balance` is `0.00` because the constructor rejected the attempt to start `account2` with a negative balance. So that object's `balance` retained its default initial value.

Reading a `BigDecimal` Value from the User and Making a Deposit

Line 21 prompts the user to enter a deposit amount for `account1`. Line 22 declares the local variable `depositAmount` to store each deposit amount the user enters. Line 22 also obtains the user input by calling the **`Scanner`'s `nextBigDecimal` method**, which converts the input to a `BigDecimal` and returns it. Lines 23–24 display the `depositAmount`. Then, line 25 calls object `account1`'s `deposit` method with the `depositAmount` as the argument. Method `deposit` ensures that the `depositAmount` is valid and, if so, adds it to the `balance`. Lines 28–31 output the `name`s and `balance`s of both `Account`s again to show that only `account1`'s `balance` has changed.

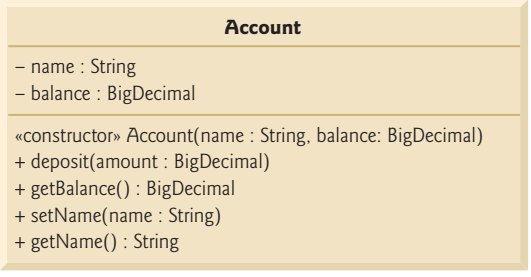
Lines 33–36 prompt for and input `account2`'s deposit amount, then display it. Line 37 calls object `account2`'s `deposit` method with `depositAmount` as the argument to add that value to the `balance`. Finally, lines 40–43 output the `name`s and `balance`s of both `Account`s again to show that only `account2`'s `balance` has changed.

Duplicated Code in Method `main`

The six statements in lines 13–14, 15–16, 28–29, 30–31, 40–41 and 42–43 are almost identical—they each output an `Account`'s `name` and `balance`. They differ only in the name of the `Account` object—`account1` or `account2`. Similarly, lines 21–25 and 33–37 are almost identical for inputting a deposit amount and depositing it into each `Account`. Duplicate code like this can create code maintenance problems when that code needs to be updated. Replacing duplicated code with calls to a method that contains one copy of that code can reduce the size of your program and improve its maintainability.

UML Class Diagram for Class Account

The following is the UML class diagram for Fig. 8.5's `Account` class. The second compartment models the `private` attributes `name` (a `String`) and `balance` (a `BigDecimal`). The third compartment models the constructor and its `name` (a `String`) and `balance` (a `BigDecimal`) parameters. The third compartment also models the class's `public` methods—`deposit` with a `BigDecimal amount` parameter, `getBalance` with a `BigDecimal` return type, `setName` with a `String name` parameter and `getName` with a `String` return type.



Generative AI

1 Prompt genAIs to explain "DRY" (don't repeat yourself) coding to help you avoid duplicate code.

2 Prompt genAIs with the code in Fig. 8.6 asking them to refactor the program to eliminate the code duplication in `main`.

8.6 Case Study: Card Shuffling and Dealing Simulation

Chapter 6's examples demonstrated arrays containing only primitive-type elements. An array's elements can have primitive or reference types. This section uses random-number generation and an array of reference-type elements—namely objects representing playing cards—to develop a class that simulates card shuffling and dealing, which you can use to implement programs that play card games.

This example contains three classes. The `Card` class (Fig. 8.7) represents a playing card with a face (`"Ace"`, `"Deuce"`, `"Three"`, …, `"Jack"`, `"Queen"`, `"King"`) and a suit (`"Hearts"`, `"Diamonds"`, `"Clubs"`, `"Spades"`). The `DeckOfCards` class (Fig. 8.8) creates a deck of 52 playing cards in which each element is a `Card` object. Finally, the `DeckOfCardsTest` class (Fig. 8.9) demonstrates class `DeckOfCards`'s card-shuffling-and-dealing capabilities.

Class Card

Class `Card` (Fig. 8.7) contains `String` instance variables `face` and `suit` that store references to a specific `Card`'s face and suit names. We declared these `final` because a real card's face and suit cannot be modified.[4] The `Card` constructor (lines 9–12) receives two `String`s to initialize `face` and `suit`. Method `toString` (lines 15–17) returns a `String` consisting of the `Card`'s `face`, `" of "` and the `Card`'s `suit` (e.g., `"Ace of Spades"`).[5] Java calls `toString`

4. We'll discuss `final` instance variables in more detail in Section 8.18.
5. You'll learn in Chapter 9 that when we provide a custom `toString` method, we are "overriding" a version "inherited" from class `Object`. As of Chapter 9, every method we explicitly override will be preceded by the annotation `@Override`, which prevents a common programming error.