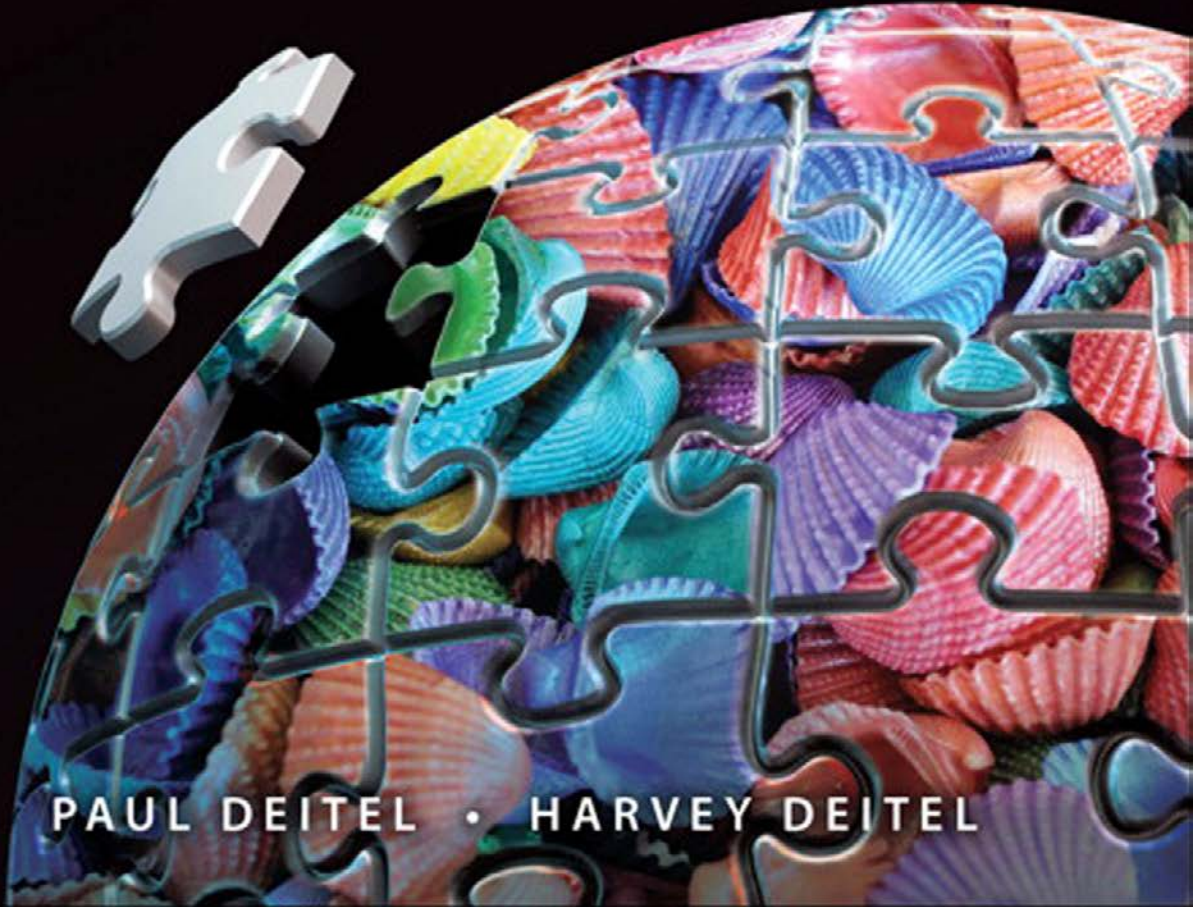
DEITEL® DEVELOPER SERIES

Java® 9 for Programmers

Interactive Java *with* **JShell**

Java Platform Module System (Project Jigsaw)



PAUL DEITEL • HARVEY DEITEL

DEITEL® DEVELOPER SERIES

The DEITEL® DEVELOPER SERIES is designed for professional programmers. The series presents focused treatments on emerging and mature technologies, including Java®, C++, C, C# and .NET, JavaScript®, Internet and web development, Android™ app development, iOS® app development, Swift™ and more. Each book in the series contains the same live-code teaching methodology used in the Deitels' HOW TO PROGRAM SERIES college textbooks—in this book, most concepts are presented in the context of completely coded, live apps.

ABOUT THE COVER

The cover art we selected reflects some key themes of ***Java® 9 for Programmers***: The art includes a colorful collection of sea shells—one of Java 9's key features is **JShell**, which is **Java 9's REPL (read-eval-print-loop) for interactive Java**. JShell is one of the most important pedagogic and productivity enhancements to Java since it was announced in 1995. We discuss JShell in detail in Chapter 23 and show how you can use it for discovery and experimentation, rapid prototyping of code segments and to learn Java faster. The shell art is overlayed onto a sphere of jigsaw puzzle pieces representing the earth. The most important new software-engineering capability in Java 9 is the **Java Platform Module System** (Chapter 27)—which was developed by Project Jigsaw.

Cover graphic: Jigsaw Puzzles, © StacieStauffSmith Photos/ShutterStock
Sea Shells, © Mopic/ShutterStock

DEITEL & ASSOCIATES, INC.

Deitel & Associates, Inc., founded by Paul Deitel and Harvey Deitel, is an internationally recognized authoring and corporate training organization, specializing in computer programming languages, object technology, Internet and web software technology, and Android and iOS app development. The company's clients over the years have included many of the world's largest corporations, government agencies, branches of the military and academic institutions. The company offers instructor-led training courses delivered at client sites worldwide on major programming languages and platforms. Through its 42-year publishing partnership with Prentice Hall/Pearson, Deitel & Associates, Inc., creates leading-edge programming professional books, college textbooks, LiveLessons video products, Learning Paths in the Safari service (`http://www.safaribooksonline.com`), e-books and REVEL™ interactive multimedia courses with integrated labs and assessment (`revel.pearson.com`). To learn more about Deitel & Associates, Inc., its text and video publications and its worldwide instructor-led, on-site training curriculum, visit `www.deitel.com` or send an email to `deitel@deitel.com`. Join the Deitel social networking communities on LinkedIn® (`bit.ly/DeitelLinkedIn`), Facebook® (`www.facebook.com/DeitelFan`), Twitter® (`twitter.com/deitel`), and YouTube™ (`www.youtube.com/DeitelTV`), and subscribe to the ***Deitel® Buzz Online*** newsletter (`www.deitel.com/newsletter/subscribe.html`).

```
1  // Fig. 10.13: Employee.java
2  // Employee abstract superclass that implements Payable.
3
4  public abstract class Employee implements Payable {
5     private final String firstName;
6     private final String lastName;
7     private final String socialSecurityNumber;
8
9     // constructor
10    public Employee(String firstName, String lastName,
11       String socialSecurityNumber) {
12       this.firstName = firstName;
13       this.lastName = lastName;
14       this.socialSecurityNumber = socialSecurityNumber;
15    }
16
17    // return first name
18    public String getFirstName() {return firstName;}
19
20    // return last name
21    public String getLastName() {return lastName;}
22
23    // return social security number
24    public String getSocialSecurityNumber() {return socialSecurityNumber;}
25
26    // return String representation of Employee object
27    @Override
28    public String toString() {
29       return String.format("%s %s%nsocial security number: %s",
30          getFirstName(), getLastName(), getSocialSecurityNumber());
31    }
32
33    // abstract method must be overridden by concrete subclasses
34    public abstract double earnings(); // no implementation here
35
36    // implementing getPaymentAmount here enables the entire Employee
37    // class hierarchy to be used in an app that processes Payables
38    public double getPaymentAmount() {return earnings();}
39 }
```

Fig. 10.13 | `Employee` `abstract` superclass that implements `Payable`.

Subclasses of `Employee` *and Interface* `Payable`

When a class implements an interface, the same *is-a* relationship as inheritance applies. Class `Employee` implements `Payable`, so we can say that an `Employee` *is a* `Payable`, and thus any object of an `Employee` subclass also *is a* `Payable`. So, if we update the class hierarchy in Section 10.5 with the new `Employee` superclass in Fig. 10.13, then `SalariedEmployees`, `HourlyEmployees`, `CommissionEmployees` and `BasePlusCommissionEmployees` are all `Payable` objects. Just as we can assign the reference of a `SalariedEmployee` subclass object to a superclass `Employee` variable, we can assign the reference of a `SalariedEmployee` object (or any other `Employee` derived-class object) to a `Payable` variable. `Invoice` im-

plements `Payable`, so an `Invoice` object also *is a* `Payable` object, and we can assign the reference of an `Invoice` object to a `Payable` variable.

Software Engineering Observation 10.10

Inheritance and interfaces are similar in their implementation of the is-a *relationship. An object of a class that implements an interface may be thought of as an object of that interface type. An object of any subclass of a class that implements an interface also can be thought of as an object of the interface type.*

Software Engineering Observation 10.11

The is-a *relationship that exists between superclasses and subclasses, and between interfaces and the classes that implement them, holds when passing an object to a method. When a method parameter receives an argument of a superclass or interface type, the method polymorphically processes the object received as an argument.*

Software Engineering Observation 10.12

Using a superclass reference, we can polymorphically invoke any method declared in the superclass and its superclasses (e.g., class `Object`*). Using an interface reference, we can polymorphically invoke any method declared in the interface, its superinterfaces (one interface can extend another) and in class* `Object`*—a variable of an interface type must refer to an object to call methods, and all objects have the methods of class* `Object`*.*

10.9.5 Using Interface `Payable` to Process `Invoices` and `Employees` Polymorphically

`PayableInterfaceTest` (Fig. 10.14) illustrates that interface `Payable` can be used to process a set of `Invoices` and `Employees` *polymorphically* in a single application. Lines 7–12 declare and initialize the four-element array `payableObjects`. Lines 8–9 place the references of `Invoice` objects in `payableObjects`' first two elements. Lines 10–11 then place the references of `SalariedEmployee` objects in `payableObjects`' last two elements. The elements are allowed to be initialized with `Invoices` and `SalariedEmployees`, because an `Invoice` *is a* `Payable`, a `SalariedEmployee` *is an* `Employee` and an `Employee` *is a* `Payable`.

```
1  // Fig. 10.14: PayableInterfaceTest.java
2  // Payable interface test program processing Invoices and
3  // Employees polymorphically.
4  public class PayableInterfaceTest {
5     public static void main(String[] args) {
6        // create four-element Payable array
7        Payable[] payableObjects = new Payable[] {
8           new Invoice("01234", "seat", 2, 375.00),
9           new Invoice("56789", "tire", 4, 79.95),
10          new SalariedEmployee("John", "Smith", "111-11-1111", 800.00),
11          new SalariedEmployee("Lisa", "Barnes", "888-88-8888", 1200.00)
12       };
13
```

Fig. 10.14 | `Payable` interface test program processing `Invoices` and `Employees` polymorphically. (Part 1 of 2.)

```
14         System.out.println(
15            "Invoices and Employees processed polymorphically:");
16
17         // generically process each element in array payableObjects
18         for (Payable currentPayable : payableObjects) {
19            // output currentPayable and its appropriate payment amount
20            System.out.printf("%n%s %npayment due: $%,.2f%n",
21               currentPayable.toString(), // could invoke implicitly
22               currentPayable.getPaymentAmount());
23         }
24      }
25   }
```

```
Invoices and Employees processed polymorphically:

invoice:
part number: 01234 (seat)
quantity: 2
price per item: $375.00
payment due: $750.00

invoice:
part number: 56789 (tire)
quantity: 4
price per item: $79.95
payment due: $319.80

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: $800.00
payment due: $800.00

salaried employee: Lisa Barnes
social security number: 888-88-8888
weekly salary: $1,200.00
payment due: $1,200.00
```

Fig. 10.14 | `Payable` interface test program processing `Invoice`s and `Employee`s polymorphically. (Part 2 of 2.)

Lines 18–23 *polymorphically* process each `Payable` object in `payableObjects`, displaying each object's `String` representation and payment amount. Line 21 invokes method `toString` via a `Payable` interface reference, even though `toString` is not declared in interface `Payable`—*all references (including those of interface types) refer to objects that extend `Object` and therefore have a `toString` method.* (Method `toString` also can be invoked *implicitly* here.) Line 22 invokes `Payable` method `getPaymentAmount` to obtain the payment amount for each object in `payableObjects`, regardless of the actual type of the object. The output reveals that each of the method calls in lines 21–22 invokes the appropriate class's `toString` and `getPaymentAmount` methods.

10.9.6 Some Common Interfaces of the Java API

You'll use interfaces extensively when developing Java applications. The Java API contains numerous interfaces, and many of the Java API methods take interface arguments and re-

turn interface values. Figure 10.15 overviews a few of the more popular interfaces of the Java API that we use in later chapters.

Interface	Description
`Comparable`	Java contains several comparison operators (e.g., <, <=, >, >=, ==, !=) that allow you to compare primitive values. However, these operators *cannot* be used to compare objects. Interface `Comparable` is used to allow objects of a class that `implements` the interface to be compared to one another. Interface `Comparable` is commonly used for ordering objects in a collection such as an `ArrayList`. We use `Comparable` in Chapter 16, Generic Collections, and Chapter 19, Generic Classes and Methods: A Deeper Look.
`Serializable`	An interface used to identify classes whose objects can be written to (i.e., serialized) or read from (i.e., deserialized) some type of storage (e.g., file on disk, database field) or transmitted across a network.
`Runnable`	Implemented by any class that represents a task to perform. Objects of such a class are often executed in parallel using a technique called *multithreading* (discussed in Chapter 21, Concurrency and Multi-Core Performance). The interface contains one method, `run`, which specifies the behavior of an object when executed.
GUI event-listener interfaces	You work with graphical user interfaces (GUIs) every day. In your web browser, you might type the address of a website to visit, or you might click a button to return to a previous site. The browser responds to your interaction and performs the desired task. Your interaction is known as an *event*, and the code that the browser uses to respond to an event is known as an *event handler*. In Chapter 12, JavaFX Graphical User Interfaces: Part 1, you'll begin learning how to build GUIs with event handlers that respond to user interactions. Event handlers are declared in classes that implement an appropriate *event-listener interface*. Each event-listener interface specifies one or more methods that must be implemented to respond to user interactions.
`AutoCloseable`	Implemented by classes that can be used with the `try`-with-resources statement (Chapter 11, Exception Handling: A Deeper Look) to help prevent resource leaks. We use this interface in Chapter 15, Files, Input/Output Streams, NIO and XML Serialization, and Chapter 22, Accessing Databases with JDBC.

Fig. 10.15 | Common interfaces of the Java API.

10.10 Java SE 8 Interface Enhancements

8

This section introduces interface features that were added in Java SE 8. We discuss these in more detail in later chapters.

10.10.1 `default` Interface Methods

Prior to Java SE 8, interface methods could be *only* `public abstract` methods. This meant that an interface specified *what* operations an implementing class must perform but not *how* the class should perform them.

As of Java SE 8, interfaces also may contain `public` **`default` methods** with *concrete* default implementations that specify *how* operations are performed when an implementing class does not override the methods. If a class implements such an interface, the class also receives the interface's `default` implementations (if any). To declare a `default` method, place the keyword `default` before the method's return type and provide a concrete method implementation.

Adding Methods to Existing Interfaces

Prior to Java SE 8, adding methods to an interface would break any implementing classes that did not implement the new methods. Recall that if you didn't implement each of an interface's methods, you had to declare your class `abstract`.

Any class that implements the original interface will *not* break when a `default` method is added—the class simply receives the new `default` method. When a class implements a Java SE 8 interface, the class "signs a contract" with the compiler that says, "I will declare all the `abstract` methods specified by the interface or I will declare my class `abstract`"—the implementing class is not required to override the interface's `default` methods, but it can if necessary.

Software Engineering Observation 10.13

Java SE 8 `default` methods enable you to evolve existing interfaces by adding new methods to those interfaces without breaking code that uses them.

Interfaces vs. `abstract` *Classes*

Prior to Java SE 8, an interface was typically used (rather than an `abstract` class) when there were no implementation details to inherit—no fields and no method implementations. With `default` methods, you can instead declare common method implementations in interfaces. This gives you more flexibility in designing your classes, because a class can implement many interfaces, but can extend only one superclass.

10.10.2 `static` Interface Methods

Prior to Java SE 8, it was common to associate with an interface a class containing `static` helper methods for working with objects that implemented the interface. In Chapter 16, you'll learn about class `Collections` which contains many `static` helper methods for working with objects that implement interfaces `Collection`, `List`, `Set` and more. For example, `Collections` method `sort` can sort objects of *any* class that implements interface `List`. With `static` interface methods, such helper methods can now be declared directly in interfaces rather than in separate classes.

10.10.3 Functional Interfaces

As of Java SE 8, any interface containing only one `abstract` method is known as a **functional interface**—these are also called SAM (single `abstract` method) interfaces. There are many such interfaces throughout the Java APIs. Some functional interfaces that you'll use in this book include:

- `ChangeListener` (Chapter 12)—You'll implement this interface to define a method that's called when the interacts with a slider graphical user interface control.

- `Comparator` (Chapter 16)—You'll implement this interface to define a method that can compare two objects of a given type to determine whether the first object is less than, equal to or greater than the second.
- `Runnable` (Chapter 21)—You'll implement this interface to define a task that may be run in parallel with other parts of your program.

Functional interfaces are used extensively with Java's lambda capabilities that we introduce in Chapter 17. Lambdas provide a shorthand notation for implementing functional interfaces.

10.11 Java SE 9 private Interface Methods

As you know, a class's `private` helper methods may be called only by the class's other methods. As of Java SE 9, you can declare helper methods in *interfaces* via **`private` interface methods**. An interface's `private` instance methods can be called directly (i.e., without an object reference) only by the interface's other instance methods. An interface's `private static` methods can be called by any of the interface's instance or `static` methods.

Common Programming Error 10.7

Including the `default` keyword in a `private` interface method's declaration is a compilation error—`default` methods must be `public`.

10.12 private Constructors

In Section 3.4, we mentioned that constructors are normally declared `public`. Sometimes it's useful to declare one or more of a class's constructors as `private`.

Preventing Object Instantiation

You can prevent client code from creating objects of a class by making the class's constructors `private`. For example, consider class `Math`, which contains only `public static` constants and `public static` methods. There's no need to create a `Math` object to use the class's constants and methods, so its constructor is `private`.

Sharing Initialization Code in Constructors

One common use of a `private` constructor is sharing initialization code among a class's other constructors. You can use delegating constructors (introduced in Fig. 8.5) to call the `private` constructor that contains the shared initialization code.

Factory Methods

Another common use of `private` constructors is to force client code to use so-called "factory methods" to create objects. A **factory method** is a `public static` method that creates and initializes an object of a specified type (possibly of the same class), then returns a reference to it. A key benefit of this architecture is that the method's return type can be an interface or a superclass (either `abstract` or concrete).[2]

2. Gamma, Erich et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.