Java® Full Throttle

Presented by
Paul Deitel, CEO, Deitel & Associates, Inc.

Oracle Java Champion

Paul Deitel

- CEO and Chief Technical Officer, Deitel & Associates, Inc.
- Oracle® Java® Champion—
  - "Java Champions are an exclusive group of passionate Java technology and community leaders who are community-nominated and selected under a project sponsored by Oracle. Java Champions get the opportunity to provide feedback, ideas, and direction that will help Oracle grow the Java Platform. … Members come from a broad cross-section of the Java community". https://community.oracle.com/community/java/java-champions
- Graduate of MIT with 40 years of experience
- 25+ years as a Java programmer/instructor/author
- Delivered hundreds of programming courses worldwide to industry, government, military and academic clients

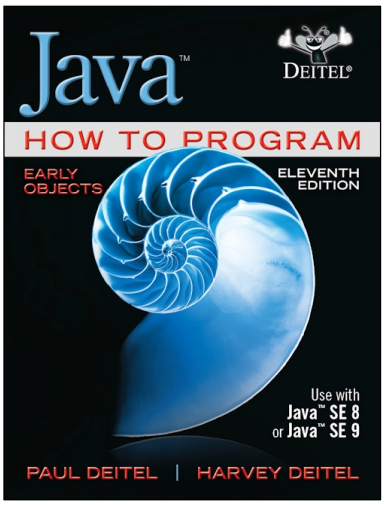
Keeping In Touch

- E-mail questions to paul@deitel.com
- https://deitel.com/contact-us
- LinkedIn: https://linkedin.com/company/deitel-&-associates
- Facebook: https://facebook.com/DeitelFan
- Twitter: https://twitter.com/deitel (@deitel)
- YouTube: https://youtube.com/DeitelTV

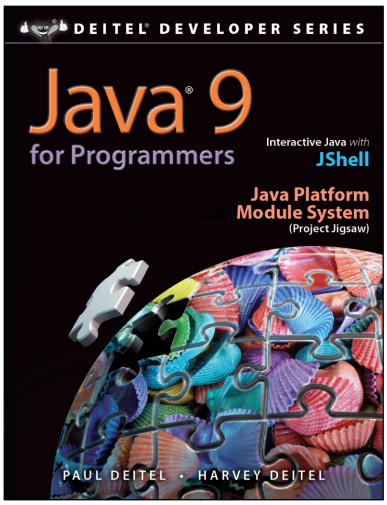
3

Based on Our Books

https://deitel.com/books/JHTP11



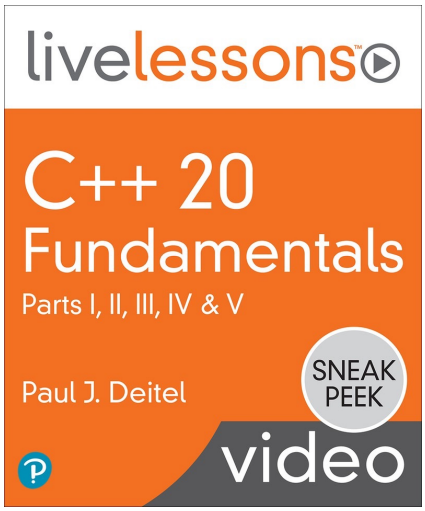
https://deitel.com/books/Java9FP

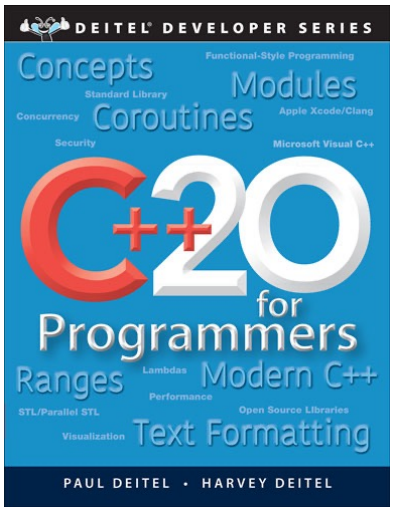


4

New on Oreilly

Early access to Lessons/Chapters 1-9 so far – will be updated post technical review

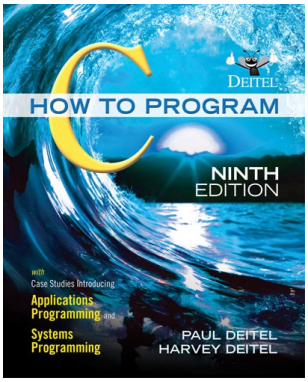




5

C How to Program, 9/e:

with Application Programming and Systems Programming Case Studies



- E-mail paul@deitel.com with questions
- A few key features:
  - Rich coverage of programming fundamentals
  - 20+ case studies from CS, AI, data science & more
  - 350+ self-check exercises with answers
  - Enhanced security/data science per ACM/IEEE
  - Use free open-source libraries and tools
  - gnuplot visualization
  - raylib game programming
  - AI: Machine learning, natural language processing
  - Robotics with the Webots simulator

6

Our Java Books and Videos on O'Reilly

- ***Java 8 and 9 Fundamentals: Modern Java Development with Lambdas, Streams, Introducing Java 9's JShell and JPMS***
  - 54+ hours
  - New videos based on Java 17 coming later this year
  - https://learning.oreilly.com/videos/java-8-fundamentals/9780133489354
- ***Java How to Program, 11/e:*** https://learning.oreilly.com/library/view/java-how-to/9780134751962/
- ***Java 9 for Programmers:*** https://learning.oreilly.com/library/view/java-9-for/9780134778167/

(cont.)

My Upcoming One-Day Full Throttle Webinars https://deitel.com/LearnWithDeitel

- June 8 — Java Full Throttle
- June 15 — Python Full Throttle
- June 22 — Python Data Science Full Throttle
- July 20 — Python Full Throttle
- July 27 — Python Data Science Full Throttle
- August — Java and both Python courses again
- **C++20 Full Throttle** & **C Full Throttle** coming in 2021

Rhythm of the Course

- Whirlwind tour of Java
- Lecture only, source-code focused presentation with some slides
- Six lecture segments
- Two 7-minute breaks in first three hours
- 45-minute meal break
- Two 7-minute breaks in last three hours

Questions

- Submit questions via chat or Q/A windows
- Will attempt to answer questions inline
- For questions I cannot get to, please e-mail me at paul@deitel.com and I'll get back to you quickly after the course

Objectives

- Compiling and running Java apps
- Built-in types, strings, input/output
- Control statements
- Methods
- Arrays
- Classes, objects, instance variables, instance methods, static methods
- Value vs. reference types

Objectives

- Inheritance, Polymorphism, Interfaces
- Exception handling
- JavaFX GUI overview
- Generic collections
- Lambdas, streams, functional interfaces
- Concurrency, threads and parallel streams
- What's New in Java 10 – 15

Chapter 1: Getting Started

For more detail, see the Before You Begin, Dive Into and Test Drive videos at:
http://bit.ly/JavaFundamentals2E

- I am using OpenJDK from
  - https://adoptopenjdk.net/
- Popular IDEs
  - NetBeans (https://netbeans.apache.org/)
  - IntelliJ IDEA (https://www.jetbrains.com/idea/)
  - Eclipse (https://www.eclipse.org/)
- I've provided projects that are compatible with NetBeans and IntelliJ
- Importing into Eclipse:
  - https://stackoverflow.com/questions/21535023/how-to-get-your-netbeans-project-into-eclipse

macOS 10.15+ Quarantined Files

- New digital signing requirements
- May get errors asking you to move files into the trash because the developer cannot be verified
- These issues are being worked on now
- To run apps correctly you can remove quarantine on the files in your JavaFX/IDE folder (replace `/Path/To` with the location of the folders on your system)
  - `xattr -r -d com.apple.quarantine /Path/To/netbeans`
  - `xattr -r -d com.apple.quarantine /Path/To/javafx-sdk-15`

macOS Users: Switching Java Versions

- List the installed versions
  - `/usr/libexec/java_home –V`
- Change Java Version (if necessary, replace 15 with the version to use)
  - `/usr/libexec/java_home -v 15 --exec javac -version`
- Set JAVA_HOME environment variable for current shell
  - `export JAVA_HOME=`/usr/libexec/java_home -v 15``
- Set JAVA_HOME in .bash_profile
  - `export JAVA_HOME="/Library/Java/JavaVirtualMachines/adoptopenjdk-15.jdk/Contents/Home"`

Chapter 2: Intro to Java

For more detail, see Lesson 2:
http://bit.ly/JavaFundamentals2ELesson02

- Basic Java applications
- Command-line I/O
- Escape sequences
- Arithmetic operators
- `if` statement
- Equality and relational operators
- Java allows only `boolean` values in conditions

Arithmetic Operators

Java operation	Operator	Algebraic expression	Java expression
Addition	+	$f + 7$	`f + 7`
Subtraction	-	$p - c$	`p - c`
Multiplication	*	bm	`b * m`
Division	/	x / y or $\frac{x}{y}$ or $x \div y$	`x / y`
Remainder	%	$r \bmod s$	`r % s`

Chapter 3: Intro to Classes

For more detail, see Lesson 3:
http://bit.ly/JavaFundamentals2ELesson03

- Classes
- Instance variables
- Methods
- `public/private`
- Primitive vs. reference types

Primitive Types vs. Reference Types

- Types divided primitive types and reference types
- Primitive types
  - `boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, `double`
  - Portable across all platforms
- All other types are reference types
- Primitive-type instance variables initialized by default
  - `byte`, `char`, `short`, `int`, `long`, `float` and `double` initialized to `0`
  - `boolean` are initialized to `false`

Primitive Types vs. Reference Types

- Reference-type variables
  - Store the location of an object in the computer's memory
  - Refer to objects in the program
  - Instance variables initialized by default to `null`
- A reference to an object is required to invoke an object's methods
- A primitive-type variable does not refer to an object and therefore cannot be used to invoke a method

Primitive Types vs. Reference Types

- No built-in decimal type in Java
- For arbitrary precision math use the `java.math` package's `BigDecimal/BigInteger`
- `BigDecimal` can be used for monetary amounts
- More robust monetary functionality provided by the JavaMoney project: https://javamoney.github.io/

Chapter 4: Control Statements 1

For more detail, see Lesson 4:
http://bit.ly/JavaFundamentals2ELesson04

- `if`
- `if…else`
- Conditional operator
- `while`
- Converting between types
- Compound assignment operators
- Increment and decrement operators

Equality and Relational Operators

Algebraic operator	Java equality or relational operator	Sample Java condition	Meaning of Java condition
Equality operators			
=	==	x == y	x is equal to y
≠	!=	x != y	x is not equal to y
Relational operators			
>	>	x > y	x is greater than y
<	<	x < y	x is less than y
≥	>=	x >= y	x is greater than or equal to y
≤	<=	x <= y	x is less than or equal to y

Conditional operator

- Conditional operator (`?:`)—shorthand `if`…`else`.
- Ternary operator (takes *three* operands)
- Operands and `?:` form a conditional expression

```
System.out.println(
   studentGrade >= 60 ? "Passed" : "Failed");
```

Explicitly and Implicitly Converting Between Primitive Types

- Integer division yields an integer result.
- Cast operator performs explicit conversion (or type cast).
- The value stored in the operand is unchanged.
- Java evaluates only arithmetic expressions in which the operands' types are *identical*.
- Promotion (or implicit conversion) performed on operands.
- In an expression containing values of the types `int` and `double`, the `int` values are promoted to `double` values for use in the expression.

Compound Assignment Operators

Assignment operator	Sample expression	Explanation	Assigns
Assume: `int c = 3, d = 5, e = 4, f = 6, g = 12;`			
`+=`	`c += 7`	`c = c + 7`	10 to c
`-=`	`d -= 4`	`d = d - 4`	1 to d
`*=`	`e *= 5`	`e = e * 5`	20 to e
`/=`	`f /= 3`	`f = f / 3`	2 to f
`%=`	`g %= 9`	`g = g % 9`	3 to g

(cont.)

Increment and Decrement Operators

Operator	Sample expression	Explanation
++ (prefix increment)	`++a`	Increment `a` by 1, then use the new value of `a` in the expression in which `a` resides.
++ (postfix increment)	`a++`	Use the current value of `a` in the expression in which `a` resides, then increment `a` by 1.
-- (prefix decrement)	`--b`	Decrement `b` by 1, then use the new value of `b` in the expression in which `b` resides.
-- (postfix decrement)	`b--`	Use the current value of `b` in the expression in which `b` resides, then decrement `b` by 1.

(cont.)

Chapter 5: Control Statements 2

For more detail, see Lesson 5:
http://bit.ly/JavaFundamentals2ELesson05

- `for`
- `do…while`
- `switch` with integer expressions
- `switch` with string expressions
- `break` and `continue`
- Logical Operators

Chapter 6: Methods

For more detail, see Lesson 6:
http://bit.ly/JavaFundamentals2ELesson06

- `static` class members
- Argument promotion and casting
- Scope
- Method overloading
- `enum` types

`static` Methods, `static` Fields and Class `Math`

- `static` (class) methods apply to the class in which they're declared
- Calling a `static` method
  - *ClassName*`.`*methodName*`(`*arguments*`)`

`static` Methods, `static` Fields and Class `Math`

- Class variables
  - Each object of a class does *not* need its own separate copy
  - All the objects of the class share one copy of a class variable
- Together a class's `static` variables and instance variables are known as its fields

`static` Methods, `static` Fields and Class `Math`

- `Math` fields for commonly used mathematical constants
  - `Math.PI` (3.141592653589793)
  - `Math.E` (2.718281828459045)
- Declared in class `Math` with the modifiers `public`, `final` and `static`
  - `public` allows you to use these fields in your own classes.
  - A field declared with keyword `final` is *constant*—its value cannot change after the field is initialized.

Why is `main` static?

- The JVM attempts to invoke the `main` method of the class you specify—at this point no objects of the class have been created.
- Declaring `main` as `static` allows the JVM to invoke `main` without creating an instance of the class.

Argument Promotion

Type	Valid promotions
`double`	None
`float`	`double`
`long`	`float` or `double`
`int`	`long`, `float` or `double`
`char`	`int`, `long`, `float` or `double`
`short`	`int`, `long`, `float` or `double` (but not `char`)
`byte`	`short`, `int`, `long`, `float` or `double` (but not `char`)
`boolean`	None (`boolean` values are not considered to be numbers in Java)

enum Types

- Declares a set of constants represented by identifiers
- Special kind of class that is introduced by the keyword `enum` and a type name
- Braces delimit an `enum` declaration's body
- Inside the braces is a comma-separated list of `enum` constants, each representing a unique value
- The identifiers in an `enum` must be unique
- Variables of an `enum` type can be assigned only the constants declared in the `enum`

Why Some Constants Are Not Defined as enum Constants

- Java does *not* allow an `int` to be compared to an `enum` constant
- Java does not provide an easy way to convert an `int` value to a particular `enum` constant
- Translating an `int` into an `enum` constant could be done with a `switch` statement
  - Cumbersome and would not improve the readability of the program (thus defeating the purpose of using an `enum`)

Scope of Declarations

- Basic scope rules:
  - parameter—body of the method in which the declaration appears
  - Local variable—from the point at which the declaration appears to the end of that block
  - Local variable in the initialization section of a `for` statement's header—`for` statement's body and the other expressions in the header
  - Method or field's scope — entire body of the class
- *Any* block may contain variable declarations
- If a local variable or parameter has the same name as a field, the field is shadowed until the block terminates

Chapter 7: Arrays and ArrayLists

For more detail, see Lesson 7:
http://bit.ly/JavaFundamentals2ELesson07

- One-dimensional arrays
- Enhanced `for` statement
- Two-dimensional arrays
- Class `Arrays`
- Generic class `ArrayList`

Pass-By-Value vs. Pass-By-Reference

- All arguments passed by value
- A method call can pass two types of values to a method
  - Copies of primitive values
  - Copies of references to objects (not the objects themselves)
- A method can interact with a referenced object by calling its `public` methods using the copy of the object's reference
  - The parameter in the called method and the argument in the calling method refer to the same object in memory

ArrayList

- Generic type
- Dynamically resizable

Chapter 8: Classes–A Deeper Look

For more detail, see Lesson 8:
http://bit.ly/JavaFundamentals2ELesson08

- The `this` reference
- Overloaded constructors
- `enum` types
- `static` class members

Garbage Collection

- Every object uses system resources, such as memory.
- The JVM performs automatic garbage collection to reclaim the *memory* occupied by objects that are no longer used.
  - When there are *no more references* to an object, the object is *eligible* to be collected.
  - Collection typically occurs when the JVM executes its garbage collector, which may not happen for a while, or even at all before a program terminates.
- Memory and other resource leaks still may occur

Garbage Collection

A Note about Class `Object`'s `finalize` Method

- Every class in Java has the methods of class `Object` (package `java.lang`), one of which is method `finalize`.
- Original intent of `finalize` was to allow the garbage collector to perform termination housekeeping on an object just before reclaiming the object's memory—in practice, it mostly caused problems
- *Never* use `finalize`—there's uncertainty as to whether it will ever get called before a program terminates
- Clean up an object when you know you do not need it anymore
  - Will discuss AutoCloseable later

`enum` Types Revisited

- An `enum` class has the following restrictions:
  - `enum` constants are *implicitly* `final`
  - `enum` constants are implicitly `static`
  - Any attempt to create an object of an `enum` type with operator `new` results in a compilation error
- `enum` constants can be used anywhere constants can be used, such as in the `case` labels of `switch` statements and to control enhanced `for` statements

enum Types Revisited

- An `enum` declaration may optionally include other components of traditional classes, such as constructors, fields and methods
- An `enum` constructor can specify any number of parameters and can be overloaded
- For every `enum`, the compiler generates the `static` method `values` that returns an array of the `enum`'s constants
- When an `enum` constant is converted to a `String`, the constant's identifier is used as the `String` representation

Chapter 9: Inheritance

For more detail, see Lesson 9:
http://bit.ly/JavaFundamentals2ELesson09

- We'll cover the following keywords in the context of Chapter 10's examples
  - `extends`
  - `super`
- `protected` class members
- `Object` class
  - https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/lang/Object.html

`protected` Members

- A class's `public` members are accessible wherever the program has a reference to an object of that class or one of its subclasses.
- A class's `private` members are accessible only within the class itself.
- A superclass's `protected` members can be accessed by members of that superclass, by members of its subclasses and by members of other classes in the *same package*
  - `protected` members also have package access.
- All `public` and `protected` superclass members retain their original access modifier when they become members of the subclass.

`protected` Members

- A superclass's `private` members are *hidden* from its subclasses
  - They can be accessed only through the `public` or `protected` methods inherited from the superclass
- Subclass methods can refer to `public` and `protected` members inherited from the superclass simply by using the member names.
- When a subclass method *overrides* an inherited superclass method, the *superclass* version of the method can be accessed from the *subclass* by preceding the superclass method name with keyword `super` and a dot (`.`) separator.

Chapter 10: Polymorphism

For more detail, see Lesson 10:
http://bit.ly/JavaFundamentals2ELesson10

- `extends`
- `super`
- Abstract classes
- Implementing a polymorphic class hierarchy
- Method overriding
- `final` methods and classes
- Interfaces
- Implementing a polymorphic hierarchy with interfaces
- Interface enhancements to support lambdas

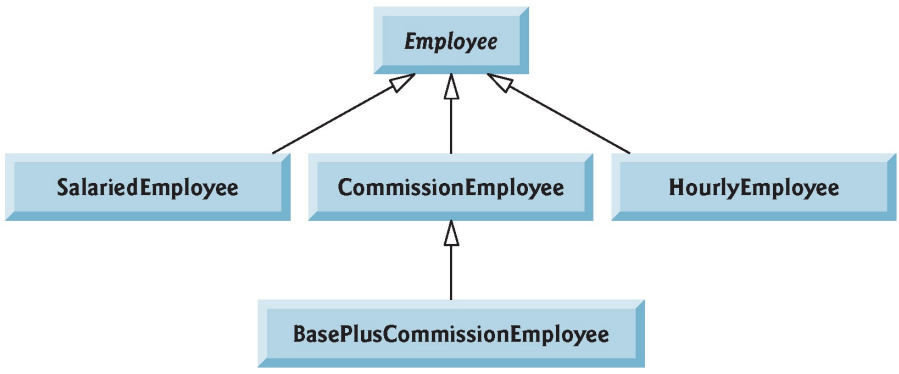


Fig. 10.2 | `Employee` hierarchy UML class diagram.

	earnings	toString
Employee	abstract	*firstName lastName* social security number: *SSN*
Salaried-Employee	weeklySalary	salaried employee: *firstName lastName* social security number: *SSN* weekly salary: *weeklySalary*

Hourly-Employee	if (hours <= 40) {
	  wage * hours
	}
	else if (hours > 40) {
	  40 * wage +
	  (hours - 40) *
	  wage * 1.5
	}
		hourly employee: *firstName lastName*
		social security number: *SSN*
		hourly wage: *wage*; hours worked: *hours*

Commission-Employee	commissionRate * grossSales	commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*
BasePlus-Commission-Employee	(commissionRate * grossSales) + baseSalary	base salaried commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*; base salary: *baseSalary*

Fig. 10.3 | Polymorphic interface for the `Employee` hierarchy classes.

`final` Methods and Classes

- A `final` method in a superclass cannot be overridden in a subclass.
  - Methods that are declared `private` are implicitly `final`, because it's not possible to override them in a subclass.
  - Methods that are declared `static` are implicitly `final`.
  - A `final` method's declaration can never change, so all subclasses use the same method implementation, and calls to `final` methods are resolved at compile time—this is known as static binding.

`final` Methods and Classes

- A `final` class cannot be extended to create a subclass.
  - All methods in a `final` class are implicitly `final`.
- Class `String` is an example of a `final` class.
  - If you were allowed to create a subclass of `String`, objects of that subclass could be used wherever `String`s are expected.
  - Since class `String` cannot be extended, programs that use `String`s can rely on the functionality of `String` objects as specified in the Java API.
  - Making the class `final` also prevents programmers from creating subclasses that might bypass security restrictions.

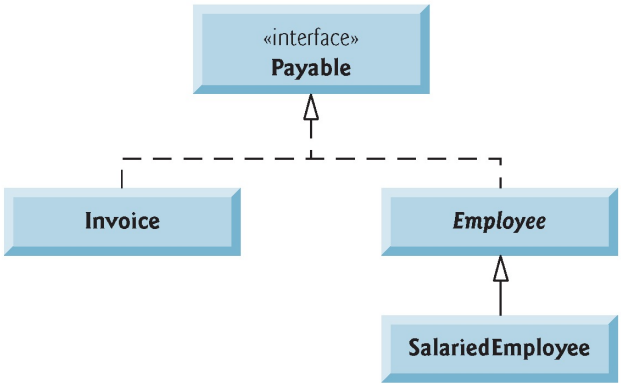


Fig. 10.10 | `Payable` hierarchy UML class diagram.

Common Interfaces

- Java API contains numerous interfaces, and many of the Java API methods take interface arguments and return interface values
- Figure 10.16 overviews a few common interfaces

Common Interfaces

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 20, Generic Classes and Methods: A Deeper Look.

Fig. 10.15 | Common interfaces of the Java API. (Part 1 of 4.)

Common Interfaces

Interface	Description
`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 23, Concurrency). The interface contains one method, `run`, which specifies the behavior of an object when executed.

Fig. 10.15 | Common interfaces of the Java API. (Part 2 of 4.)

Common Interfaces

Interface	Description
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.

Fig. 10.15 | Common interfaces of the Java API. (Part 3 of 4.)

Common Interfaces

Interface	Description
`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 24, Accessing Databases with JDBC.

Fig. 10.15 | Common interfaces of the Java API. (Part 4 of 4.)

Java 8 Interface Enhancements—Default Methods

- Prior to Java 8, *only* `public abstract` methods
- As of Java 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

Java 8 Interface Enhancements—Default Methods

Useful for Evolving Interfaces

- Any class that implements the original interface will *not* break when a `default` method is added
  - The class simply receives the new default method
- The implementing class is not required to override the interface's `default` methods, but it can

Java 8 Interface Enhancements—Default Methods

Interfaces vs. `abstract` Classes

- With `default` methods, you can declare common method implementations in interfaces rather than `abstract` classes
- More flexible hierarchy design—a class can implement many interfaces, but can extend only one superclass

Java 8 Interface Enhancements—Static Methods

- Prior to Java 8, it was common to associate with an interface a class containing `static` helper methods for working with objects that implemented the interface
  - E.g., class `Collections` contains many `static` helper methods for working with objects that implement interfaces `Collection`, `List`, `Set` and more
  - `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

Functional Interfaces

- As of Java 8, any interface containing only one `abstract` method is known as a functional interface—also called a SAM (Single Abstract Method) interface
- Functional interfaces we use in our book include:
  - `ActionListener` (Chapter 12)—define a method that's called when the user clicks a button
  - `Comparator` (Chapter 16)—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 23)—define a task that may be run in parallel with other parts of your program

Java 9 `private` Interface Methods

- 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

`private` Constructors

- Sometimes it's useful to declare one or more of a class's constructors as `private`
- Prevent client code from creating objects of a class
  - Class `Math`, which contains only `public static` constants and `public static` methods
- Sharing initialization code in constructors
  - Use delegating constructors (Chapter 8) to call the `private` constructor that contains the shared initialization code

`private` Constructors

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

Chapter 11: Exception Handling

For more detail, see Lesson 11:
http://bit.ly/JavaFundamentals2ELesson11

- Exceptions
- `try/catch/finally`
- Multi-`catch`
- `throw`
- `throws`
- Exception hierarchy: Checked and unchecked exceptions
- `try`-with-resources and the `AutoCloseable` interface

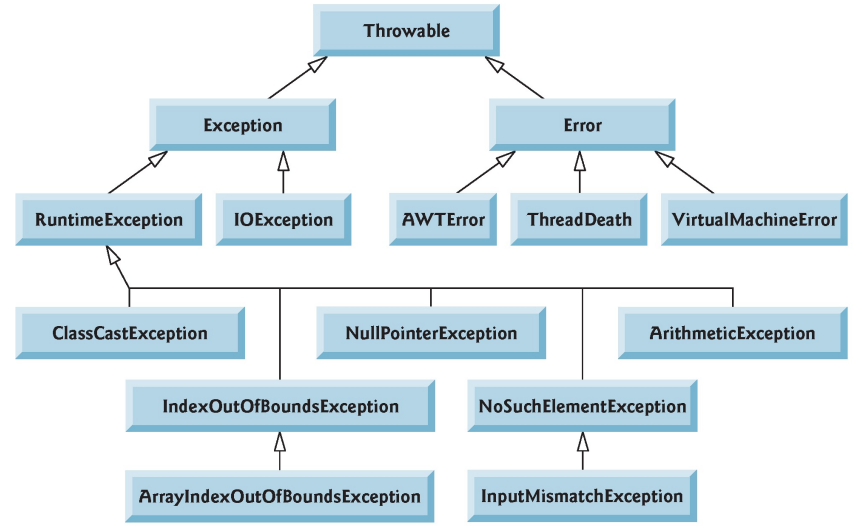


Fig. 11.4 | Portion of class `Throwable`'s inheritance hierarchy.

11.12 `try`-with-Resources: Automatic Resource Deallocation

- Typically resource-release code should be placed in a `finally` block to ensure that a resource is released, regardless of whether there were exceptions when the resource was used in the corresponding try block.
- `try`-with-resources (introduced in Java SE 7)—simplifies writing code in which you obtain one or more resources, use them in a `try` block and release them in a corresponding finally block.
- Each resource must be an object of a class that implements the `AutoCloseable` interface—and thus provides a `close` method.

11.12 `try`-with-Resources: Automatic Resource Deallocation (cont.)

- The general form of a `try`-with-resources statement is

```
try (ClassName theObject = new ClassName()) {
   // use theObject here
}
catch (Exception e) {
   // catch exceptions that occur while using the resource
}
```

- Creates a *ClassName* object, uses it in the `try` block, then calls its `close` method to release any resources used by the object
- Can allocate multiple resources in the parentheses following try by separating them with a semicolon (`;`).

Chapter 12: JavaFX Part 1

For more detail, see Java How to Program, 11/e, Chapter 13:
https://learning.oreilly.com/library/view/java-how-to/9780134751962/

- JavaFX app window structure
- SceneBuilder and FXML
- Application and controller classes
- No longer part of the JDK
  - Instructions for getting started
    - https://openjfx.io/openjfx-docs/#introduction
  - Install the JavaFX SDK and create a PATH_TO_FX environment variable pointing to its `lib` folder
    - Maven/Gradle users can download modules into projects as necessary (described in the instructions at the link above)

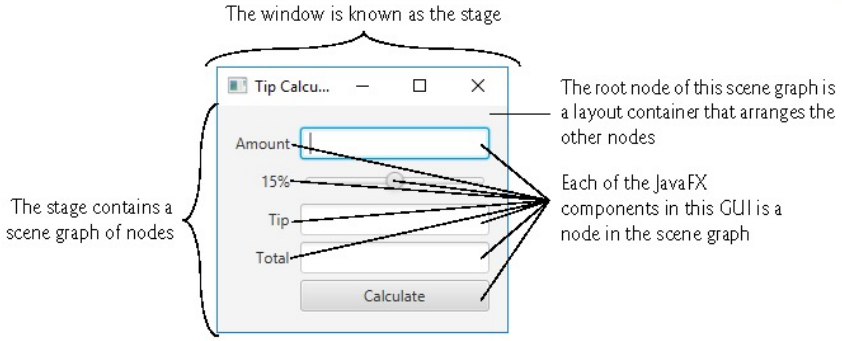


Fig. 12.1 | JavaFX app window parts.

Chapter 13: JavaFX Part 2

For more detail, see Java How to Program, 11/e, Chapter 13:
https://learning.oreilly.com/library/view/java-how-to/9780134751962/

- Java FX property bindings
- Customizing ListViews with custom cell factories.

Chapter 16: Generic collections

For more detail, see Lesson 16:
http://bit.ly/JavaFundamentals2ELesson16

- Generic collections
- Type wrapper classes
- Boxing and unboxing
- `Collections` class algorithms
- Iterators
- Factory methods for immutable `Lists`, `Sets` and `Maps`.

Interface	Description
`Collection`	The root interface in the collections hierarchy from which interfaces `Set`, `Queue` and `List` are derived.
`Set`	A collection that does *not* contain duplicates.
`List`	An ordered collection that *can* contain duplicate elements.
`Map`	A collection that associates keys to values and *cannot* contain duplicate keys. `Map` does not derive from `Collection`.
`Queue`	Typically a *first-in, first-out* collection that models a *waiting line*; other orders can be specified.

Fig. 16.1 | Some collections-framework interfaces.

Type-Wrapper Classes

- Each primitive type has a type-wrapper class (in package `java.lang`).
  - `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long` and `Short`
- Manipulate primitive-type values as objects
  - Store in collections
  - May not be necessary in the future (Project Valhalla)

77

Type-Wrapper Classes

- Numeric type-wrapper classes—`Byte`, `Short`, `Integer`, `Long`, `Float` and `Double`—extend class `Number`
- The type-wrapper classes are `final` classes, so you cannot extend them
- Primitive types do not have methods, so the methods related to a primitive type are located in the corresponding type-wrapper class

78

Autoboxing and Auto-Unboxing

- A boxing conversion converts a value of a primitive type to an object of the corresponding type-wrapper class.
- An unboxing conversion converts an object of a type-wrapper class to a value of the corresponding primitive type

Interface `Collection` and Class `Collections`

- Interface `Collection` contains bulk operations for *adding*, *clearing* and *comparing* objects in a collection
- A `Collection` can be converted to an array.
- Interface `Collection` provides a method that returns an `Iterator` object, which allows a program to walk through the collection and remove elements from the collection during the iteration
- Class `Collections` provides `static` methods that *search*, *sort* and perform other operations on collections

Java 9: Convenience Factory Methods for Immutable Collections

- Java 9 added static *convenience factory methods* to interfaces `List`, `Set` and `Map` for creating small *immutable* collections
- Pass the elements as arguments to a convenience factory method, which creates the collection
- Return custom collection objects that are optimized to store small collections
- Randomized iteration order for `Sets`/`Maps`

Chapter 17: Lambdas and Streams

For more detail, see Lesson 17:
http://bit.ly/JavaFundamentals2ELesson17

- Java's functional programming techniques
- Lambdas
- Streams
- Stream pipelines formed from stream sources, intermediate operations and terminal operations
- Filter/map/reduce operations
- Implement functional interfaces with lambdas
- Lambda event handlers

Lambdas and Streams

- Changes the way you think about Java programming
- Write certain kinds of programs faster, simpler, more concisely and with fewer bugs than with previous techniques
- Such programs can be easier to parallelize so that you can take advantage of multi-core architectures to enhance performance—a key goal of lambdas and streams

External Iteration

- In counter-controlled iteration, you typically determine *what* you want to accomplish then specify precisely *how* to accomplish it using a `for` loop
- *What*: Sum the integers from 1 through 10

```
int total = 0;

for (int number = 1; number <= 10; number++) {
   total += number;
}
```

- *How*: Process each value of `number` from 1 through 10, adding `number`'s current value to `total` once per loop iteration and incrementing number after each addition operation
- Known as external iteration, because you specify all the iteration details

External Iteration is Error Prone

- Requires two variables (`total` and `number`) that the code mutates during each loop iteration
- Mutating leads to possible errors

External Iteration is Error Prone

- Several other opportunities for errors:
  - initialize `total` incorrectly
  - initialize the loop's control variable `number` incorrectly
  - use the wrong loop-continuation condition
  - increment control variable `number` incorrectly
  - incorrectly add each value of `number` to the `total`
- As tasks get more complicated, understanding *how* the code works gets in the way of understanding *what* it does
  - Makes the code harder to read, debug and modify, and more likely to contain errors

Specify *What* Rather Than *How*

- Let's specify *what* to do rather than *how* to do it
- Fig. 17.3: Summing with a stream and reduction
- Declarative programming (specifying *what*) vs. imperative programming (specifying *how*)
- Two simple tasks
  - Producing the numbers in a closed range (1–10)
  - Calculating their sum

Internal Iteration

- Known as internal iteration, because `IntStream` handles all the iteration details—a key aspect of functional programming
- `IntStream` knows *how* to perform each of these tasks
- We did not specify *how* to iterate through the elements or declare and use any mutable variables
- Primary potential for error in line 9 of Fig. 17.3 is specifying the incorrect starting and/or ending values as arguments
- Once you're used to it, stream pipeline code also can be easier to read

Mapping and Lambdas

- Most stream pipelines also contain intermediate operations that specify tasks to perform on a stream's elements before a terminal operation produces a result
- Mapping transforms a stream's elements to new values, producing a stream with the same number of elements
- Mapped elements can have different types from the original stream's elements

Mapping and Lambdas

- External Iteration: Sum the even integers 2 – 20:

```
int total = 0;
for (int number = 2; number <= 20; number += 2) {
   total += number;
}
```

- Figure 17.4 reimplements this with streams and internal iteration

Mapping and Lambdas

- Stream pipeline:
  - Create the data source
  - Map each element (`x`) in the stream to 2 times that element
    - Result is a stream of even integers `2`, `4`, `6`, `8`, `10`, `12`, `14`, `16`, `18`, `20`.
  - Reduce the stream's elements to a single value—the `sum` of the elements
    - Terminal operation that *initiates* the pipeline's processing, then `sums` the stream's elements

Lambda Expressions

- Many intermediate and terminal stream operations receive methods as arguments
- Method `map`'s argument in line 10 is a lambda expression, representing an anonymous method

```
(int x) -> {return x * 2;}
```

- Lambdas are methods that can be treated as data
  - Pass as arguments to other methods (or lambdas)
  - Assign to variables for later use
  - Return from methods

Lambda Syntax

- Parameter list followed by the arrow token (`->`) and a body, as in:
  `(`*parameterList*`) -> {`*statements*`}`
- This lambda receives an `int`, multiplies its value by `2` and returns the result
  ```
  (int x) -> {return x * 2;}
  ```
- The body is a statement block that may contain multiple statements enclosed in curly braces
- Compiler infers return type `int`, because parameter `x` is an `int` and `2` is an `int`—multiplying an `int` by an `int` yields an `int` result
- The preceding lambda is similar to the following method
  ```
  int multiplyBy2(int x) {
     return x * 2;
  }
  ```


Lambda Syntax

- Parameter type(s) may be omitted if they can be inferred
  ```
  (x) -> {return x * 2;}
  ```
- For one-expression body, the `return` keyword, curly braces and semicolon may be omitted
  ```
  (x) -> x * 2
  ```
- For one parameter, parentheses may be omitted
  ```
  x -> x * 2
  ```
- Lambdas can have empty parameter lists
  ```
  () -> System.out.println("Welcome to lambdas!")
  ```
- Method references (later)

Intermediate and Terminal Operations

- `map` is an intermediate operation
- Intermediate operations are lazy
  - Do not perform any operations until a terminal operation is called to produce a result
  - Allows library developers to optimize stream performance
  - E.g., looking for the first `Person` with the last name `"Jones"` in 1,000,000 `Person` objects—rather than processing all 1,000,000, processing terminates on first match
- `sum` is a terminal operation
- Terminal operations are eager
  - Perform the requested operation when they're called

Common intermediate stream operations	
`filter`	Returns a stream containing only the elements that satisfy a condition (known as a *predicate*). The new stream often has fewer elements than the original stream.
`distinct`	Returns a stream containing only the unique elements—duplicates are eliminated.
`limit`	Returns a stream with the specified number of elements from the beginning of the original stream.
`map`	Returns a stream in which each of the original stream's elements is mapped to a new value (possibly of a different type)—for example, mapping numeric values to the squares of the numeric values or mapping numeric grades to letter grades (A, B C, D or F). The new stream has the same number of elements as the original stream.
`sorted`	Returns a stream in which the elements are in sorted order. The new stream has the same number of elements as the original stream. We'll show how to specify both ascending and descending order.

Fig. 17.5 | Common intermediate stream operations.

Common terminal stream operations	
`forEach`	Performs processing on every element in a stream (for example, display each element).
Reduction operations—*Take all values in the stream and return a single value*	
`average`	Returns the *average* of the elements in a numeric stream.
`count`	Returns the *number of elements* in the stream.
`max`	Returns the *maximum* value in a stream.
`min`	Returns the *minimum* value in a stream.
`reduce`	Reduces the elements of a collection to a *single value* using an associative accumulation function (for example, a lambda that adds two elements and returns the sum).

Fig. 17.6 | Common terminal stream operations.

Filtering

- Another common intermediate stream operation is filtering elements to select those that match a condition—known as a predicate
- Select the even integers in the range 1–10, multiply each by 3 and sum the results

```
int total = 0;

for (int x = 1; x <= 10; x++) {
   if (x % 2 == 0) { // if x is even
      total += x * 3;
   }
}
```

- Figure 17.7 reimplements this loop using streams

Filtering

- `IntStream` method `filter` receives as its argument a method that takes one parameter and returns a `boolean` result
- For each element in the stream, `filter` calls the method that it receives as an argument, passing to the method the current stream element
- If the method's return value is `true`, the corresponding element becomes part of the intermediate stream that `filter` returns

How Elements Move Through Stream Pipelines

- Each new stream is simply an object representing the *processing steps* that have been specified to that point in the pipeline
- Intermediate-operation method calls add to the set of processing steps to perform on each stream element
- Last stream object in the pipeline contains all the processing steps to perform on each stream element
- Intermediate operations' processing steps are applied for a given stream element before they are applied to the next stream element

How Elements Move Through Stream Pipelines

- So the stream pipeline in Fig. 17.7 operates as follows:
  - For each element
  - If the element is an even integer
  - Multiply the element by 3 and add the result to the total
- Let's modify the pipeline to see this

Method References

- For a lambda that simply calls another method, you can replace the lambda with that method's name—known as a method reference
  - Compiler converts into an appropriate lambda
- Fig. 17.8 obtains random numbers in the range 1–6 using streams to create the random values and method references to help display the results

Method References

- Bound instance method reference
  - *objectName*`::`*instanceMethodName*
  - Specific object to the left of `::` must be used to call the instance method to the right of `::`
    - `System.out::println`
  - Each stream element is passed to the method
- May also have unbound instance method references
  - *ClassName*`::`*instanceMethodName*
  - Specified method is called on each stream element

Method References

- `static` method reference
  - *ClassName*`::`*staticMethodName*
  - `String::valueOf`
  - Compiler converts `String::valueOf` into a one-parameter lambda that calls `valueOf`, passing the current stream element as an argument

Terminal Operation `collect`

- `Stream` terminal operation `collect` uses a collector to gather the stream's elements into a single object—often a collection
- Similar to a reduction, but returns an object containing the stream's elements
  - `reduce` returns a single value of the stream's element type
- Predefined collector returned by the `static` `Collectors` method `joining`
  - Creates a concatenated `String` representation of the stream's elements separated by the `joining` method's argument

Functional Interfaces

- A functional interface contains exactly one `abstract` method (may contain `default` and `static` methods)
- Functional interfaces are used extensively in functional-style Java programming

Functional Interfaces

- Functional programmers work with pure functions that have referential transparency—they:
  - depend only on their parameters
  - have no side-effects
  - do not maintain any state
- Pure functions are methods that implement functional interfaces—typically defined as lambdas
- State changes occur by passing data from method to method
- No data is shared

Functional Interfaces

- Package `java.util.function` contains several functional interfaces
- Six basic generic functional interfaces, plus many specializations for primitive types and special cases

Interface	Description
`BinaryOperator<T>`	Represents a method that takes two parameters of the same type and returns a value of that type. Performs a task using the parameters (such as a calculation) and returns the result. The lambdas you passed to `IntStream` method `reduce` (Section 17.7) implemented `IntBinaryOperator`—an `int` specific version of `BinaryOperator`.
`Consumer<T>`	Represents a one-parameter method that returns `void`. Performs a task using its parameter, such as outputting the object, invoking a method of the object, etc. The lambda you passed to `IntStream` method `forEach` (Section 17.6) implemented interface `IntConsumer`—an `int`-specialized version of `Consumer`. Later sections present several more examples of `Consumers`.

Fig. 17.10 | The six basic generic functional interfaces in package `java.util.function`.

Interface	Description
`Function<T,R>`	Represents a one-parameter method that performs a task on the parameter and returns a result—possibly of a different type than the parameter. The lambda you passed to `IntStream` method `mapToObj` (Section 17.6) implemented interface `IntFunction`—an `int`-specialized version of `Function`. Later sections present several more examples of `Functions`.
`Predicate<T>`	Represents a one-parameter method that returns a `boolean` result. Determines whether the parameter satisfies a condition. The lambda you passed to `IntStream` method `filter` (Section 17.4) implemented interface `IntPredicate`—an `int`-specialized version of `Predicate`. Later sections present several more examples of `Predicates`.

Fig. 17.10 | The six basic generic functional interfaces in package `java.util.function`.

Interface	Description
`Supplier<T>`	Represents a no-parameter method that returns a result. Often used to create a collection object in which a stream operation's results are placed. You'll see several examples of `Suppliers` starting in Section 17.13.
`UnaryOperator<T>`	Represents a one-parameter method that returns a result of the same type as its parameter. The lambdas you passed in Section 17.3 to `IntStream` method `map` implemented `IntUnaryOperator`—an `int`-specialized version of `UnaryOperator`. Later sections present several more examples of `UnaryOperators`.

Fig. 17.10 | The six basic generic functional interfaces in package `java.util.function`.

Aside: Composing Lambda Expressions

- Many functional interfaces provide `default` methods that enable you to compose functionality
- For example, consider the interface `IntPredicate`, which contains three `default` methods:
  - `and`—performs a logical AND with short-circuit evaluation between the `IntPredicate` on which it's called and the `IntPredicate` it receives as an argument
  - `negate`—reverses the `boolean` value of the `IntPredicate` on which it's called
  - `or`—performs a logical OR with short-circuit evaluation between the `IntPredicate` on which it's called and the `IntPredicate` it receives as an argument
- Use to compose more complex conditions

Aside: Composing Lambda Expressions

- Consider these `IntPredicates`:

```
IntPredicate even = value -> value % 2 == 0;
IntPredicate greaterThan5 = value -> value > 5;
```

- To locate all the even integers greater than 5 in an `IntStream`, you could pass to `IntStream` method `filter`:

```
even.and(greaterThan5)
```

- Like `IntPredicate`, generic functional interface `Predicate` represents a method that returns a `boolean` indicating whether its argument satisfies a condition
  - `Predicate` also contains methods `and` and `or` for combining predicates, and `negate` for reversing a predicate's `boolean` value

Lambda Event Handlers

- Event-listener interfaces with one `abstract` method—like `ChangeListener`—are functional interfaces
- For such interfaces, you can implement event handlers with lambdas
- Consider the following `Slider` event handler from Fig. 12.23:

```
tipPercentageSlider.valueProperty().addListener(
   new ChangeListener<Number>() {
      @Override
      public void changed(ObservableValue<? extends Number> ov,
         Number oldValue, Number newValue) {
         tipPercentage =
            BigDecimal.valueOf(newValue.intValue() / 100.0);
         tipPercentageLabel.setText(percent.format(tipPercentage));
      }
   }
);
```

Lambda Event Handlers (cont.)

- Can be implemented more concisely with a lambda as

```
tipPercentageSlider.valueProperty().addListener(
   (ov, oldValue, newValue) -> {
      tipPercentage =
         BigDecimal.valueOf(newValue.intValue() / 100.0);
      tipPercentageLabel.setText(percent.format(tipPercentage));
   });
```

- For a simple event handler, a lambda significantly reduces the amount of code you need to write

Chapter 23: Concurrent Processing and Parallel Streams

For more detail, see Lesson 20: http://bit.ly/JavaFundamentals2ELesson20

- Creating threads of execution
- `sort` vs. `parallelSort`
- Parallel Streams

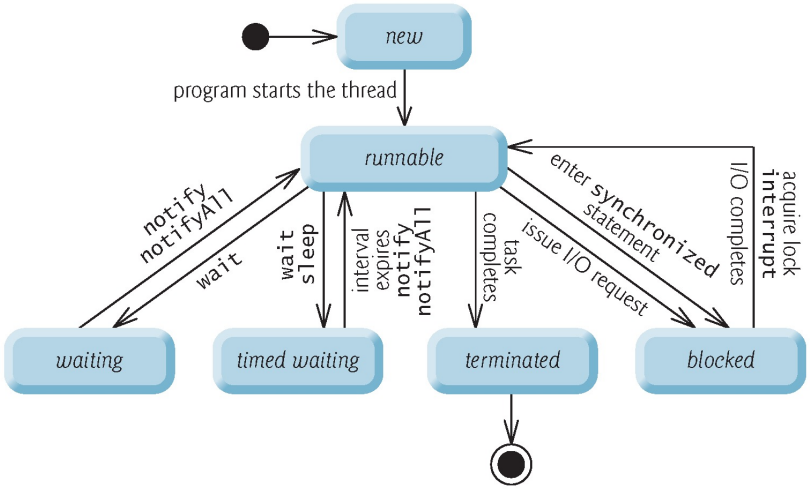


Fig. 23.1 | Thread life-cycle UML state diagram.

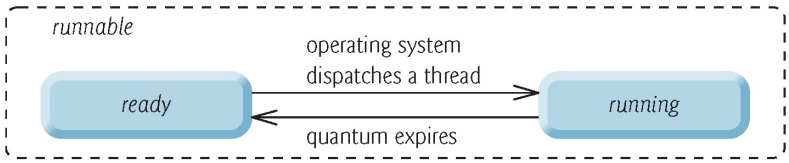


Fig. 23.2 | Operating system's internal view of Java's *runnable* state.

23.3 Creating and Executing Threads with the Executor Framework

- A `Runnable` object represents a "task" that can execute concurrently with other tasks.
- The `Runnable` interface declares the single method `run`, which contains the code that defines the task that a `Runnable` object should perform.
- When a thread executing a `Runnable` is created and started, the thread calls the `Runnable` object's `run` method, which executes in the new thread.

23.3 Creating and Executing Threads with the Executor Framework (cont.)

- Use the `Executor` interface to manage execution of `Runnables`.
- Typically creates and manages a thread pool.
- Can reuse existing threads and can improve performance by optimizing the number of threads.
- Method `execute` accepts a `Runnable` as an argument.
- Assigns every `Runnable` passed to its `execute` method to one of the available threads in the thread pool.
- If there are no available threads, the `Executor` creates a new thread or waits for a thread to become available.

23.3 Creating and Executing Threads with the Executor Framework (cont.)

- The `ExecutorService` interface extends `Executor` and declares methods for managing the life cycle of an `Executor`.
- Create via `static` methods in class `Executors`.
- `newCachedThreadPool` returns an `ExecutorService` that creates new threads as they're needed by the application.
- `shutdown` tells `ExecutorService` to stop accepting new tasks, but continues executing existing ones.

What's New in Java 10 – 16

- New API methods
- Behind the scenes performance and security improvements
- Advanced developer features
- See links in the next several slides

What's New in Java 10

- Local variable type inference with `var`
- Performance improvement features
- JVM can recognize when it's running in a Docker container
- https://dzone.com/articles/whats-new-in-java-10
- https://www.oracle.com/technetwork/java/javase/10-relnote-issues-4108729.html

What's New in Java 11

- JavaFX now developed separately from the JDK
- `var` in lambda parameters
- Run single-source-file applications directly with
  - `java` *filename*`.java`
- New HttpClient
- Java EE modules removed
- https://dzone.com/articles/90-new-features-and-apis-in-jdk-11
- https://dzone.com/articles/90-new-features-and-apis-in-jdk-11-part-2
- https://www.oracle.com/technetwork/java/javase/11-relnote-issues-5012449.html

What's New in Java 12

- https://dzone.com/articles/39-new-features-and-apis-in-jdk-12
- https://www.oracle.com/technetwork/java/javase/12-relnote-issues-5211422.html

What's New in Java 13

- https://dzone.com/articles/81-new-features-and-apis-in-jdk-13

What's New in Java 14

- https://www.azul.com/whats-new-in-jdk14-latest-release/

What's New in Java 15

- Nice summary of language changes on a version by version basis:
  - https://docs.oracle.com/en/java/javase/15/language/java-language-changes.html
- Sealed classes (preview): https://openjdk.java.net/jeps/360
- Records (preview 2): https://openjdk.java.net/jeps/384
- Pattern matching `instanceof` operator (preview 2): https://openjdk.java.net/jeps/375
- Textblocks (final): https://openjdk.java.net/jeps/378

What's New in Java 16

- https://openjdk.java.net/projects/jdk/16/
  - Pattern matching `instanceof` finalized
    - https://openjdk.java.net/jeps/394
  - Preparing for value types so things like `new Integer(7)` will produce warnings
    - https://openjdk.java.net/jeps/390
  - `records` finalized
    - https://openjdk.java.net/jeps/395
  - Sealed classes preview 2 – likely to be finalized in LTS release 17
    - https://openjdk.java.net/jeps/397

What's New in Java 16

- Vector API (Incubator)
  - "vector computations that reliably compile at runtime to optimal vector hardware instructions on supported CPU architectures and thus achieve superior performance to equivalent scalar computations"
  - https://openjdk.java.net/jeps/338

What's New in Java 17 (LTS)

- https://openjdk.java.net/projects/jdk/17/
- Vector API (Incubator)
  - JEP 356: Enhanced Pseudo-Random Number Generators https://openjdk.java.net/jeps/356
  - JEP 398: Deprecate the Applet API for Removal (Finally) https://openjdk.java.net/jeps/398
  - JEP 403: Strongly Encapsulate JDK Internals https://openjdk.java.net/jeps/403
  - JEP 409: Sealed Classes https://openjdk.java.net/jeps/409
  - More will come over the next couple of months

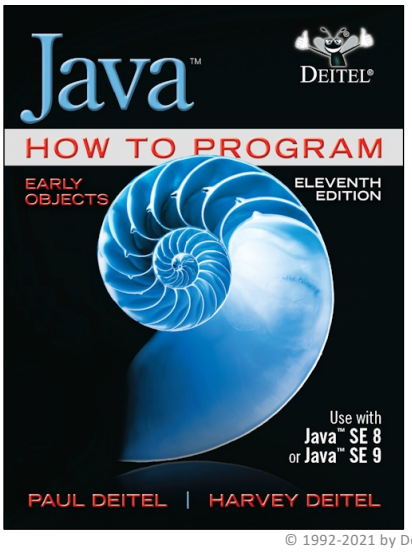
What's New in Java 17 (LTS)

- JEP 406: Pattern Matching for `switch` (Preview)
  - The next step in `switch` expressions
  - https://openjdk.java.net/jeps/406
- Example from the JEP:

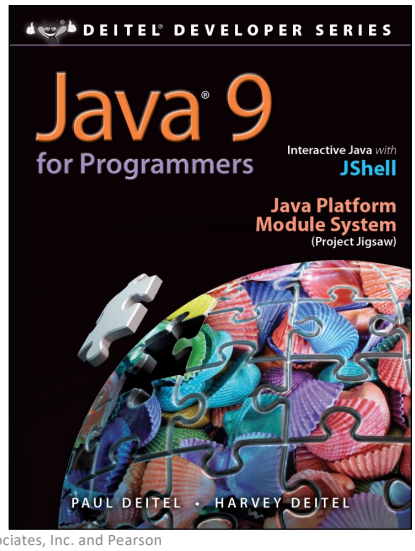
```
static String formatterPatternSwitch(Object o) {
    return switch (o) {
        case Integer i -> String.format("int %d", i);
        case Long l -> String.format("long %d", l);
        case Double d -> String.format("double %f", d);
        case String s -> String.format("String %s", s);
        default -> o.toString();
    };
}
```

Wrap-Up!

https://deitel.com/books/JHTP11



https://deitel.com/books/Java9FP



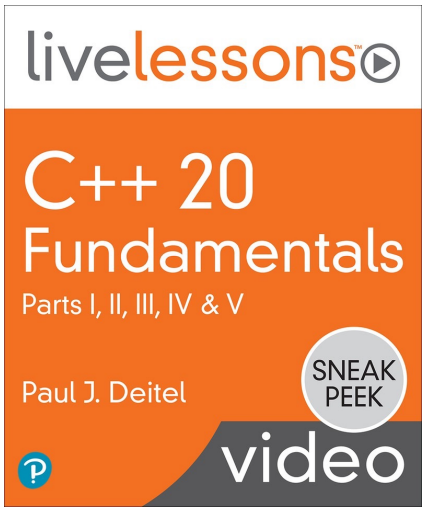
Our Java Books and Videos on O'Reilly

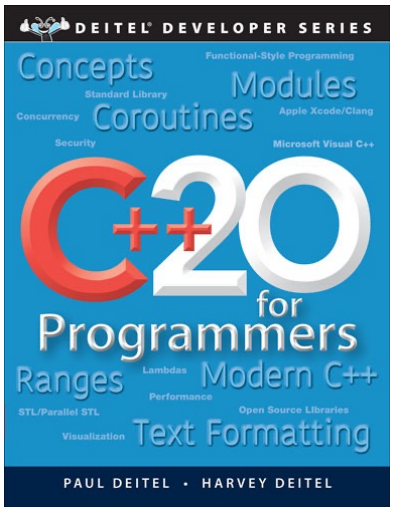
- ***Java 8 and 9 Fundamentals: Modern Java Development with Lambdas, Streams, Introducing Java 9's JShell and JPMS***
  - https://learning.oreilly.com/videos/java-8-fundamentals/9780133489354
- ***Java How to Program, 11/e***
  - https://learning.oreilly.com/library/view/java-how-to/9780134751962/
- ***Java 9 for Programmers***
  - https://learning.oreilly.com/library/view/java-9-for/9780134778167/

(cont.)

New on Oreilly

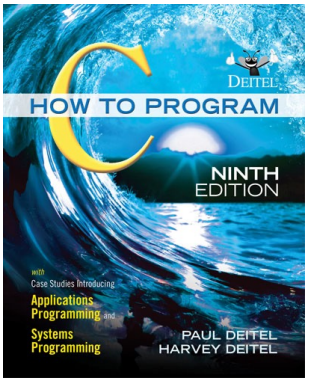
Early access to Lessons/Chapters 1-9 so far – will be updated post technical review





C How to Program, 9/e:

with Application Programming and Systems Programming Case Studies



- E-mail paul@deitel.com with questions
- A few key features:
  - Rich coverage of programming fundamentals
  - 20+ case studies from CS, AI, data science & more
  - 350+ self-check exercises with answers
  - Enhanced security/data science per ACM/IEEE
  - Use free open-source libraries and tools
  - gnuplot visualization
  - raylib game programming
  - AI: Machine learning, natural language processing
  - Robotics with the Webots simulator

My Upcoming One-Day Full Throttle Webinars https://deitel.com/LearnWithDeitel

- June 8 — Java Full Throttle
- June 15 — Python Full Throttle
- June 22 — Python Data Science Full Throttle
- July 20 — Python Full Throttle
- July 27 — Python Data Science Full Throttle
- August — Java and both Python courses again
- **C++20 Full Throttle** & **C Full Throttle** coming in 2021

Additional Resources

- Latest OpenJDK releases
  - https://adoptopenjdk.net/
- NetBeans
  - http://netbeans.apache.org/download/
- IntelliJ IDEA Community Edition
  - https://www.jetbrains.com/idea/download/
- Eclipse IDE
  - https://www.eclipse.org/downloads/

Keeping In Touch

- E-mail questions to paul@deitel.com
- LinkedIn: http://linkedin.com/company/deitel-&-associates
- Facebook: http://facebook.com/DeitelFan
- Twitter: http://twitter.com/deitel (@deitel)
- YouTube: http://youtube.com/DeitelTV