Chapter 11
Exception Handling: A Deeper Look

Java How to Program, 11/e

Questions? E-mail paul.deitel@deitel.com

OBJECTIVES

In this chapter you'll:

- Learn why exception handling is an effective mechanism for responding to runtime problems.
- Use `try` blocks to delimit code in which exceptions might occur.
- Use `throw` to indicate a problem.
- Use `catch` blocks to specify exception handlers.
- Learn when to use exception handling.

OBJECTIVES (cont.)

- Understand the exception class hierarchy.
- Use the `finally` block to release resources.
- Chain exceptions by catching one exception and throwing another.
- Create user-defined exceptions.
- Use the debugging feature `assert` to state conditions that should be true at a particular point in a method.
- Learn how `try`-with-resources can automatically release a resource when the `try` block terminates.

OUTLINE

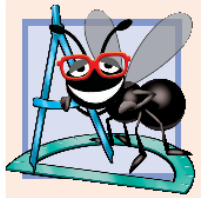
OUTLINE (cont.)

11.1 Introduction

- Exception handling
- Exception—an indication of a problem that occurs during a program's execution.
  - The name "exception" implies that the problem occurs infrequently.
- With exception handling, a program can continue executing (rather than terminating) after dealing with a problem.
  - Mission-critical or business-critical computing.
  - Robust and fault-tolerant programs (i.e., programs that can deal with problems as they arise and continue executing).

11.1 Introduction (Cont.)

- `ArrayIndexOutOfBoundsException` occurs when an attempt is made to access an element past either end of an array.
- `ClassCastException` occurs when an attempt is made to cast an object that does not have an *is-a* relationship with the type specified in the cast operator.
- A `NullPointerException` occurs when a `null` reference is used where an object is expected.
- Only classes that extend `Throwable` (package `java.lang`) directly or indirectly can be used with exception handling.



Software Engineering Observation 11.1

In industry, you're likely to find that companies have strict design, coding, testing, debugging and maintenance standards. These often vary among companies. Companies often have their own exception-handling standards that are sensitive to the type of application, such as real-time systems, high-performance mathematical calculations, big data, network-based distributed systems, etc. This chapter's tips provide observations consistent with these types of industry policies.

Chapter	Sample of exceptions used
Chapter 7	`ArrayIndexOutOfBoundsException`
Chapters 8–10	`IllegalArgumentException`
Chapter 11	`ArithmeticException`, `InputMismatchException`
Chapter 15	`SecurityException`, `FileNotFoundException`, `IOException`, `ClassNotFoundException`, `IllegalStateException`, `FormatterClosedException`, `NoSuchElementException`
Chapter 16	`ClassCastException`, `UnsupportedOperationException`, `NullPointerException`, custom exception types
Chapter 20	`ClassCastException`, custom exception types

Fig. 11.1 | Various exception types that you'll see throughout this book

Chapter	Sample of exceptions used
Chapter 21	`IllegalArgumentException`, custom exception types
Chapter 23	`InterruptedException`, `IllegalMonitorStateException`, `ExecutionException`, `CancellationException`
Chapter 24	`SQLException`, `IllegalStateException`, `PatternSyntaxException`
Chapter 28	`MalformedURLException`, `EOFException`, `SocketException`, `InterruptedException`, `UnknownHostException`
Chapter 31	`SQLException`

Fig. 11.1 | Various exception types that you'll see throughout this book

11.2 Example: Divide by Zero without Exception Handling

- Exceptions are thrown (i.e., the exception occurs) by a method detects a problem and is unable to handle it.
- Stack trace—information displayed when an exception occurs and is not handled.
- Information includes:
  - The name of the exception in a descriptive message that indicates the problem that occurred
  - The method-call stack (i.e., the call chain) at the time it occurred. Represents the path of execution that led to the exception method by method.
- This information helps you debug the program.

11.2 Example: Divide by Zero without Exception Handling (Cont.)

- Java does not allow division by zero in integer arithmetic.
  - Throws an `ArithmeticException`.
  - Can arise from a several problems, so an error message (e.g., "`/ by zero`") provides more specific information.
- Java *does* allow division by zero with floating-point values.
  - Such a calculation results in the value positive or negative infinity
  - Floating-point value that displays as `Infinity` or `-Infinity`.
  - If 0.0 is divided by 0.0, the result is NaN (not a number), which is represented as a floating-point value that displays as `NaN`.

```
1  // Fig. 11.2: DivideByZeroNoExceptionHandling.java
2  // Integer division without exception handling.
3  import java.util.Scanner;
4
5  public class DivideByZeroNoExceptionHandling {
6     // demonstrates throwing an exception when a divide-by-zero occurs
7     public static int quotient(int numerator, int denominator) {
8        return numerator / denominator; // possible division by zero
9     }
10
11    public static void main(String[] args) {
12       Scanner scanner = new Scanner(System.in);
13
14       System.out.print("Please enter an integer numerator: ");
15       int numerator = scanner.nextInt();
```

Fig. 11.2 | Integer division without exception handling. (Part 1 of 3.)

```
16          System.out.print("Please enter an integer denominator: ");
17          int denominator = scanner.nextInt();
18
19          int result = quotient(numerator, denominator);
20          System.out.printf(
21             "%nResult: %d / %d = %d%n", numerator, denominator, result);
22       }
23    }
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

Fig. 11.2 | Integer division without exception handling. (Part 2 of 3.)

```
Please enter an integer numerator: 100
Please enter an integer denominator: 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at DivideByZeroNoExceptionHandling.quotient(
           DivideByZeroNoExceptionHandling.java:8)
        at DivideByZeroNoExceptionHandling.main(
           DivideByZeroNoExceptionHandling.java:19)
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: hello
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Unknown Source)
        at java.util.Scanner.next(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at DivideByZeroNoExceptionHandling.main(
            DivideByZeroNoExceptionHandling.java:17)
```

Fig. 11.2 | Integer division without exception handling. (Part 3 of 3.)

11.2 Example: Divide by Zero without Exception Handling (Cont.)

- Prior examples that input numeric values assumed that the user would input a proper integer value.
- Users sometimes make mistakes and input noninteger values.
- An `InputMismatchException` occurs when `Scanner` method `nextInt` receives a `String` that does not represent a valid integer.
- If a stack trace contains "`Unknown Source`" for a particular method, the debugging symbols for that method's class were not available to the JVM—this is typically the case for the classes of the Java API.

11.2 Example: Divide by Zero without Exception Handling (Cont.)

- Last line of the stack trace started the call chain.
- Each line contains the class name and method followed by the filename and line number.
- The top row of the call chain indicates the throw point—the initial point at which the exception occurred.

11.3 Example: Handling `ArithmeticException`s and `InputMismatchException`s

- The application in Fig. 11.2 uses exception handling to process any `ArithmeticException`s and `InputMistmatchException`s that arise.
- If the user makes a mistake, the program catches and handles (i.e., deals with) the exception—in this case, allowing the user to try to enter the input.

```
1  // Fig. 11.3: DivideByZeroWithExceptionHandling.java
2  // Handling ArithmeticExceptions and InputMismatchExceptions.
3  import java.util.InputMismatchException;
4  import java.util.Scanner;
5
6  public class DivideByZeroWithExceptionHandling
7  {
8     // demonstrates throwing an exception when a divide-by-zero occurs
9     public static int quotient(int numerator, int denominator)
10       throws ArithmeticException {
11       return numerator / denominator; // possible division by zero
12    }
13
```

Fig. 11.3 | Handling `ArithmeticExceptions` and `InputMismatchExceptions`. (Part 1 of 5.)

```
14      public static void main(String[] args) {
15         Scanner scanner = new Scanner(System.in);
16         boolean continueLoop = true; // determines if more input is needed
17
18         do {
19            try { // read two numbers and calculate quotient
20               System.out.print("Please enter an integer numerator: ");
21               int numerator = scanner.nextInt();
22               System.out.print("Please enter an integer denominator: ");
23               int denominator = scanner.nextInt();
24
25               int result = quotient(numerator, denominator);
26               System.out.printf("%nResult: %d / %d = %d%n", numerator,
27                  denominator, result);
28               continueLoop = false; // input successful; end looping
29            }
```

Fig. 11.3 | Handling `ArithmeticExceptions` and `InputMismatchExceptions`. (Part 2 of 5.)

```
30            catch (InputMismatchException inputMismatchException) {
31               System.err.printf("%nException: %s%n",
32                  inputMismatchException);
33               scanner.nextLine(); // discard input so user can try again
34               System.out.printf(
35                  "You must enter integers. Please try again.%n%n");
36            }
37            catch (ArithmeticException arithmeticException) {
38               System.err.printf("%nException: %s%n", arithmeticException);
39               System.out.printf(
40                  "Zero is an invalid denominator. Please try again.%n%n");
41            }
42         } while (continueLoop);
43      }
44   }
```

Fig. 11.3 | Handling `ArithmeticExceptions` and `InputMismatchExceptions`. (Part 3 of 5.)

```
Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 0

Exception: java.lang.ArithmeticException: / by zero
Zero is an invalid denominator. Please try again.

Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

Fig. 11.3 | Handling `ArithmeticExceptions` and `InputMismatchExceptions`. (Part 4 of 5.)

```
Please enter an integer numerator: 100
Please enter an integer denominator: hello

Exception: java.util.InputMismatchException
You must enter integers. Please try again.

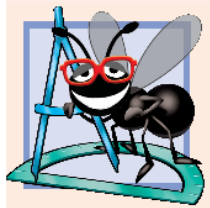
Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

Fig. 11.3 | Handling `ArithmeticExceptions` and `InputMismatchExceptions`. (Part 5 of 5.)

11.3 Example: Handling ArithmeticExceptions and InputMismatchExceptions (Cont.)

- `try` block encloses
  - code that might `throw` an exception
  - code that should not execute if an exception occurs.
- Consists of the keyword `try` followed by a block of code enclosed in curly braces.



Software Engineering Observation 11.2

Exceptions may surface through explicitly mentioned code in a `try` block, through deeply nested method calls initiated by code in a `try` block or from the Java Virtual Machine as it executes Java bytecodes.

11.3 Example: Handling `ArithmeticExceptions` and `InputMismatchExceptions` (Cont.)

- `catch` block (also called a `catch` clause or exception handler) *catches* and *handles* an exception.
  - Begins with the keyword `catch` followed by an exception parameter in parentheses and a block of code enclosed in curly braces.
- At least one `catch` block or a `finally` block (Section 11.6) *must* immediately follow the `try` block.
- The exception parameter identifies the exception type the handler can process.
  - The parameter's name enables the `catch` block to interact with a caught exception object.

11.3 Example: Handling `ArithmeticExceptions` and `InputMismatchExceptions` (Cont.)

- When an exception occurs in a `try` block, the `catch` block that executes is the first one whose type matches the type of the exception that occurred.
- Use the `System.err` (standard error stream) object to output error messages.
  - By default, displays data to the *command prompt*.

Common Programming Error 11.1

It's a syntax error to place code between a `try` block and its corresponding `catch` blocks.

11.3 Example: Handling `ArithmeticException`s and `InputMismatchException`s (Cont.)

Multi-`catch`

- If the bodies of several `catch` blocks are identical, you can use the multi-`catch` feature (introduced in Java SE 7) to catch those exception types in a *single* `catch` handler and perform the same task.
- The syntax for a *multi-`catch`* is:
  - `catch (`*Type1* `|` *Type2* `|` *Type3* `e)`
- Each exception type is separated from the next with a vertical bar (|).
- The preceding line of code indicates that *any* of the types (or their subclasses) can be caught in the exception handler.
- Any number of `Throwable` types can be specified in a multi-catch.

11.3 Example: Handling `ArithmeticException`s and `InputMismatchException`s (Cont.)

- Uncaught exception—one for which there are no matching `catch` blocks.
- Recall that previous uncaught exceptions caused the application to terminate early.
  - This does not always occur as a result of uncaught exceptions.
- Java uses a multithreaded model of program execution.
  - Each thread is a *concurrent activity*.
  - One program can have many threads.
  - If a program has only *one* thread, an uncaught exception will cause the program to terminate.
  - If a program has multiple threads, an uncaught exception will terminate *only* the thread in which the exception occurred.

11.3 Example: Handling `ArithmeticExceptions` and `InputMismatchExceptions` (Cont.)

- If an exception occurs in a `try` block, the `try` block terminates immediately and program control transfers to the first matching `catch` block.
- After the exception is handled, control resumes after the last `catch` block.
- Known as the termination model of exception handling.
  - Some languages use the resumption model of exception handling, in which, after an exception is handled, control resumes just after the throw point.

11.3 Example: Handling `ArithmeticException`s and `InputMismatchException`s (Cont.)

- If no exceptions are thrown in a `try` block, the `catch` blocks are *skipped* and control continues with the first statement after the `catch` blocks
  - We'll learn about another possibility when we discuss the `finally` block in Section 11.6.
- The `try` block and its corresponding `catch` and/or `finally` blocks form a `try` statement.

11.3 Example: Handling `ArithmeticException`s and `InputMismatchException`s (Cont.)

- When a `try` block terminates, local variables declared in the block go out of scope.
  - The *local variables* of a `try` block are not accessible in the corresponding `catch` blocks.
- When a `catch` block *terminates*, *local variables* declared within the `catch` block (including the exception parameter) also *go out of scope*.
- Any remaining `catch` blocks in the `try` statement are *ignored*, and execution resumes at the first line of code after the `try`…`catch` sequence.
  - A `finally` block, if one is present.

11.3 Example: Handling ArithmeticExceptions and InputMismatchExceptions (Cont.)

- `throws` clause—specifies the exceptions a method *might* throw if problems occur.
  - Must appear after the method's parameter list and before the body.
  - Contains a comma-separated list of the exception types.
    - May be thrown by statements in the method's body or by methods called from there.
  - Clients of a method with a `throws` clause are thus informed that the method might throw exceptions.

Error-Prevention Tip 11.1

Read a method's online API documentation before using it in a program. The documentation specifies exceptions thrown by the method (if any) and indicates reasons why such exceptions may occur. Next, read the online API documentation for the specified exception classes. The documentation for an exception class typically contains potential reasons that such exceptions occur. Finally, provide for handling those exceptions in your program.

11.3 Example: Handling ArithmeticExceptions and InputMismatchExceptions (Cont.)

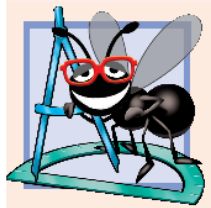
- When a method throws an exception, the method terminates and does not return a value, and its *local variables go out of scope*.
  - If the local variables were references to objects and there were no other references to those objects, the objects would be available for *garbage collection*.

11.4 When to Use Exception Handling

- Exception handling is designed to process synchronous errors, which occur when a statement executes.
- Common examples in this book:
  - out-of-range array indices
  - arithmetic overflow
  - division by zero
  - invalid method parameters
  - thread interruption

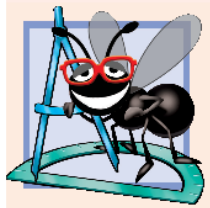
11.4 When to Use Exception Handling (Cont.)

- Exception handling is not designed to process problems associated with asynchronous events
  - disk I/O completions
  - network message arrivals
  - mouse clicks and keystrokes



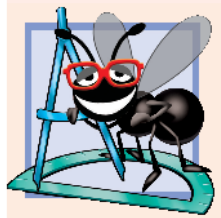
Software Engineering Observation 11.3

Incorporate your exception-handling and error-recovery strategy into your system from the inception of the design process—including these after a system has been implemented can be difficult.



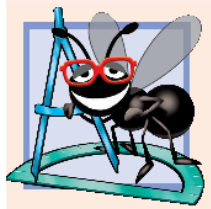
Software Engineering Observation 11.4

Exception handling provides a single, uniform technique for documenting, detecting and recovering from errors. This helps programmers working on large projects understand each other's error-processing code.



Software Engineering Observation 11.5

A great variety of situations can generate exceptions—some exceptions are easier to recover from than others.



Software Engineering Observation 11.6

Sometimes you can prevent an exception by validating data first. For example, before you perform integer division, you can ensure that the denominator is not zero, which prevents the `ArithmeticException` that occurs when you divide by zero.

11.5 Java Exception Hierarchy

- Exception classes inherit directly or indirectly from class `Exception`, forming an *inheritance hierarchy*.
  - Can extend this hierarchy with your own exception classes.
- Figure 11.3 shows a small portion of the inheritance hierarchy for class `Throwable` (a subclass of `Object`), which is the superclass of class `Exception`.
  - Only `Throwable` objects can be used with the exception-handling mechanism.
- Class `Throwable` has two subclasses: `Exception` and `Error`.

11.5 Java Exception Hierarchy (Cont.)

- Class `Exception` and its subclasses represent exceptional situations that can occur in a Java program
  - These can be caught and handled by the application.
- Class `Error` and its subclasses represent abnormal situations that happen in the JVM.
  - `Errors` happen infrequently.
  - These should not be caught by applications.
  - Applications usually cannot recover from `Errors`.

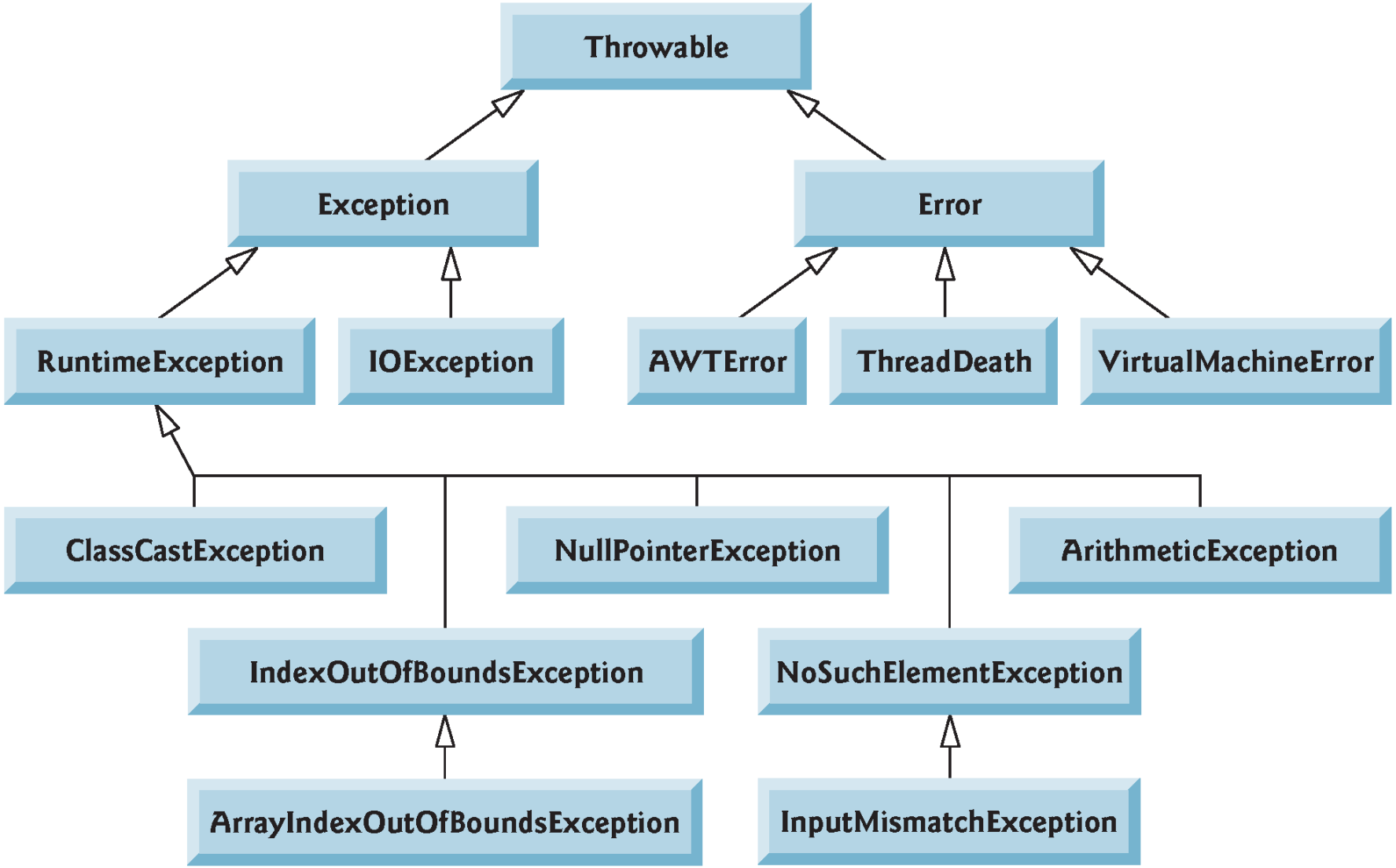


Fig. 11.4 | Portion of class `Throwable`'s inheritance hierarchy.

11.5 Java Exception Hierarchy (Cont.)

- Checked exceptions vs. unchecked exceptions.
  - Compiler enforces a catch-or-declare requirement for *checked* exceptions.
- An exception's type determines whether it is checked or unchecked.
- Direct or indirect subclasses of class `RuntimeException` (package `java.lang`) are *unchecked* exceptions.
- Typically caused by defects in your program's code, e.g.:
- `ArrayIndexOutOfBoundsExceptions`
- `ArithmeticExceptions`
- Subclasses of `Exception` but not `RuntimeException` are *checked* exceptions.
  - Caused by conditions that are not in the control of the program—e.g., in file processing, the program can't open a file if it does not exist.

11.5 Java Exception Hierarchy (Cont.)

- The compiler *checks* each method call and method declaration to determine whether the method throws a checked exception.
  - If so, the compiler verifies that the checked exception is *caught* or is *declared* in a `throws` clause—this is known as the `catch-or-declare` requirement.
- `throws` clause specifies the exceptions a method throws.
  - Such exceptions are typically not caught in the method's body.

11.5 Java Exception Hierarchy (Cont.)

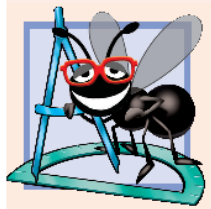
- To satisfy the *catch* part of the *catch-or-declare requirement*, the code that generates the exception must be wrapped in a `try` block and must provide a `catch` handler for the checked-exception type (or one of its superclasses).
- To satisfy the *declare* part of the *catch-or-declare requirement*, the method must provide a `throws` clause containing the checked-exception type after its parameter list and before its method body.
- If the catch-or-declare requirement is not satisfied, the compiler will issue an error message.

Error-Prevention Tip 11.2

You must deal with checked exceptions. This results in more robust code than would be created if you were able to simply ignore them.

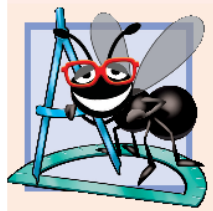
Common Programming Error 11.2

If a subclass method overrides a superclass method, it's an error for the subclass method to list more exceptions in its `throws` clause than the superclass method does. However, a subclass's `throws` clause can contain a subset of a superclass's `throws` clause.



Software Engineering Observation 11.7

If your method calls other methods that throw checked exceptions, those exceptions must be caught or declared. If an exception can be handled meaningfully in a method, the method should catch the exception rather than declare it.

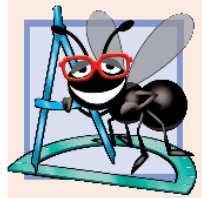


Software Engineering Observation 11.8

Checked exceptions represent problems from which programs often can recover, so programmers are required to deal with them.

11.5 Java Exception Hierarchy (Cont.)

- The compiler does *not* examine the code to determine whether an unchecked exception is caught or declared.
  - These typically can be *prevented* by proper coding.
  - For example, an `ArithmeticException` can be avoided if a method ensures that the denominator is not zero *before* performing.
- Unchecked exceptions are not required to be listed in a method's `throws` clause.
  - Even if they are, it's not required that such exceptions be caught by an application.



Software Engineering Observation 11.9

Although the compiler does not enforce the `catch`-or-declare requirement for unchecked exceptions, provide appropriate exception-handling code when it's known that such exceptions might occur. For example, a program should process the `NumberFormatException` from `Integer` method `parseInt`, even though `NumberFormatException` is an indirect subclass of `RuntimeException` (and thus an unchecked exception). This makes your programs more robust.

11.5 Java Exception Hierarchy (Cont.)

- If a `catch` handler is written to catch *superclass* exception objects, it can also catch all objects of that class's *subclasses*.
- This enables catch to handle related exceptions *polymorphically*.
- You can catch each subclass individually if those exceptions require different processing.

11.5 Java Exception Hierarchy (Cont.)

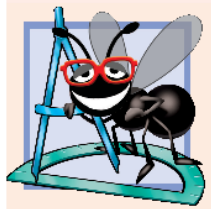
- If *multiple* `catch` blocks match a particular exception type, only the *first* matching `catch` block executes.
- It's a compilation error to catch the *exact same type* in two different `catch` blocks associated with a particular `try` block.

Common Programming Error 11.3

Placing a `catch` block for a superclass exception type before other `catch` blocks that catch subclass exception types would prevent those `catch` blocks from executing, so a compilation error occurs.

Error-Prevention Tip 11.3

Catching subclass types individually is subject to error if you forget to test for one or more of the subclass types explicitly; catching the superclass guarantees that objects of all subclasses will be caught. Positioning a `catch` block for the superclass type after all other subclass `catch` blocks ensures that all subclass exceptions are eventually caught.



Software Engineering Observation 11.10

In industry, throwing or catching type `Exception` is discouraged—we use it in this chapter simply to demonstrate exception-handling mechanics. In subsequent chapters, we generally throw and catch more specific exception types.

11.6 `finally` Block

- Programs that obtain certain resources must return them to the system to avoid so-called resource leaks.
  - In programming languages such as C and C++, the most common resource leak is a *memory leak*.
  - Java automatically garbage collects memory no longer used by programs, thus avoiding most memory leaks.
  - Other types of resource leaks can occur.
    - Files, database connections and network connections that are not closed properly might not be available for use in other programs.
- The `finally` block (which consists of the `finally` keyword, followed by code enclosed in curly braces), sometimes referred to as the `finally` clause, is optional.

Error-Prevention Tip 11.4

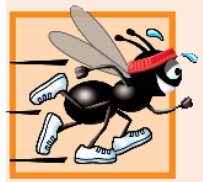
A subtle issue is that Java does not entirely eliminate memory leaks. Java will not garbage-collect an object until there are no remaining references to it. Thus, if you erroneously keep references to unwanted objects, memory leaks can occur.

11.6 `finally` Block (Cont.)

- `finally` block will execute *whether or not* an exception is thrown in the corresponding `try` block.
- `finally` block will execute if a `try` block exits by using a `return`, `break` or `continue` statement or simply by reaching its closing right brace.
- `finally` block will *not* execute if the application *exits early* from a `try` block by calling method `System.exit`.

Error-Prevention Tip 11.5

The `finally` block is an ideal place to release resources acquired in a `try` block (such as opened files), which helps eliminate resource leaks.



Performance Tip 11.1

Always release a resource explicitly and at the earliest possible moment at which it's no longer needed. This makes resources available for reuse as early as possible, thus improving resource utilization and program performance.

11.6 `finally` Block (Cont.)

- If an exception that occurs in a `try` block cannot be caught by one of that `try` block's `catch` handlers, control proceeds to the `finally` block.
- Then the program passes the exception to the next outer `try` block—normally in the calling method—where an associated `catch` block might catch it.
  - This process can occur through many levels of `try` blocks.
  - The exception could go *uncaught*.
- If a `catch` block throws an exception, the `finally` block still executes.
  - Then the exception is passed to the next outer `try` block—again, normally in the calling method.

11.6 `finally` Block (Cont.)

- Because a `finally` block always executes, it typically contains *resource-release code*.
- Suppose a resource is allocated in a `try` block.
  - If no exception occurs, control proceeds to the `finally` block, which frees the resource. Control then proceeds to the first statement after the `finally` block.
  - If an exception occurs, the `try` block *terminates*. The program catches and processes the exception in one of the corresponding `catch` blocks, then the `finally` block *releases the resource* and control proceeds to the first statement after the `finally` block.
  - If the program doesn't catch the exception, the `finally` block *still* releases the resource and an attempt is made to catch the exception in a calling method.

```
1  // Fig. 11.5: UsingExceptions.java
2  // try...catch...finally exception handling mechanism.
3
4  public class UsingExceptions {
5     public static void main(String[] args) {
6        try {
7           throwException();
8        }
9        catch (Exception exception) { // exception thrown by throwException
10          System.err.println("Exception handled in main");
11       }
12
13       doesNotThrowException();
14    }
15
```

Fig. 11.5 | `try...catch...finally` exception-handling mechanism. (Part 1 of 4.)

```
16      // demonstrate try...catch...finally
17      public static void throwException() throws Exception {
18         try { // throw an exception and immediately catch it
19            System.out.println("Method throwException");
20            throw new Exception(); // generate exception
21         }
22         catch (Exception exception) { // catch exception thrown in try
23            System.err.println(
24               "Exception handled in method throwException");
25            throw exception; // rethrow for further processing
26
27            // code here would not be reached; would cause compilation errors
28
29         }
30         finally { // executes regardless of what occurs in try...catch
31            System.err.println("Finally executed in throwException");
32         }
33
34         // code here would not be reached; would cause compilation errors
35      }
```

Fig. 11.5 | `try...catch...finally` exception-handling mechanism. (Part 2 of 4.)

```
36
37       // demonstrate finally when no exception occurs
38       public static void doesNotThrowException() {
39          try { // try block does not throw an exception
40             System.out.println("Method doesNotThrowException");
41          }
42          catch (Exception exception) { // does not execute
43             System.err.println(exception);
44          }
45          finally { // executes regardless of what occurs in try...catch
46             System.err.println("Finally executed in doesNotThrowException");
47          }
48
49          System.out.println("End of method doesNotThrowException");
50       }
51    }
```

Fig. 11.5 | `try...catch...finally` exception-handling mechanism. (Part 3 of 4.)

```
Method throwException
Exception handled in method throwException
Finally executed in throwException
Exception handled in main
Method doesNotThrowException
Finally executed in doesNotThrowException
End of method doesNotThrowException
```

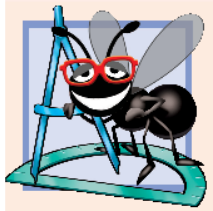
Fig. 11.5 | `try...catch...finally` exception-handling mechanism. (Part 4 of 4.)

11.6 `finally` Block (Cont.)

- `System.out` and `System.err` are streams—a sequence of bytes.
  - `System.out` (the standard output stream) displays output
  - `System.err` (the standard error stream) displays errors
- Output from these streams can be redirected (e.g., to a file).
- Using two different streams enables you to easily separate error messages from other output.
  - Data output from `System.err` could be sent to a log file
  - Data output from `System.out` can be displayed on the screen

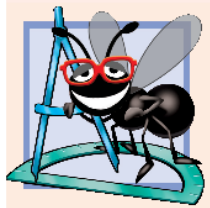
11.6 `finally` Block (Cont.)

- `throw` statement—indicates that an exception has occurred.
  - Used to throw exceptions.
  - Indicates to client code that an error has occurred.
  - Specifies an object to be thrown.
  - The operand of a `throw` can be of any class derived from class `Throwable`.



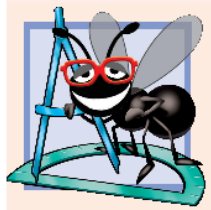
Software Engineering Observation 11.11

When `toString` is invoked on any `Throwable` object, its resulting `String` includes the descriptive `string` that was supplied to the constructor, or simply the class name if no `string` was supplied.



Software Engineering Observation 11.12

An exception can be thrown without containing information about the problem that occurred. In this case, simply knowing that an exception of a particular type occurred may provide sufficient information for the handler to process the problem correctly.



Software Engineering Observation 11.13

Throw exceptions from constructors to indicate that the constructor parameters are not valid—this prevents an object from being created in an invalid state.

11.6 `finally` Block (Cont.)

- Rethrow an exception
  - Done when a `catch` block, cannot process that exception or can only partially process it.
  - Defers the exception handling (or perhaps a portion of it) to another `catch` block associated with an outer `try` statement.
- Rethrow by using the `throw` keyword, followed by a reference to the exception object that was just caught.
- When a rethrow occurs, the *next enclosing `try` block* detects the exception, and that `try` block's `catch` blocks attempt to handle it.

Common Programming Error 11.4

If an exception has not been caught when control enters a `finally` block and the `finally` block throws an exception that's not caught in the `finally` block, the first exception will be lost and the exception from the `finally` block will be returned to the calling method.

Error-Prevention Tip 11.6

Avoid placing in a `finally` block code that can `throw` an exception. If such code is required, enclose the code in a `try`…`catch` within the `finally` block.

Common Programming Error 11.5

Assuming that an exception thrown from a `catch` block will be processed by that `catch` block or any other `catch` block associated with the same `try` statement can lead to logic errors.

Good Programming Practice 11.1

Exception handling removes error-processing code from the main line of a program's code to improve program clarity. Do not place `try...catch... finally` around every statement that may throw an exception. This decreases readability. Rather, place one `try` block around a significant portion of your code, follow the `try` with `catch` blocks that handle each possible exception and follow the `catch` blocks with a single `finally` block (if one is required).

11.7 Stack Unwinding and Obtaining Information from an Exception Object

- Stack unwinding—When an exception is thrown but not caught in a particular scope, the method-call stack is "unwound"
- An attempt is made to `catch` the exception in the next outer `try` block.
- All local variables in the unwound method go out of scope and control returns to the statement that originally invoked that method.
- If a `try` block encloses that statement, an attempt is made to `catch` the exception.
- If a `try` block does not enclose that statement or if the exception is not caught, stack unwinding occurs again.

```
1  // Fig. 11.6: UsingExceptions.java
2  // Stack unwinding and obtaining data from an exception object.
3
4  public class UsingExceptions {
5     public static void main(String[] args) {
6        try {
7           method1();
8        }
9        catch (Exception exception) { // catch exception thrown in method1
10          System.err.printf("%s%n%n", exception.getMessage());
11          exception.printStackTrace();
12
13          // obtain the stack-trace information
14          StackTraceElement[] traceElements = exception.getStackTrace();
15
```

Fig. 11.6 | Stack unwinding and obtaining data from an exception object. (Part 1 of 4.)

```
16            System.out.printf("%nStack trace from getStackTrace:%n");
17            System.out.println("Class\t\tFile\t\t\tLine\tMethod");
18
19            // loop through traceElements to get exception description
20            for (StackTraceElement element : traceElements) {
21               System.out.printf("%s\t", element.getClassName());
22               System.out.printf("%s\t", element.getFileName());
23               System.out.printf("%s\t", element.getLineNumber());
24               System.out.printf("%s%n", element.getMethodName());
25            }
26         }
27      }
28
```

Fig. 11.6 | Stack unwinding and obtaining data from an exception object. (Part 2 of 4.)

```
29      // call method2; throw exceptions back to main
30      public static void method1() throws Exception {
31         method2();
32      }
33
34      // call method3; throw exceptions back to method1
35      public static void method2() throws Exception {
36         method3();
37      }
38
39      // throw Exception back to method2
40      public static void method3() throws Exception {
41         throw new Exception("Exception thrown in method3");
42      }
43   }
```

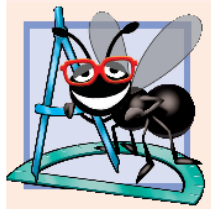
Fig. 11.6 | Stack unwinding and obtaining data from an exception object. (Part 3 of 4.)

```
Exception thrown in method3

java.lang.Exception: Exception thrown in method3
        at UsingExceptions.method3(UsingExceptions.java:41)
        at UsingExceptions.method2(UsingExceptions.java:36)
        at UsingExceptions.method1(UsingExceptions.java:31)
        at UsingExceptions.main(UsingExceptions.java:7)

Stack trace from getStackTrace:
Class            File                         Line    Method
UsingExceptions UsingExceptions.java    41      method3
UsingExceptions UsingExceptions.java    36      method2
UsingExceptions UsingExceptions.java    31      method1
UsingExceptions UsingExceptions.java    7       main
```

Fig. 11.6 | Stack unwinding and obtaining data from an exception object. (Part 4 of 4.)



Software Engineering Observation 11.14

Occasionally, you might want to ignore an exception by writing a `catch` handler with an empty body. Before doing so, ensure that the exception doesn't indicate a condition that code higher up the stack might want to know about or recover from.

11.7 Stack Unwinding and Obtaining Information from an Exception Object (cont.)

- `Throwable` methods `printStackTrace` and `getStackTrace` each process the entire method-call stack
- When debugging, this can be inefficient
- you may be interested only in stack frames corresponding to methods of a specific class
- Java SE 9 introduces the Stack-Walking API (class `StackWalker` in package java.lang), which uses lambdas and streams (Chapter 17) to access method-call-stack information in a more efficient manner
- Learn more about this API at: http://openjdk.java.net/jeps/259

11.8 Chained Exceptions

- Sometimes a method responds to an exception by throwing a different exception type that is specific to the current application.
- If a `catch` block throws a new exception, the original exception's information and stack trace are *lost*.
- Earlier Java versions provided no mechanism to wrap the original exception information with the new exception's information.
  - This made debugging such problems particularly difficult.
- Chained exceptions enable an exception object to maintain the complete stack-trace information from the original exception.
- For any chained exception, you can get the `Throwable` that initially caused that exception by calling `Throwable` method `getCause`.

```
1  // Fig. 11.7: UsingChainedExceptions.java
2  // Chained exceptions.
3
4  public class UsingChainedExceptions {
5     public static void main(String[] args) {
6        try {
7           method1();
8        }
9        catch (Exception exception) { // exceptions thrown from method1
10          exception.printStackTrace();
11       }
12    }
13
```

Fig. 11.7 | Chained exceptions. (Part 1 of 3.)

```
14      // call method2; throw exceptions back to main
15      public static void method1() throws Exception {
16         try {
17            method2();
18         }
19         catch (Exception exception) { // exception thrown from method2
20            throw new Exception("Exception thrown in method1", exception);
21         }
22      }
23
24      // call method3; throw exceptions back to method1
25      public static void method2() throws Exception {
26         try {
27            method3();
28         }
29         catch (Exception exception) { // exception thrown from method3
30            throw new Exception("Exception thrown in method2", exception);
31         }
32      }
```

Fig. 11.7 | Chained exceptions. (Part 2 of 3.)

```
33
34      // throw Exception back to method2
35      public static void method3() throws Exception {
36         throw new Exception("Exception thrown in method3");
37      }
38   }
```

```
java.lang.Exception: Exception thrown in method1
        at UsingChainedExceptions.method1(UsingChainedExceptions.java:17)
        at UsingChainedExceptions.main(UsingChainedExceptions.java:7)
Caused by: java.lang.Exception: Exception thrown in method2
        at UsingChainedExceptions.method2(UsingChainedExceptions.java:27)
        at UsingChainedExceptions.method1(UsingChainedExceptions.java:17)
        ... 1 more
Caused by: java.lang.Exception: Exception thrown in method3
        at UsingChainedExceptions.method3(UsingChainedExceptions.java:36)
        at UsingChainedExceptions.method2(UsingChainedExceptions.java:27)
        ... 2 more
```

Fig. 11.7 | Chained exceptions. (Part 3 of 3.)

11.9 Declaring New Exception Types

- Sometimes it's useful to declare your own exception classes that are specific to the problems that can occur when another programmer uses your reusable classes.
- A new exception class must extend an existing exception class to ensure that the class can be used with the exception-handling mechanism.

11.9 Declaring New Exception Types (cont.)

- A typical new exception class contains only four constructors:
  - one that takes no arguments and passes a default error message `String` to the superclass constructor;
  - one that receives a customized error message as a `String` and passes it to the superclass constructor;
  - one that receives a customized error message as a `String` and a `Throwable` (for chaining exceptions) and passes both to the superclass constructor;
  - and one that receives a `Throwable` (for chaining exceptions) and passes it to the superclass constructor.

Good Programming Practice 11.2

Associating each type of serious execution-time malfunction with an appropriately named `Exception` class improves program clarity.

Software Engineering Observation 11.15

Most programmers will not need to declare their own exception classes. Before defining your own, study the existing ones in the Java API and try to choose one that already exists. If there is not an appropriate existing class, try to extend a related exception class. For example, if you're creating a new class to represent when a method attempts a division by zero, you might extend class `ArithmeticException` because division by zero occurs during arithmetic. If the existing classes are not appropriate superclasses for your new exception class, decide whether your new class should be a checked or an unchecked exception class. If clients should be required to handle the exception, the new exception class should be a checked exception (i.e., extend `Exception` but not `RuntimeException`). The client application should be able to reasonably recover from such an exception. If the client code should be able to ignore the exception (i.e., the exception is an unchecked one), the new exception class should extend `RuntimeException`.

Good Programming Practice 11.3

By convention, all exception-class names should end with the word `Exception`.

11.10 Preconditions and Postconditions

- Programmers spend significant amounts of time maintaining and debugging code.
- To facilitate these tasks and to improve the overall design, they can specify the expected states before and after a method's execution.
- These states are called preconditions and postconditions, respectively.

11.10 Preconditions and Postconditions (Cont.)

- A precondition must be true when a method is invoked.
  - Describes constraints on method parameters and any other expectations the method has about the current state of a program just before it begins executing.
  - If the preconditions are not met, the method's behavior is undefined.
  - You should never expect consistent behavior if the preconditions are not satisfied.

11.10 Preconditions and Postconditions (Cont.)

- A postcondition is true after the method successfully returns.
  - Describes constraints on the return value and any other side effects the method may have.
  - When calling a method, you may assume that a method fulfills all of its postconditions.
  - If writing your own method, document all postconditions so that others know what to expect when they call your method, and you should make certain that your method honors all its postconditions if its preconditions are met.
- When preconditions or postconditions are not met, methods typically throw exceptions.

11.10 Preconditions and Postconditions (Cont.)

- As an example, examine `String` method `charAt`, which has one `int` parameter—an index in the `String`.
  - For a precondition, method `charAt` assumes that `index` is greater than or equal to zero and less than the length of the `String`.
  - If the precondition is met, the postcondition states that the method will return the character at the position in the `String` specified by the parameter `index`.
  - Otherwise, the method throws an `Index-Out-Of-Bounds-Exception`.
  - We trust that method `charAt` satisfies its postcondition, provided that we meet the precondition.
  - We need not be concerned with the details of how the method actually retrieves the character at the index.

11.10 Preconditions and Postconditions (Cont.)

- Some programmers state preconditions and postconditions informally as part of the general method specification, while others prefer a more formal approach by explicitly defining them.
- State the preconditions and postconditions in a comment before the method declaration.
- Stating the preconditions and postconditions before writing a method will also help guide you as you implement the method.

11.11 Assertions

- When implementing and debugging a class, it's sometimes useful to state conditions that should be true at a particular point in a method.
- Assertions help ensure a program's validity by catching potential bugs and identifying possible logic errors during development.
- Preconditions and postconditions are two types of assertions.

11.11 Assertions (Cont.)

- Java includes two versions of the `assert` statement for validating assertions programatically.
- `assert` evaluates a `boolean` expression and, if `false`, throws an `AssertionError` (a subclass of `Error`).

  assert *expression*;
  - throws an `AssertionError` if *expression is `false`.*

  assert *expression1 : expression2;*
  - evaluates *expression1* and throws an `AssertionError` with *expression2* as the error message if expression1 is `false`.
- Can be used to programmatically implement preconditions and postconditions or to verify any other *intermediate* states that help you ensure your code is working correctly.

11.11 Assertions (Cont.)

- You use assertions primarily for debugging and identifying logic errors in an application.
- You must explicitly enable assertions when executing a program
  - They reduce performance.
  - They are unnecessary for the program's user.
- To enable assertions, use the `java` command's `-ea` command-line option, as in

```
java -ea AssertTest
```

```
1  // Fig. 11.8: AssertTest.java
2  // Checking with assert that a value is within range
3  import java.util.Scanner;
4
5  public class AssertTest {
6     public static void main(String[] args) {
7        Scanner input = new Scanner(System.in);
8
9        System.out.print("Enter a number between 0 and 10: ");
10       int number = input.nextInt();
11
12       // assert that the value is >= 0 and <= 10
13       assert (number >= 0 && number <= 10) : "bad number: " + number;
14
15       System.out.printf("You entered %d%n", number);
16    }
17 }
```

Fig. 11.8 | Checking with `assert` that a value is within range. (Part 1 of 2.)

```
Enter a number between 0 and 10: 5
You entered 5
```

```
Enter a number between 0 and 10: 50
Exception in thread "main" java.lang.AssertionError: bad number: 50
        at AssertTest.main(AssertTest.java:13)
```

Fig. 11.8 | Checking with `assert` that a value is within range. (Part 2 of 2.)

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.
- An alternative notation—the `try`-with-resources statement (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.

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

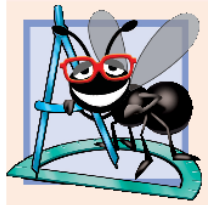
- For example, a file-processing application could process a file with a `try`-with-resources statement to ensure that the file is closed properly when it's no longer needed.
- Each resource must be an object of a class that implements the `AutoCloseable` interface—and thus provides a `close` method.

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
}
```

- *ClassName* is a class that implements the `AutoCloseable` interface.



Software Engineering Observation 11.16

Users shouldn't encounter `AssertionErrors`—these should be used only during program development. For this reason, you shouldn't catch `AssertionErrors`. Instead, allow the program to terminate, so you can see the error message, then locate and fix the source of the problem. You should not use `assert` to indicate runtime problems in production code (as we did in Fig. 11.8 for demonstration purposes)—use the exception mechanism for this purpose.

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

- This code creates an object of type *ClassName* and uses it in the `try` block, then calls its `close` method to release any resources used by the object.
- The `try`-with-resources statement *implicitly* calls the Object's `close` method *at the end of the `try` block*.
- You can allocate multiple resources in the parentheses following try by separating them with a semicolon (`;`).

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

- Java 8 introduced effectively `final` local variables
- If the compiler can infer that the variable could have been declared `final`, because its enclosing method never modifies the variable after it's declared and initialized, then the variable is effectively `final`
  - Frequently are used with lambdas (Chapter 17, Lambdas and Streams).
- As of Java SE 9, you can create an `AutoCloseable` object and assign it to a local variable that's explicitly declared `final` or that's effectively `final`
- Then, you can use it in a `try`-with-resources statement that releases the object's resources at the end of the `try` block.

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

- ```
  ClassName theObject = new ClassName();
  try (theObject) {
     // use theObject here, then release its resources at
     // the end of the try block
  }
  catch (Exception e) {
     // catch exceptions that occur while using the resource
  }
  ```
- As before, you can separate with a semicolon (;) multiple `AutoCloseable` objects in the parentheses following try
  ```