Chapter 11
Exception Handling: A Deeper Look

Objectives

- 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.
- 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.

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.

11.2 Example: Divide by Zero without Exception Handling

- Exceptions are thrown (i.e., the exception occurs) when 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.

```
1  // Fig. 11.1: DivideByZeroNoExceptionHandling.java
2  // Integer division without exception handling.
3  import java.util.Scanner;
4
5  public class DivideByZeroNoExceptionHandling
6  {
7     // demonstrates throwing an exception when a divide-by-zero occurs
8     public static int quotient( int numerator, int denominator )
9     {
10       return numerator / denominator; // possible division by zero
11    } // end method quotient
12
```

Fig. 11.1 | Integer division without exception handling. (Part 1 of 3.)

```
13      public static void main( String[] args )
14      {
15         Scanner scanner = new Scanner( System.in ); // scanner for input
16
17         System.out.print( "Please enter an integer numerator: " );
18         int numerator = scanner.nextInt();
19         System.out.print( "Please enter an integer denominator: " );
20         int denominator = scanner.nextInt();
21
22         int result = quotient( numerator, denominator );
23         System.out.printf(
24            "\nResult: %d / %d = %d\n", numerator, denominator, result );
25      } // end main
26   } // end class DivideByZeroNoExceptionHandling
```

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

Result: 100 / 7 = 14
```

Fig. 11.1 | 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:10)
        at DivideByZeroNoExceptionHandling.main(
           DivideByZeroNoExceptionHandling.java:22)
```

```
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:20)
```

Fig. 11.1 | Integer division without exception handling. (Part 3 of 3.)

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

- The top "at" line of the call chain indicates the throw point—the initial point at which the exception occurs.
- As you read a stack trace top to bottom, the first "at" line that contains your class name and method name is typically the point in the program that led to the exception.
- An InputMismatchException occurs when `Scanner` method `nextInt` receives a `String` that does not represent a valid integer.

```
1  // Fig. 11.2: 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    {
12       return numerator / denominator; // possible division by zero
13    } // end method quotient
14
15    public static void main( String[] args )
16    {
17       Scanner scanner = new Scanner( System.in ); // scanner for input
18       boolean continueLoop = true; // determines if more input is needed
19
```

Fig. 11.2 | Handling `ArithmeticExceptions` and `InputMismatchExceptions`. (Part 1 of 4.)

```
20          do
21          {
22             try // read two numbers and calculate quotient
23             {
24                System.out.print( "Please enter an integer numerator: " );
25                int numerator = scanner.nextInt();
26                System.out.print( "Please enter an integer denominator: " );
27                int denominator = scanner.nextInt();
28
29                int result = quotient( numerator, denominator );
30                System.out.printf( "\nResult: %d / %d = %d\n", numerator,
31                   denominator, result );
32                continueLoop = false; // input successful; end looping
33             } // end try
34             catch ( InputMismatchException inputMismatchException )
35             {
36                System.err.printf( "\nException: %s\n",
37                   inputMismatchException );
38                scanner.nextLine(); // discard input so user can try again
39                System.out.println(
40                   "You must enter integers. Please try again.\n" );
41             } // end catch
```

Fig. 11.2 | Handling `ArithmeticException`s and `InputMismatchException`s. (Part 2 of 4.)

```
42          catch ( ArithmeticException arithmeticException )
43          {
44             System.err.printf( "\nException: %s\n", arithmeticException );
45             System.out.println(
46                "Zero is an invalid denominator. Please try again.\n" );
47          } // end catch
48       } while ( continueLoop ); // end do...while
49    } // end main
50 } // end class DivideByZeroWithExceptionHandling
```

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

Result: 100 / 7 = 14
```

Fig. 11.2 | Handling `ArithmeticException`s and `InputMismatchException`s. (Part 3 of 4.)

```
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
```

```
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. 11.2 | Handling `ArithmeticExceptions` and `InputMismatchExceptions`. (Part 4 of 4.)

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 11.1

Exceptions may surface through explicitly mentioned code in a `try` *block, through calls to other methods, 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` and is 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.*

Multi-catch

```
catch (Type1 | Type2 | Type3 e)
```

1

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

- Uncaught exception—one for which there are no matching `catch` blocks.
- Java uses a multithreaded model of program execution.
  - Each thread is a parallel activity.
  - One program can have many threads.
  - If a program has only one thread, an uncaught exception will cause the program to terminate.
  - [redacted]ultiple threads, an uncaught exception will terminat[redacted]hread where the exception occurred.

11.3 Example: Handling `ArithmeticExceptions` and `InputMismatchExceptions` (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 `ArithmeticExceptions` and `InputMismatchExceptions` (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 throws.
  - Appears after the method's parameter list and before the method's body.
  - Contains a comma-separated list of the exceptions that the method will throw if various problems occur.
    - May be thrown by statements in the method's body or by methods called from the body.
  - Method can throw exceptions of the classes listed in its `throws` clause or of their subclasses.
  - Clients of a method with a `throws` clause are thus informed that the method may throw exceptions.

Error-Prevention Tip 11.1

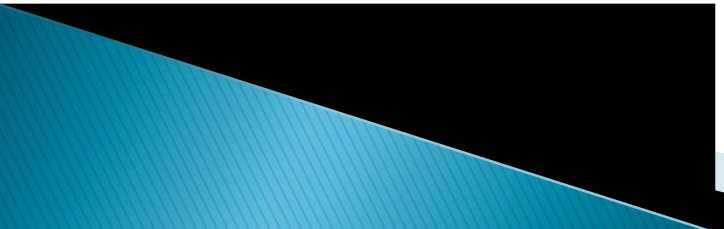
Read the online API documentation for a method before using it in a program. The documentation specifies the exceptions thrown by the method (if any) and indicates reasons why such exceptions may occur. Next, read the online API documentation for the specified exception classes. The documentation for 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
  - unsuccessful memory allocation



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

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.
  - `Error`s happen infrequently.
  - These should not be caught by applications.
  - Applications usually cannot recover from `Error`s.

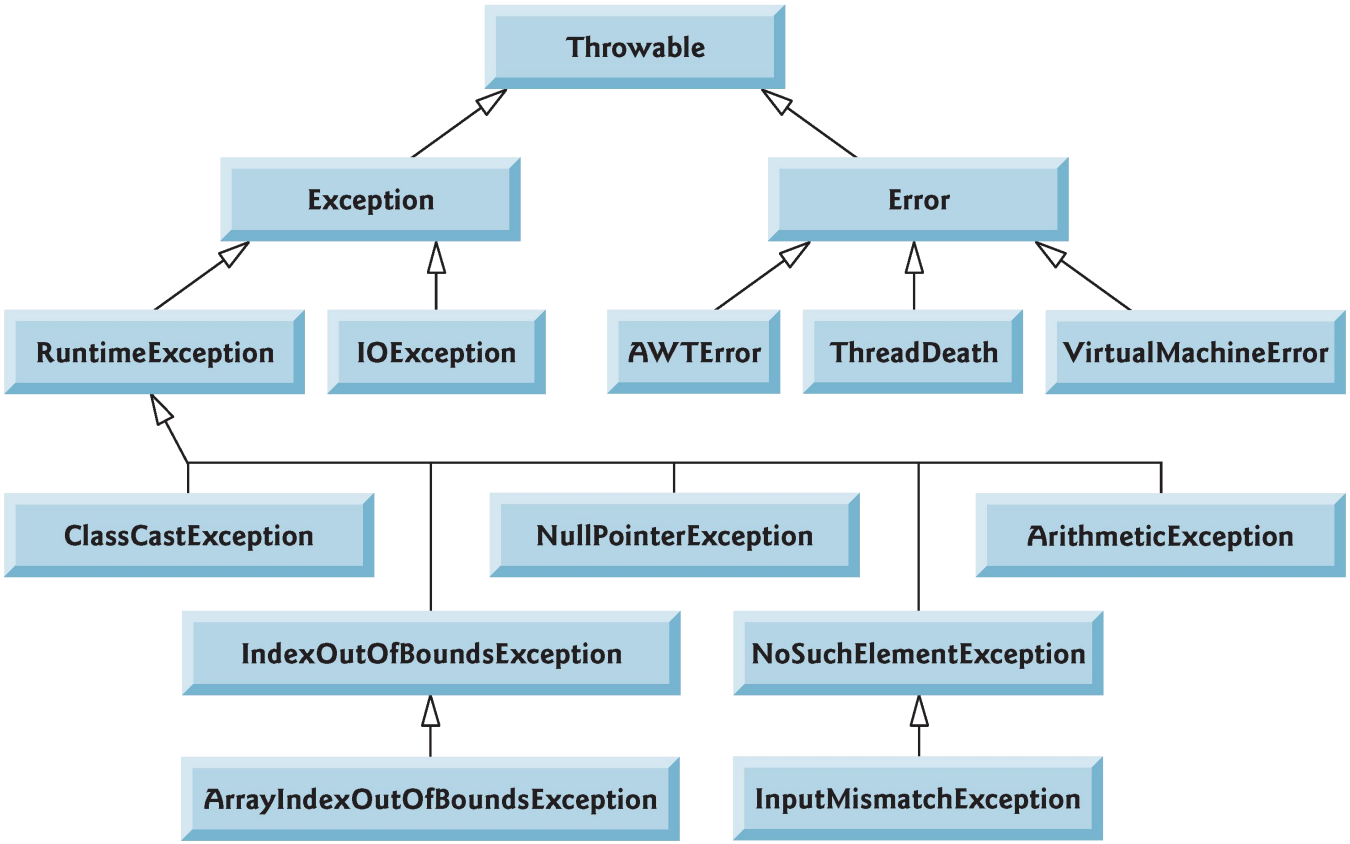


Fig. 11.3 | Portion of class `Throwable`'s inheritance hierarchy.

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., `ArrayIndexOutOfBoundsException`s).
- 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 because the file does not exist.

11.5 Java Exception Hierarchy (Cont.)

- Classes that inherit from class `Error` are considered to be *unchecked*.

11.5 Java Exception Hierarchy (Cont.)

- Checked Exception:
  - The compiler *checks* each method call and method declaration to determine whether the method throws checked exceptions.
  - If so, the compiler verifies that the checked exception is caught or is declared in a `throws` clause.
  - `throws` clause specifies the exceptions a method throws.
    - Such exceptions are typically not caught in the method's body.
  - If the catch-or-declare requirement is not satisfied, the compiler will ge indicating that the exception must be caught or declared.

11.5 Java Exception Hierarchy (Cont.)

- unchecked exception :
  - The compiler does not check 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 attempting to perform the division.
  - 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.

11.5 Java Exception Hierarchy (Cont.)

- A `catch` parameter of a superclass-type can also catch all of that exception type's subclass types.
- You can also catch each subclass type individually if those exceptions require different processing.

11.5 Java Exception Hierarchy (Cont.)

- If there 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 11.5

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.*

11.6 `finally` Block

- The `finally` block is used for resource deallocation.
  - Placed after the last `catch` block.
- `finally` block will execute whether or not an exception is thrown in the corresponding `try` block.

11.6 `finally` Block (Cont.)

- `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 terminates immediately by calling method `System.exit`.

ReturnFinallyTest.jav
TestException.java

```
1  // Fig. 11.4: UsingExceptions.java
2  // try...catch...finally exception handling mechanism.
3
4  public class UsingExceptions
5  {
6     public static void main( String[] args )
7     {
8        try
9        {
10          throwException(); // call method throwException
11       } // end try
12       catch ( Exception exception ) // exception thrown by throwException
13       {
14          System.err.println( "Exception handled in main" );
15       } // end catch
16
17       doesNotThrowException();
18    } // end main
19
20    // demonstrate try...catch...finally
21    public static void throwException() throws Exception
22    {
```

Fig. 11.4 | `try`...`catch`...`finally` exception-handling mechanism. (Part 1 of 4.)

```
23          try // throw an exception and immediately catch it
24          {
25             System.out.println( "Method throwException" );
26             throw new Exception(); // generate exception
27          } // end try
28          catch ( Exception exception ) // catch exception thrown in try
29          {
30             System.err.println(
31                "Exception handled in method throwException" );
32             throw exception; // rethrow for further processing
33
34             // code here would not be reached; would cause compilation errors
35
36          } // end catch
37          finally // executes regardless of what occurs in try...catch
38          {
39             System.err.println( "Finally executed in throwException" );
40          } // end finally
41
42          // code here would not be reached; would cause compilation errors
43
44       } // end method throwException
45
```

Fig. 11.4 | `try...catch...finally` exception-handling mechanism. (Part 2 of 4.)

```
46      // demonstrate finally when no exception occurs
47      public static void doesNotThrowException()
48      {
49         try // try block does not throw an exception
50         {
51            System.out.println( "Method doesNotThrowException" );
52         } // end try
53         catch ( Exception exception ) // does not execute
54         {
55            System.err.println( exception );
56         } // end catch
57         finally // executes regardless of what occurs in try...catch
58         {
59            System.err.println(
60               "Finally executed in doesNotThrowException" );
61         } // end finally
62
63         System.out.println( "End of method doesNotThrowException" );
64      } // end method doesNotThrowException
65   } // end class UsingExceptions
```

Fig. 11.4 | `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. 11.4 | `try`...`catch`...`finally` exception-handling mechanism. (Part 4 of 4.)

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 11.7

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 11.8

An object 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 11.9

Exceptions can be thrown from constructors. When an error is detected in a constructor, an exception should be thrown to avoid creating an improperly formed object.

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 11.6

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 11.5

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

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. 11.5: UsingExceptions.java
2  // Stack unwinding and obtaining data from an exception object.
3
4  public class UsingExceptions
5  {
6     public static void main( String[] args )
7     {
8        try
9        {
10          method1(); // call method1
11       } // end try
12       catch ( Exception exception ) // catch exception thrown in method1
13       {
14          System.err.printf( "%s\n\n", exception.getMessage() );
15          exception.printStackTrace(); // print exception stack trace
16
17          // obtain the stack-trace information
18          StackTraceElement[] traceElements = exception.getStackTrace();
19
20          System.out.println( "\nStack trace from getStackTrace:" );
21          System.out.println( "Class\t\tFile\t\t\tLine\tMethod" );
22
```

Fig. 11.5 | Stack unwinding and obtaining data from an exception object. (Part 1 of 3.)

```
23            // loop through traceElements to get exception description
24            for ( StackTraceElement element : traceElements )
25            {
26               System.out.printf( "%s\t", element.getClassName() );
27               System.out.printf( "%s\t", element.getFileName() );
28               System.out.printf( "%s\t", element.getLineNumber() );
29               System.out.printf( "%s\n", element.getMethodName() );
30            } // end for
31         } // end catch
32      } // end main
33
34      // call method2; throw exceptions back to main
35      public static void method1() throws Exception
36      {
37         method2();
38      } // end method method1
39
40      // call method3; throw exceptions back to method1
41      public static void method2() throws Exception
42      {
43         method3();
44      } // end method method2
```

Fig. 11.5 | Stack unwinding and obtaining data from an exception object. (Part 2 of 3.)

```
45
46      // throw Exception back to method2
47      public static void method3() throws Exception
48      {
49         throw new Exception( "Exception thrown in method3" );
50      } // end method method3
51   } // end class UsingExceptions
```

```
Exception thrown in method3

java.lang.Exception: Exception thrown in method3
        at UsingExceptions.method3(UsingExceptions.java:49)
        at UsingExceptions.method2(UsingExceptions.java:43)
        at UsingExceptions.method1(UsingExceptions.java:37)
        at UsingExceptions.main(UsingExceptions.java:10)

Stack trace from getStackTrace:
Class            File                         Line    Method
UsingExceptions UsingExceptions.java    49      method3
UsingExceptions UsingExceptions.java    43      method2
UsingExceptions UsingExceptions.java    37      method1
UsingExceptions UsingExceptions.java    10      main
```

Fig. 11.5 | Stack unwinding and obtaining data from an exception object. (Part 3 of 3.)

Error-Prevention Tip 11.6

An exception that's not caught in an application causes Java's default exception handler *to run. This displays the name of the exception, a descriptive message that indicates the problem that occurred and a complete execution stack trace. In an application with a single thread of execution, the application terminates. In an application with multiple threads, the thread that caused the exception terminates.*

Error-Prevention Tip 11.7

`Throwable` method `toString` (inherited by all `Throwable` subclasses) returns a `String` containing the name of the exception's class and a descriptive message.

Software Engineering Observation 11.10

Never provide a `catch` *handler with an empty body—this effectively ignores the exception. At least use* `printStackTrace` *to output an error message to indicate that a problem exists.*

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.
- Chained exceptions enable an exception object to maintain the complete stack-trace information from the original

```
1  // Fig. 11.6: UsingChainedExceptions.java
2  // Chained exceptions.
3
4  public class UsingChainedExceptions
5  {
6     public static void main( String[] args )
7     {
8        try
9        {
10          method1(); // call method1
11       } // end try
12       catch ( Exception exception ) // exceptions thrown from method1
13       {
14          exception.printStackTrace();
15       } // end catch
16    } // end main
17
```

Fig. 11.6 | Chained exceptions. (Part 1 of 4.)

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

Fig. 11.6 | Chained exceptions. (Part 2 of 4.)

```
31      // call method3; throw exceptions back to method1
32      public static void method2() throws Exception
33      {
34         try
35         {
36            method3(); // call method3
37         } // end try
38         catch ( Exception exception ) // exception thrown from method3
39         {
40            throw new Exception( "Exception thrown in method2", exception );
41         } // end catch
42      } // end method method2
43
44      // throw Exception back to method2
45      public static void method3() throws Exception
46      {
47         throw new Exception( "Exception thrown in method3" );
48      } // end method method3
49   } // end class UsingChainedExceptions
```

Fig. 11.6 | Chained exceptions. (Part 3 of 4.)

```
java.lang.Exception: Exception thrown in method1
        at UsingChainedExceptions.method1(UsingChainedExceptions.java:27)
        at UsingChainedExceptions.main(UsingChainedExceptions.java:10)
Caused by: java.lang.Exception: Exception thrown in method2
        at UsingChainedExceptions.method2(UsingChainedExceptions.java:40)
        at UsingChainedExceptions.method1(UsingChainedExceptions.java:23)
        ... 1 more
Caused by: java.lang.Exception: Exception thrown in method3
        at UsingChainedExceptions.method3(UsingChainedExceptions.java:47)
        at UsingChainedExceptions.method2(UsingChainedExceptions.java:36)
        ... 2 more
```

Fig. 11.6 | Chained exceptions. (Part 4 of 4.)

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.

Software Engineering Observation 11.11

If possible, indicate exceptions from your methods by using existing *exception classes, rather than creating new ones. The Java API contains many exception classes that might be suitable for the type of problems your methods need to indicate.*

Good Programming Practice 11.3

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

Software Engineering Observation 11.12

When defining your own exception type, study the existing exception classes in the Java API and 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. The new exception class should be a checked exception (i.e., extend* `Exception` *but not* `RuntimeException`*) if clients should be* required *to handle the exception. The client application should be able to reasonably recover from such an exception. The new exception class should extend* `RuntimeException` *if the client code should be able to ignore the exception (i.e., the exception is an unchecked one).*

Good Programming Practice 11.4

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

11.10 Assertions

- 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 Assertions (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 Assertions (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 `IndexOutOfBoundsException`.
  - 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 [illegible] character at the index.

11.11 Assertions (Cont.)

- 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.

```
1  // Fig. 11.7: AssertTest.java
2  // Checking with assert that a value is within range
3  import java.util.Scanner;
4
5  public class AssertTest
6  {
7     public static void main( String[] args )
8     {
9        Scanner input = new Scanner( System.in );
10
11       System.out.print( "Enter a number between 0 and 10: " );
12       int number = input.nextInt();
13
14       // assert that the value is >= 0 and <= 10
15       assert ( number >= 0 && number <= 10 ) : "bad number: " + number;
16
17       System.out.printf( "You entered %d\n", number );
18    } // end main
19 } // end class AssertTest
```

Fig. 11.7 | 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:15)
```

Fig. 11.7 | Checking with `assert` that a value is within range. (Part 2 of 2.)

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
```

11.11 Assertions (Cont.)

- Users should not encounter any `AssertionError`s through normal execution of a properly written program.
  - Such errors should only indicate bugs in the implementation.
  - As a result, you should never catch an `AssertionError`.
  - Allow the program to terminate when the error occurs, so you can see the error message, then locate and fix the source of the problem.
- Since application users can choose not to enable assertions at runtime
  - You *should not* use `assert` to indicate runtime problems in production code.
  - You *should* use the exception mechanism for this purpose.

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 (ClassName theObject = new ClassName()) {
    // 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
}
```

where ClassName is a class that implements AutoCloseable.

This code creates a ClassName object, uses it in the try block, then calls its close method at the end of the try block—or, if an exception occurs, at the end of a catch block—to release the object's resources.