

13

Exception Handling



It is common sense to take a method and try it. If it fails, admit it frankly and try another. But above all, try something.

— Franklin Delano Roosevelt

O! throw away the worser part of it, And live the purer with the other half.

— William Shakespeare

If they're running and they don't look where they're going I have to come out from somewhere and catch them.

— Jerome David Salinger

O infinite virtue! com'st thou smiling from the world's great snare uncaught?

— William Shakespeare



OBJECTIVES

In this chapter you will learn:

- **How exception and error handling works.**
- **To use try, throw and catch to detect, indicate and handle exceptions, respectively.**
- **To use the finally block to release resources.**
- **How stack unwinding enables exceptions not caught in one scope to be caught in another scope.**
- **How stack traces help in debugging.**
- **How exceptions are arranged in an exception class hierarchy.**
- **To declare new exception classes.**
- **To create chained exceptions that maintain complete stack trace information.**



- 13.1 Introduction**
- 13.2 Exception-Handling Overview**
- 13.3 Example: Divide By Zero Without Exception Handling**
- 13.4 Example: Handling `ArithmeticExceptions` and `InputMismatchExceptions`**
- 13.5 When to Use Exception Handling**
- 13.6 Java Exception Hierarchy**
- 13.7 `finally` block**
- 13.8 Stack Unwinding**
- 13.9 `printStackTrace`, `getStackTrace` and `getMessage`**
- 13.10 Chained Exceptions**
- 13.11 Declaring New Exception Types**
- 13.12 Preconditions and Postconditions**
- 13.13 Assertions**
- 13.14 Wrap-Up**



13.1 Introduction

- **Exception – an indication of a problem that occurs during a program's execution**
- **Exception handling – resolving exceptions that may occur so program can continue or terminate gracefully**
- **Exception handling enables programmers to create programs that are more robust and fault-tolerant**



Error-Prevention Tip 13.1

Exception handling helps improve a program's fault tolerance.



13.1 Introduction

- **Examples**

- **ArrayIndexOutOfBoundsException** – an attempt is made to access an element past the end of an array
- **ClassCastException** – an attempt is made to cast an object that does not have an *is-a* relationship with the type specified in the cast operator
- **NullPointerException** – when a **null** reference is used where an object is expected



13.2 Exception-Handling Overview

- **Intermixing program logic with error-handling logic can make programs difficult to read, modify, maintain and debug**
- **Exception handling enables programmers to remove error-handling code from the “main line” of the program’s execution**
- **Improves clarity**
- **Enhances modifiability**



Performance Tip 13.1

If the potential problems occur infrequently, intermixing program and error-handling logic can degrade a program's performance, because the program must perform (potentially frequent) tests to determine whether the task executed correctly and the next task can be performed.



13.3 Example: Divide By Zero Without Exception Handling

- **Thrown exception** – an exception that has occurred
- **Stack trace**
 - Name of the exception in a descriptive message that indicates the problem
 - Complete method-call stack
- **ArithmeticException** – can arise from a number of different problems in arithmetic
- **Throw point** – initial point at which the exception occurs, top row of call chain
- **InputMismatchException** – occurs when `Scanner` method `nextInt` receives a string that does not represent a valid integer



```

1 // Fig. 13.1: DivideByZeroNoExceptionHandling.java
2 // An application that attempts to divide by zero.
3 import java.util.Scanner;
4
5 public class DivideByZeroNoExceptionHandling
6 {
7     // demonstrates throwing an exception
8     public static int quotient( int numerator,
9                               int denominator )
10    {
11        return numerator / denominator; // possible division by zero
12    } // end method quotient
13
14    public static void main( String args[] )
15    {
16        Scanner scanner = new Scanner( System.in ); // scanner for input
17
18        System.out.print( "Please enter an integer numerator: " );
19        int numerator = scanner.nextInt();
20        System.out.print( "Please enter an integer denominator: " );
21        int denominator = scanner.nextInt();
22
23        int result = quotient( numerator, denominator );
24        System.out.printf(
25            "\nResult: %d / %d = %d\n", numerator, denominator, result );
26    } // end main
27 } // end class DivideByZeroNoExceptionHandling

```

Attempt to divide; denominator may be zero

Read input; exception occurs if input is not a valid integer

```

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: 7
```

```
Result: 100 / 7 = 14
```

```
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)
```



13.4 Example: Handling ArithmeticExceptions and InputMismatchExceptions

- **With exception handling, the program catches and handles (i.e., deals with) the exception**
- **Next example allows user to try again if invalid input is entered (zero for denominator, or non-integer input)**



Enclosing Code in a try Block

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



Software Engineering Observation 13.1

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.



Catching Exceptions

- **catch block** – catches (i.e., receives) and handles an exception, contains:
 - Begins with keyword **catch**
 - Exception parameter in parentheses – exception parameter identifies the exception type and enables **catch** block to interact with caught exception object
 - Block of code in curly braces that executes when exception of proper type occurs
- **Matching catch block** – the type of the exception parameter matches the thrown exception type exactly or is a superclass of it
- **Uncaught exception** – an exception that occurs for which there are no matching **catch** blocks
 - Cause program to terminate if program has only one thread; Otherwise only current thread is terminated and there may be adverse effects to the rest of the program



Common Programming Error 13.1

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



Common Programming Error 13.2

Each catch block can have only a single parameter—specifying a comma-separated list of exception parameters is a syntax error.



Termination Model of Exception Handling

- **When an exception occurs:**
 - **try** block terminates immediately
 - Program control transfers to first matching **catch** block
- **After exception is handled:**
 - Termination model of exception handling – program control does not return to the throw point because the **try** block has expired; Flow of control proceeds to the first statement after the last **catch** block
 - Resumption model of exception handling – program control resumes just after throw point
- **try** statement – consists of **try** block and corresponding **catch** and/or **finally** blocks



Common Programming Error 13.3

Logic errors can occur if you assume that after an exception is handled, control will return to the first statement after the throw point.



Error-Prevention Tip 13.2

With exception handling, a program can continue executing (rather than terminating) after dealing with a problem. This helps ensure the kind of robust applications that contribute to what is called mission-critical computing or business-critical computing.



Good Programming Practice 13.1

Using an exception parameter name that reflects the parameter's type promotes clarity by reminding the programmer of the type of exception being handled.



Using the throws Clause

- **throws clause** – specifies the exceptions a method may throws
 - Appears after method's parameter list and before the method's body
 - Contains a comma-separated list of exceptions
 - Exceptions can be thrown by statements in method's body or by methods called in method's body
 - Exceptions can be of types listed in **throws** clause or subclasses



Error-Prevention Tip 13.3

If you know that a method might throw an exception, include appropriate exception-handling code in your program to make it more robust.



Error-Prevention Tip 13.4

Read the online API documentation for a method before using that method in a program. The documentation specifies the exceptions thrown by the method (if any) and indicates reasons why such exceptions may occur. Then provide for handling those exceptions in your program.



Error-Prevention Tip 13.5

Read the online API documentation for an exception class before writing exception-handling code for that type of exception. The documentation for an exception class typically contains potential reasons that such exceptions occur during program execution.



```

1 // Fig. 13.2: DivideByZeroWithExceptionHandling.java
2 // An exception-handling example that checks for divide-by-zero.
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; //
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 done = false; // input is needed
19         do
20         {
21             try // read two numbers and calculate quotient
22             {
23                 system.out.print( "Please enter an integer numerator: " );
24                 int numerator = scanner.nextInt();
25                 system.out.print( "Please enter an integer denominator: " );
26                 int denominator = scanner.nextInt();
27
28

```

throws clause specifies that method `quotient` may throw an `ArithmeticException`

Repetition statement loops until `try` block completes successfully

`try` block attempts to read input and perform division

Retrieve input; `InputMismatchException` thrown if input not valid integers



```

29 int result = quotient( numerator, denominator );
30 System.out.printf( "\nResult: %d / %d = %d\n", numerator
31     denominator, result );
32 continueLoop = false;
33 } // end try
34 catch ( InputMismatchException
35 {
36     System.err.printf( "\nException: %s\n",
37         inputMismatchException );
38     scanner.nextLine(); // discard input so user can try
39     System.out.println(
40         "You must enter integers. Please try again." );
41 } // end catch
42 catch ( ArithmeticException arithmeticException )
43 {
44     System.err.printf( "\nException: %s\n",
45         arithmeticException );
46     System.out.println(
47         "Zero is an invalid denominator. Please try again." );
48 } // end catch
49 } while ( continueLoop ); // end do...while
50 } // end main
51 } // end class DivideByZeroWithExceptionHandling

```

Call method `quotient`, which may throw `ArithmeticException`

If we have reached this point, input was valid and denominator was non-zero, so looping can stop

Catching `InputMismatchException` (user has entered non-integer input)

Exception parameters

Read invalid input but do nothing with it

Catching `ArithmeticException` (user has entered zero for denominator)

If line 32 was never successfully reached, loop continues and user can 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: 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
```



13.5 When to Use Exception Handling

- **Exception handling designed to process synchronous errors**
- **Synchronous errors – occur when a statement executes**
- **Asynchronous errors – occur in parallel with and independent of the program's flow of control**



Software Engineering Observation 13.2

Incorporate your exception-handling strategy into your system from the design process's inception. Including effective exception handling after a system has been implemented can be difficult.



Software Engineering Observation 13.3

Exception handling provides a single, uniform technique for processing problems. This helps programmers working on large projects understand each other's error-processing code.



Software Engineering Observation 13.4

Avoid using exception handling as an alternate form of flow of control. These “additional” exceptions can “get in the way” of genuine error-type exceptions.



Software Engineering Observation 13.5

Exception handling simplifies combining software components and enables them to work together effectively by enabling predefined components to communicate problems to application-specific components, which can then process the problems in an application-specific manner.



13.6 Java Exception Hierarchy

- All exceptions inherit either directly or indirectly from class **Exception**
- Exception classes form an inheritance hierarchy that can be extended
- Class **Throwable**, superclass of **Exception**
 - Only **Throwable** objects can be used with the exception-handling mechanism
 - Has two subclasses: **Exception** and **Error**
 - Class **Exception** and its subclasses represent exception situations that can occur in a Java program and that can be caught by the application
 - Class **Error** and its subclasses represent abnormal situations that could happen in the JVM – it is usually not possible for a program to recover from **Errors**



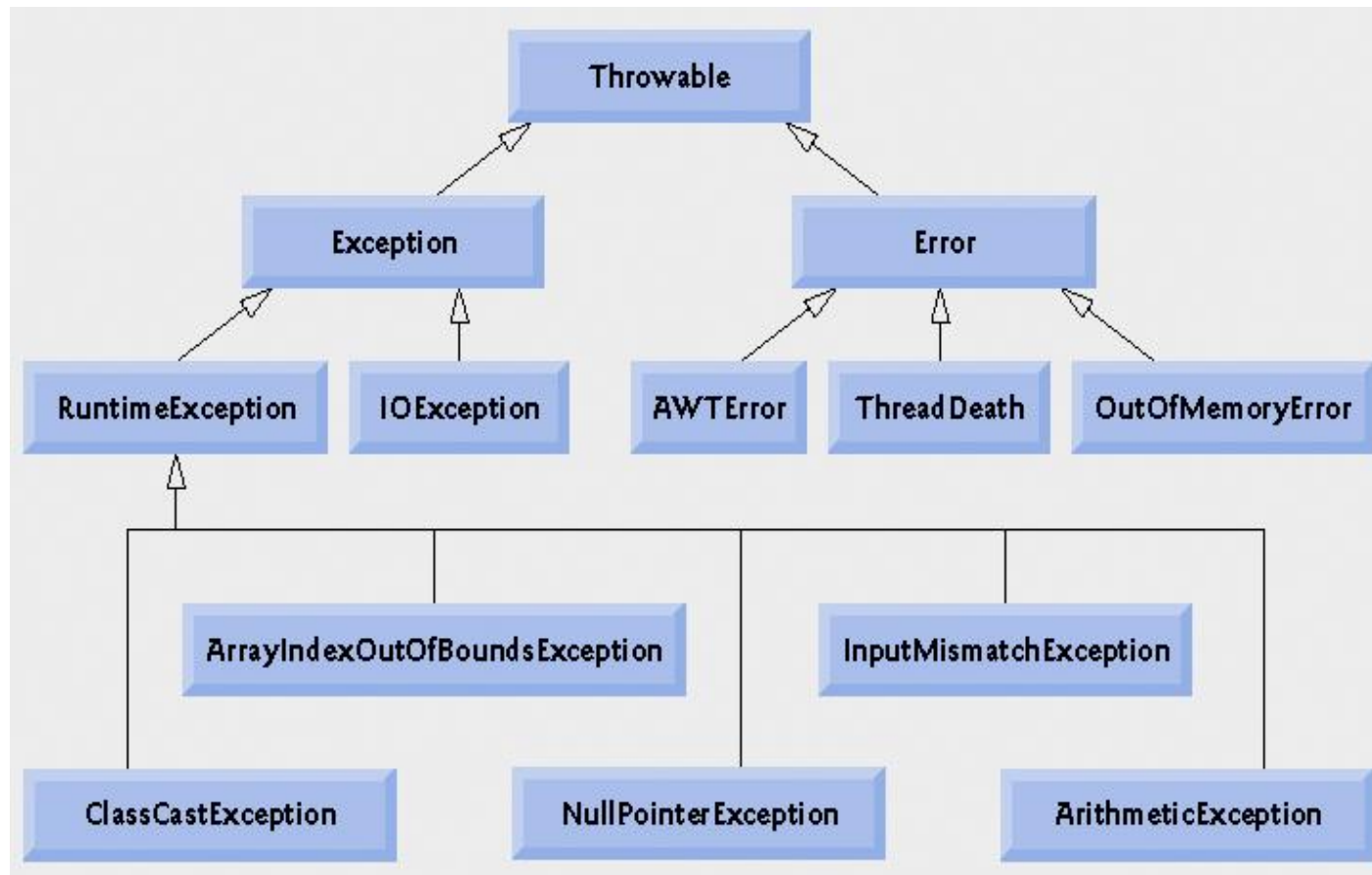


Fig. 13.3 | Portion of class Throwable's inheritance hierarchy.

13.6 Java Exception Hierarchy

- **Two categories of exceptions: checked and unchecked**
- **Checked exceptions**
 - Exceptions that inherit from class `Exception` but not from `RuntimeException`
 - Compiler enforces a catch-or-declare requirement
 - Compiler checks each method call and method declaration to determine whether the method **throws** checked exceptions. If so, the compiler ensures that the checked exception is caught or is declared in a **throws** clause. If not caught or declared, compiler error occurs.
- **Unchecked exceptions**
 - Inherit from class `RuntimeException` or class `Error`
 - Compiler does not check code to see if exception is caught or declared
 - If an unchecked exception occurs and is not caught, the program terminates or runs with unexpected results
 - Can typically be prevented by proper coding



Software Engineering Observation 13.6

Programmers are forced to deal with checked exceptions. This results in more robust code than would be created if programmers were able to simply ignore the exceptions.



Common Programming Error 13.4

A compilation error occurs if a method explicitly attempts to throw a checked exception (or calls another method that throws a checked exception) and that exception is not listed in that method's throws clause.



Common Programming Error 13.5

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



Software Engineering Observation 13.7

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



Software Engineering Observation 13.8

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



13.6 Java Exception Hierarchy

- **catch block catches all exceptions of its type and subclasses of its type**
- **If there are multiple catch blocks that match a particular exception type, only the first matching catch block executes**
- **It makes sense to use a catch block of a superclass when all the catch blocks for that class's subclasses will perform the same functionality**



Error-Prevention Tip 13.6

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 for subclasses of that superclass ensures that all subclass exceptions are eventually caught.



Common Programming Error 13.6

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



13.7 `finally` block

- Programs that obtain certain resources must return them explicitly to avoid resource leaks
- `finally` block
 - Consists of `finally` keyword followed by a block of code enclosed in curly braces
 - Optional in a `try` statement
 - If present, is placed after the last `catch` block
 - Executes whether or not an exception is thrown in the corresponding `try` block or any of its corresponding `catch` blocks
 - Will not execute if the application exits early from a `try` block via method `System.exit`
 - Typically contains resource-release code



Error-Prevention Tip 13.7

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



```
try
{
    statements
    resource-acquisition statements
} // end try
catch ( AKindOfException exception1 )
{
    exception-handling statements
} // end catch
.
.
.
catch ( AnotherKindOfException exception2 )
{
    exception-handling statements
} // end catch
finally
{
    statements
    resource-release statements
} // end finally
```

Fig. 13.4 | Position of the finally block after the last catch block in a try statement.



13.7 finally block

- If no exception occurs, **catch** blocks are skipped and control proceeds to **finally** block.
- After the **finally** block executes control proceeds to first statement after the **finally** block.
- If exception occurs in the **try** block, program skips rest of the **try** block. First matching the **catch** block executes and control proceeds to the **finally** block. If exception occurs and there are no matching **catch** blocks, control proceeds to the **finally** block. After the **finally** block executes, the program passes the exception to the next outer the **try** block.
- If **catch** block throws an exception, the **finally** block still executes.



Performance Tip 13.2

Always release each resource explicitly and at the earliest possible moment at which it is no longer needed. This makes resources immediately available to be reused by your program or by other programs, thus improving resource utilization.



Error-Prevention Tip 13.8

Because the `finally` block is guaranteed to execute whether or not an exception occurs in the corresponding `try` block, this block is an ideal place to release resources acquired in a `try` block. This is also an effective way to eliminate resource leaks. For example, the `finally` block should close any files opened in the `try` block.



13.7 finally block

- **Standard streams**
 - `System.out` – standard output stream
 - `System.err` – standard error stream
- **`System.err` can be used to separate error output from regular output**
- **`System.err.println` and `System.out.println` display data to the command prompt by default**



```
1 // Fig. 13.5: UsingExceptions.java
2 // Demonstration of the try...catch...finally exception handling
3 // mechanism.
4
5 public class UsingExceptions
6 {
7     public static void main( String args[] )
8     {
9         try
10        {
11            throwException(); // call method throwException
12        } // end try
13        catch ( Exception exception )
14        {
15            System.err.println( "Exception handled in main" );
16        } // end catch
17
18        doesNotThrowException();
19    } // end main
20
```

Call method that throws an exception



```
21 // demonstrate try...catch...finally
22 public static void throwException() throws Exception
23 {
24     try // throw an exception and immediately catch it
25     {
26         System.out.println( "Method throwException" );
27         throw new Exception(); // generate exception
28     } // end try
29     catch ( Exception exception ) // ca
30     {
31         System.err.println(
32             "Exception handled in method throwException" );
33         throw exception; // rethrow for further processing
34
35         // any code here would not be
36
37     } // end catch
38     finally // executes regardless of what occurs in try...catch
39     {
40         System.err.println
41     } // end finally
42
43     // any code here would not be reached, exception rethrown in catch
44
```

Create new Exception and throw it

Throw previously created Exception

finally block executes even though
exception is rethrown in catch block



```

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

```

finally block executes even though no exception is thrown

```

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

```



Throwing Exceptions Using the throw Statement

- **throw statement** – used to throw exceptions
- **Programmers can throw exceptions themselves from a method if something has gone wrong**
- **throw statement consists of keyword throw followed by the exception object**



Software Engineering Observation 13.9

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 13.10

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



Software Engineering Observation 13.11

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



Rethrowing Exceptions

- **Exceptions are rethrown when a `catch` block decides either that it cannot process the exception or that it can only partially process it**
- **Exception is deferred to outer `try` statement**
- **Exception is rethrown by using keyword `throw` followed by a reference to the exception object**



Common Programming Error 13.7

If an exception has not been caught when control enters a `finally` block and the `finally` block throws an exception that is 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 13.9

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



Common Programming Error 13.8

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 13.2

Java's exception-handling mechanism is intended to remove 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 makes programs difficult to read. Rather, place one `try` block around a significant portion of your code, follow that `try` block with `catch` blocks that handle each possible exception and follow the `catch` blocks with a single `finally` block (if one is required).



13.8 Stack Unwinding

- **Stack unwinding** – When an exception is thrown but not caught in a particular scope, the method-call stack is “unwound,” and an attempt is made to catch the exception in the next outer **try** block.
- **When unwinding occurs:**
 - The method in which the exception was not caught terminates
 - All local variables in that method go out of scope
 - Control returns to the statement that originally invoked the method – if a try block encloses the method call, an attempt is made to catch the exception.



```
1 // Fig. 13.6: UsingExceptions.java
2 // Demonstration of stack unwinding.
3
4 public class UsingExceptions
5 {
6     public static void main( String args[] )
7     {
8         try // call throwExce
9         {
10             throwException();
11         } // end try
12         catch ( Exception exception ) // exception thrown in throwException
13         {
14             System.err.println( "Except
15         } // end catch
16     } // end main
17
```

Call method that throws an exception

Catch exception that may occur in the
above try block, including the call to
method throwException



```

18 // throwException throws exception that is not caught in this method
19 public static void throwException() throws Exception
20 {
21     try // throw an exception and catch it in main
22     {
23         System.out.println( "Method throwException" );
24         throw new Exception(); // generate exception
25     } // end try
26     catch ( RuntimeException runtimeEx
27     {
28         System.err.println(
29             "Exception handled in method throwException" );
30     } // end catch
31     finally // finally block executes before
32     {           control returns to outer try block
33         System.err.println( "Finally is always executed" );
34     } // end finally
35 } // end method throwException
36 } // end class UsingExceptions

```

Method throws exception

Throw new exception; Exception not caught in current try block, so handled in outer try block

finally block executes before control returns to outer try block

Method throwException
Finally is always executed
Exception handled in main



13.9 `printStackTrace`, `getStackTrace` and `getMessage`

- **Methods in class `Throwable` retrieve more information about an exception**
 - **`printStackTrace`** – outputs stack trace to standard error stream
 - **`getStackTrace`** – retrieves stack trace information as an array of `StackTraceElement` objects; enables custom processing of the exception information
 - **`getMessage`** – returns the descriptive string stored in an exception



Error-Prevention Tip 13.10

An exception that is 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 13.11

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



13.9 printStackTrace, getStackTrace and getMessage

- **StackTraceElement methods**
 - `getClassName`
 - `getFileName`
 - `getLineNumber`
 - `getMethodName`
- **Stack trace information follows pattern –**
`className.methodName(fileName:lineNumber)`



```

1 // Fig. 13.7: UsingExceptions.java
2 // Demonstrating getMessage and printStackTrace from class Exception.
3
4 public class UsingExceptions
5 {
6     public static void main(
7     {
8         try
9         {
10             method1(); // call method1
11         } // end try
12         catch ( Exception exception ) // catch exception thrown
13         {
14             System.err.printf( "%s\n\n", exception.getMessage() );
15             exception.printStackTrace(); // print exception stack
16
17             // obtain the stack-trace information
18             StackTraceElement[] traceElements = exception.getStackTrace();
19

```

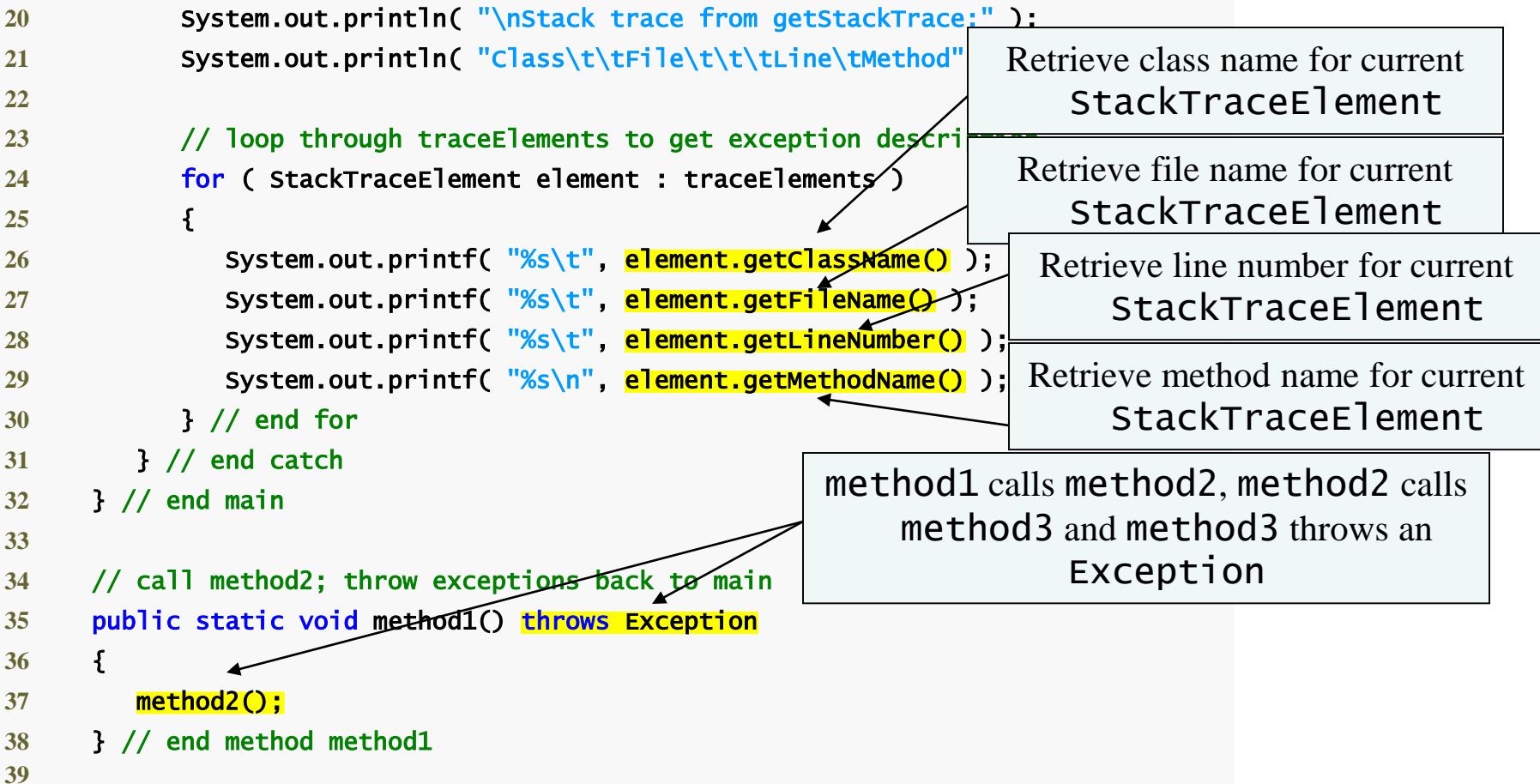
Call to method1, method1 calls method2, method2 calls method3 and method3 throws a new Exception

Display descriptive string of exception thrown in method3

Retrieve stack information as an array of StackTraceElement objects

Display stack trace for exception thrown in method3





```

40 // call method3; throw exceptions back to method1
41 public static void method2() throws Exception
42 {
43     method3();
44 } // end method method2
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

```

method2 calls method3, which throws an Exception

Exception created and thrown

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



Software Engineering Observation 13.12

Never ignore an exception you catch. At least use `printStackTrace` to output an error message. This will inform users that a problem exists, so that they can take appropriate actions.



13.10 Chained Exceptions

- **Chained exceptions enable an exception object to maintain the complete stack-trace information when an exception is thrown from a catch block**
- **Users can retrieve information about original exception**
- **Stack trace from a chained exception displays how many chained exceptions remain**



```
1 // Fig. 13.8: UsingChainedExceptions.java
```

```
2 // Demonstrating 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         {
```

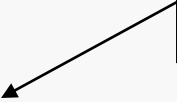
```
14             exception.printStackTrace();
```

```
15         } // end catch
```

```
16     } // end main
```

```
17
```

Catch exception from **method1** as well
as any associated chained exceptions



```
18 // call method2; throw exceptions back to main
19 public static void method1() throws Exceptionw
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 try
29 } // end method method1
30
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
```

Catch exception from method2, throw new exception to be chained with earlier exceptions

Catch exception from method3, throw new exception to be chained with earlier exceptions



```
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
```

Original thrown exception



```
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
```



13.11 Declaring New Exception Types

- **You can declare your own exception classes that are specific to the problems that can occur when another program uses your reusable classes**
- **New exception class must extend an existing exception class**
- **Typically contains only two constructors**
 - **One takes no arguments, passes a default exception messages to the superclass constructor**
 - **One that receives a customized exception message as a string and passes it to the superclass constructor**



Software Engineering Observation 13.13

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



Good Programming Practice 13.3

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



Software Engineering Observation 13.14

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 are 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. (cont...)



Software Engineering Observation 13.14

The new exception class should be a checked exception (i.e., extend `Exception` but not `RuntimeException`) if possible 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 exception).



Good Programming Practice 13.4

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



13.12 Preconditions and Postconditions

- **Preconditions and postconditions are the states before and after a method's execution**
- **Used to facilitate debugging and improve design**
- **You should state the preconditions and postconditions in a comment before the method declaration**



13.12 Preconditions and Postconditions

- **Preconditions**

- **Condition that must be true when the method is invoked**
- **Describe method parameters and any other expectations the method has about the current state of a program**
- **If preconditions not met, method's behavior is undefined**

- **Postconditions**

- **Condition that is true after the method successfully returns**
- **Describe 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**



13.13 Assertions

- **Assertions are conditions that should be true at a particular point in a method**
- **Help ensure a program's validity by catching potential bugs**
- **Preconditions and Postconditions are two kinds of assertions**
- **Assertions can be stated as comments or assertions can be validated programmatically using the `assert` statement**



13.13 Assertions

- **assert statement**
 - Evaluates a **boolean** expression and determines whether it is **true** or **false**
 - Two forms
 - **assert *expression*; --** **AssertionError** is thrown if *expression* is **false**
 - **assert *expression1* : *expression2*; --** **AssertionError** is thrown if *expression1* is **false**, *expression2* is error message
 - Used to verify intermediate states to ensure code is working correctly
 - Used to implement preconditions and postconditions programmatically
- **By default, assertions are disabled**
- **Assertions can be enabled with the **-ea** command-line option**



```

1 // Fig. 13.9: AssertTest.java
2 // Demonstrates the assert statement
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 =
13             // assert statement
14             // assert that the absolute value is >= 0
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

```

Message to be displayed with
AssertionError

If number is less than 0 or greater
than 10, AssertionError occurs

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)

