Exception Handling

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).

Error-Prevention Tip 11.1

Exception handling helps improve a program's fault tolerance.

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.

Error-Handling Overview

- Programs frequently test conditions to determine how program execution should proceed.
- Consider the following pseudocode:
  - *Perform a task*
    - » *If the preceding task did not execute correctly*
      Perform error processing
    - » *Perform next task*
  - *If the preceding task did not execute correctly*
    Perform error processing
  - *…*
    - Begins by performing a task; then tests whether it executed correctly.
    - If not, perform error processing.
    - Otherwise, continue with the next task.
- Intermixing program and error-handling logic in this manner can make programs difficult to read, modify, maintain and debug—especially in large applications.

Error-Handling Overview (Cont.)

- Exception handling enables you to remove error-handling code from the "main line" of program execution
  - Improves program clarity
  - Enhances modifiability
- Handle any exceptions you choose
  - All exceptions
  - All exceptions of a certain type
  - All exceptions of a group of related types (i.e., related through a superclass).
- Such flexibility reduces the likelihood that errors will be overlooked, thus making programs more robust.

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.

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. 11.1: 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
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 );
```

JVM throws exception if `denominator` is 0

User could type invalid input

User could type invalid input (including 0)

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


```
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
```

```
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)
```

Causes division by 0; stack trace shows what led to the exception

Fig. 11.1 | Integer division without exception handling. (Part 2 of 3.)

```
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)
```

User typed non-integer value; stack trace shows what led to the exception

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

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

- Last "at" line in the stack trace started the call chain.
- Each line contains the class name and method followed by the file name and line number.
- 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.

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

- Prior examples that read numeric values from the user 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.

Example: Handling `ArithmeticExceptions` and `InputMismatchExceptions`

- The application in uses exception handling to process any `ArithmeticExceptions` and `InputMistmatchExceptions` 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 again.

```
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
```

Exception type thrown by several methods of class `Scanner`

Indicates that this method *might* throw an `ArithmeticException`

Fig. 11.2 | Handling `ArithmeticException`s and `InputMismatchException`s. (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
```

Starts a block of code in which an exception might occur; block also contains code that should not execute if an exception occurs

Catches and processes `InputMismatchExceptions`

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
```

Catches and processes ArithmeticExceptions

```
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
```

We purposely displayed the exception's error message

```
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
```

We purposely displayed the exception's error message

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

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.*

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 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.

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.*

Common Programming Error 11.2

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

Example: Handling `ArithmeticExceptions` and `InputMismatchExceptions` (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 parallel 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 where the exception occurred.

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.

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.
- The `try` block and its corresponding `catch` and/or `finally` blocks form a `try` statement.

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.

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.

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.

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

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

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 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`.

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.

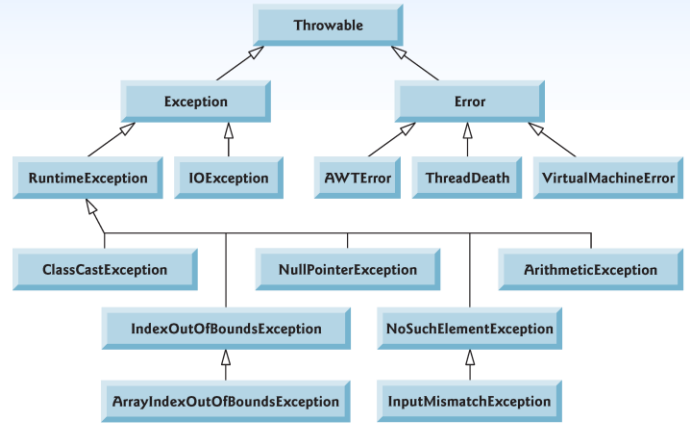

Fig. 11.3 | Portion of class `Throwable`'s inheritance hierarchy.

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.

Java Exception Hierarchy (Cont.)

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

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 indicating that the exception must be caught or declared.

Java Exception Hierarchy (Cont.)

- 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.

Java Exception Hierarchy (Cont.)

- A `catch` parameter of a superclass-type can also catch all of that exception type's subclass types.
  - Enables `catch` to handle related er-rors with a concise notation
  - Allows for polymorphic processing of related exceptions
  - Catching related exceptions in one `catch` block makes sense only if the handling behavior is the same for all subclasses.
- You can also catch each subclass type individually if those exceptions require different processing.

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.

`finally` Block

- Programs that obtain resources must return them to the system explicitly to avoid so-called resource leaks.
  - In programming languages such as C and C++, the most common kind of 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 is used for resource deallocation.
  - Placed after the last `catch` block.

```
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. 11.4 | A `try` statement with a `finally` block.

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

`finally` Block (Cont.)

- Because a `finally` block almost 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.

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.

`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.


```
1  // Fig. 11.5: 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
```

Starts a call chain in which an exception will be thrown

Starts a call chain in which no exceptions occur

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

```
20    // demonstrate try...catch...finally
21    public static void throwException() throws Exception
22    {
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
```

This method might throw an `Exception` (this is a *checked* type)

Throws a new `Exception` that is caught at line 28 and thrown again at line 32

Rethrowing the exception means that it is not considered to have been handled

This block executes even though line 32 in the `catch` handler threw an exception; then the method terminates

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


```
45
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
```

This method does not throw any exceptions

This `try` block will execute all of its statements correctly

This `catch` handler will be skipped; no exceptions occur

This `finally` block still executes

Program control continues here

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.)

`finally` Block (Cont.)

- Both `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

`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`.

`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.

Stack Unwinding

- 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.
3
4  public class UsingExceptions
5  {
6     public static void main( String[] args )
7     {
8        try // call throwException to demonstrate stack unwinding
9        {
10          throwException(); <- Calls a method that might throw an exception
11       } // end try
12       catch ( Exception exception ) // exception thrown in throwException <- Catches the exception and displays a message
13       {
14          System.err.println( "Exception handled in main" );
15       } // end catch
16    } // end main
17
```

Fig. 11.6 | Stack unwinding. (Part 1 of 2.)

```
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 runtimeException ) // catch incorrect type
27        {
28           System.err.println(
29              "Exception handled in method throwException" );
30        } // end catch
31        finally // finally block always executes
32        {
33           System.err.println( "Finally is always executed" );
34        } // end finally
35     } // end method throwException
36  } // end class UsingExceptions
```

This method might throw an `Exception` (this is a *checked* type)

Throws a new `Exception` that is not caught by an exception handler in this method's scope

The `finally` block executes before the method terminates (stack unwinding) and the exception is returned to the caller

```
Method throwException
Finally is always executed
Exception handled in main
```

Fig. 11.6 | Stack unwinding. (Part 2 of 2.)

`printStackTrace`, `getStackTrace` and `getMessage`

- `Throwable` method `printStackTrace` outputs the stack trace to the standard error stream.
  - Helpful in testing and debugging.
- `Throwable` method `getStackTrace` retrieves the stack-trace information.
- `Throwable` method `getMessage` returns the descriptive string stored in an exception.
- To output the stack-trace information to streams other than the standard error stream:
  - Use the information returned from `getStackTrace` and output it to another stream
  - Use one of the overloaded versions of method `printStackTrace`

```
1  // Fig. 11.7: UsingExceptions.java
2  // Throwable methods getMessage, getStackTrace and printStackTrace.
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
```

Starts the call chain that will lead to an exception in this program

None of the other methods catch the exception; so the stack is unwound and the exception is caught here

Gets an array of `StackTraceElement`s

Fig. 11.7 | `Throwable` methods `getMessage`, `getStackTrace` and `printStackTrace`. (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
```

`StackTraceElement` methods returns the class name, file name, line number and method name for a particular stack frame

This method might throw an `Exception` (this is a *checked* type)

Continues the call chain to `method2`

This method might throw an `Exception` (this is a *checked* type)

Continues the call chain to `method3`

Fig. 11.7 | `Throwable` methods `getMessage`, `getStackTrace` and `printStackTrace`. (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
```

This method might throw an `Exception` (this is a *checked* type)

Throws a new `Exception` and begins stack unwinding

```
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
```

Shows just the error message that was stored in the `Exception` object

Shows the complete error message and stack trace

Shows the stack trace information obtained from `StackTraceElements`

Fig. 11.7 | `Throwable` methods `getMessage`, `getStackTrace` and `printStackTrace`. (Part 3 of 3.)

`printStackTrace`, `getStackTrace` and `getMessage` (Cont.)

- An exception's `getStackTrace` method obtains the stack-trace information as an array of `StackTraceElement` objects.
  - `StackTraceElement`'s methods `getClassName`, `getFileName`, `getLineNumber` and `getMethodName` get the class name, file name, line number and method name, respectively, for that `StackTraceElement`.
- Each `StackTraceElement` represents one method call on the method-call stack.

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.


```
1  // Fig. 11.8: 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
```

Catches the chained exception and displays the stack trace

Fig. 11.8 | Chained exceptions. (Part 1 of 3.)

```
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
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
```

Creates a new exception with a custom message; chains the exception thrown by `method2`

Creates a new exception with a custom message; chains the exception thrown by `method3`

Fig. 11.8 | Chained exceptions. (Part 2 of 3.)


```
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
```

Original 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
```

Notice that the chained exceptions appear in the stack trace information

Fig. 11.8 | Chained exceptions. (Part 3 of 3.)

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.

Declaring New Exception Types

- 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.