Java Programming

Exception handling

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).

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 to debug the program.

Exception handling

- `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.

Exception handling

- Following keywords are used in Exceptional Handling

  - try
  - catch
  - finally
  - throws
  - throw

Example: Divide by Zero without Exception Handling

- 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`.

Example: Divide by Zero without Exception Handling

- 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`.

Example: Divide by Zero without Exception Handling

```java
// Fig. 11.1: DivideByZeroNoExceptionHandling.java
// Integer division without exception handling.
import java.util.Scanner;

public class DivideByZeroNoExceptionHandling
{
   // demonstrates throwing an exception when a divide-by-zero occurs
   public static int quotient( int numerator, int denominator )
   {
      return numerator / denominator; // possible division by zero
   } // end method quotient

```

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

Example: Divide by Zero without Exception Handling

```java
    public static void main( String[] args )
    {
       Scanner scanner = new Scanner( System.in ); // scanner for input

       System.out.print( "Please enter an integer numerator: " );
       int numerator = scanner.nextInt();
       System.out.print( "Please enter an integer denominator: " );
       int denominator = scanner.nextInt();

       int result = quotient( numerator, denominator );
       System.out.printf(
          "\nResult: %d / %d = %d\n", numerator, denominator, result );
    } // end main
 } // 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.)

Example: Divide by Zero without Exception Handling

```
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.)

Example: Handling `ArithmeticExceptions` and `InputMismatchExceptions`

- The application in Fig. 11.2 uses exception handling to process any `ArithmeticException`s and `InputMistmatchException`s that arise.

- If the user makes a mistake, the program catches and handles (i.e., deals with) the exception—in this case, allowing the user to try to enter the input again.

Example: Handling ArithmeticExceptions and InputMismatchExceptions

```java
// Fig. 11.2: DivideByZeroWithExceptionHandling.java
// Handling ArithmeticExceptions and InputMismatchExceptions.
import java.util.InputMismatchException;
import java.util.Scanner;

public class DivideByZeroWithExceptionHandling
{
   // demonstrates throwing an exception when a divide-by-zero occurs
   public static int quotient( int numerator, int denominator )
      throws ArithmeticException
   {
      return numerator / denominator; // possible division by zero
   } // end method quotient

   public static void main( String[] args )
   {
      Scanner scanner = new Scanner( System.in ); // scanner for input
      boolean continueLoop = true; // determines if more input is needed

```

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

Example: Handling ArithmeticExceptions and InputMismatchExceptions

```java
do
{
   try // read two numbers and calculate quotient
   {
      System.out.print( "Please enter an integer numerator: " );
      int numerator = scanner.nextInt();
      System.out.print( "Please enter an integer denominator: " );
      int denominator = scanner.nextInt();

      int result = quotient( numerator, denominator );
      System.out.printf( "\nResult: %d / %d = %d\n", numerator,
         denominator, result );
      continueLoop = false; // input successful; end looping
   } // end try
   catch ( InputMismatchException inputMismatchException )
   {
      System.err.printf( "\nException: %s\n",
         inputMismatchException );
      scanner.nextLine(); // discard input so user can try again
      System.out.println(
         "You must enter integers. Please try again.\n" );
   } // end catch
```

Fig. 11.2 | Handling ArithmeticExceptions and InputMismatchExceptions. (Part 2 of 4.)

Example: Handling ArithmeticExceptions and InputMismatchExceptions

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

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

Result: 100 / 7 = 14
```

Fig. 11.2 | Handling ArithmeticExceptions and InputMismatchExceptions. (Part 3 of 4.)

Example: Handling ArithmeticExceptions and InputMismatchExceptions

```
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.)

Example: Handling `ArithmeticExceptions` and `InputMismatchExceptions`

- `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.
- `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.

Example: Handling `ArithmeticExceptions` and `InputMismatchExceptions`

- 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.
- 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

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

Example: Handling `ArithmeticExceptions` and `InputMismatchExceptions`

- `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.

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

- 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 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`.

Java Exception Hierarchy

- 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.

Java Exception Hierarchy

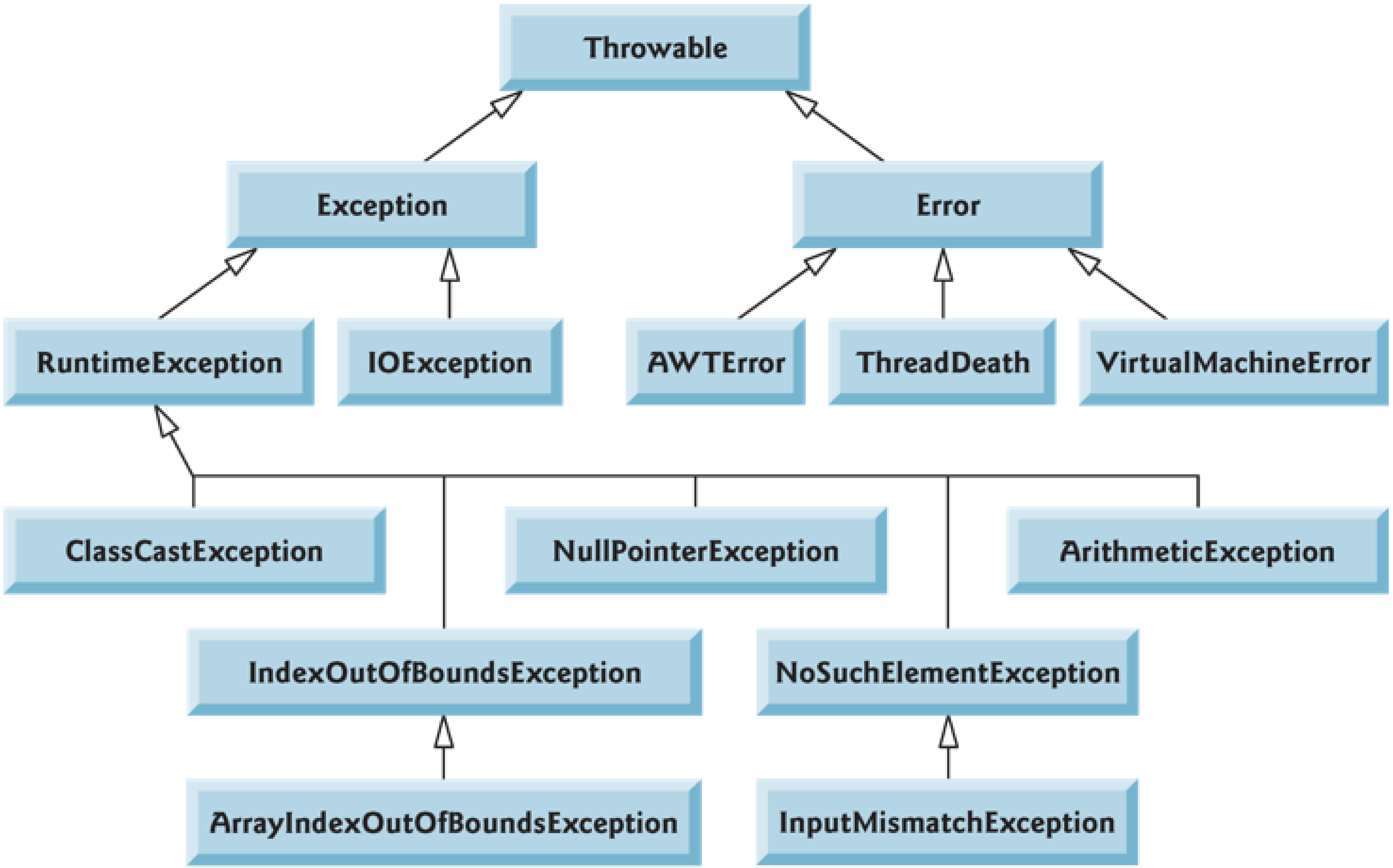

Fig. 11.3 | Portion of class `Throwable`'s inheritance hierarchy.

`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.

`finally` Block

- `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

- 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.

`finally` Block

```java
// Fig. 11.4: UsingExceptions.java
// try...catch...finally exception handling mechanism.

public class UsingExceptions
{
   public static void main( String[] args )
   {
      try
      {
         throwException(); // call method throwException
      } // end try
      catch ( Exception exception ) // exception thrown by throwException
      {
         System.err.println( "Exception handled in main" );
      } // end catch

      doesNotThrowException();
   } // end main

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

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

finally Block

```java
      try // throw an exception and immediately catch it
      {
         System.out.println( "Method throwException" );
         throw new Exception(); // generate exception
      } // end try
      catch ( Exception exception ) // catch exception thrown in try
      {
         System.err.println(
            "Exception handled in method throwException" );
         throw exception; // rethrow for further processing

         // code here would not be reached; would cause compilation errors

      } // end catch
      finally // executes regardless of what occurs in try...catch
      {
         System.err.println( "Finally executed in throwException" );
      } // end finally

      // code here would not be reached; would cause compilation errors

   } // end method throwException

```

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

`finally` Block

```java
   // demonstrate finally when no exception occurs
   public static void doesNotThrowException()
   {
      try // try block does not throw an exception
      {
         System.out.println( "Method doesNotThrowException" );
      } // end try
      catch ( Exception exception ) // does not execute
      {
         System.err.println( exception );
      } // end catch
      finally // executes regardless of what occurs in try...catch
      {
         System.err.println(
            "Finally executed in doesNotThrowException" );
      } // end finally

      System.out.println( "End of method doesNotThrowException" );
   } // end method doesNotThrowException
} // end class UsingExceptions
```

Fig. 11.4 | `try`...`catch`...`finally` exception-handling mechanism. (Part 3 of 4.)

`finally` Block

```
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.)

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.

Stack Unwinding and Obtaining Information from an Exception Object

```java
// Fig. 11.5: UsingExceptions.java
// Stack unwinding and obtaining data from an exception object.

public class UsingExceptions
{
   public static void main( String[] args )
   {
      try
      {
         method1(); // call method1
      } // end try
      catch ( Exception exception ) // catch exception thrown in method1
      {
         System.err.printf( "%s\n\n", exception.getMessage() );
         exception.printStackTrace(); // print exception stack trace

         // obtain the stack-trace information
         StackTraceElement[] traceElements = exception.getStackTrace();

         System.out.println( "\nStack trace from getStackTrace:" );
         System.out.println( "Class\t\tFile\t\t\tLine\tMethod" );

```

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

Stack Unwinding and Obtaining Information from an Exception Object

```java
        // loop through traceElements to get exception description
        for ( StackTraceElement element : traceElements )
        {
           System.out.printf( "%s\t", element.getClassName() );
           System.out.printf( "%s\t", element.getFileName() );
           System.out.printf( "%s\t", element.getLineNumber() );
           System.out.printf( "%s\n", element.getMethodName() );
        } // end for
     } // end catch
   } // end main

   // call method2; throw exceptions back to main
   public static void method1() throws Exception
   {
      method2();
   } // end method method1

   // call method3; throw exceptions back to method1
   public static void method2() throws Exception
   {
      method3();
   } // end method method2
```

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

Stack Unwinding and Obtaining Information from an Exception Object

```java

    // throw Exception back to method2
    public static void method3() throws Exception
    {
       throw new Exception( "Exception thrown in method3" );
    } // end method method3
 } // 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.)

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.

- 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.