Week 3 – Exception Handling

Outline
3.1 Introduction
3.2 When Exception Handling Should Be Used
3.3 Other Error-Handling Techniques
3.4 Basics of Java Exception Handling
3.5 try Blocks
3.6 Throwing an Exception
3.7 Catching an Exception
3.8 Exception-Handling Example: Divide by Zero
3.9 Rethrowing an Exception
3.10 throws Clause
3.11 Constructors, Finalizers and Exception Handling
3.12 Exceptions and Inheritance
3.13 finally Block
3.14 Using printStackTrace and getMessage
3.1 Introduction

• Exception handling
  – Exception
    • Indication of problem during execution
      – E.g., array out of bounds
    – Handles errors
  – Class Exception
3.2 When Exception Handling Should Be Used

• Uses of exception handling
  – When method cannot complete its task
  – Process exceptions from program components
  – Handle exceptions in a uniform manner in large projects
3.3 Other Error-Handling Techniques

- Using no error-handling
  - Not for mission critical applications
- Exit application on error
  - Program must return resources
3.4 Basics of Java Exception Handling

• A method detects an error and throws an exception
  – Exception handler processes the error
    • The error is considered caught and handled in this model
• Code that could generate errors put in try blocks
  – Code for error handling enclosed in a catch block
  – The finally always executes with or without an error
• Keyword throws tells exceptions of a method
• Termination model of exception handling
  – The block in which the exception occurs expires
3.5 try Blocks

- The **try** block structure
  ```java
  try {
  statements that may throw an exception
  }
  catch ( ExceptionType exceptionReference ) {
  statements to process an exception
  }
  ```

- A **try** followed by any number of **catch** blocks
3.6 Throwing an Exception

• The **throw** statement
  – Indicates an exception has occurred
  – Operand any class derived from **Throwable**

• **Subclasses of** **Throwable**
  – Class **Exception**
    • Problems that should be caught
  – Class **Error**
    • Serious exception should not be caught

• **Control moves from** **try** block to catch **block**
3.7 Catching an Exception

- Handler catches exception
  - Executes code in `catch` block
  - Should only catch `Exceptions`
- Program terminates if no appropriate handler
- Single `catch` can handle multiple exceptions
- Many ways to write exception handlers
3.8 Exception-Handling Example: Divide by Zero

- Common programming mistake
- Throws `ArithmeticException`
// Fig. 3.1: DivideByZeroException.java
// Definition of class DivideByZeroException.
// Used to throw an exception when a
divide-by-zero is attempted.
public class DivideByZeroException extends ArithmeticException {

    // no-argument constructor specifies default error message
    public DivideByZeroException()
    {
        super( "Attempted to divide by zero" );
    }

    // constructor to allow customized error message
    public DivideByZeroException( String message )
    {
        super( message );
    }
}

// end class DivideByZeroException
// Fig. 3.2: DivideByZeroTest.java
// A simple exception handling example.
// Checking for a divide-by-zero-error.

// Java core packages
import java.awt.*;
import java.awt.event.*;
import java.text.DecimalFormat;

// Java extension packages
import javax.swing.*;

public class DivideByZeroTest extends JFrame
    implements ActionListener {

    private JTextField inputField1, inputField2, outputField;
    private int number1, number2;
    private double result;

    // set up GUI
    public DivideByZeroTest()
    {
        super( "Demonstrating Exceptions" );

        // get content pane and set its layout
        Container container = getContentPane();
        container.setLayout( new GridLayout( 3, 2 ) );

        // set up label and inputField1
        container.add(
            new JLabel( "Enter numerator ", SwingConstants.RIGHT ) );
        inputField1 = new JTextField( 10 );
        container.add( inputField1 );
```java
// set up label and inputField2; register listener
container.add(
    new JLabel( "Enter denominator and press Enter ",
        SwingConstants.RIGHT ) );
inputField2 = new JTextField( 10 );
container.add( inputField2 );
inputField2.addActionListener( this );

// set up label and outputField
container.add(
    new JLabel( "RESULT ", SwingConstants.RIGHT ) );
outputField = new JTextField();
container.add( outputField );

setSize( 425, 100 );
setVisible( true );

// process GUI events
public void actionPerformed( ActionEvent event )
{
    DecimalFormat precision3 = new DecimalFormat( "0.000" );
    outputField.setText( "" );   // clear outputField

    // read two numbers and calculate quotient
    try {
        number1 = Integer.parseInt( inputField1.getText() );
        number2 = Integer.parseInt( inputField2.getText() );

        result = quotient( number1, number2 );
        outputField.setText( precision3.format( result ) );
    }
}
```
// process improperly formatted input
try {
    String input = scanner.nextLine();
    int numerator = parseInt(input);
    int denominator = parseInt(input);
    if (denominator == 0) {
        throw new DivideByZeroException();
    }
    return (double) numerator / denominator;
} catch (NumberFormatException numberFormatException) {
    JOptionPane.showMessageDialog(this, "You must enter two integers", "Invalid Number Format", JOptionPane.ERROR_MESSAGE);
} catch (ArithmeticException arithmeticException) {
    JOptionPane.showMessageDialog(this, arithmeticException.toString(), "Arithmetic Exception", JOptionPane.ERROR_MESSAGE);
}

} // end method quotient

// method quotient demonstrated throwing an exception
// when a divide-by-zero error occurs
public double quotient( int numerator, int denominator )
    throws DivideByZeroException
{   if ( denominator == 0 )
    throw new DivideByZeroException();
    return (double) numerator / denominator;
}
97 // execute application
98 public static void main( String args[] )
99 {
100 DivideByZeroTest application = new DivideByZeroTest();
101 application.setDefaultCloseOperation(
102 JFrame.EXIT_ON_CLOSE );
103 }
104 }
105 // end class DivideByZeroTest

Program Output

Demonstrating Exceptions
Enter numerator 100
Enter denominator and press Enter 7
RESULT 14.286

Demonstrating Exceptions
Enter numerator 100
Enter denominator and press Enter hello
RESULT

Demonstrating Exceptions
Enter numerator 100
Enter denominator and press Enter 0
RESULT

Arithmetic Exception
DivideByZeroException: Attempted to divide by zero
OK

© 2002 Prentice Hall, Inc.
All rights reserved.
3.9 Rethrowing an Exception

• Rethrow exception if `catch` cannot handle it
3.10 throws Clause

• Lists the exceptions thrown by a method

```java
int functionName ( paramterList )
  throws ExceptionType1, ExceptionType2,…
{
  // method body
}
```

• **RuntimeExceptions** occur during execution
  – `ArrayIndexOutOfBoundsException`
  – `NullPointerException`

• Declare exceptions a method throws
3.11 Constructors, Finalizers, and Exception Handling

- Throw exception if constructor causes error
- **Finalize** called when object garbage collected
3.12 Exceptions and Inheritance

• Inheritance of exception classes
  – Allows polymorphic processing of related exceptions
3.13 finally Block

• Resource leak
  – Caused when resources are not released by a program

• The `finally` block
  – Appears after `catch` blocks
  – Always executes
  – Use to release resources
```java
// Fig. 3.9: UsingExceptions.java
// Demonstration of the try-catch-finally
// exception handling mechanism.
public class UsingExceptions {

    // execute application
    public static void main( String args[] )
    {
        // call method throwException
        try { throwException(); }

        // catch Exceptions thrown by method throwException
        catch ( Exception exception )
        {
            System.err.println( "Exception handled in main" );
        }

        doesNotThrowException();
    }

    // demonstrate try/catch/finally
    public static void throwException() throws Exception
    {
        // throw an exception and immediately catch it
        try {
            System.out.println( "Method throwException" );
            throw new Exception(); // generate exception
        }
    }

    // method does not throw an exception...
    private static void doesNotThrowException()
    {
        System.out.println( "Method doesNotThrowException" );
    }
}
```
// catch exception thrown in try block

```java
32   // catch exception thrown in try block
33   catch ( Exception exception )
34   {
35       System.err.println(
36           "Exception handled in method throwException" );
37       throw exception;  // rethrow for further processing
38   }
39   // any code here would not be reached
40   
41   // this block executes regardless of what occurs in
42   // try/catch
43   finally {
44       System.err.println(
45           "Finally executed in throwException" );
46   }
47   // any code here would not be reached
48   
49   // demonstrate finally when no exception occurs
50   public static void doesNotThrowException()
51   {
52       // try block does not throw an exception
53       try {
54           System.out.println( "Method doesNotThrowException" );
55       }
56   }  // catch does not execute, because no exception thrown
57   catch( Exception exception )
58   {
59       System.err.println( exception.toString() );
60   }
61   // any code here would not be reached
62   
63   // this block executes regardless of what occurs in
64   // try/catch
65   finally {
66       System.err.println( "Finally executed in doesNotThrowException" );
67   }
68   // any code here would not be reached
69   
70   // demonstrate finally when no exception occurs
71   public static void doesNotThrowException()
72   {
73       // try block does not throw an exception
74       try {
75           System.out.println( "Method doesNotThrowException" );
76       }  // catch does not execute, because no exception thrown
77       catch( Exception exception )
78       {
79           System.err.println( exception.toString() );
80       }
81   }  // any code here would not be reached
82   
83   // finally block executes, even though Exception thrown
```

© 2002 Prentice Hall, Inc.
All rights reserved.
// this block executes regardless of what occurs in
// try/catch
finally {
    System.err.println(
        "Finally executed in doesNotThrowException" );
}

System.out.println(
    "End of method doesNotThrowException" );

} // end class UsingExceptions

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 !
```java
public class UsingExceptions {

    // execute application
    public static void main( String args[] )
    {
        // call throwException to demonstrate stack unwinding
        try {
            throwException();
        }

        // catch exception thrown in throwException
        catch ( Exception exception ) {
            System.err.println( "Exception handled in main" );
        }
    }
}
```
public static void throwException() throws Exception {
   // throw an exception and catch it in main
   try {
      System.out.println( "Method throwException" );
      throw new Exception();      // generate exception
   }

   // catch is incorrect type, so Exception not caught
   catch( RuntimeException runtimeException ) {
      System.err.println( "Exception handled in method throwException" );
   }

   // finally block always executes
   finally {
      System.err.println( "Finally is always executed" );
   }
}

19   // throwException throws an exception that is not caught in
20   // the body of this method
21   public static void throwException() throws Exception {
22   // throw an exception and catch it in main
23   try {
24   System.out.println( "Method throwException" );
25   throw new Exception();      // generate exception
26   }
27
28   // catch is incorrect type, so Exception not caught
29   catch( RuntimeException runtimeException ) {
30   System.err.println( "Exception handled in method throwException" );
31   }
32
33   // finally block always executes
34   finally {
35   System.err.println( "Finally is always executed" );
36   }
37
38   }  // end class UsingExceptions
39
40
41   }  // end class UsingExceptions

Method throwException
Finally is always executed
Exception handled in main
3.14 Using `printStackTrace` and `getMessage`

- Method `printStackTrace`
  - Prints the method call stack
- `Throwable` class
  - Method `getMessage` retrieves `informationString`
```java
// Fig. 3.11: UsingExceptions.java
// Demonstrating the getMessage and printStackTrace
// methods inherited into all exception classes.
public class UsingExceptions {

    // execute application
    public static void main( String args[] )
    {
        // call method1
        try {
            method1();
        }

        // catch Exceptions thrown from method1
        catch ( Exception exception ) {
            System.err.println( exception.getMessage() + "\n" );
            exception.printStackTrace();
        }

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

        // call method3; throw exceptions back to method1
        public static void method2() throws Exception
        {
            method3();
        }
    }
}
```
33    // throw Exception back to method2
34    public static void method3() throws Exception
35    {
36        throw new Exception( "Exception thrown in method3" );
37    }
38
39    } // end class Using Exceptions

Exception thrown in method3

java.lang.Exception: Exception thrown in method3
    at UsingExceptions.method3(UsingExceptions.java:36)
    at UsingExceptions.method2(UsingExceptions.java:30)
    at UsingExceptions.method1(UsingExceptions.java:24)
    at UsingExceptions.main(UsingExceptions.java:11)