

Module 8

Exception Handling

Adapted from Absolute Java, Rose Williams, *Binghamton University*

Introduction to Exception Handling

- Sometimes the best outcome can be when nothing unusual happens
- However, the case where exceptional things happen must also be prepared for
 - Java exception handling facilities are used when the invocation of a method may cause something exceptional to occur

Introduction to Exception Handling

- Java library software (or programmer-defined code) provides a mechanism that signals when something unusual happens
 - This is called *throwing an exception*
- In another place in the program, the programmer must provide code that deals with the exceptional case
 - This is called *handling the exception*

try-throw-catch Mechanism

- The basic way of handling exceptions in Java consists of the *try-throw-catch* trio
- The **try** block contains the code for the basic algorithm
 - It tells what to do when everything goes smoothly
- It is called a **try** block because it "tries" to execute the case where all goes as planned
 - It can also contain code that throws an exception if something unusual happens

```
try
{
    CodeThatMayThrowAnException
}
```

try-throw-catch Mechanism

`throw new`

`ExceptionClassName(PossiblySomeArguments) ;`

- When an exception is thrown, the execution of the surrounding `try` block is stopped
 - Normally, the flow of control is transferred to another portion of code known as the `catch` block
- The value thrown is the argument to the `throw` operator, and is always an object of some exception class
 - The execution of a `throw` statement is called *throwing an exception*

try-throw-catch Mechanism

- A **throw** statement is similar to a method call:

```
throw new ExceptionClassName(SomeString) ;
```

- In the above example, the object of class ***ExceptionClassName*** is created using a string as its argument
 - This object, which is an argument to the **throw** operator, is the exception object thrown
- Instead of calling a method, a **throw** statement calls a **catch** block

try-throw-catch Mechanism

- When an exception is thrown, the **catch** block begins execution
 - The **catch** block has one parameter
 - The exception object thrown is plugged in for the **catch** block parameter
- The execution of the **catch** block is called *catching the exception*, or *handling the exception*
 - Whenever an exception is thrown, it should ultimately be handled (or caught) by some **catch** block

try-throw-catch Mechanism

```
catch(Exception e)
{
    ExceptionHandlingCode
}
```

- A **catch** block looks like a method definition that has a parameter of type *Exception* class
 - It is not really a method definition, however
- A **catch** block is a separate piece of code that is executed when a program encounters and executes a **throw** statement in the preceding **try** block
 - A **catch** block is often referred to as an *exception handler*
 - It can have at most one parameter

try-throw-catch Mechanism

```
catch(Exception e) { . . . }
```

- The identifier **e** in the above **catch** block heading is called the **catch** block parameter
- The **catch** block parameter does two things:
 1. It specifies the type of thrown exception object that the **catch** block can catch (e.g., an **Exception** class object above)
 2. It provides a name (for the thrown object that is caught) on which it can operate in the **catch** block
 - Note: The identifier **e** is often used by convention, but any non-keyword identifier can be used

try-throw-catch Mechanism

- When a **try** block is executed, two things can happen:
 1. No exception is thrown in the **try** block
 - The code in the **try** block is executed to the end of the block
 - The **catch** block is skipped
 - The execution continues with the code placed after the **catch** block

try-throw-catch Mechanism

2. An exception is thrown in the **try** block and caught in the **catch** block
 - The rest of the code in the **try** block is skipped
 - Control is transferred to a following **catch** block (in simple cases)
 - The thrown object is plugged in for the **catch** block parameter
 - The code in the **catch** block is executed
 - The code that follows that **catch** block is executed (if any)

Exception Classes

- There are more exception classes than just the single class **Exception**
 - There are more exception classes in the standard Java libraries
 - New exception classes can be defined like any other class
- All predefined exception classes have the following properties:
 - There is a constructor that takes a single argument of type **String**
 - The class has an accessor method **getMessage** that can recover the string given as an argument to the constructor when the exception object was created
- All programmer-defined classes should have the same properties

Exception Classes from Standard Packages

- Numerous predefined exception classes are included in the standard packages that come with Java

- For example:

`IOException`

`NoSuchMethodException`

`FileNotFoundException`

`NumberFormatException`

- Many exception classes must be imported in order to use them

```
import java.io.IOException;
```

Exception Classes from Standard Packages

- The predefined exception class `Exception` is the root class for all exceptions
 - Every exception class is a descendent class of the class `Exception`
 - Although the `Exception` class can be used directly in a class or program, it is most often used to define a derived class
 - The class `Exception` is in the `java.lang` package, and so requires no `import` statement

Using the `getMessage` Method

```
. . . // method code
try
{
    . . .
    throw new Exception(StringArgument) ;
    . . .
}
catch (Exception e)
{
    String message = e.getMessage() ;
    System.out.println(message) ;
    System.exit(0) ;
} . . .
```

Using the `getMessage` Method

- Every exception has a `String` instance variable that contains some message
 - This string typically identifies the reason for the exception
- In the previous example, `StringArgument` is an argument to the `Exception` constructor
- This is the string used for the value of the string instance variable of exception `e`
 - Therefore, the method call `e.getMessage()` returns this string

Multiple `catch` Blocks

- A `try` block can potentially throw any number of exception values, and they can be of differing types
 - In any one execution of a `try` block, at most one exception can be thrown (since a `throw` statement ends the execution of the `try` block)
 - However, different types of exception values can be thrown on different executions of the `try` block

Multiple `catch` Blocks

- Each `catch` block can only catch values of the exception class type given in the `catch` block heading
- Different types of exceptions can be caught by placing more than one `catch` block after a `try` block
 - Any number of `catch` blocks can be included, but they must be placed in the correct order

Catch the More Specific Exception First

- When catching multiple exceptions, the order of the `catch` blocks is important
 - When an exception is thrown in a `try` block, the `catch` blocks are examined in order
 - The first one that matches the type of the exception thrown is the one that is executed

Catch the More Specific Exception First

```
catch (Exception e)
{ . . . }
catch (NegativeNumberException e)
{ . . . }
```

- Because a **NegativeNumberException** is a type of **Exception**, all **NegativeNumberExceptions** will be caught by the first **catch** block before ever reaching the second block
 - The catch block for **NegativeNumberException** will never be used!
- For the correct ordering, simply reverse the two blocks

Throwing an Exception in a Method

- Sometimes it makes sense to throw an exception in a method, but not catch it in the same method
 - Some programs that use a method should just end if an exception is thrown, and other programs should do something else
 - In such cases, the program using the method should enclose the method invocation in a **try** block, and catch the exception in a **catch** block that follows
- In this case, the method itself would not include **try** and **catch** blocks
 - However, it would have to include a **throws clause**

Declaring Exceptions in a **throws** Clause

- If a method can throw an exception but does not catch it, it must provide a warning
 - This warning is called a *throws clause*
 - The process of including an exception class in a throws clause is called *declaring the exception*

`throws AnException //throws clause`

- The following states that an invocation of `aMethod` could throw `AnException`

```
public void aMethod() throws AnException
```

Declaring Exceptions in a **throws** Clause

- If a method can throw more than one type of exception, then separate the exception types by commas

```
public void aMethod() throws  
    AnException, AnotherException
```

- If a method throws an exception and does not catch it, then the method invocation ends immediately

The Catch or Declare Rule

- Most ordinary exceptions that might be thrown within a method **must** be accounted for in one of two ways:
 1. The code that can throw an exception is placed within a **try** block, and the possible exception is caught in a **catch** block within the same method
 2. The possible exception can be declared at the start of the method definition by placing the exception class name in a **throws** clause

The Catch or Declare Rule

- The first technique handles an exception in a **catch** block
- The second technique is a way to shift the exception handling responsibility to the method that invoked the exception throwing method
- The invoking method must handle the exception, unless it too uses the same technique to pass it up
- Ultimately, every exception that is thrown should eventually be caught by a **catch** block in some method that does not just declare the exception class in a **throws** clause

The Catch or Declare Rule

- In any one method, both techniques can be mixed
 - Some exceptions may be caught, and others may be declared in a **throws** clause
- However, these techniques must be used consistently with a given exception
 - If an exception is not declared, then it must be handled within the method
 - If an exception is declared, then the responsibility for handling it is shifted to some other calling method
 - Note that if a method definition encloses an invocation of a second method, and the second method can throw an exception and does not catch it, then the first method must catch or declare it

What Happens If an Exception is Never Caught?

- If every method up to and including the main method simply includes a **throws** clause for an exception, that exception may be thrown but never caught
 - In a GUI program (i.e., a program with a windowing interface), nothing happens - but the user may be left in an unexplained situation, and the program may no longer be reliable
 - In non-GUI programs, this causes the program to terminate with an error message giving the name of the exception class
- Every well-written program should eventually catch every exception by a **catch** block in some method

Nested `try-catch` Blocks

- It is possible to place a `try` block and its following catch blocks inside a larger `try` block, or inside a larger `catch` block
 - If a set of `try-catch` blocks are placed inside a larger `catch` block, different names must be used for the `catch` block parameters in the inner and outer blocks, just like any other set of nested blocks
 - If a set of `try-catch` blocks are placed inside a larger `try` block, and an exception is thrown in the inner `try` block that is not caught, then the exception is thrown to the outer `try` block for processing, and may be caught in one of its `catch` blocks

The **finally** Block

- The **finally** block contains code to be executed whether or not an exception is thrown in a **try** block
 - If it is used, a **finally** block is placed after a **try** block and its following **catch** blocks

```
try
{ . . . }
catch (ExceptionClass1 e)
{ . . . }
. . .
catch (ExceptionClassN e)
{ . . . }
finally
{
    CodeToBeExecutedInAllCases
}
```

The **finally** Block

- If the **try-catch-finally** blocks are inside a method definition, there are three possibilities when the code is run:
 1. The **try** block runs to the end, no exception is thrown, and the **finally** block is executed
 2. An exception is thrown in the **try** block, caught in one of the **catch** blocks, and the **finally** block is executed
 3. An exception is thrown in the **try** block, there is no matching **catch** block in the method, the **finally** block is executed, and then the method invocation ends and the exception object is thrown to the enclosing method

Exception Handling with the `Scanner` Class

- The `nextInt` method of the `Scanner` class can be used to read `int` values from the keyboard
- However, if a user enters something other than a well-formed `int` value, an `InputMismatchException` will be thrown
 - Unless this exception is caught, the program will end with an error message
 - If the exception is caught, the `catch` block can give code for some alternative action, such as asking the user to reenter the input

The

InputMismatchException

- The `InputMismatchException` is in the standard Java package `java.util`
 - A program that refers to it must use an `import` statement, such as the following:

```
import java.util.InputMismatchException;
```

Exception Controlled Loops

- Sometimes it is better to simply loop through an action again when an exception is thrown, as follows:

```
boolean done = false;
while (! done)
{
    try
    {
        CodeThatMayThrowAnException
        done = true;
    }
    catch (SomeExceptionClass e)
    {
        SomeMoreCode
    }
}
```

An Exception Controlled Loop (Part 1 of 3)

Display 9.11 An Exception Controlled Loop

```
1 import java.util.Scanner;
2 import java.util.InputMismatchException;

3 public class InputMismatchExceptionDemo
4 {
5     public static void main(String[] args)
6     {
7         Scanner keyboard = new Scanner(System.in);
8         int number = 0; //to keep compiler happy
9         boolean done = false;
```

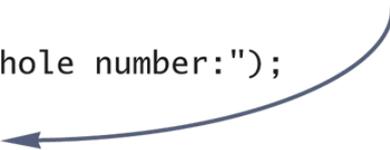
(continued)

An Exception Controlled Loop (Part 2 of 3)

Display 9.11 An Exception Controlled Loop

```
10     while (! done)
11     {
12         try
13         {
14             System.out.println("Enter a whole number:");
15             number = keyboard.nextInt();
16             done = true;
17         }
18         catch(InputMismatchException e)
19         {
20             keyboard.nextLine();
21             System.out.println("Not a correctly written whole number.");
22             System.out.println("Try again.");
23         }
24     }
25     System.out.println("You entered " + number);
26 }
27 }
```

If nextInt throws an exception, the try block ends and so the boolean variable done is not set to true.



(continued)

An Exception Controlled Loop (Part 3 of 3)

Display 9.11 An Exception Controlled Loop

SAMPLE DIALOGUE

```
Enter a whole number:  
forty two  
Not a correctly written whole number.  
Try again.  
Enter a whole number:  
fortytwo  
Not a correctly written whole number.  
Try again.  
Enter a whole number:  
42  
You entered 42
```

Re-throwing an Exception

- A **catch** block can contain code that throws an exception
 - Sometimes it is useful to catch an exception and then, depending on the string produced by **getMessage** (or perhaps something else), throw the same or a different exception for handling further up the chain of exception handling blocks