

Chapter 8

Exception Handling

- Basic Exception Handling
 - » the mechanics of exceptions
- Defining and Using Exceptions
 - » some "simple" cases
- Reality Check
 - » guidelines for more realistic situations

Exception Handling Overview

- A way of organizing a program into sections for the normal case and the exceptional case
 - » exception examples:
division by zero
incorrect type of input
- A way of implementing programs incrementally
 - » write and debug the code for normal operation first
 - » add code for the exceptional case later
- Simplifies development, testing, debugging and maintenance
 - » errors are easier to isolate

Warning:

- The example programs in this chapter are simplified for instructional purposes.
- Real programs are more complicated and usually have a somewhat different organization.
- More about this later, after the mechanics of exception handling, their definition and use have been explained

Some Terminology

- *Throwing an exception*: either Java itself or your code signals when something unusual happens
- *Catching an exception*: responding to an exception by executing a part of the program specifically written for the exception
 - » also called *handling an exception*
- The normal case is handled in a `try` block
- The exceptional case is handled in a `catch` block
- The catch block takes a parameter of type `Exception`
 - » it is called the *catch-block parameter*
 - » `e` is a commonly used name for it
- If an exception is *thrown*, execution in the `try` block ends and control passes to the `catch` block(s) after the `try` block

try-throw-catch Threesome

Basic code organization:

```
try
{
    <code to try>
    if(test condition)
        throw new Exception("Message to display");
    <more code>
}
catch(Exception e)
{
    <exception handling code>
}

<possibly more code>
```

try-throw-catch Program Flow

Try block

Statements execute up to the conditional `throw` statement

If the condition is `true` the exception is thrown

- » control passes to the `catch` block(s) after the `try` block

Else the condition is `false`

- » the exception is not thrown
- » the remaining statements in the `try` block (those following the conditional throw) are executed

Catch block

Executes if an exception is thrown

- » may terminate execution with `exit` statement
- » if it does not exit, execution resumes after the `catch` block

Statements after the Catch block

Executes if either the exception is not thrown or if it is thrown but the `catch` block does not exit

An Example of Exception Handling

ExceptionDemo
try and catch blocks

try block

throw statement

catch block

```
try
{
    System.out.println("Enter number of donuts:");
    donutCount = SavitchIn.readLineInt();

    System.out.println("Enter number of glasses of milk:");
    milkCount = SavitchIn.readLineInt();

    if (milkCount < 1)
        throw new Exception("Exception: No Milk!");

    donutsPerGlass = donutCount/(double)milkCount;
    System.out.println(donutCount + " donuts.");
    System.out.println(milkCount + " glasses of milk.");
    System.out.println("You have " + donutsPerGlass
        + " donuts for each glass of milk.");
}

catch(Exception e)
{
    System.out.println(e.getMessage());
    System.out.println("Go buy some milk.");
    System.out.println("Program aborted.");
    System.exit(0);
}
```

Flow of Control with Exception Handling

```
try
{
    System.out.println("Enter number of donuts:");
    donutCount = SavitchIn.readInt();
    System.out.println("Enter number of glasses of milk:");
    milkCount = SavitchIn.readLineInt();
    if (milkCount < 1)
        throw new Exception("Exception: No Milk!");
    donutsPerGlass = donutCount / (double)milkCount;
    System.out.println(donutCount + " donuts per glass");
    System.out.println(milkCount + " glasses of milk");
    System.out.println("You have " + donutsPerGlass + " donuts per glass");
}
catch (Exception e)
{
    System.out.println(e.getMessage());
    System.out.println("Go buy some milk.");
}
System.out.println("End of program.");
```

Assume user enters a positive number for number of glasses, so no exception is thrown.

Not executed when there's no exception thrown.

Main method from Exception-Demo program

Flow of Control with Exception Handling

```
try
{
    System.out.println("Enter number of donuts:");
    donutCount = SavitchIn.readInt();
    System.out.println("Enter number of glasses of milk:");
    milkCount = SavitchIn.readLineInt();
    if (milkCount < 1)
        throw new Exception("Exception: No Milk!");
    donutsPerGlass = donutCount / (double)milkCount;
    System.out.println(donutCount + " donuts.");
    System.out.println(milkCount + " glasses of milk.");
    System.out.println("You have " + donutsPerGlass);
}
catch (Exception e)
{
    System.out.println(e.getMessage());
    System.out.println("Go buy some milk.");
}
System.out.println("End of program.");
```

Assume user enters zero or a negative number for number of glasses, so an exception is thrown.

Not executed when an exception is thrown

Main method from Exception-Demo program

More about the **catch**-Block

- Although it may look similar to a method definition

The `catch`-block is **not** a method definition!

- Every `Exception` has a `getMessage` method
 - » it retrieves the string given to the exception object when it was thrown, e.g.

```
throw new Exception("This message is retrieved");
```

- A `catch`-block applies only to an immediately preceding `try` block
 - » if no exception is thrown, the catch block is ignored

Predefined Exception Classes

- `Exception` is the root class of all exceptions
- Many predefined classes throw exceptions
 - » the documentation or interface will tell you
 - » the exceptions thrown are often also predefined
- Some common predefined exceptions:
 - » `IOException`
 - » `ClassNotFoundException`, **and**
 - » `FileNotFoundException`

Code Organization when Using an Object that May Throw an Exception

```
Sample object = new SampleClass();
try
{
    <Possibly some code>
    object.doStuff();//may throw IOException
    <Possibly some more code>
}
catch(IOException e)
{
    <Code to handle the IOException, probably
    including this line:>
    System.out.println(e.getMessage());
}
```

- Predefined exceptions usually include a meaningful message that is retrieved with `getMessage`

ArrayIndexOutOfBoundsException

- thrown if your program attempts to use an array index that is out of bounds (too big or a negative number)
- does not need to be caught or accounted for in any way
- normally is not caught in a catch block
- functions more like a run-time error than a regular exception

Defining Your Own Exception Classes

```
public class DivideByZeroException extends Exception
{
    public DivideByZeroException()
    {
        super("Dividing by Zero!");
    }
    public DivideByZeroException(String message)
    {
        super(message);
    }
}
```

- Must be derived from some already-defined exception class
- Often the only methods you need to define are constructors.

Java Tip: Preserve `getMessage` When You Define Exception Classes

```
throw new Exception("This is a big exception!");
```

This string is stored in an instance variable in the exception object and is returned by the `getMessage` method.

To preserve the correct `getMessage` behavior in `Exception` classes that you define, include:

- a constructor with a string parameter that begins with a call to `super`

```
public DivideByZeroException(String message)
{
    super(message);
}
```
- a default constructor that passes a default message to the `super` constructor

```
public DivideByZeroException()
{
    super("Dividing by Zero!");
}
```

When to Define Your Own Exception Class

- When you use a `throw` statement in your code, you should usually define your own exception class.
- If you use a predefined, more general exception class, then your `catch`-block will have to be general.
- A general `catch`-block could also catch exceptions that should be handled somewhere else.
- A specific `catch`-block for your own exception class will catch the exceptions it should and pass others on.

Example: Using the **Divide- ByZero- Exception** Class

Excerpt from
DivideByZero-
ExceptionDemo

```
public void doIt()
{
    try
    {
        System.out.println("Enter numerator:");
        numerator = SavitchIn.readLineInt();
        System.out.println("Enter denominator:");
        denominator = SavitchIn.readLineInt();
        if (denominator == 0)
            throw new DivideByZeroException();
        quotient = numerator/denominator;
        System.out.println(numerator + "/"
            + denominator
            + " = " + quotient);
    }
    catch(DivideByZeroException e)
    {
        System.out.println(e.getMessage());
        secondChance();
    }
}
```

Passing the Buck – Declaring Exceptions

When defining a method you must include a `throws`-clause to declare any exception that might be thrown but is not caught in the method.

- Use a *throws-clause* to "pass the buck" to whatever method calls it (pass the responsibility for the `catch` block to the method that calls it)
 - » that method can also pass the buck, but eventually some method must catch it
- This tells other methods
"If you call me, you must handle any exceptions that I throw."

Example: **throws**-Clause

DoDivision

- It may throw a `DivideByZeroException` in the method `normal`
- But the `catch` block is in `main`
- So `normal` must include a *throws-clause* in the first line of the constructor definition:

```
public void normal() throws DivideByZeroException
{
    <statements to define the normal method>
}
```

More about Passing the Buck

Good programming practice:

Every exception thrown should eventually be caught in some method

- Normally exceptions are either caught in a `catch` block or *deferred* to the calling method in a `throws`-clause
- If a method throws an exception, the `catch` block must be in that method unless it is deferred by a `throws`-clause
 - » if the calling method also defers with a `throws`-clause, its calling program is expected to have the `catch` block, etc., up the line all the way to `main`, until a `catch` block is found

Uncaught Exceptions

- In any one method you can catch some exceptions and defer others
- If an exception is not caught in the method that throws it or any of its calling methods, either:
 - » the program ends, or,
 - » in the case of a GUI using Swing, the program may become unstable
- "Exceptions" derived from the classes `Error` and `RuntimeError` do not need a `catch` block or `throws`-clause
 - » they look like exceptions, but they are not descendants of `Exception`

throws-Clauses in Derived Classes

- You cannot add exceptions to the `throws`-clause of a redefined method in a derived class
 - » only exceptions in the `throws` -clause of the parent class's method can be in the `throws` -clause of the redefined method in the derived class
- In other words, you cannot throw any exceptions that are not either caught in a `catch` block or already listed in the `throws` -clause of the same method in the base class
- You can, however, declare fewer exceptions in the `throws` -clause of the redefined method

Multiple Exceptions and **catch** Blocks in a Method

- Methods can throw more than one exception
- The `catch` blocks immediately following the try block are searched in sequence for one that catches the exception type
 - » the first catch block that handles the exception type is the only one that executes
- Specific exceptions are derived from more general types
 - » both the specific and general types from which they are derived will handle exceptions of the more specific type
- So put the `catch` blocks for the more specific, derived, exceptions early and the more general ones later

Catch the more specific exception first.

Warning Revisited:

- As stated earlier, the example programs in this chapter are simplified for instructional purposes
 - » catch blocks are sometimes defined in the method that throws the exception
- Real programs are more complicated and usually have a somewhat different organization
 - » catch blocks are usually defined in a different method than the throw

MethodA throws
MyException
but defers
catching it (by
using a throws-
clause:

```
public void MethodA() throws MyException  
{  
    throw new MyException("Bla Bla Bla");  
}
```

Typical Program Organization for Exception Handling in Real Programs

MethodB, which
calls MethodA,
catches
MyException
exceptions:

```
public void MethodB()  
{  
    try  
    {  
        MethodA();//May throw MyException exception  
    }  
    catch(MyException e)  
    {  
        <statements to handle MyException exceptions>  
    }  
}
```

Exception Handling: Reality Check

- Exception handling can be overdone
 - » use it sparingly and only in certain ways
- If the way an exceptional condition is handled depends on how and where the method is invoked, then it is better to use exception handling and let the programmer handle the exception (by writing the catch block and choosing where to put it)
- Otherwise it is better to avoid throwing exceptions
- An example of this technique is shown in the case study **A Line-Oriented Calculator**

Case Study:

A Line-Oriented Calculator

- Preliminary version with no exception handling written first
- Exception when user enters unknown operator
- Three choices for handling exception:
 - » Catch the exception in the method `evaluate()` (where it is thrown)
 - » Declare `evaluate()` as throwing the exception and catch it in `doCalculation()`
 - » Declare both `evaluate()` and `doCalculation()` as throwing the exception and handle it in `main()`
- Asks user to re-enter the calculation, so it uses the third option
- Also includes an exception for division by zero

Calculator **evaluate** Method

```
public double evaluate(char op, double n1, double n2)
    throws DivideByZeroException, UnknownOpException
{
    double precision = 0.001;
    ...
    case '+':
        answer = n1 + n2;
        break;
        cases for - and * ops not shown here—see text for complete code
    ...
    case '/':
        if ((Math.abs(n2) < precision)
                throw new DivideByZeroException();
        answer = n1/n2;
        break;
    default:
        throw new UnknownOpException(op);
    ...
}
```

Any number smaller than **precision** is treated as zero because it is usually best to avoid using the equality operator with floating-point numbers.

Throws exception if **op** is any **char** other than **+, -, *, /**

Calculator `UnknownOpException` Class

```
public class UnknownOpException extends Exception
```

```
{  
    public UnknownOpException()  
    {  
        super("UnknownOpException");  
    }  
}
```

Provides an easy way to make the unknown op character part of the exception message.

```
public UnknownOpException(char op)  
{  
    super (op + " is an unknown operator.");  
}
```

```
public UnknownOpException(String message)  
{  
    super(message);  
}  
}
```

Note that all three constructors provide a way for the exception object to have a meaningful message.

The **finally** Block

At this stage of your programming you may not have much use for the `finally` block, but it is included for completeness - you may have find it useful later

- You can add a `finally` block after the `try/catch` blocks
- `finally` blocks execute whether or not `catch` block(s) execute
- Code organization using `finally` block:

```
try block
catch block
finally
{
    <Code to be executed whether or not an exception is thrown>
}
```

Three Possibilities for a **try-catch-finally** Block

- The `try`-block runs to the end and no exception is thrown.
 - » The `finally`-block runs after the `try`-block.
- An exception is thrown in the `try`-block and caught in the matching `catch`-block.
 - » The `finally`-block runs after the `catch`-block.
- An exception is thrown in the `try`-block and there is no matching `catch`-block.
 - » The `finally`-block is executed before the method ends.
 - » Code that is after the `catch`-blocks but not in a `finally`-block would not be executed in this situation.

Summary

Part 1

- An exception is an object descended from the `Exception` class
- Descendents of the `Error` class act like exceptions but are not
- Exception handling allows you to design code for the normal case separately from that for the exceptional case
- You can use predefined exception classes or define your own
- Exceptions can be thrown by:
 - » certain Java statements
 - » methods from class libraries
 - » explicit use of the `throw` statement
- An exception can be thrown in either
 - » a `try` block, or
 - » a method definition without a `try` block, but in this case the call to the method must be placed inside a `try` block

Summary

Part 2

- An exception is caught in a catch block
- When a method might throw an exception but does not have a `catch` block to catch it, usually the exception class must be listed in the `throws`-clause for the method
- A try block may be followed by more than one catch block
 - » more than one catch block may be capable of handling the exception
 - » the first catch block that can handle the exception is the only one that executes
 - » so put the most specific catch blocks first and the most general last
- Every exception class has a `getMessage` method to retrieve a text message description of the exception caught
- Do not overuse exceptions