

# JAVA OOP: EXCEPTION HANDLING\*

R.G. (Dick) Baldwin

This work is produced by OpenStax-CNX and licensed under the  
Creative Commons Attribution License 3.0<sup>†</sup>

## Abstract

Baldwin discusses and illustrates many of the details having to do with exception handling in Java.

## 1 Table of Contents

- Preface (p. 1)
  - Viewing tip (p. 1)
    - \* Figures (p. 1)
    - \* Listings (p. 2)
- Preview (p. 2)
- Discussion and sample code (p. 2)
- Summary (p. 26)
- What's next? (p. 26)
- Miscellaneous (p. 26)

## 2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

### 2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

#### 2.1.1 Figures

- Figure 1 (p. 5) . Throwable constructors.
- Figure 2 (p. 6) . Methods of the Throwable class.
- Figure 3 (p. 10) . Compiler error from an unhandled checked exception.
- Figure 4 (p. 11) . Another compiler error.
- Figure 5 (p. 14) . Output from program that throws ArithmeticException.
- Figure 6 (p. 15) . Syntax of a try block.

---

\*Version 1.2: Aug 8, 2012 11:20 pm -0500

<sup>†</sup><http://creativecommons.org/licenses/by/3.0/>

- Figure 7 (p. 16) . Syntax of a catch block.
- Figure 8 (p. 19) . Output produced by the finally block.
- Figure 9 (p. 20) . Syntax for declaring that a method throws exceptions.
- Figure 10 (p. 21) . Example of a throw statement.
- Figure 11 (p. 23) . Output from the for loop.
- Figure 12 (p. 24) . Output from the exception handler.
- Figure 13 (p. 24) . Output from code following the catch block.

### 2.1.2 Listings

- Listing 1 (p. 9) . Sample program with no exception handling code.
- Listing 2 (p. 10) . Sample program that fixes one compiler error.
- Listing 3 (p. 11) . Sample program that fixes the remaining compiler error.
- Listing 4 (p. 13) . A sample program that throws an exception.
- Listing 5 (p. 18) . The power of the finally block.
- Listing 6 (p. 22) . The class named MyException.
- Listing 7 (p. 22) . The try block.
- Listing 8 (p. 23) . A matching catch block.
- Listing 9 (p. 24) . Code following the catch block.
- Listing 10 (p. 24) . Complete program listing for Excep16.

## 3 Preview

This module explains Exception Handling in Java. The discussion includes the following topics:

- What is an exception?
- How do you throw and catch exceptions?
- What do you do with an exception once you have caught it?
- How do you make use of the exception class hierarchy provided by the Java development environment?

This module will cover many of the details having to do with exception handling in Java. By the end of the module, you should know that the use of exception handling is not optional in Java, and you should have a pretty good idea how to use exception handling in a beneficial way.

## 4 Discussion and sample code

### Introduction

Stated simply, the exception-handling capability of Java makes it possible for you to:

- Monitor for exceptional conditions within your program
- Transfer control to special exception-handling code (*which you design*) if an exceptional condition occurs

### The basic concept

This is accomplished using the keywords: **try** , **catch** , **throw** , **throws** , and **finally** . The basic concept is as follows:

- You **try** to execute the statements contained within a block of code. (*A block of code is a group of one or more statements surrounded by curly brackets.*)
- If you detect an exceptional condition within that block, you **throw** an exception object of a specific type.

- You **catch** and process the exception object using code that you have designed.
- You optionally execute a block of code, designated by **finally** , which needs to be executed whether or not an exception occurs. (*Code in the **finally** block is normally used to perform some type of cleanup.*)

### Exceptions in code written by others

There are also situations where you don't write the code to **throw** the exception object, but an exceptional condition that occurs in code written by someone else transfers control to exception-handling code that you write.

For example, the **read** method of the **InputStream** class throws an exception of type **IOException** if an exception occurs while the **read** method is executing. In this case, you are responsible only for the code in the **catch** block and optionally for the code in the **finally** block.

*(This is the reason that you must surround the call to **System.in.read()** with a **try** block followed by a **catch** block, or optionally declare that your method **throws** an exception of type **IOException**.)*

### Exception hierarchy, an overview

When an exceptional condition causes an exception to be *thrown* , that exception is represented by an object instantiated from the class named **Throwable** or one of its subclasses.

Here is part of what Sun has to say about the **Throwable** class:

*"The **Throwable** class is the superclass of all errors and exceptions in the Java language. Only objects that are instances of this class (or one of its subclasses) are thrown by the Java Virtual Machine or can be thrown by the Java **throw** statement. Similarly, only this class or one of its subclasses can be the argument type in a **catch** clause."*

Sun goes on to say:

*"Instances of two subclasses, **Error** and **Exception** , are conventionally used to indicate that exceptional situations have occurred. Typically, these instances are freshly created in the context of the exceptional situation so as to include relevant information (such as stack trace data)."*

### The Error and Exception classes

The virtual machine and many different methods in many different classes throw *exceptions* and *errors* . I will have quite a lot more to say about the classes named **Error** and **Exception** later in this module.

### Defining your own exception types

You may have concluded from the Sun quotation given above that you can define and **throw** exception objects of your own design, and if you did, that is a correct conclusion. (*Your new class must extend **Throwable** or one of its subclasses.*)

### The difference between Error and Exception

As mentioned above, the **Throwable** class has two subclasses:

- **Error**
- **Exception**

### What is an error?

What is the difference between an **Error** and an **Exception** ? Paraphrasing David Flanagan and his excellent series of books entitled *Java in a Nutshell* , an **Error** indicates that a non-recoverable error has occurred that should not be caught. Errors usually cause the Java virtual machine to display a message and exit.

Sun says the same thing in a slightly different way:

*"An **Error** is a subclass of **Throwable** that indicates serious problems that a reasonable application should not try to catch. Most such errors are abnormal conditions."*

For example, one of the subclasses of **Error** is named **VirtualMachineError** . This error is *"Thrown to indicate that the Java Virtual Machine is broken or has run out of resources necessary for it to continue operating. "*

### What is an exception?

Paraphrasing Flanagan again, an **Exception** indicates an abnormal condition that must be properly handled to prevent program termination.

Sun explains it this way:

*"The class **Exception** and its subclasses are a form of **Throwable** that indicates conditions that a reasonable application might want to catch."*

As of JDK 1.4.0, there are more than fifty known subclasses of the **Exception** class. Many of these subclasses themselves have numerous subclasses, so there is quite a lot of material that you need to become familiar with.

### **The RuntimeException class**

One subclass of **Exception** is the class named **RuntimeException**. As of JDK 1.4.0, this class has about 30 subclasses, many which are further subclassed. The class named **RuntimeException** is a very important class.

### **Unchecked exceptions**

The **RuntimeException** class, and its subclasses, are important not so much for what they do, but for what they don't do. I will refer to exceptions instantiated from **RuntimeException** and its subclasses as *unchecked* exceptions.

Basically, an unchecked exception is a type of exception that you can optionally handle, or ignore. If you elect to ignore the possibility of an unchecked exception, and one occurs, your program will terminate as a result. If you elect to handle an unchecked exception and one occurs, the result will depend on the code that you have written to handle the exception.

### **Checked exceptions**

All exceptions instantiated from the **Exception** class, or from subclasses of **Exception** other than **RuntimeException** and its subclasses must either be:

- Handled with a **try** block followed by a **catch** block, or
- Declared in a **throws** clause of any method that can throw them

In other words, checked exceptions *cannot be ignored* when you write the code in your methods. According to Flanagan, the exception classes in this category represent routine abnormal conditions that should be anticipated and caught to prevent program termination.

### **Checked by the compiler**

Your code must anticipate and either handle or declare checked exceptions. Otherwise, your program won't compile. (*These are exception types that are checked by the compiler.*)

### **Throwable constructors and methods**

As mentioned above, all errors and exceptions are subclasses of the **Throwable** class. As of JDK 1.4.0, the **Throwable** class provides four constructors and about a dozen methods. The four constructors are shown in Figure 1 (p. 5) .

---

### Throwable constructors.

```
Throwable()  
  
Throwable(String message)  
  
Throwable(String message,  
           Throwable cause)  
  
Throwable(Throwable cause)
```

**Figure 1:** Throwable constructors.

---

The first two constructors have been in Java for a very long time. Basically, these two constructors allow you to construct an exception object with, or without a **String** message encapsulated in the object.

#### **New to JDK 1.4**

The last two constructors are new in JDK 1.4.0. These two constructors are provided to support the *cause facility*. The *cause facility* is new in release 1.4. It is also known as the *chained exception*<sup>1</sup> facility. (*I won't cover this facility in this module. Rather, I plan to cover it in a series of future modules.*)

#### **Methods of the Throwable class**

Figure 2 (p. 6) shows some of the methods of the **Throwable** class. (*I omitted some of the methods introduced in JDK 1.4 for the reasons given above.*)

---

<sup>1</sup>[http://softwaredev.earthweb.com/java/article/0,12082\\_1431531\\_1,00.html](http://softwaredev.earthweb.com/java/article/0,12082_1431531_1,00.html)

---

### Methods of the Throwable class.

```

fillInStackTrace()
getStackTrace()
printStackTrace().
setStackTrace(StackTraceElement[] stackTrace)

getLocalizedMessage()
getMessage()
toString()

```

**Figure 2:** Methods of the Throwable class.

---

#### The StackTrace

The first four methods in Figure 2 (p. 6) deal with the *StackTrace*. In case you are unfamiliar with the term *StackTrace*, this is a list of the methods executed in sequence that led to the exception. (*This is what you typically see on the screen when your program aborts with a runtime error that hasn't been handled.*)

#### Messages

The two methods dealing with messages provide access to a **String** message that may be encapsulated in the exception object. The **getMessage** class simply returns the message that was encapsulated when the object was instantiated. (*If no message was encapsulated, this method returns null.*)

The **getLocalizedMessage** method is a little more complicated to use. According to Sun, "*Subclasses may override this method in order to produce a locale-specific message.*"

#### The toString method

The **toString** method is inherited from the **Object** class and overridden in the exception subclass to "*return a short description of the **Throwable***".

#### Inherited methods

All exception objects inherit the methods of the **Throwable** class, which are listed in Figure 2 (p. 6). Thus, any of these methods may be called by the code in the **catch** block in its attempt to successfully handle the exception.

For example, exceptions may have a message encapsulated in the exception object, which can be accessed using the **getMessage** method. You can use this to display a message describing the error or exception.

You can also use other methods of the **Throwable** class to:

- Display a stack trace showing where the exception or error occurred
- Produce a **String** representation of the exception object

#### So, what is an exception?

According to the online book entitled *The Java Tutorial*<sup>2</sup> by Campione and Walrath:

*"The term exception is shorthand for the phrase "exceptional event". It can be defined as follows:*

---

<sup>2</sup><http://java.sun.com/docs/books/tutorial/>

*Definition: An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions."*

When an exceptional condition occurs within a method, the method may instantiate an exception object and hand it off to the runtime system to deal with it. This is accomplished using the **throw** keyword. (This is called *throwing an exception*.)

To be useful, the exception object should probably contain information about the exception, including its type and the state of the program when the exception occurred.

### Handling the exception

At that point, the runtime system becomes responsible for finding a block of code designed to handle the exception.

The runtime system begins its search with the method in which the exception occurred and searches backwards through the call stack until it finds a method that contains an *appropriate* exception handler (*catch block*).

An exception handler is *appropriate* if the type of the exception thrown is the same as the type of exception handled by the handler, or is a subclass of the type of exception handled by the handler.

Thus, the requirement to handle an exception that has been thrown progresses up through the call stack until an appropriate handler is found to handle the exception. If no appropriate handler is found, the runtime system and the program terminate.

(If you have ever had a program terminate with a **NullPointerException** , then you know how program termination works).

According to the jargon, the exception handler that is chosen is said to *catch the exception*.

### Advantages of using exception handling

According to Campione and Walrath, exception handling provides the following advantages over "traditional" error management techniques:

- Separating Error Handling Code from "Regular" Code
- Propagating Errors Up the Call Stack
- Grouping Error Types and Error Differentiation

### Separating error handling code from regular code

I don't plan to discuss these advantages in detail. Rather, I will simply refer you to The Java Tutorial<sup>3</sup> and other good books where you can read their discussions. However, I will comment briefly.

Campione and Walrath provide a good illustration where they show how a simple program having about six lines of code get "bloated" into about 29 lines of very confusing code through the use of traditional error management techniques. Not only does the program suffer bloat, the logical flow of the original program gets lost in the clutter of the modified program.

They then show how to accomplish the same error management using exception handling. Although the version with exception handling contains about seventeen lines of code, it is orderly and easy to understand. The additional lines of code do not cause the original logic of the program to get lost.

### You must still do the hard work

However, the use of exception handling does not spare you from the hard work of detecting, reporting, and handling errors. What it does is provide a means to separate the details of what to do when something out-of-the-ordinary happens from the normal logical flow of the program code.

### Propagating exceptions up the call stack

Sometimes it is desirable to propagate exception handling up the call stack and let the corrective action be taken at a higher level.

For example, you might provide a class with methods that implement a *stack* . One of the methods of your class might be to *pop* an element off the stack.

---

<sup>3</sup><http://java.sun.com/docs/books/tutorial/>

What should your program do if a using program attempts to pop an element off an empty stack? That decision might best be left to the user of your stack class, and you might simply propagate the notification up to the calling method and let that method take the corrective action.

### Grouping exception types

When an exception is thrown, an object of one of the exception classes is passed as a parameter. Objects are instances of classes, and classes fall into an inheritance hierarchy in Java. Therefore, a natural hierarchy can be created, which causes exceptions to be grouped in logical ways.

For example, going back to the stack example, you might create an exception class that applies to all exceptional conditions associated with an object of your stack class. Then you might extend that class into other classes that pertain to specific exceptional conditions, such as *push* exceptions, *pop* exceptions, and *initialization* exceptions.

When your code throws an exception object of a specific type, that object can be caught by an exception handler designed either to:

- Catch on the basis of a group of exceptions, or
- Catch on the basis of a subgroup of that group, or
- Catch on the basis of one of the specialized exceptions.

In other words, an exception handler can catch exceptions of the class specified by the type of its parameter, or can catch exceptions of any subclass of the class specified by the type of its parameter.

### More detailed information on exception handling

As explained earlier, except for **Throwable** objects of type **Error** and for **Throwable.Exception** objects of type **RuntimeException**, Java programs must either *handle* or *declare* all **Exception** objects that are thrown. Otherwise, the compiler will refuse to compile the program.

In other words, all exceptions other than those specified above are *checked* by the compiler, and the compiler will refuse to compile the program if the exceptions aren't handled or declared. As a result, exceptions other than those specified above are often referred to as *checked* exceptions.

### Catching an exception

Just to make certain that we are using the same terminology, a method *catches* an exception by providing an exception handler whose parameter type is appropriate for that type of exception object. (*I will more or less use the terms **catch** block and exception handler interchangeably.*)

The type of the parameter in the **catch** block must be the class from which the exception was instantiated, or a superclass of that class that resides somewhere between that class and the **Throwable** class in the inheritance hierarchy.

### Declaring an exception

If the code in a method can throw a checked exception, and the method does not provide an exception handler for the type of exception object thrown, the method must *declare* that it can throw that exception. The **throws** keyword is used in the method declaration to declare that it **throws** an exception of a particular type.

Any checked exception that can be thrown by a method is part of the method's programming interface (see the *read* method of the *InputStream* class, which throws **IOException**, for example). Users of a method must know about the exceptions that a method can throw in order to be able to handle them. Thus, you must declare the exceptions that the method can throw in the method signature.

### Checked exceptions

Checked exceptions are all exception objects instantiated from subclasses of the **Exception** class other than those of the **RuntimeException** class.

Exceptions of all **Exception** subclasses other than **RuntimeException** are checked by the compiler and will result in compiler errors if they are neither *caught* nor *declared*.

You will learn how you can create your own exception classes later. Whether your exception objects become checked or not depends on the class that you extend when you define your exception class.

(*If you extend a checked exception class, your new exception type will be a checked exception. Otherwise, it will not be a checked exception.*)

### Exceptions that can be thrown within the scope of a method

The exceptions that can be thrown within the scope of a method include not only exceptions which are thrown by code written into the method, but also includes exceptions thrown by methods called by that method, or methods called by those methods, etc.

According to Campione and Walrath,

*"This ... includes any exception that can be thrown while the flow of control remains within the method. Thus, this ... includes both exceptions that are thrown directly by the method with Java's throw statement, and exceptions that are thrown indirectly by the method through calls to other methods."*

### Sample programs

Now it's time to take a look at some sample code designed to deal with exceptions of the types delivered with the JDK. Initially I won't include exception classes that are designed for custom purposes. However, I will deal with exceptions of those types later in the module.

The first three sample programs will illustrate the successive stages of dealing with checked exceptions by either catching or declaring those exceptions.

### Sample program with no exception handling code

The first sample program shown in Listing 1 (p. 9) neither catches nor declares the **InterruptedException** which can be thrown by the **sleep** method of the **Thread** class.

### Listing 1: Sample program with no exception handling code.

```

/*File Excep11.java
Copyright 2002, R.G.Baldwin
Tested using JDK 1.4.0 under Win2000
*****/
import java.lang.Thread;

class Excep11{
    public static void main(
        String[] args){
        Excep11 obj = new Excep11();
        obj.myMethod();
    }//end main
    //-----//

    void myMethod(){
        Thread.currentThread().sleep(1000);
    }//end myMethod
}//end class Excep11

```

### A possible InterruptedException

The code in the **main** method of Listing 1 (p. 9) calls the method named **myMethod**. The method named **myMethod** calls the method named **sleep** of the **Thread** class. The method named **sleep** declares that it throws **InterruptedException**.

**InterruptedException** is a checked exception. The program illustrates the failure to either catch or declare **InterruptedException** in the method named **myMethod**.

As a result, this program won't compile. The compiler error is similar to that shown in Figure 3 (p. 10). Note the caret in the last line that points to the point where the compiler detected the problem.

---

### Compiler error from an unhandled checked exception.

```

unreported exception
java.lang.InterruptedException;
must be caught or declared to be thrown
    Thread.currentThread().sleep(1000);
        ^

```

**Figure 3:** Compiler error from an unhandled checked exception.

---

As you can see, the compiler detected a problem where the **sleep** method was called, because the method named **myMethod** failed to deal properly with an exception that can be thrown by the **sleep** method.

#### Sample program that fixes one compiler error

The next version of the program, shown in Listing 2 (p. 10) , fixes the problem identified with the call to the **sleep** method, by declaring the exception in the signature for the method named **myMethod** .

**Listing 2: Sample program that fixes one compiler error.**

```

/*File Excep12.java
Copyright 2002, R.G.Baldwin
Tested using JDK 1.4.0 under Win2000
*****/
import java.lang.Thread;

class Excep12{
    public static void main(
        String[] args){
        Excep12 obj = new Excep12();
        obj.myMethod();
    }//end main
    //-----//

    void myMethod()
        throws InterruptedException{
        Thread.currentThread().sleep(1000);
    }//end myMethod
}//end class Excep12

```

#### Another possible InterruptedException

As was the case in the previous program, this program also illustrates a failure to catch or declare an **InterruptedException** . However, in this case, the problem has moved up one level in the call stack relative to the problem with the program in Listing 1 (p. 9) .

This program also fails to compile, producing a compiler error similar to that shown in Figure 4 (p. 11). Note that the caret indicates that the problem is associated with the call to **myMethod**.

---

#### Another compiler error.

```

unreported exception
java.lang.InterruptedException;
must be caught or declared to be thrown
    obj.myMethod();
        ^

```

**Figure 4:** Another compiler error.

---

#### Didn't solve the problem

Simply declaring a checked exception doesn't solve the problem. Ultimately, the exception must be handled if the compiler problem is to be solved.

*(Note, however, that it is possible to declare that the **main** method throws a checked exception, which will cause the compiler to ignore it and allow your program to compile.)*

The program in Listing 2 (p. 10) eliminated the compiler error identified with the call to the method named **sleep**. This was accomplished by declaring that the method named **myMethod** throws *InterruptedException*. However, this simply passed the exception up the call stack to the next higher-level method in the stack. This didn't solve the problem, it simply handed it off to another method to solve.

The problem still exists, and is now identified with the call to **myMethod** where it will have to be handled in order to make the compiler error go away.

#### Sample program that fixes the remaining compiler error

The version of the program shown in Listing 3 (p. 11) fixes the remaining compiler error. This program illustrates both declaring and handling a checked exception. This program compiles and runs successfully.

#### Listing 3: Sample program that fixes the remaining compiler error.

```

/*File Excep13.java
Copyright 2002, R.G.Baldwin

Tested using JDK 1.4.0 under Win2000
*****/
import java.lang.Thread;

class Excep13{
    public static void main(
        String[] args){
        Excep13 obj = new Excep13();
        try{//begin try block

```

```

        obj.myMethod();
    }catch(InterruptedException e){
        System.out.println(
            "Handle exception here");
    }//end catch block
} //end main
//-----//

void myMethod()
    throws InterruptedException{
    Thread.currentThread().sleep(1000);
} //end myMethod
} //end class Excep13

```

### The solution to the problem

This solution to the problem is accomplished by surrounding the call to **myMethod** with a **try** block, which is followed immediately by an *appropriate* **catch** block. In this case, an appropriate **catch** block is one whose parameter type is either **InterruptedException**, or a superclass of **InterruptedException**.

(Note, however, that the superclass cannot be higher than the **Throwable** class in the inheritance hierarchy.)

#### The myMethod method declares the exception

As in the previous version, the method named **myMethod** (declares the exception and passes it up the call stack to the method from which it was called.

#### The main method handles the exception

In the new version shown in Listing 3 (p. 11), the **main** method provides a **try** block with an *appropriate* **catch** block for dealing with the problem (*although it doesn't actually deal with it in any significant way*). This can be interpreted as follows:

- Try to execute the code within the **try** block.
- If an exception occurs, search for a **catch** block that matches the type of object thrown by the exception.
- If such a **catch** block can be found, immediately transfer control to the catch block without executing any of the remaining code in the **try** block.

(For simplicity, this program didn't have any remaining code. Some later sample programs will illustrate code being skipped due to the occurrence of an exception.)

#### Not a method call

Note that this transfer of control is not a method call. It is an unconditional transfer of control. There is no *return* from a catch block.

#### Matching catch block was found

In this case, there was a matching **catch** block to receive control. In the event that an **InterruptedException** is thrown, the program would execute the statement within the body of the **catch** block, and then transfer control to the code following the final **catch** block in the group of **catch** blocks (*in this case, there was only one catch block*).

#### No output is produced

It is unlikely that you will see any output when you run this program, because it is unlikely that an **InterruptedException** will be thrown. (*I didn't provide any code that will cause such an exception to occur.*)

#### A sample program that throws an exception

Now let's look at the sample program in Listing 4 (p. 13), which throws an exception and deals with it. This program illustrates the implementation of exception handling using the try/catch block structure.

**Listing 4: A sample program that throws an exception.**

```

/*File Excep14.java
Copyright 2002, R. G. Baldwin

Tested with JDK 1.4.0 under Win2000
*****/

class Excep14{
    public static void main(
        String[] args){
        try{
            for(int cnt = 2; cnt >-1; cnt--){
                System.out.println(
                    "Running. Quotient is: "
                    + 6/cnt);
            }//end for-loop
        }//end try block
        catch(ArithmeticException e){
            System.out.println(
                "Exception message is: "
                + e.getMessage()
                + "\nStacktrace shows:");
            e.printStackTrace();
            System.out.println(
                "String representation is\n " +
                e.toString());
            System.out.println(
                "Put corrective action here");
        }//end catch block
        System.out.println(
            "Out of catch block");
    }//end main
}

} //end class Excep14

```

**Keeping it simple**

I try to keep my sample programs as simple as possible, introducing the minimum amount of complexity necessary to illustrate the main point of the program. It is easy to write a *really simple* program that throws an unchecked **ArithmeticException**. Therefore, the program in Listing 4 (p. 13) was written to throw an **ArithmeticException**. This was accomplished by trying to perform an integer divide by zero.

**The try/catch structure is the same ...**

It is important to note that the *try/catch* structure illustrated in Listing 4 (p. 13) would be the same whether the exception is checked or unchecked. The main difference is that you are not required by the compiler to handle unchecked exceptions and you are required by the compiler to either handle or declare checked exceptions.

**Throwing an ArithmeticException**

The code in Listing 4 (p. 13) executes a simple counting loop inside a **try** block. During each iteration, the counting loop divides the integer 6 by the value of the counter. When the value of the counter goes to zero, the runtime system tries to perform an integer divide by zero operation, which causes it to throw an **ArithmeticException**.

### Transfer control immediately

At that point, control is transferred directly to the **catch** block that follows the **try** block. This is an *appropriate catch* block because the type of parameter declared for the **catch** block is **ArithmeticException**. It matches the type of the object that is thrown.

*(It would also be appropriate if the declared type of the parameter were a superclass of **ArithmeticException**, up to and including the class named **Throwable**. **Throwable** is a direct subclass of **Object**. If you were to declare the parameter type for the **catch** block as **Object**, the compiler would produce an incompatible type error.)*

### Calling methods inside the catch block

Once control enters the **catch** block, three of the methods of the **Throwable** class are called to cause information about the situation to be displayed on the screen. The output produced by the program is similar to that shown in Figure 5 (p. 14).

---

### Output from program that throws **ArithmeticException**.

```
Running. Quotient is: 3
Running. Quotient is: 6
Exception message is: / by zero
Stacktrace shows:
java.lang.ArithmeticException:
 / by zero
 at Excep14.main(Excep14.java:35)
String representation is
java.lang.ArithmeticException:
 / by zero
Put corrective action here
Out of catch block
```

**Figure 5:** Output from program that throws **ArithmeticException**.

---

### Key things to note

The key things to note about the code in Listing 4 (p. 13) and the output in Figure 5 (p. 14) are:

- The code to be protected is contained in a **try** block.
- The **try** block is followed immediately by an appropriate **catch** block.
- When an exception is thrown within the **try** block, control is transferred immediately to the **catch** block with the matching or appropriate parameter type.
- Although the code in the **catch** block simply displays the current state of the program, it could contain code that attempts to rectify the problem.
- Once the code in the **catch** block finishes executing, control is passed to the next executable statement following the **catch** block, which in this program is a print statement.

### Doesn't attempt to rectify the problem

This program doesn't attempt to show how an actual program might recover from an exception of this sort. However, it is clear that (*rather than experiencing automatic and unconditional termination*) the program remains in control, and in some cases, recovery might be possible.

This sample program illustrates **try** and **catch**. The use of **finally**, will be discussed and illustrated later.

#### A nuisance problem explained

While we are at it, I would be remiss in failing to mention a nuisance problem associated with exception handling.

As you may recall, the scope of a variable in Java is limited to the block of code in which it is declared. A block is determined by enclosing code within a pair of matching curly brackets: {...}.

Since a pair of curly brackets is required to define a **try** block, the scope of any variables or objects declared inside the **try** block is limited to the **try** block.

While this is not an insurmountable problem, it may require you to modify your programming style in ways that you find distasteful. In particular, if you need to access a variable both within and outside the **try** block, you must declare it before entering the **try** block.

#### The process in more detail

Now that you have seen some sample programs to help you visualize the process, let's discuss the process in more detail.

#### The try block

According to Campione and Walrath,

*"The first step in writing any exception handler is putting the Java statements within which an exception can occur into a try block. The try block is said to govern the statements enclosed within it and defines the scope of any exception handlers (established by subsequent catch blocks) associated with it."*

Note that the terminology being used by Campione and Walrath treats the **catch block** as the "exception handler" and treats the **try** block as something that precedes one or more exception handlers. I don't disagree with their terminology. I mention it only for the purpose of avoiding confusion over terminology.

#### The syntax of a try block

The general syntax of a **try** block, as you saw in the previous program, has the keyword **try** followed by one or more statements enclosed in a pair of matching curly brackets, as shown in Figure 6 (p. 15).

---

#### Syntax of a try block.

```
try{
  //java statements
} //end try block
```

**Figure 6:** Syntax of a try block.

---

#### Single statement and multiple exceptions

You may have more than one statement that can throw one or more exceptions and you will need to deal with all of them.

You could put each such statement that might throw exceptions within its own **try** block and provide separate exception handlers for each **try** block.

*(Note that some statements, particularly those that call other methods, could potentially throw many different types of exceptions.)*

Thus a **try** block consisting of a single statement might require many different exception handlers or **catch** blocks following it.

#### Multiple statements and multiple exceptions

You could put all or several of the statements that might throw exceptions within a single **try** block and associate multiple exception handlers with it. There are a number of practical issues involved here, and only you can decide in any particular instance which approach would be best.

#### The catch blocks must follow the try block

However you decide to do it, the exception handlers associated with a **try** block must be placed immediately following their associated **try** block. If an exception occurs within the **try** block, that exception is handled by the appropriate exception handler associated with the **try** block. If there is no appropriate exception handler associated with the **try** block, the system attempts to find an appropriate exception handler in the next method up the call stack.

A **try** block must be accompanied by at least one **catch** block (or one **finally** block). Otherwise, a compiler error that reads something like 'try' without 'catch' or 'finally' will occur.

#### The catch block(s)

Continuing with what Campione and Walrath have to say:

*"Next, you associate exception handlers with a try block by providing one or more catch blocks directly after the try block."*

There can be no intervening code between the end of the **try** block and the beginning of the first **catch** block, and no intervening code between **catch** blocks.

#### Syntax of a catch block

The general form of a **catch** block is shown in Figure 7 (p. 16) .

---

#### Syntax of a catch block.

```

    catch(ThrowableObjectType paramName){
    //Java statements to handle the
    // exception
    }//end catch block

```

**Figure 7:** Syntax of a catch block.

---

The declaration for the **catch** block requires a single argument as shown. The syntax for the argument declaration is the same as an argument declaration for a method.

#### Argument type specifies type of matching exception object

The argument type declares the type of exception object that a particular **catch** block can handle. The type must be **Throwable** , or a subclass of the **Throwable** class discussed earlier.

#### A parameter provides the local name

Also, as in a method declaration, there is a parameter, which is the name by which the handler can refer to the exception object. For example, in an earlier program, I used statements such as `e.getMessage()` to access an instance method of an exception object caught by the exception handler. In that case, the name of the parameter was `e`.

You access the instance variables and methods of exception objects the same way that you access the instance variables and methods of other objects.

#### Proper order of catch blocks

According to Campione and Walrath:

*"The catch block contains a series of legal Java statements. These statements are executed if and when the exception handler is called. The runtime system calls the exception handler when the handler is the first one in the call stack whose type matches that of the exception thrown."*

Therefore, the order of your exception handlers is very important, particularly if you have some handlers, which are further up the exception hierarchy than others.

Those handlers that are designed to handle exception types furthest from the root of the hierarchy tree ( **Throwable** ) should be placed first in the list of exception handlers.

Otherwise, an exception handler designed to handle a specific type of object may be preempted by another handler whose exception type is a superclass of that type, if the superclass exception handler appears earlier in the list of exception handlers.

#### Catching multiple exception types with one handler

Exception handlers that you write may be more or less specialized. In addition to writing handlers for very specialized exception objects, the Java language allows you to write general exception handlers that handle multiple types of exceptions.

#### A hierarchy of Throwable classes

Java exceptions are **Throwable** objects (*instances of the **Throwable** class or a subclass of the **Throwable** class*).

The Java standard library contains numerous classes that are subclasses of **Throwable** and thus build a hierarchy of **Throwable** classes.

According to Campione and Walrath:

*"Your exception handler can be written to handle any class that inherits from **Throwable**. If you write a handler for a "leaf" class (a class with no subclasses), you've written a specialized handler: it will only handle exceptions of that specific type. If you write a handler for a "node" class (a class with subclasses), you've written a general handler: it will handle any exception whose type is the node class or any of its subclasses."*

#### You have a choice

Therefore, when writing exception handlers, you have a choice. You can write a handler whose exception type corresponds to a node in the inheritance hierarchy, and it will be appropriate to **catch** exceptions of that type, or any subclass of that type.

Alternately, you can write a handler whose exception type corresponds to a *leaf*, in which case, it will be appropriate to **catch** exceptions of that type only.

And finally, you can mix and match, writing some exception handlers whose type corresponds to a node, and other exception handlers whose type corresponds to a leaf. In all cases, however, be sure to position your exception handlers in reverse subclass order, with the furthest subclass from the root appearing first, and the root class appearing last.

#### The finally block

And finally (*no pun intended*), Campione and Walrath tell us:

*"Java's finally block provides a mechanism that allows your method to clean up after itself regardless of what happens within the try block. Use the finally block to close files or release other system resources."*

To elaborate, the **finally** block can be used to provide a mechanism for cleaning up open files, etc., before allowing control to be passed to a different part of the program. You accomplish this by writing the cleanup code within a **finally** block.

#### Code in finally block is always executed

It is important to remember that the runtime system always executes the code within the **finally** block regardless of what happens within the **try** block.

If no exceptions are thrown, none of the code in **catch** blocks is executed, but the code in the **finally** block is executed.

If an exception is thrown and the code in an exception handler is executed, once the execution of that code is complete, control is passed to the **finally** block and the code in the **finally** block is executed.

*(There is one important exception to the above. If the code in the **catch** block terminates the program by executing **System.exit(0)**, the code in the **finally** block will not be executed.)*

#### The power of the finally block

The sample program shown in Listing 5 (p. 18) illustrates the power of the **finally** block.

#### Listing 5: The power of the finally block.

```

/*File Excep15.java
Copyright 2002, R. G. Baldwin

Tested with JDK 1.4.0 under Win2000
*****/

class Excep15{
    public static void main(
        String[] args){
        new Excep15().aMethod();
    }//end main
    //-----//

    void aMethod(){
        try{
            int x = 5/0;
        }//end try block
        catch(ArithmeticException e){
            System.out.println(
                "In catch, terminating aMethod");
            return;
        }//end catch block

        finally{
            System.out.println(
                "Executing finally block");
        }//end finally block

        System.out.println(
            "Out of catch block");
    }//end aMethod
}

} //end class Excep15

```

#### Execute return statement in catch block

The code in Listing 5 (p. 18) forces an **ArithmeticException** by attempting to do an integer divide by zero. Control is immediately transferred to the matching **catch** block, which prints a message and then executes a **return** statement.

Normally, execution of a **return** statement terminates the method immediately. In this case, however, before the method terminates and returns control to the calling method, the code in the **finally** block is executed. Then control is transferred to the **main** method, which called this method in the first place.

Figure 8 (p. 19) shows the output produced by this program.

---

### Output produced by the finally block.

```
In catch, terminating aMethod
Executing finally block
```

**Figure 8:** Output produced by the finally block.

---

This program demonstrates that the **finally** block really does have the final word.

#### Declaring exceptions thrown by a method

Sometimes it is better to handle exceptions in the method in which they are detected, and sometimes it is better to pass them up the call stack and let another method handle them.

In order to pass exceptions up the call stack, you must *declare* them in your method signature.

To *declare* that a method throws one or more exceptions, you add a **throws** clause to the method signature for the method. The **throws** clause is composed of the **throws** keyword followed by a comma-separated list of all the exceptions thrown by that method.

The **throws** clause goes after the method name and argument list and before the curly bracket that defines the scope of the method.

Figure 9 (p. 20) shows the syntax for declaring that a method **throws** four different types of exceptions.

---

### Syntax for declaring that a method throws exceptions.

```
void myMethod() throws
    InterruptedException,
    MyException,
    HerException,
    UrException
{
    //method code
} //end myMethod()
```

**Figure 9:** Syntax for declaring that a method throws exceptions.

---

Assuming that these are checked exceptions, any method calling this method would be required to either handle these exception types, or continue passing them up the call stack. Eventually, some method must handle them or the program won't compile.

*(Note however that while it might not represent good programming practice, it is allowable to declare that the **main** method **throws** exceptions. This is a way to avoid handling checked exceptions and still get your program to compile.)*

#### The **throw** keyword

Before your code can **catch** an exception, some Java code must **throw** one. The exception can be thrown by code that you write, or by code that you are using that was written by someone else.

Regardless of who wrote the code that throws the exception, it's always thrown with the Java **throw** keyword. At least that is true for exceptions that are thrown by code written in the Java language.

*(Exceptions such as **ArithmeticException** are also thrown by the runtime system, which is probably not written using Java source code.)*

#### A single argument is required

When formed into a statement, the **throw** keyword requires a single argument, which must be a reference to an object instantiated from the **Throwable** class, or any subclass of the **Throwable** class. Figure 10 (p. 21) shows an example of such a statement.

---

### Example of a throw statement.

```
throw new myThrowableClass("Message");
```

**Figure 10:** Example of a throw statement.

---

If you attempt to throw an object that is not instantiated from **Throwable** or one of its subclasses, the compiler will refuse to compile your program.

#### Defining your own exception classes

Now you know how to write exception handlers for those exception objects that are thrown by the runtime system, and thrown by methods in the standard class library.

It is also possible for you to define your own exception classes, and to cause objects of those classes to be thrown whenever an exception occurs. In this case, you get to decide just what constitutes an exceptional condition.

For example, you could write a data-processing application that processes integer data obtained via a TCP/IP link from another computer. If the specification for the program indicates that the integer value 10 should never be received, you could use an occurrence of the integer value 10 to cause an exception object of your own design to be thrown.

#### Choosing the exception type to throw

Before throwing an exception, you must decide on its type. Basically, you have two choices in this regard:

- Use an exception class written by someone else, such as the myriad of exception classes defined in the Java standard library.
- Define an exception class of your own.

#### An important question

So, an important question is, when should you define your own exception classes and when should you use classes that are already available. Because this is only one of many design issues, I'm not going to try to give you a ready answer to the question. However, I will refer you to The Java Tutorial <sup>4</sup> by Campione and Walrath where you will find a checklist to help you make this decision.

#### Choosing a superclass to extend

If you decide to define your own exception class, it must be a subclass of **Throwable**. You must decide which class you will extend.

The two existing subclasses of **Throwable** are **Exception** and **Error**. Given the earlier description of **Error** and its subclasses, it is not likely that your exceptions would fit the **Error** category. (*In concept, errors are reserved for serious hard errors that occur deep within the system.*)

#### Checked or unchecked exception

Therefore, your new class should probably be a subclass of **Exception**. If you make it a subclass of **RuntimeException**, it won't be a checked exception. If you make it a subclass of **Exception**, but not a subclass of **RuntimeException**, it will be a checked exception.

---

<sup>4</sup><http://java.sun.com/docs/books/tutorial/>

Only you can decide how far down the **Exception** hierarchy you want to go before creating a new branch of exception classes that are unique to your application.

#### Naming conventions

Many Java programmers append the word **Exception** to the end of all class names that are subclasses of **Exception**, and append the word **Error** to the end of all class names that are subclasses of **Error**.

#### One more sample program

Let's wrap up this module with one more sample program named **Excep16**. We will define our own exception class in this program. Then we will **throw**, **catch** and process an exception object instantiated from that class.

#### Discuss in fragments

This program is a little longer than the previous programs, so I will break it down and discuss it in fragments. A complete listing of the program is shown in Listing 10 (p. 24).

The class definition shown in Listing 6 (p. 22) is used to construct a custom exception object that encapsulates a message. Note that this class extends **Exception**. (*Therefore, it is a checked exception.*)

#### Listing 6: The class named MyException.

```
class MyException extends Exception{
MyException(String message){//constr
    super(message);
} //end constructor
} //end MyException class
```

The constructor for this class receives an incoming **String** message parameter and passes it to the constructor for the superclass. This makes the message available for access by the **getMessage** method called in the catch block.

#### The try block

Listing 7 (p. 22) shows the beginning of the **main** method, including the entire **try** block

#### Listing 7: The try block.

```
class Excep16{//controlling class
public static void main(
    String[] args){
try{
    for(int cnt = 0; cnt < 5; cnt++){
        //Throw a custom exception, and
        // pass message when cnt == 3
        if(cnt == 3) throw
            new MyException("3");
        //Transfer control before
        // processing for cnt == 3
        System.out.println(
            "Processing data for cnt = "
                + cnt);
    } //end for-loop
} //end try block
```

The **main** method executes a **for** loop (*inside the try block*) that guarantees that the variable named **cnt** will reach a value of 3 after a couple of iterations.

Once during each iteration, (*until the value of `cnt` reaches 3*) a print statement inside the `for` loop displays the value of `cnt` . This results in the output shown in Figure 11 (p. 23) .

---

### Output from the for loop.

```
Processing data for cnt = 0
Processing data for cnt = 1
Processing data for cnt = 2
```

**Figure 11:** Output from the for loop.

---

### What happens when `cnt` equals 3?

However, when the value of `cnt` equals 3, the `throw` statement in Listing 7 (p. 22) is executed. This causes control to transfer immediately to the matching `catch` block following the `try` block (*see Listing 8 (p. 23)* ). During this iteration, the print statement following the `throw` statement is not executed. Therefore, the output never shows a value for `cnt` greater than 2, as shown in Figure 11 (p. 23) .

### The catch block

Listing 8 (p. 23) shows a `catch` block whose type matches the type of exception thrown in Listing 7 (p. 22) .

### Listing 8: A matching catch block.

```
catch(MyException e){
System.out.println(
    "In exception handler, "
    + "get the message\n"
    + e.getMessage());
} //end catch block
```

When the `throw` statement is executed in Listing 7 (p. 22) , control is transferred immediately to the `catch` block in Listing 8 (p. 23) . No further code is executed within the `try` block.

A reference to the object instantiated as the argument to the `throw` keyword in Listing 7 (p. 22) is passed as a parameter to the `catch` block. That reference is known locally by the name `e` inside the `catch` block.

### Using the incoming parameter

The code in the `catch` block calls the method named `getMessage` (*inherited from the `Throwable` class*) on the incoming parameter and displays that message on the screen. This produces the output shown in Figure 12 (p. 24) .

---

### Output from the exception handler.

```
In exception handler, get the message
3
```

**Figure 12:** Output from the exception handler.

---

### When the catch block finishes execution ...

When the code in the `catch` block has completed execution, control is transferred to the first executable statement following the `catch` block as shown in Listing 9 (p. 24) .

### Listing 9: Code following the catch block.

```
        System.out.println(
            "Out of catch block");
    }//end main
} //end class Excep16
```

That executable statement is a print statement that produces the output shown in Figure 13 (p. 24) .

---

### Output from code following the catch block.

```
Out of catch block
```

**Figure 13:** Output from code following the catch block.

---

### Complete program listing

A complete listing of the program named `Excep16` is shown in Listing 10 (p. 24) .

### Listing 10: Complete program listing for Excep16.

```
/*File Excep16.java
Copyright 2002, R. G. Baldwin
```

Illustrates defining, throwing, catching, and processing a custom exception object that contains a message.

Tested using JDK 1.4.0 under Win 2000

The output is:

```
Processing data for cnt = 0
```

```
Processing data for cnt = 1
```

```
Processing data for cnt = 2
```

```
In exception handler, get the message
```

```
3
```

```
Out of catch block
```

```
*****/
```

```
//The following class is used to
// construct a customized exception
// object containing a message
class MyException extends Exception{
    MyException(String message){//constr
        super(message);
    }//end constructor
}//end MyException class
//=====//
```

```
class Excep16{//controlling class
    public static void main(
        String[] args){
        try{
            for(int cnt = 0; cnt < 5; cnt++){
                //Throw a custom exception, and
                // pass message when cnt == 3
                if(cnt == 3) throw
                    new MyException("3");
                //Transfer control before
                // processing for cnt == 3
                System.out.println(
                    "Processing data for cnt = "
                        + cnt);
            }//end for-loop
        }//end try block
        catch(MyException e){
            System.out.println(
                "In exception handler, "
                    + "get the message\n"
                    + e.getMessage());
        }//end catch block
    }//-----//
```

```
System.out.println(  
    "Out of catch block");  
} //end main
```

## 5 Summary

This module has covered many of the details having to do with exception handling in Java. By now, you should know that the use of exception handling is not optional in Java, and you should have a pretty good idea how to use exception handling in a beneficial way.

Along the way, the discussion has included the following topics:

- What is an exception?
- How do you throw and catch exceptions?
- What do you do with an exception once you have caught it?
- How do you make use of the exception class hierarchy provided by the Java development environment?

## 6 What's next?

That concludes the "Essence of OOP" portion of the ITSE 2321 sub-collection. The series is continued in the ITSE 2317 sub-collection.

## 7 Miscellaneous

This section contains a variety of miscellaneous information.

**NOTE: Housekeeping material**

- Module name: Java OOP: Exception Handling
- File: Java1630.htm
- Published: September 3, 2002
- Revised: August 8, 2012

**NOTE: Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

**Affiliation** : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-