Java Exception Handling

Exception Handling

Introduction

We try to avoid them, but it's an unfortunate fact: Errors occur in software programs. However, if you handle errors properly, you'll greatly improve your program’s readability, reliability and maintainability. The Java programming language uses exceptions for error handling. This document describes when and how to handle errors using exceptions.

What Is an Exception?

The term exception is shorthand for the phrase "exceptional event".

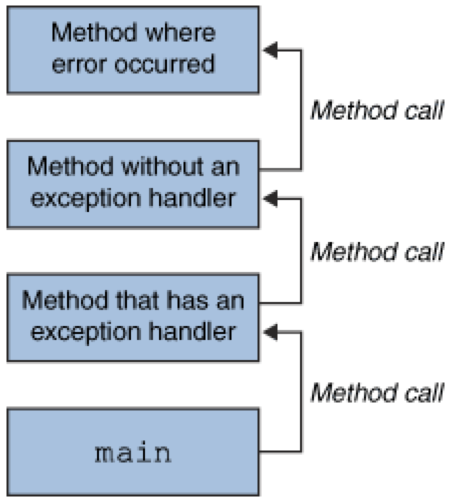
Definition: An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions during the execution of a program.

When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an exception object, contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called throwing an exception.

Exception Handling

Exceptions

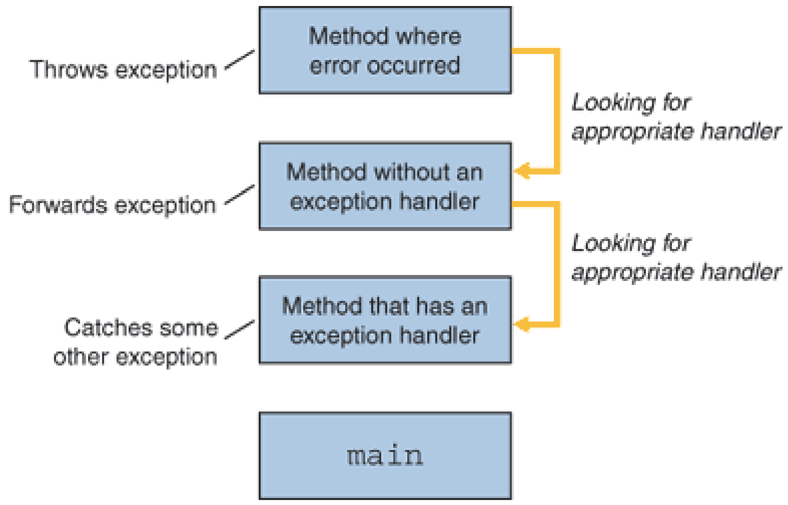
After a method throws an exception, the runtime system attempts to find something to handle it. The set of possible "somethings" to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred. The list of methods is known as the call stack:


Method where error occurred
Method call
Method without an exception handler
Method call
Method that has an exception handler
Method call
main

Exception Handling

Exceptions

The runtime system searches the call stack for a method that contains a block of code that can handle the exception. This block of code is called an exception handler. The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order that the methods were called. When an appropriate handler is found, the run-time system passes the exception to the handler.
An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler. The exception handler chosen is said to catch the exception. If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, as shown in the next figure, the runtime system (and, consequently, the program) terminates.


Throws exception
Method where error occurred
Looking for appropriate handler
Forwards exception
Method without an exception handler
Looking for appropriate handler
Catches some other exception
Method that has an exception handler
main

Exception Handling

Exceptions

The Java runtime system requires that a method must either catch or specify all checked exceptions that can be thrown by that method. This requirement has several components that need further description: "catch," "specify," "checked exceptions," and "exceptions that can be thrown by that method."

Catch: A method can catch an exception by providing an exception handler for that type of exception.

Specify: A method specifies that it can throw exceptions by using the throws clause in the method declaration. This is also called “propagating” the exception.

Checked exceptions

There are two kinds of exceptions: runtime exceptions and non-runtime exceptions. Runtime exceptions occur within the Java runtime system: arithmetic exceptions (such as dividing by zero), pointer exceptions (such as trying to access an object’s members through a null reference), and indexing exceptions (such as trying to access an array element with an index that is too large or too small). A method does not have to catch or specify runtime exceptions, although it may.

Non-runtime exceptions are exceptions that occur in code outside of the Java runtime system. For example, exceptions that occur during I/O are non-runtime exceptions. The compiler ensures that non-runtime exceptions are caught or specified; thus, they are also called checked exceptions.

Exception Handling

Checked exceptions

Some consider the fact that you do not have to catch or specify runtime exceptions a loophole in the exception-handling mechanism. Many programmers are tempted to use runtime exceptions instead of checked exceptions so that they don't have to catch or specify them. In general, this is not recommended.

Exceptions that can be thrown within the scope of the method:

The exceptions that a method can throw include

- Any exception thrown directly by the method with the throw statement
- Any exception thrown indirectly by calling another method that throws an exception

Catching and Handling Exceptions

The following example shows you how to use the three components of an exception handler—the try, catch, and finally blocks—to write an exception handler.

The example defines and implements a class named *ListOfNumbers*. Upon construction, *ListOfNumbers* creates a Vector that contains ten *Integer* elements with sequential values 0 through 9. The *ListOfNumbers* class also defines a method named *writeList* that writes the list of numbers into a text file called OutFile.txt. (This example uses output classes defined in java.io).

Exception Handling

Catching and Handling Exceptions

```
// Note: This class won’t compile by design!
import java.io.*;
import java.util.Vector;

public class ListOfNumbers {

    private Vector victor;
    private static final int SIZE = 10;

    public ListOfNumbers () {
        victor = new Vector(SIZE);
        for (int i = 0; i < SIZE; i++) {
            victor.addElement(new Integer(i));
        }
    }

    public void writeList() {
        PrintWriter out = new PrintWriter(
                     new FileWriter("OutFile.txt"));

        for (int i = 0; i < SIZE; i++) {
            out.println("Value at: " + i + " = " +
                     victor.elementAt(i));
        }

        out.close();
    }
}
```

Exception Handling

Catching and Handling Exceptions

The first line in boldface is a call to the *FileWriter* constructor. The constructor initializes an output stream on a file. If the file cannot be opened, the constructor throws an *IOException*. The second line in boldface is a call to the Vector class's *elementAt* method, which throws an *ArrayIndexOutOfBoundsException* if the value of its argument is too small (less than zero) or too large (larger than the number of elements currently contained by the Vector).

If you try to compile the ListOfNumbers (in a .java source file) class, the compiler prints an error message about the exception thrown by the *FileWriter* constructor. However, it does not display an error message about the exception thrown by elementAt. The reason is that the exception thrown by the constructor, *IOException*, is a checked exception and the one thrown by the elementAt method, *ArrayIndexOutOfBoundsException*, is a runtime exception. The Java programming language requires only that a program handle checked exceptions, so you get only one error message.

Next we'll show exception handlers that catch and handle those exceptions.

Exception Handling

The try Block

The first step in constructing an exception handler is to enclose the code that might throw an exception within a try block.

In general, a try block looks like this:

```
try
{
    code
}
catch and finally blocks . . .
```

The segment in the example code contains one or more legal lines of code that could throw an exception. (The catch and finally blocks are explained in following sections.)

To construct an exception handler for the *writeList* method from the *ListOfNumbers* class, you need to enclose the exception-throwing statements of the *writeList* method within a try block.

There is more than one way to do this. You can put each line of code that might throw an exception within its own try block and provide separate exception handlers for each. Or, you can put all the *writeList* code within a single try block and associate multiple handlers with it.

The following listing uses one try block for the entire method because the code in question is very short:

Exception Handling

The try Block

```
...
private Vector victor;
private static final int SIZE = 10;
...
PrintWriter out = null;

try {
    System.out.println("Entered try statement");
    out = new PrintWriter(new FileWriter("OutFile.txt"));
    for (int i = 0; i < SIZE; i++) {
        out.println("Value at: " + i + " = "
         + victor.elementAt(i));
    }
}
catch and finally statements . . .
```

If an exception occurs within the try block, that exception is handled by an exception handler associated with it. To associate an exception handler with a try block, you must put a catch block after it.

Exception Handling

The catch Block(s)

You associate exception handlers with a try block by providing one or more catch blocks directly after the try block. No code can be between the end of the try block and the beginning of the first catch block:

```
try {
    ...
} catch (ExceptionType1 name) {
    ...
} catch (ExceptionType2 name) {
    ...
} ...
```

Each catch block is an exception handler and handles the type of exception indicated by its argument. The argument type, ExceptionType, declares the type of exception that the handler can handle and must be the name of a class that inherits from the Throwable class. The handler can refer to the exception with name.

The catch block contains code that is executed if and when the exception handler is invoked. The runtime system invokes the exception handler when the handler is the first one in the call stack whose ExceptionType matches the type of the exception thrown. The system considers it a match if the thrown object can legally be assigned to the exception handler's argument.

Exception Handling

The catch Block(s)

Here are two exception handlers for the writeList method — one for two types of checked exceptions that can be thrown within the try statement:

```
try
{
    ...
}
catch (FileNotFoundException e)
{
    System.err.println("FileNotFoundException: " + e.getMessage());
    throw new SampleException(e);
}
catch (IOException e)
{
    System.err.println("Caught IOException: " + e.getMessage());
}
```

Both handlers print an error message. The second handler does nothing else. By catching any IOException that's not caught by the first handler, it allows the program to continue executing.

The first handler, in addition to printing a message, throws a *user-defined* exception. In this example, the *FileNotFoundException*, when caught, causes a user-defined exception called *SampleException* to be thrown. You might want to do this if you want your program to handle a this exception in this situation in a specific way.

Exception handlers can do more than just print error messages or halt the program. They can do error recovery, prompt the user to make a decision, or propagate the error up to a higher-level handler using chained exceptions.

Exception Handling

The finally Block

The final step in setting up an exception handler is to clean up before allowing control to be passed to a different part of the program. You do this by enclosing the clean up code within a *finally* block. The finally block is *optional* and provides a mechanism to clean up regardless of what happens within the try block. Use the finally block to close files or to release other system resources.

The *try* block of the writeList method that you've been working with here opens a *PrintWriter*. The program should close that stream before exiting the writeList method. This poses a somewhat complicated problem because writeList's try block can exit in one of three ways.

1. The new FileWriter statement fails and throws an IOException.
2. The victor.elementAt(i) statement fails and throws an ArrayIndexOutOfBoundsException.
3. Everything succeeds and the try block exits normally.

The runtime system always executes the statements within the finally block regardless of what happens within the try block. So it's the perfect place to perform cleanup

Exception Handling

The finally Block

The following finally block for the writeList method cleans up and closes the PrintWriter.

```
finally
{
    if (out != null)
    {
        System.out.println("Closing PrintWriter");
        out.close();
    }
    else
    {
        System.out.println("PrintWriter not open");
    }
}
```

Exception Handling

The finally Block

In the *writeList* example, you could provide for cleanup without the intervention of a finally block.

For example, you could put the code to close the *PrintWriter* at the end of the try block and again within the exception handler for *ArrayIndexOutOfBoundsException*, as shown here:

```
try
{
    ...
    out.close();       // don't do this; it duplicates code
}
catch (FileNotFoundException e)
{
    out.close();       // don't do this; it duplicates code
    System.err.println("Caught: FileNotFoundException: " + e.getMessage());
    throw new RuntimeException(e);
}
catch (IOException e)
{
    System.err.println("Caught IOException: " + e.getMessage());
}
```

However, this duplicates code, thus making the code difficult to read and error prone if you later modify it. For example, if you add to the try block code that can throw a new type of exception, you have to remember to close the *PrintWriter* within the new exception handler.

Exception Handling

Putting It All Together

The previous sections describe how to construct the try, catch, and finally code blocks for the writeList method in the ListOfNumbers class. When all of the components are put together, the writeList method looks like this:

```
public void writeList()
{
        PrintWriter out = null;
        try
        {
            System.out.println("Entering try statement");
            out = new PrintWriter(new FileWriter("OutFile.txt"));
            for (int i = 0; i < SIZE; i++)
                out.println("Value at: " + i + " = " + victor.elementAt(i));
        }
        catch (ArrayIndexOutOfBoundsException e)
        {
            System.err.println("Caught " + "ArrayIndexOutOfBoundsException: " +
                                     e.getMessage());
        }
        catch (IOException e)
        {
            System.err.println("Caught IOException: " + e.getMessage());
        }
        finally
        {
            if (out != null)
            {
                System.out.println("Closing PrintWriter");
                out.close();
            } else
                System.out.println("PrintWriter not open");
        }
    }
```

Exception Handling

Specifying the Exceptions Thrown by a Method

The previous section showed you how to write an exception handler for the *writeList* method in the *ListOfNumbers* class. Sometimes, it’s appropriate for your code to catch exceptions that can occur within it. In other cases, however, it’s better to let a method further up the call stack handle the exception.

For example, if you were providing the *ListOfNumbers* class as part of a package of classes, you probably couldn’t anticipate the needs of all the users of your package. In this case, it’s better to not catch the exception and to allow a method further up the call stack to handle it.

If the *writeList* method doesn’t catch the checked exceptions that can occur within it, the *writeList* method must specify that it can throw these exceptions. Let’s modify the original *writeList* method to specify the exceptions that it can throw *instead* of catching them. To remind you, here’s the original version of the writeList method that won’t compile:

```
// Note: This method won’t compile by design!
public void writeList()
{
    PrintWriter out =
                new PrintWriter(new FileWriter("OutFile.txt"));
    for (int i = 0; i < SIZE; i++)
    {
        out.println("Value at: " + i + " = " + victor.elementAt(i));
    }
    out.close();
}
```

Exception Handling

Specifying the Exceptions Thrown by a Method

To specify that *writeList* can throw two exceptions, you add a *throws* clause to the method declaration for the *writeList* method. The *throws* clause comprises the throws keyword followed by a comma-separated list of all the exceptions thrown by that method. The clause goes after the method name and argument list and before the brace that defines the scope of the method. Here's an example:

```
public void writeList() throws IOException, ArrayIndexOutOfBoundsException
{
      […]
}
```

Remember that *ArrayIndexOutOfBoundsException* is a runtime exception, so you don't *have* to specify it in the throws clause, although you can. You could just write this:

```
public void writeList() throws IOException
```

Exception Handling

How to Throw Exceptions

Before you can catch an exception, some code somewhere must throw one. Any code can throw an exception: your code, code from a package written by someone else (such as the packages that come with the Java platform), or the Java runtime environment. Regardless of what throws the exception, it's always thrown with the throw statement.

As you have probably noticed, the Java platform provides numerous exception classes. All these classes are descendants of the *Throwable* class and all allow programs to differentiate among the various types of exceptions that can occur during the execution of a program.

You also can create your own exception classes to represent problems that can occur within the classes that you write. In fact, if you are a package developer, you might have to create your own set of exception classes so as to allow your users to differentiate an error that can occur in your package from errors that occur in the Java platform or other packages.

Exception Handling

The throw Statement

All methods use the throw statement to throw an exception. The throw statement requires a single argument: a throwable object. Throwable objects are instances of any subclass of the Throwable class. Here's an example of a throw statement:

```
throw someThrowableObject;
```

Let's look at the throw statement in context. The following pop method is taken from a class that implements a common stack object. The method removes the top element from the stack and returns the object:

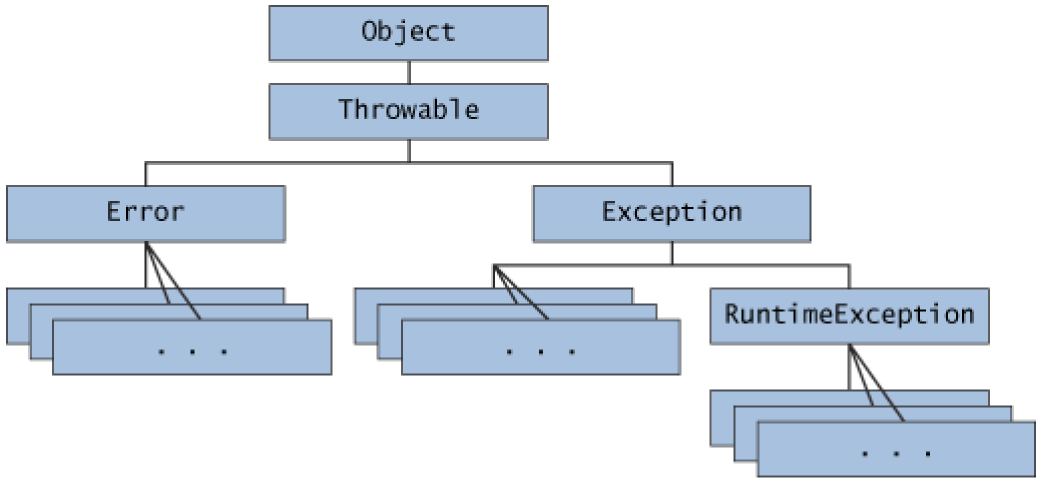
```
public Object pop() throws EmptyStackException
{
    Object obj;

    if (size == 0)
        throw new EmptyStackException();

    obj = objectAt(SIZE - 1);
    setObjectAt(SIZE - 1, null);
    size--;
    return obj;
}
```

The *pop* method checks whether any elements are on the stack. If the stack is empty (its size is equal to 0), *pop* instantiates a new *EmptyStackException* object (a member of java.util) and throws it. Note that the declaration of the *pop* method contains a *throws* clause. *EmptyStackException* is a checked exception, and the *pop* method makes no effort to catch it. Hence, the method must use the *throws* clause to declare that it can throw that type of exception.

Exception Handling

Throwable Class and Its Subclasses

The objects that inherit from the Throwable class include direct descendants (objects that inherit directly from the Throwable class) and indirect descendants (objects that inherit from children or grandchildren of the Throwable class). The figure below illustrates the class hierarchy of the Throwable class and its most significant subclasses. As you can see, Throwable has two direct descendants: Error and Exception



Exception Handling

Error Class

When a dynamic linking failure or other “hard” failure in the Java Virtual Machine occurs, the virtual machine throws an Error. Simple programs typically *do not* catch or throw Errors.

Exception Class

Most programs throw and catch objects that derive from the *Exception* class. An Exception indicates that a problem occurred, but it is not a serious system problem. Most programs you write will throw and catch Exceptions.

The Exception class has many descendants defined in the Java platform. These descendants indicate various types of exceptions that can occur. For example, *IllegalAccessException* signals that a particular method could not be found, and *NegativeArraySizeException* indicates that a program attempted to create an array with a negative size.

One Exception subclass has special meaning: *RuntimeException*. RuntimeExceptions are exceptions that occur within the Java Virtual Machine during runtime. An example of a runtime exception is *NullPointerException*, which occurs when a method tries to access a member of an object through a null reference.

Exception Handling

Chained Exceptions

Chained exceptions allow you to *re-throw* an exception, providing additional information without losing the original cause of the exception. The chained exception API was introduced in 1.4 by adding a *cause* property of type *Throwable* to exceptions. Two methods and two constructors were added to Throwable, the class from which all exceptions inherit.

Since every *Throwable* can have a cause, each exception can have a cause, which itself can have a cause, and so on. The methods and constructors in Throwable that support chained exceptions are:

```
Throwable getCause()
Throwable initCause(Throwable)
Throwable(String, Throwable)
Throwable(Throwable)
```

The Throwable argument to *initCause* and the Throwable constructors is the exception that caused the current exception. *getCause* returns the exception that caused the current exception, and *initCause* returns the current exception.

Exception Handling

Chained Exceptions

The following example shows how to use a chained exception:

```
try
{
...
}
catch (IOException e)
{
     throw new SampleException("Other IOException", e);
}
```

In this example, when an *IOException* is caught, a new *SampleException* exception is created with the original cause attached and the chain of exceptions is thrown up to the next higher-level exception handler.

Exception Handling

Accessing Stack Trace Information

Now let’s suppose that the higher-level exception handler wants to dump the stack trace in its own format.

Definition: A stack trace provides information on the execution history of the current thread and lists the names of the classes and methods that were called at the point when the exception occurred. A stack trace is a useful debugging tool that you'll normally take advantage of when an exception has been thrown.

The following code shows how to call the *getStackTrace* method on the exception object:

```
catch (Exception cause)
{
    StackTraceElement elements[] = cause.getStackTrace();

    for (int i = 0; n = elements.length; i < n; i++)
    {
        System.err.println(elements[i].getFileName() + ":"
                + elements[i].getLineNumber()
                + ">> " + elements[i].getMethodName() + "()");
    }
}
```

Exception Handling

Creating Your Own Exception Classes

When faced with choosing the type of exception to throw, you can either use one written by someone else—the Java platform provides exception classes that you can use—or you can write one of your own. You should write your own exception classes if you answer yes to any of the following questions. Otherwise, you can probably use someone else's.

- Do you need an exception type that isn't represented by those in the Java platform?
- Would it help your users if they could differentiate your exceptions from those thrown by classes written by other vendors?
- Does your code throw more than one related exception?
- If you use someone else's exceptions, will your users have access to those exceptions? A similar question is - should your package be independent and self-contained?

Suppose you are writing a linked list class that you're planning to distribute as freeware. Your linked list class supports the following methods, among others:

`objectAt(int n)`
Returns the object in the nth position in the list. Throws an exception if the argument is less than 0 or larger than the number of objects currently in the list.

`firstObject()`
Returns the first object in the list. Throws an exception if the list contains no objects.

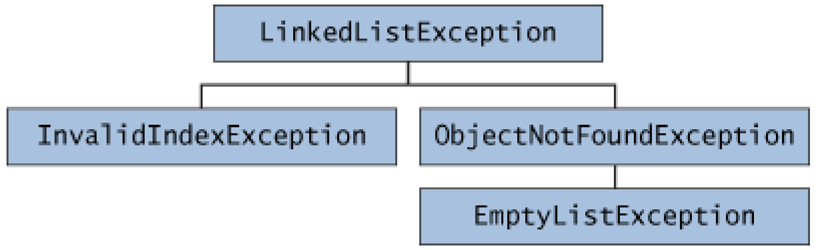
`indexOf(Object o)`
Searches the list for the specified Object and returns its position in the list. Throws an exception if the object passed into the method is not in the list.

Exception Handling

Creating Your Own Exception Classes

The linked list class can throw multiple exceptions, and it would be convenient to be able to catch all exceptions thrown by the linked list with one exception handler. Also, if you plan to distribute your linked list in a package, all related code should be packaged together. Thus, the linked list should provide its own set of exception classes.

The next figure illustrates one possible class hierarchy for the exceptions thrown by the linked list.



Exception Handling

Choosing a Superclass

Any Exception subclass can be used as the parent class of *LinkedListException*. However, a quick perusal of those subclasses shows that they are inappropriate because they are either too specialized or completely unrelated to *LinkedListException*. Therefore, the parent class of *LinkedListException* should be *Exception*.

Most applets and applications that you write will throw objects that are Exceptions. Errors are normally used for serious, hard errors in the system, such as those that prevent the Java Virtual Machine from running.

Note: For readable code, it's good practice to append the string Exception to the names of all classes that inherit (directly or indirectly) from the Exception class.

Exception Handling

Summary

A program can use exceptions to indicate that an error occurred. To throw an exception, you use the throw statement and provide it with an exception object — a descendant of *Throwable* — to provide information about the specific error that occurred. A method that throws an uncaught checked exception must include a throws clause in its declaration.

A program can catch exceptions by using a combination of the try, catch, and finally blocks. The try block identifies a block of code in which an exception can occur. The catch block identifies a block of code, known as an exception handler, which can handle a particular type of exception. The finally block identifies a block of code that cleans up regardless of whether an exception occurred within the try block. The try statement must should contain at least one catch block or a finally block and may have multiple catch blocks.

The class of the exception object indicates the type of the exception thrown. The exception object can contain further information about the error, including an error message. With the chained exception feature added in 1.4, an exception can point to the exception that caused it, which can in turn point to the exception that caused it, and so on.

Exception Handling

Java Exception Examples

A method (“setSpeed”) with an exception thrown & handled

```
public void setSpeed (double speedIn)
{
  try
  {
    if (speedIn <= 0.0)
      throw new InvalidRangeException (“setSpeed(double): “ + speedIn);
  }
  catch(InvalidRangeException e)
  {
    System.out.println (e.getMessage());
    e.printStackTrace();
    System.exit (0);
  }
    speed = speedIn;
}
```

Exception Handling

Java Exception Examples

A method ("setSpeed") with an exception Thrown & Propagated

```
public void setSpeed (double speedIn) throws InvalidRangeException
{
    if (speedIn <= 0.0)
      throw new InvalidRangeException ("setSpeed(double): " + speedIn);

    speed = speedIn ;
}
```

Note – Unless otherwise specified, our project methods will "Throw & Propagate" Exceptions.

When throwing Exceptions in our project applications, NEVER simply throw "Exception" – we will throw ONLY subclasses of "Exception. ("Exception" is too generic!).
These Exception subclasses should have useful names like InvalidDataException, InvalidIndexException, NotFoundException, NullValuePassedException, etc.

Exception Handling

Java Exception Examples

Example: Throw & Propagate Exception subclass as we should do in our project

```
public void setLength (double len) throws InvalidDataException
{
    if (len <= 0.0)
    {
        throw new InvalidDataException (getClass ().getName () +
                                         ").setLength(double)" , len);
    }
    length = len;
}
```

When a method “throws” an exception that it does not handle (i.e., it “propagates” it), the method itself should be declared to “throw” the propagated exception:

```
public void setLength (double len) throws InvalidDataException
```

Exception Handling

Java Exception Examples

Our “propagated” Exceptions will be “handled” at strategic points throughout the application – places at which the “problem” can be properly dealt with.

Example: Handling “Propagated” Exceptions

Examine the following method examples:

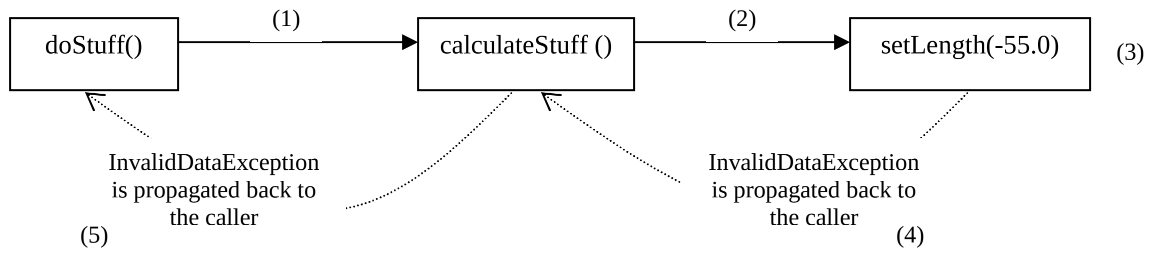
```
public void doStuff()
{
   try
   {
      calculateStuff();
   }
      catch(InvalidDataException e)
      {
         Logger.logError(e.getMessage());
         e.printStackTrace();
         System.exit(0);
      }
}

public void calculateStuff()
            throws InvalidDataException
{
   setLength (-55.0);
}
```

```
public void setLength(double len)
                 throws InvalidDataException
{
   if ( len <= 0.0 )
   {
      throw new InvalidDataException(getClass().getName() +
                                    ").setLength(double)", len ) ;
   }
   length = len ;
}
```

Exception Handling

Java Exception Examples

A call to “doStuff()” results in the following behavior:


doStuff()
(1)
calculateStuff ()
(2)
setLength(-55.0)
(3)
InvalidDataException is propagated back to the caller
(4)
InvalidDataException is propagated back to the caller
(5)

1. From within the “try” block, method “doStuff()” calls “calculateStuff()”.

2. Then “calculateStuff()” calls “setLength(-55.0)”

3. Next “setLength” ends up throwing “InvalidDataException” because the parameter passed in is “<= 0.0”.

   a. This Exception is not handled in “setLength” so therefore it is “propagated” back to the caller (“calculateStuff()”). Since “setLength” propagates it’s Exceptions, the method “setLength” must be declared to throw “InvalidDataException”.

   b. As mentioned, the Exception is “propagated” back to the caller – so the Exception is propagated to “calculateStuff()”.

Exception Handling

Java Exception Examples

4. Since the method “calculateStuff()” calls a method that propagates exceptions that “calculateStuff()” does not handle, then “calculateStuff()”must be declared to throw “InvalidDataException” and will then “propagate” the exception back to the caller (“doStuff()”) as well.

5. Finally, the Exception is propagated back to “doStuff()”. The “doStuff()” method *does* handle “InvalidDataException” (there is a “catch: block” for it) so the exception is finally handled there and is not propagated any further. Since “doStuff()” handles the Exception and does not propagate it further, the “doStuff()” method is *not* defined as throwing any Exceptions.