Exceptions

- Java uses exceptions as a way of signaling serious problems when you execute a program.
- The standard classes use them extensively.
- Since they arise in your Java programs when things go wrong, they are a very basic consideration when you are designing and writing your programs.

- In this lecture you'll learn:
  - What an exception is
  - How you handle exceptions in your programs
  - The standard exceptions in Java
  - How to guarantee a particular block of code will always be executed
  - How to define and use your own types of exceptions
  - How to throw exceptions in your programs

The Idea Behind Exceptions

- An exception usually signals an error and is so called because errors in your Java programs are bound to be the exception rather than the rule—by definition!
- An exception doesn't always indicate an error though—it can also signal some particularly unusual event in your program that deserves special attention.
- If you try to deal with some and often highly unusual error conditions that might arise in the midst of the code that deals with the normal operation of the program, your program structure will soon become very complicated and difficult to understand.
- One major benefit of having an error signaled by an exception is that it separates the code that deals with errors from the code that is executed when things are moving along smoothly.
- Another positive aspect of exceptions is that they provide a way of enforcing a response to particular errors.
- With many kinds of exceptions, you must include code in your program to deal with them; otherwise, your code will not compile.

Cont'd

- One important idea to grasp is that not all errors in your programs need to be signaled by exceptions.
- Exceptions should be reserved for the unusual or catastrophic situations that can arise.
- A user entering incorrect input to your program for instance is a normal event and should be handled without recourse to exceptions.
- The reason for this is that dealing with exceptions involves quite a lot of processing overhead, so if your program is handling exceptions a lot of the time it will be a lot slower than it needs to be.

What is an exception?

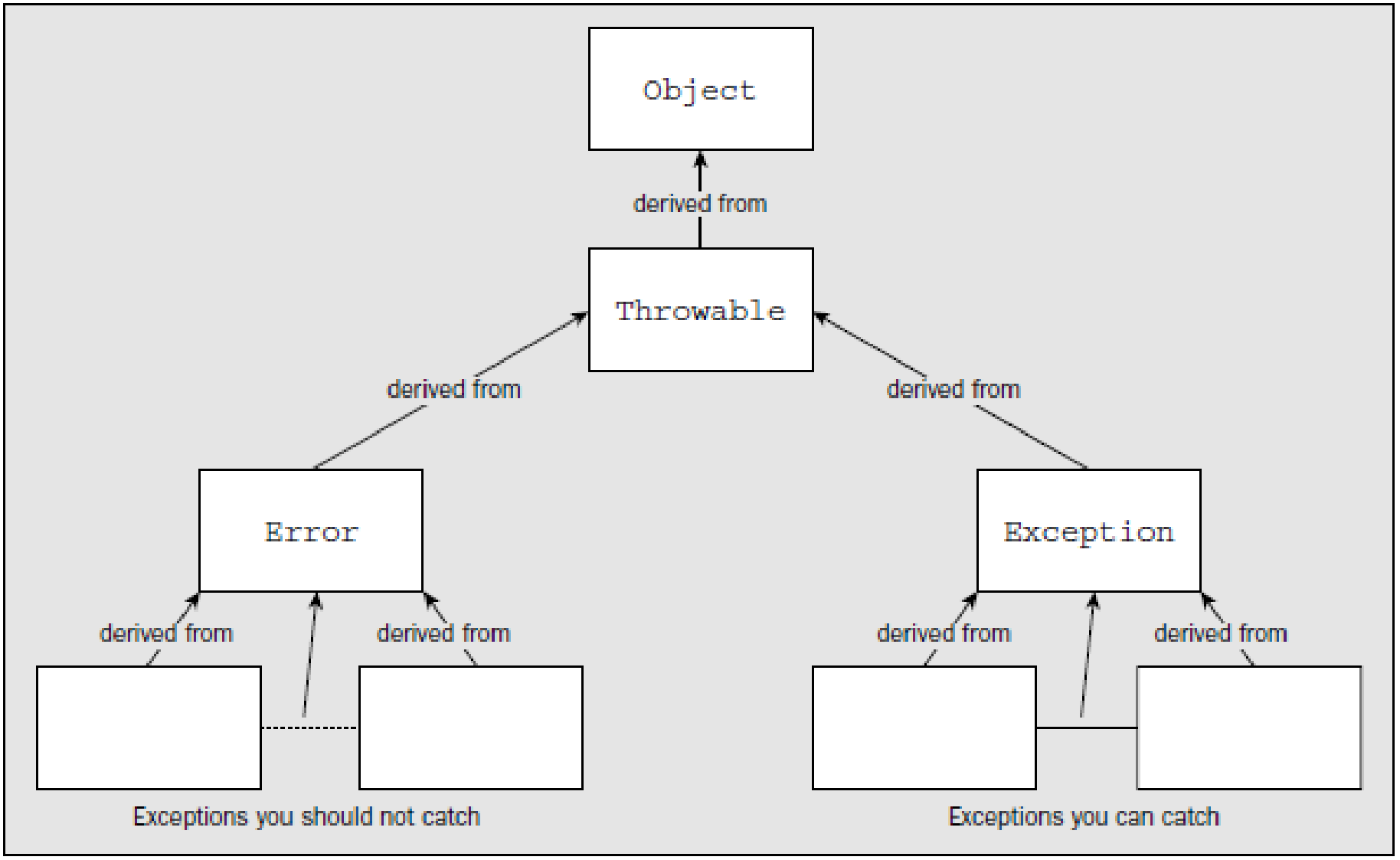
- An exception in Java is an object that's created when an abnormal situation arises in your program.
- This exception object has fields that store information about the nature of the problem.
- The exception is said to be thrown—that is, the object identifying the exceptional circumstance is tossed as an argument to a specific piece of program code that has been written specifically to deal with that kind of problem.
- The code receiving the exception object as a parameter is said to catch it.

The situations that cause exceptions are quite diverse, but they fall into four broad categories:

Code or data errors	For example, you attempt an invalid cast of an object, you try to use an array index that's outside the limits for the array, or an integer arithmetic expression has a zero divisor.
Standard method exceptions	For example, if you use the `substring()` method in the `String` class, it can throw a `StringIndexOutOfBoundsException` exception.
Throwing your own exceptions	You'll see later in this chapter how you can throw a few of your own when you need to.
Java errors	These can be due to errors in executing the Java Virtual Machine, which runs your compiled program, but usually arise as a consequence of an error in your program.

Types of Exceptions

- An exception is always an object of some subclass of the standard class Throwable.
- This is true for exceptions that you define and throw yourself, as well as the standard exceptions that arise due to errors in your code.
- It's also true for exceptions that are thrown by methods in one or another of the standard packages.
- Two direct subclasses of the class Throwable—the class Error and the class Exception—cover all the standard exceptions.
- Both these classes themselves have subclasses that identify specific exception conditions.


Object
derived from
Throwable
derived from
derived from
Error
Exception
derived from
derived from
derived from
derived from
Exceptions you should not catch
Exceptions you can catch

Error Exceptions

The exceptions that are defined by the `Error` class and its subclasses are characterized by the fact that they all represent conditions that you aren't expected to do anything about, so you aren't expected to catch them. `Error` has three direct subclasses—`ThreadDeath`, `LinkageError`, and `VirtualMachineError`:

- The first of these sounds the most serious, but in fact it isn't. A `ThreadDeath` exception is thrown whenever an executing thread is deliberately stopped, and for the thread to be destroyed properly, you should not catch this exception. In some circumstances you might want to catch it—for clean-up operations, for example—in which case you must be sure to rethrow the exception to allow the thread to die peacefully. When a `ThreadDeath` exception is thrown and not caught, it's the thread that ends, not the program.
- The `LinkageError` exception class has subclasses that record serious errors with the classes in your program. Incompatibilities between classes or attempting to create an object of a nonexistent class type are the sorts of things that cause these exceptions to be thrown.
- The `VirtualMachineError` class has four subclasses that specify exceptions that will be thrown when a catastrophic failure of the Java Virtual Machine occurs. You aren't prohibited from trying to deal with these exceptions, but in general, there's little point in attempting to catch them.

The exceptions that correspond to objects of classes derived from `LinkageError` and `VirtualMachineError` are all the result of catastrophic events or conditions. You can do little or nothing to recover from them during the execution of the program. In these sorts of situations, all you can usually do is read the error message that is generated by the exception being thrown and then, particularly in the case of a `LinkageError` exception, try to figure out what might be wrong with your code to cause the problem.

RuntimeException Exceptions

For almost all the exceptions that are represented by subclasses of the `Exception` class, you must include code in your programs to deal with them if your code may cause them to be thrown. If a method in your program has the potential to generate an exception of a type that has `Exception` as a superclass, you must either handle the exception within the method or register that your method may throw such an exception. If you don't, your program will not compile. You'll see in a moment how to handle exceptions and how to specify that a method can throw an exception.

One group of subclasses of `Exception` that is exempted from this is comprised of those derived from `RuntimeException`. The reason that `RuntimeException` exceptions are treated differently, and that the compiler allows you to ignore them, is that they generally arise because of serious errors in your code. In most cases you can do little to recover the situation. However, in some contexts for some of these exceptions, this is not always the case, and you may well want to include code to recognize them. Quite a lot of subclasses of `RuntimeException` are used to signal problems in various packages in the Java class library. Let's look at the exception classes that have `RuntimeException` as a base that are defined in the `java.lang` package.

The subclasses of `RuntimeException` defined in the standard package `java.lang` are:

Class Name	Exception Condition Represented
`ArithmeticException`	An invalid arithmetic condition has arisen, such as an attempt to divide an integer value by zero.
`IndexOutOfBoundsException`	You've attempted to use an index that is outside the bounds of the object it is applied to. This may be an array, a `String` object, or a `Vector` object. The `Vector` class is defined in the standard package `java.util`. You will be looking into the `Vector` class in Chapter 14.
`NegativeArraySizeException`	You tried to define an array with a negative dimension.
`NullPointerException`	You used an object variable containing `null`, when it should refer to an object for proper operation — for example, calling a method or accessing a data member.
`ArrayStoreException`	You've attempted to store an object in an array that isn't permitted for the array type.
`ClassCastException`	You've tried to cast an object to an invalid type — the object isn't of the class specified, nor is it a subclass or a superclass of the class specified.
`IllegalArgumentException`	You've passed an argument to a method that doesn't correspond with the parameter type.
`SecurityException`	Your program has performed an illegal operation that is a security violation. This might be trying to read a file on the local machine from an applet.
`IllegalMonitorStateException`	A thread has tried to wait on the monitor for an object that the thread doesn't own. (You'll look into threads in Chapter 16.)
`IllegalStateException`	You tried to call a method at a time when it was not legal to do so.
`UnsupportedOperationException`	This is thrown if you request an operation to be carried out that is not supported.

Other Subclasses of Exception

- For all the other classes derived from the class Exception, the compiler will check that you've either
  - handled the exception in a method where the exception may be thrown or
  - that you've indicated that the method can throw such an exception.

- If you do neither, your code won't compile.

- Apart from a few that have RuntimeException as a base, all exceptions thrown by methods in the Java class library are of a type that you must deal with.

Dealing with Exceptions

- As discussed, if your code can throw exceptions other than those of type Error or type RuntimeException, you must do something about it.
- Whenever you write code that can throw an exception, you have a choice.
- You can supply code within the method to deal with any exception that is thrown, or you can essentially ignore it by enabling the method containing the exception-throwing code to pass it on to the code that called the method.

Specifying the Exceptions a Method Can Throw

Suppose you have a method that can throw an exception that is neither a subclass of `RuntimeException` nor of `Error`. This could be an exception of type `IOException`, for example, which can be thrown if your method involves some file input or output operations. If the exception isn't caught and disposed of in the method, you must at least declare that the exception can be thrown. But how do you do that?

You do it simply by adding a `throws` clause in the definition of the method. Suppose you write a method that uses the methods from classes that support input/output that are defined in the package `java.io`. You'll see in the chapters devoted to I/O operations that some of these can throw exceptions represented by objects of classes `IOException` and `FileNotFoundException`. Neither of these is a subclass of `RuntimeException` or `Error`, so the possibility of an exception being thrown needs to be declared. Since the method can't handle any exceptions it might throw, for the simple reason that you don't know how to do it yet, it must be defined as:

```
double myMethod() throws IOException, FileNotFoundException {
  // Detail of the method code...
}
```

As the preceding fragment illustrates, to declare that your method can throw exceptions you just put the `throws` keyword after the parameter list for the method. Then add the list of classes for the exceptions that might be thrown, separated by commas. This has a knock-on effect—if another method calls this method, it too must take account of the exceptions this method can throw. After all, calling a method that can throw an exception is clearly code where an exception may be thrown. The calling method definition must either deal with the exceptions or declare that it can throw these exceptions as well. It's a simple choice. You either pass the buck or decide that the buck stops here. The compiler checks for this and your code will not compile if you don't do one or the other. The reasons for this will become obvious when you look at the way a Java program behaves when it encounters an exception.

Handling Exceptions

If you want to deal with the exceptions where they occur, you can include three kinds of code blocks in a method to handle them — `try`, `catch`, and `finally` blocks:

- A `try` block encloses code that may give rise to one or more exceptions. Code that can throw an exception that you want to catch must be in a `try` block.
- A `catch` block encloses code that is intended to handle exceptions of a particular type that may be thrown in the associated `try` block. I'll get to how a `catch` block is associated with a `try` block in a moment.
- The code in a `finally` block is always executed before the method ends, regardless of whether any exceptions are thrown in the `try` block.

The try Block

When you want to catch an exception, the code in the method that might cause the exception to be thrown must be enclosed in a `try` block. Code that can cause exceptions need not be in a `try` block, but in this case, the method containing the code won't be able to catch any exceptions that are thrown and the method must declare that it can throw the types of exceptions that are not caught.

A `try` block is simply the keyword `try`, followed by braces enclosing the code that can throw the exception:

```
try {
  // Code that can throw one or more exceptions
}
```

Although I am discussing primarily exceptions that you must deal with here, a `try` block is also necessary if you want to catch exceptions of type `Error` or `RuntimeException`. When you come to a working example in a moment, you will use an exception type that you don't have to catch, simply because exceptions of this type are easy to generate.

The catch Block

You enclose the code to handle an exception of a given type in a `catch` block. The `catch` block must immediately follow the `try` block that contains the code that may throw that particular exception. A `catch` block consists of the keyword `catch` followed by a single parameter between parentheses that identifies the type of exception that the block is to deal with. This is followed by the code to handle the exception enclosed between braces:

```
try {
  // Code that can throw one or more exceptions

} catch(ArithmeticException e) {
  // Code to handle the exception
}
```

This `catch` block handles only `ArithmeticException` exceptions. This implies that this is the only kind of exception that can be thrown in the `try` block. If others can be thrown, this won't compile. I will come back to handling multiple exception types in a moment.

In general, the parameter for a `catch` block must be of type `Throwable` or one of the subclasses of the class `Throwable`. If the class that you specify as the parameter type has subclasses, the `catch` block will be expected to process exceptions of that class type, plus all subclasses of the class. If you specified the parameter to a `catch` block as type `RuntimeException`, for example, the code in the `catch` block would be invoked for exceptions defined by the class `RuntimeException`, or any of its subclasses.

Example

```
public class TestTryCatch {
  public static void main(String[] args) {
    int i = 1;
    int j = 0;

    try {
      System.out.println("Try block entered " + "i = "+ i + " j = "+j);
      System.out.println(i/j);                    // Divide by 0 - exception thrown
      System.out.println("Ending try block");

    } catch(ArithmeticException e) {              // Catch the exception
      System.out.println("Arithmetic exception caught");
    }

    System.out.println("After try block");
    return;
  }
}
```

try catch Bonding

- The try and catch blocks are bonded together.
- You must not separate them by putting statements between the two blocks, or even by putting braces around the try keyword and the try block itself.
- If you have a loop block that is also a try block, the catch block that follows is also part of the loop.

EXAMPLE: A Loop Block That Is a try Block

```
public class TestLoopTryCatch {
  public static void main(String[] args) {

    int i = 12;

    for(int j=3 ;j>=-1 ; j--)
      try {
        System.out.println("Try block entered " + "i = "+ i + " j = "+j);
        System.out.println(i/j);                    // Divide by 0 - exception thrown
       System.out.println("Ending try block");

      } catch(ArithmeticException e) {              // Catch the exception
        System.out.println("Arithmetic exception caught");
      }

    System.out.println("After try block");
    return;
  }
}
```

Multiple catch Blocks

If a `try` block can throw several different kinds of exception, you can put several `catch` blocks after the `try` block to handle them:

```
try {
  // Code that may throw exceptions

} catch(ArithmeticException e) {
  // Code for handling ArithmeticException exceptions
} catch(IndexOutOfBoundsException e) {
  // Code for handling IndexOutOfBoundsException exceptions
}
// Execution continues here...
```

Exceptions of type `ArithmeticException` will be caught by the first `catch` block, and exceptions of type `IndexOutOfBoundsException` will be caught by the second. Of course, if an `ArithmeticException` exception is thrown, only the code in that `catch` block will be executed. When it is complete, execution continues with the statement following the last `catch` block.

When you need to catch exceptions of several different types that may be thrown in a `try` block, the order of the `catch` blocks can be important. When an exception is thrown, it will be caught by the first `catch` block that has a parameter type that is the same as that of the exception, or a type that is a superclass of the type of the exception. An extreme case would be if you specified the `catch` block parameter as type `Exception`. This will catch any exception that is of type `Exception`, or of a class type that is derived from `Exception`. This includes virtually all the exceptions you are likely to meet in the normal course of events.

This has implications for multiple `catch` blocks relating to exception class types in a hierarchy. The `catch` blocks must be in sequence with the most derived type first, and the most basic type last. Otherwise, your code will not compile. The simple reason for this is that if a `catch` block for a given class type precedes a `catch` block for a type that is derived from the first, the second `catch` block can never be executed, and the compiler will detect that this is the case.

Suppose you have a `catch` block for exceptions of type `ArithmeticException` and another for exceptions of type `Exception` as a catch-all. If you write them in the following sequence, exceptions of type `ArithmeticException` could never reach the second `catch` block because they will always be caught by the first:

```
// Invalid catch block sequence - won't compile!
try {
  // try block code

} catch(Exception e) {
  // Generic handling of exceptions
} catch(ArithmeticException e) {
  // Specialized handling for these exceptions
}
```

Of course, this won't get past the compiler—it would be flagged as an error.

To summarize—if you have `catch` blocks for several exception types in the same class hierarchy, you must put the `catch` blocks in order, starting with the lowest subclass first and then progressing to the highest superclass.

The finally Block

- The immediate nature of an exception being thrown means that execution of the try block code breaks off, regardless of the importance of the code that follows the point at which the exception was thrown.
- This introduces the possibility that the exception leaves things in an unsatisfactory state.
- You might have opened a file, for example, and because an exception was thrown, the code to close the file is not executed.
- The finally block provides the means for you to clean up at the end of executing a try block.
- You use a finally block when you need to be sure that some particular code is run before a method returns, no matter what exceptions are thrown within the associated try block.
- A finally block is always executed, regardless of whether or not exceptions are thrown during the execution of the associated try block.
- If a file needs to be closed, or a critical resource released, you can guarantee that it will be done if the code to do it is put in a finally block.
- The finally block has a very simple structure:

```
finally {
    // Clean-up code to be executed last
}
```

- Just like a catch block, a finally block is associated with a particular try block, and it must be located immediately following any catch blocks for the try block.
- If there are no catch blocks, then you position the finally block immediately after the try block.
- If you don't do this, your program will not compile.

Structuring a Method

You've looked at the blocks you can include in the body of a method, but it may not always be obvious how they are combined. The first thing to get straight is that a `try` block plus any corresponding `catch` blocks and the `finally` block all bunch together in that order:

```
try {
  // Code that may throw exceptions...

} catch(ExceptionType1 e) {
  // Code to handle exceptions of type ExceptionType1 or subclass
} catch(ExceptionType2 e) {
  // Code to handle exceptions of type ExceptionType2 or subclass
... // more catch blocks if necessary
} finally {
  // Code always to be executed after try block code
}
```

You can't have just a `try` block by itself. Each `try` block must always be followed by at least one block that is either a `catch` block or a `finally` block.

You must not include other code between a `try` block and its `catch` blocks, or between the `catch` blocks and the `finally` block. You can have other code that doesn't throw exceptions after the `finally` block, and you can have multiple `try` blocks in a method. In this case, your method might be structured as shown below

```
double doSomething( int aParam)
   throws ExceptionType1, ExceptionType2{

   //Code that does not throw exceptions

   //Set of try/catch/finally blocks...

   //Code that does not throw exceptions

   //Set of try/catch/finally blocks...

   //Code that does not throw exceptions

   //Set of try/catch/finally blocks...

   //Code that does not throw exceptions

   //...

}
```

Typical Structure

```
try{
   //Code that does throw exceptions
}
```

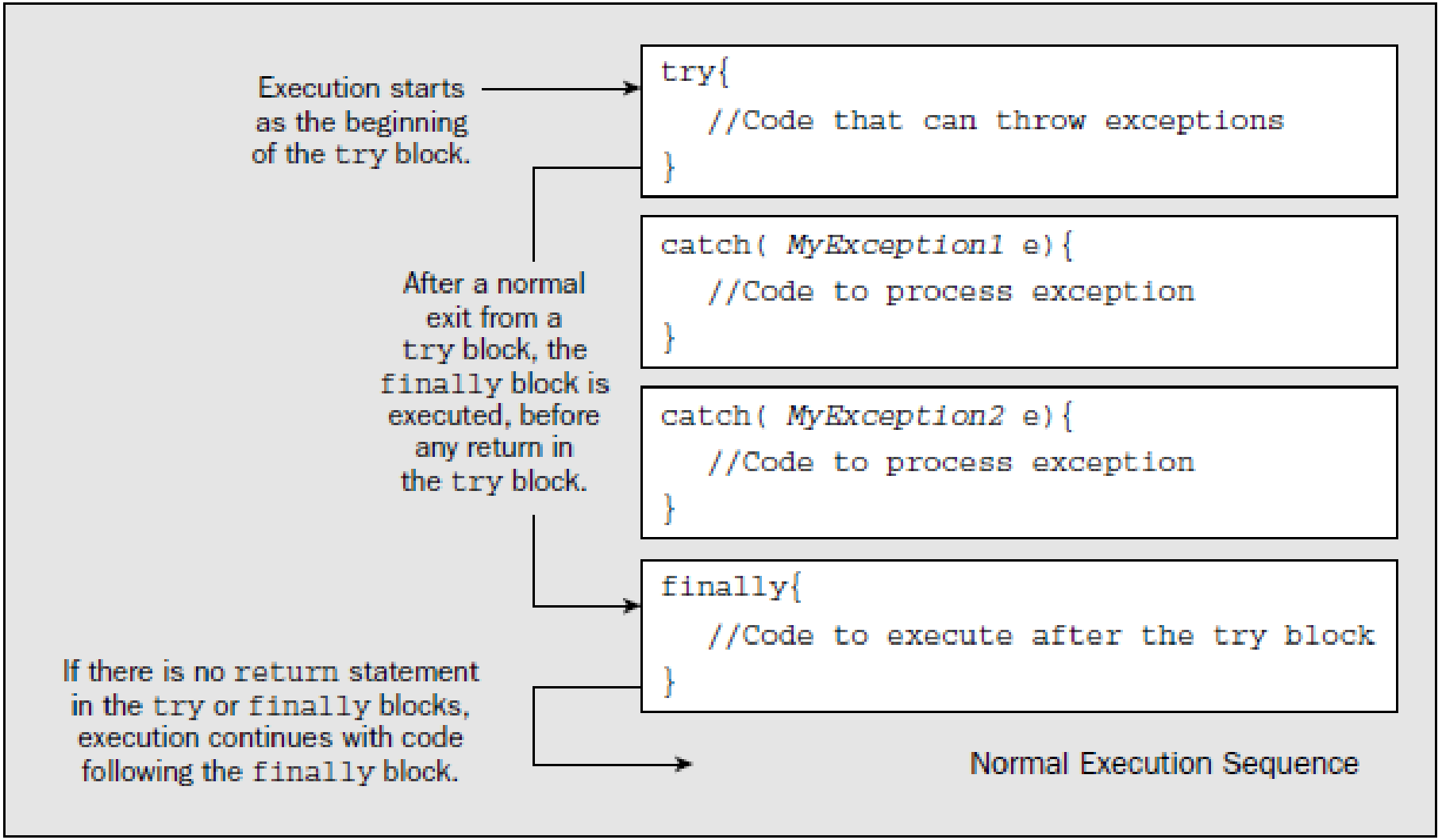
```
catch( MyException1 e){
   //Code to process exception
}
```

```
catch( MyException2 e){
   //Code to process exception
}
```

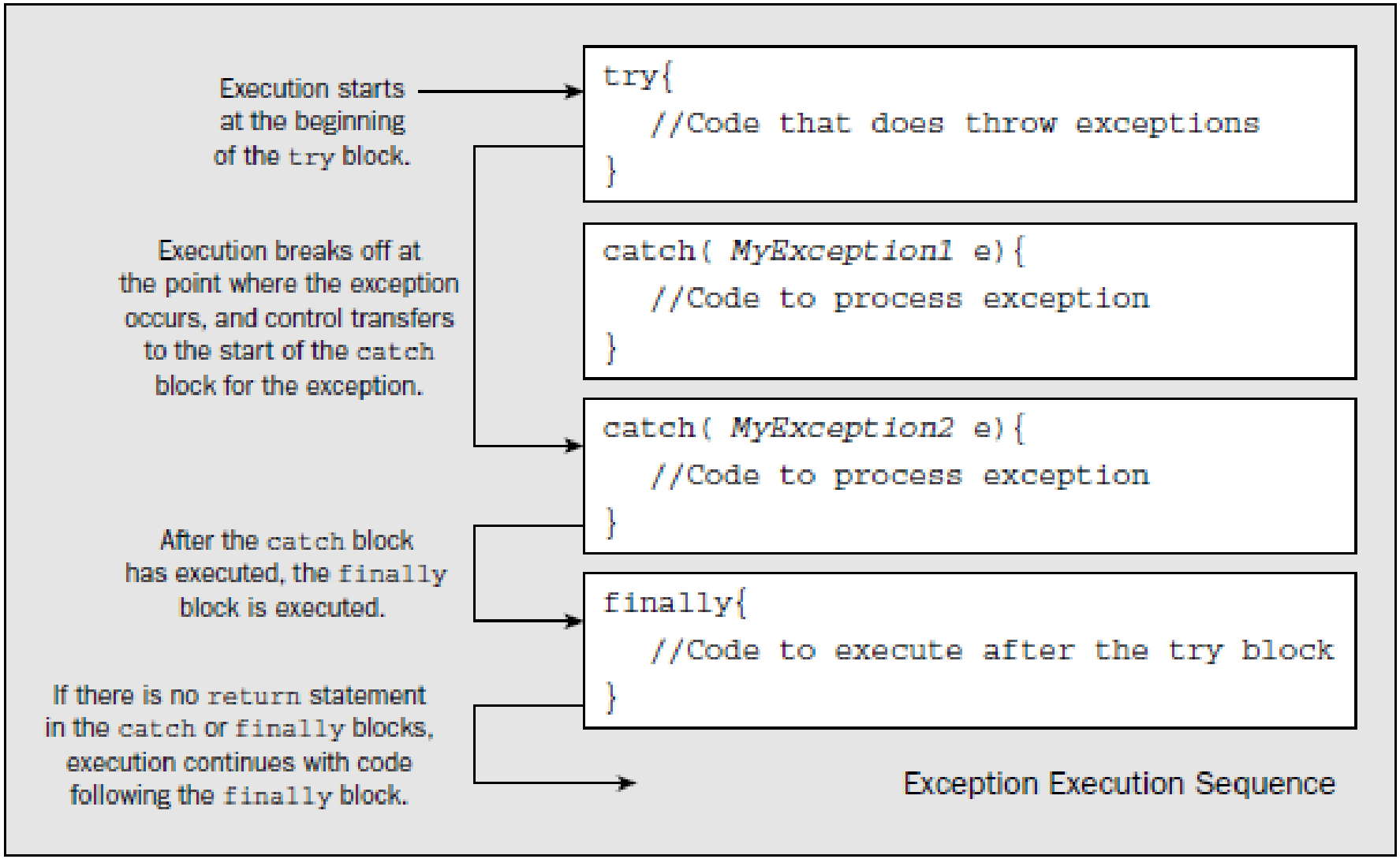
```
finally{
   //Code to execute after the try block
}
```

In general there can be as many `catch` blocks as required, and there may be none. The `finally` block is optional if there is a `catch` block.

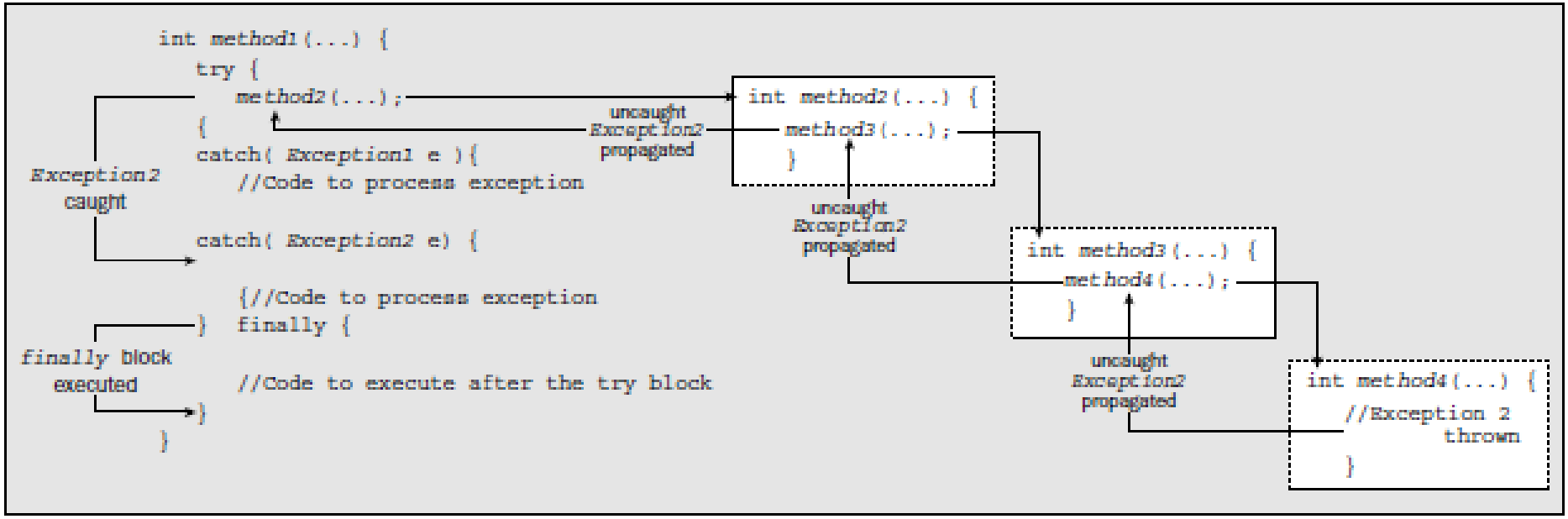
The sequence of execution when no exceptions occur



Execution When an Exception Is Thrown



Execution When an Exception Is Not Caught



Nested try Blocks

```
try {
    try {
        //1st inner try block code...
    } catch( Exception1 ) {
        //...
    }
    //Outer try block code...
    try {
        //2nd inner try block code...
    }catch( Exception1 e ) {
        //try block code...
    }

}catch( Exception2 e ){
    //Outer catch block code...
}
```

Exceptions of type `Exception2` thrown anywhere in here that are not caught will be caught by the `catch` block for the outer `try` block.

Rethrowing Exceptions

Even though you may need to recognize that an exception has occurred in a method by implementing a `catch` clause for it, this is not necessarily the end of the matter. In many situations, the calling program may need to know about it—perhaps because it will affect the continued operation of the program or because the calling program may be able to compensate for the problem.

If you need to pass an exception that you have caught on to the calling program, you can rethrow it from within the `catch` block using a `throw` statement. For example:

```
try {
  // Code that originates an arithmetic exception

} catch(ArithmeticException e) {
  // Deal with the exception here
  throw e;                    // Rethrow the exception to the calling program
}
```

The `throw` statement is the keyword `throw` followed by the exception object to be thrown.

Exception Objects

- Well, you now understand how to put try blocks together with catch blocks and finally blocks in your methods.
- You may be thinking at this point that it seems a lot of trouble to go to just to display a message when an exception is thrown.
- You may be right, but whether you can do very much more depends on the nature and context of the problem.
- In many situations a message may be the best you can do, although you can produce messages that are a bit more informative than those you've used so far in our examples.
- The exception object that is passed to a catch block can provide additional information about the nature of the problem that originated it.
- To understand more about this, let's first look at the members of the base class for exceptions Throwable because these will be inherited by all exception classes and are therefore contained in every exception object that is thrown.

The Throwable Class

The `Throwable` class is the class from which all Java exception classes are derived — that is, every exception object will contain the methods defined in this class. The `Throwable` class has two constructors: a default constructor and a constructor that accepts an argument of type `String`. The `String` object that is passed to the constructor is used to provide a description of the nature of the problem causing the exception. Both constructors are public.

Objects of type `Throwable` contain two items of information about an exception:

- A message, which I have just referred to as being initialized by a constructor
- A record of the execution stack at the time the object was created

The `Throwable` class has the following public methods that enable you to access the message and the stack trace:

Method	Description
`getMessage()`	This returns the contents of the message, describing the current exception. This will typically be the fully qualified name of the exception class (it will be a subclass of `Throwable`) and a brief description of the exception.
`printStackTrace()`	This will output the message and the stack trace to the standard error output stream — which is the screen in the case of a console program.
`printStackTrace(PrintStream s)`	This is the same as the previous method except that you specify the output stream as an argument. Calling the previous method for an exception object `e` is equivalent to: `e.printStackTrace(System.err);`

Another method, `fillInStackTrace()`, will update the stack trace to the point at which this method is called. For example, if you put a call to this method in the `catch` block:

```
e.fillInStackTrace();
```

the line number recorded in the stack record for the method in which the exception occurred will be the line where `fillInStackTrace()` is called. The main use of this is when you want to rethrow an exception (so it will be caught by the calling method) and record the point at which it is rethrown. For example:

```
e.fillInStackTrace();                    // Record the throw point
throw e;                                 // Rethrow the exception
```

Standard Exceptions

- The majority of predefined exception classes in Java don't add further information about the conditions that created the exception.
- The type alone serves to differentiate one exception from another in most cases.
- This general lack of additional information is because it can be gleaned in the majority of cases only by prior knowledge of the computation that is being carried out when the exception occurs, and the only person who is privy to that is you, since you're writing the program.
- This should spark the glimmer of an idea.
- If you need more information about the circumstances surrounding an exception, you're going to have to obtain it and, equally important, communicate it to the appropriate point in your program.
- This leads to the notion of defining your own exceptions.

Defining Your Own Exceptions

There are two basic reasons for defining your own exception classes:

- You want to add information when a standard exception occurs, and you can do this by rethrowing an object of your own exception class.
- You may have error conditions that arise in your code that warrant the distinction of a special exception class.

However, you should bear in mind that there's a lot of overhead in throwing exceptions, so it is not a valid substitute for "normal" recovery code that you would expect to be executed frequently. If you have recovery code that will be executed often, then it doesn't belong in a `catch` block, but rather in something like an `if-else` statement.

Defining an Exception Class

Your exception classes must always have `Throwable` as a superclass; otherwise, they will not define an exception. Although you can derive them from any of the standard exception classes, your best policy is to derive them from the `Exception` class. This will allow the compiler to keep track of where such exceptions are thrown in your program and check that they are either caught or declared as thrown in a method. If you use `RuntimeException` or one of its subclasses, the compiler checking for `catch` blocks of your exception class will be suppressed.

Let's go through an example of how you define an exception class:

```
public class DreadfulProblemException extends Exception {
  // Constructors
  public DreadfulProblemException(){ }           // Default constructor

  public DreadfulProblemException(String s) {
    super(s);                                    // Call the base class constructor
  }
}
```

This is the minimum you should supply in your exception class definition. By convention, your exception class should include a default constructor and a constructor that accepts a `String` object as an argument. The message stored in the superclass `Exception` (in fact, in `Throwable`, which is the superclass of `Exception`) will automatically be initialized with the name of your class, whichever constructor for your class objects is used. The `String` passed to the second constructor will be appended to the name of the class to form the message stored in the exception object.

Of course, you can add other constructors. In general, you'll want to do so, particularly when you're rethrowing your own exception after a standard exception has been thrown. In addition, you'll typically want to add instance variables to the class that store additional information about the problem, plus methods that will enable the code in a `catch` block to get at the data. Since your exception class is ultimately derived from `Throwable`, the stack trace information will be automatically available for your exceptions.

Throwing Your Own Exception

As you saw earlier, you throw an exception with a statement that consists of the `throw` keyword, followed by an exception object. This means you can throw your own exception with the following statements:

```
DreadfulProblemException e = new DreadfulProblemException();
throw e;
```

The method will cease execution at this point — unless the code snippet above is in a `try` or a `catch` block with an associated `finally` clause, the contents of which will be executed before the method ends. The exception will be thrown in the calling program at the point where this method was called. The message in the exception object will consist only of the qualified name of the exception class.

If you wanted to add a specific message to the exception, you could define it as:

```
DreadfulProblemException e = new DreadfulProblemException("Uh-Oh, trouble.");
```

An Exception Handling Strategy

You should think through what you want to achieve with the exception handling code in your program. There are no hard and fast rules. In some situations you may be able to correct a problem and enable your program to continue as though nothing happened. In other situations, outputting the stack trace and a fast exit will be the best approach — a fast exit being achieved by calling the `exit()` method in the `System` class. Here you'll take a look at some of the things you need to weigh when deciding how to handle exceptions.

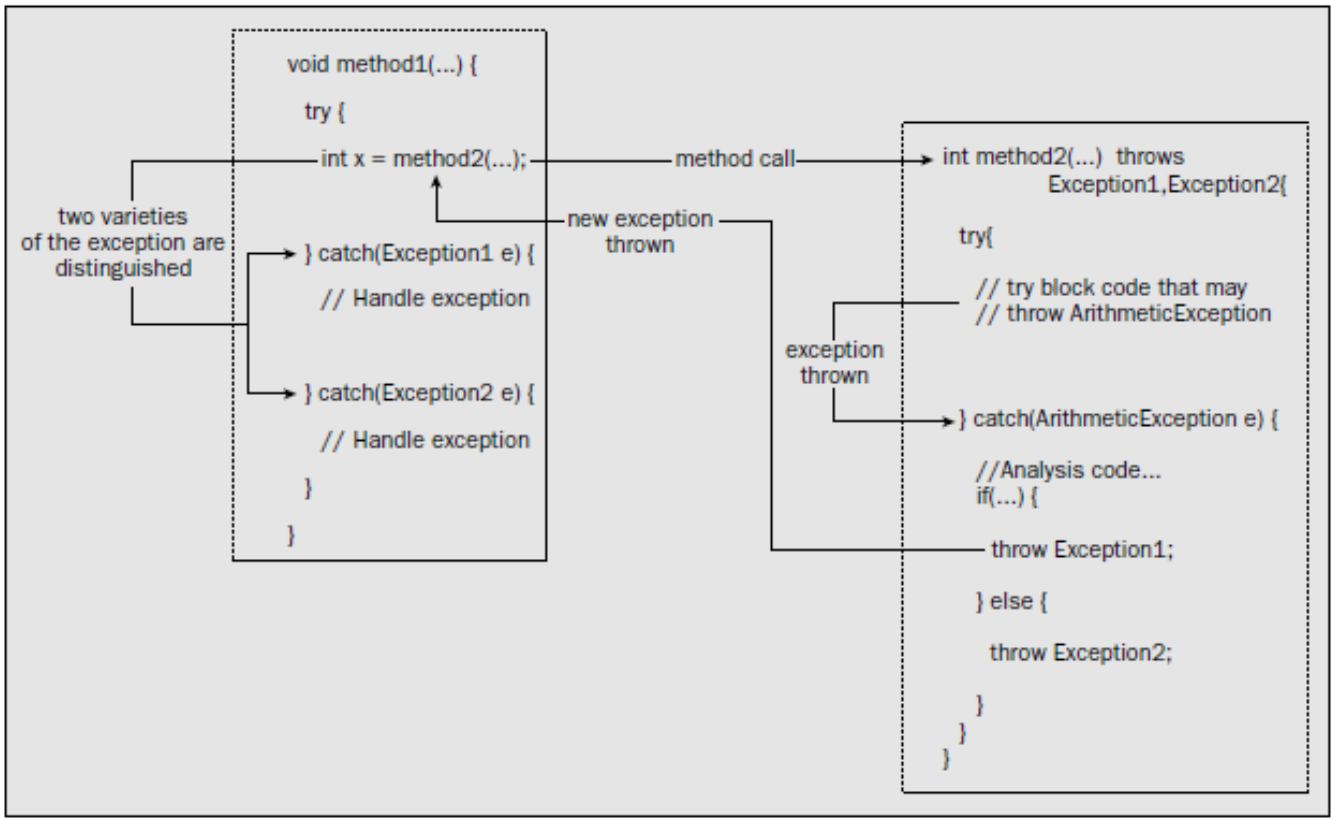
Consider the last example where you handled arithmetic and index-out-of-bounds exceptions in the `divide()` method. While this was a reasonable demonstration of the way the various blocks worked, it wasn't a satisfactory way of dealing with the exceptions in the program for at least two reasons.

- First, it does not make sense to catch the arithmetic exceptions in the `divide()` method without passing them on to the calling method. After all, it was the calling method that set the data up, and only the calling program has the potential to recover the situation.
- Second, by handling the exceptions completely in the `divide()` method, you allow the calling program to continue execution without any knowledge of the problem that arose. In a real situation this would undoubtedly create chaos, as further calculations would proceed with erroneous data.

You could have simply ignored the exceptions in the `divide()` method. This might not be a bad approach in this particular situation, but the first problem the calling program would have is determining the source of the exception. After all, such exceptions might also arise in the calling program itself. A second consideration could arise if the `divide()` method were more complicated. There could be several places where such exceptions might be thrown, and the calling method would have a hard time distinguishing them.

An Example of an Exception Class

- Another possibility is to catch the exceptions in the method where they originate and then pass them on to the calling program.
- You can pass them on by throwing new exceptions that provide more granularity in identifying the problem (by having more than one exception type or by providing additional data within the new exception type).



Defining Your Own Exception Class

```
public class ZeroDivideException extends Exception {
        private int index = -1; // Index of array element causing error
        // Default Constructor
        public ZeroDivideException(){ }
        // Standard constructor
        public ZeroDivideException(String s) {
                super(s); // Call the base constructor
        }
        public ZeroDivideException(int index) {
                super("/ by zero"); // Call the base constructor
                this.index = index; // Set the index value
        }
        // Get the array index value for the error
        public int getIndex() {
                return index; // Return the index value
        }
}
```

Using the Exception Class

You need to use the exception class in two contexts—in the `divide()` method when you catch a standard `ArithmeticException` and in the calling method `main()` to catch the new exception. Let's modify `divide()` first:

```
public static int divide(int[] array, int index) throws ZeroDivideException {
  try {
    System.out.println("First try block in divide() entered");
    array[index + 2] = array[index]/array[index + 1];
    System.out.println("Code at end of first try block in divide()");
    return array[index + 2];

  } catch(ArithmeticException e) {
    System.out.println("Arithmetic exception caught in divide()");
    throw new ZeroDivideException(index + 1);    // Throw new exception
  } catch(ArrayIndexOutOfBoundsException e) {
    System.out.println(
                "Index-out-of-bounds index exception caught in divide()");
  }
  System.out.println("Executing code after try block in divide()");
  return array[index + 2];
}
```

The first change is to add the `throws` clause to the method definition. Without this you'll get an error message from the compiler. The second change adds a statement to the `catch` block for `ArithmeticException` exceptions that throws a new exception.

This new exception needs to be caught in the calling method `main()`:

```
public static void main(String[] args) {
  int[] x = {10, 5, 0};                  // Array of three integers

  // This block only throws an exception if method divide() does
  try {
    System.out.println("First try block in main()entered");
    System.out.println("result = " + divide(x,0));  // No error
    x[1] = 0;                            // Will cause a divide by zero
    System.out.println("result = " + divide(x,0));  // Arithmetic error
    x[1] = 1;                            // Reset to prevent divide by zero
    System.out.println("result = " + divide(x,1));  // Index error
  } catch(ZeroDivideException e) {
    int index = e.getIndex();            // Get the index for the error
    if(index > 0) {                      // Verify it is valid and now fix the array
      x[index] = 1;                      // ...set the divisor to 1...
      x[index + 1] = x[index - 1];       // ...and set the result
      System.out.println("Zero divisor corrected to " + x[index]);
    }
  } catch(ArithmeticException e) {
    System.out.println("Arithmetic exception caught in main()");
  } catch(ArrayIndexOutOfBoundsException e) {
    System.out.println("Index-out-of-bounds exception caught in main()");
  }
  System.out.println("Outside first try block in main()");
}
```

SUMMARY

- Exceptions identify errors that arise in your program.
- Exceptions are objects of subclasses of the `Throwable` class.
- Java includes a set of standard exceptions that may be thrown automatically, as a result of errors in your code, or may be thrown by methods in the standard classes in Java.
- If a method throws exceptions that aren't caught, and aren't represented by subclasses of the `Error` class or by subclasses of the `RuntimeException` class, then you must identify the exception classes in a `throws` clause in the method definition.
- If you want to handle an exception in a method, you must place the code that may generate the exception in a `try` block. A method may have several `try` blocks.
- Exception handling code is placed in a `catch` block that immediately follows the `try` block that contains the code that can throw the exception. A `try` block can have multiple `catch` blocks that each deals with a different type of exception.
- A `finally` block is used to contain code that must be executed after the execution of a `try` block, regardless of how the `try` block execution ends. A `finally` block will always be executed before execution of the method ends.
- You can throw an exception by using a `throw` statement. You can throw an exception anywhere in a method. You can also rethrow an existing exception in a `catch` block to pass it to the calling method.
- You can define your own exception classes that, in general, should be derived from the class `Exception`.