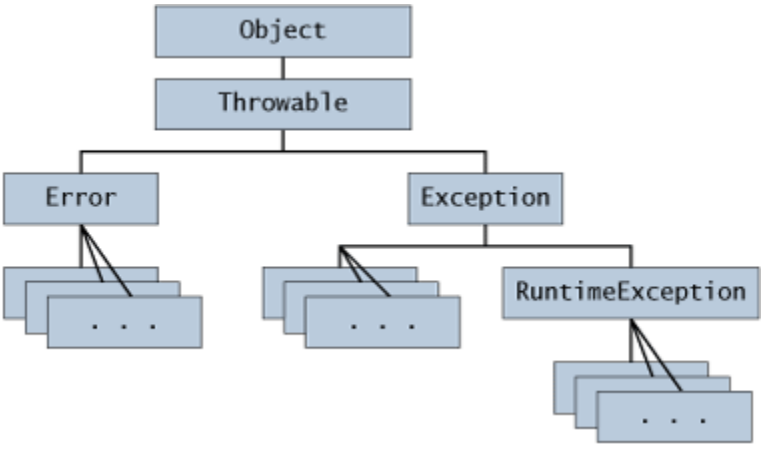
Exception Handling



1 The term "exception" means "exceptional condition" and is an occurrence that alters the normal program flow. A bunch of things can lead to exceptions, including hardware failures, resource exhaustion, and good old bugs. When an exceptional event occurs in Java, an exception is said to be "thrown." The code that's responsible for doing something about the exception is called an "exception handler," and it "catches" the thrown exception.

2 Therefore, we need a way to tell the JVM what code to execute when a certain exception happens. To do this, we use the try and catch keywords. The try is used to define a block of code in which exceptions may occur. This block of code is called a guarded region (which really means "risky code goes here"). One or more catch clauses match a specific exception (or group of exceptions—more on that later) to a block of code that handles it. Here's how it looks in pseudocode:

```
1. try {
2. // This is the first line of the "guarded region"
3. // that is governed by the try keyword.
4. // Put code here that might cause some kind of exception.
5. // We may have many code lines here or just one.
6. }
7. catch(MyFirstException) {
8. // Put code here that handles this exception.
9. // This is the next line of the exception handler.
10. // This is the last line of the exception handler.
11. }
12. catch(MySecondException) {
13. // Put code here that handles this exception
14. }
15.
16. // Some other unguarded (normal, non-risky) code begins here
```

1 Notice that the catch blocks immediately follow the try block. This is a requirement; if you have one or more catch blocks, they must immediately follow the try block. Additionally, the catch blocks must all follow each other, without any other statements or blocks in between. Also, the order in which the catch blocks appear matters,

2 Execution of the guarded region starts at line 2. If the program executes all the way past line 5 with no exceptions being thrown, execution will transfer to line 15 and continue downward.

However, if at any time in lines 2 through 5 (the try block) an exception is thrown of type MyFirstException, execution will immediately transfer to line 7. Lines 8 through 10 will then be executed so that the entire catch block runs, and then execution will transfer to line 15 and continue.

3 Note that if an exception occurred on, say, line 3 of the try block, the rest of the lines in the try block (4 and 5) would never be executed. Once control jumps to the catch block, it never returns to complete the balance of the try block.

4 A finally block encloses code that is always executed at some point after the try block, whether an exception was thrown or not.

5 If there was an exception thrown, the finally block executes immediately after the proper catch block completes. Let's look at another pseudocode example:

```
1: try {
2: // This is the first line of the "guarded region".
3: }
4: catch(MyFirstException) {
5: // Put code here that handles this exception
6: }
7: catch(MySecondException) {
8: // Put code here that handles this exception
9: }
10: finally {
11: // Put code here to release any resource we
12: // allocated in the try clause.
13: }
14:
15: // More code here
```

1 If there are no exceptions thrown in the try block, execution transfers to line 11, the first line of the finally block. On the other hand, if a MySecondException is thrown while the code in the try block is executing, execution transfers to the first line of that exception handler, line 8 in the catch clause. After all the code in the catch clause is executed, the program moves to line 11, the first line of the finally clause. Repeat after me: finally always runs!

2 Using a finally block allows the cleanup code to execute even when there isn't a catch clause. The following legal code demonstrates a try with a finally but no catch:

```
try {
        // do stuff
    } finally {
        //clean up
                }
```

The following legal code demonstrates a try, catch, and finally:

```
try{
        //do stuff
}catch(SomeException e)
        {
        //Do exception handling
        }finally
                {
                //clean up
                }
```

exception	Description
ArrayIndexOutOfBoundsException	Thrown when attempting to access an array with an invalid index value (either negative or beyond the length of the array).
ClassCastException	Thrown when attempting to cast a reference variable to a type that fails the IS-A test.
IllegalArgumentException	Thrown when a method receives an argument formatted differently than the method expects.

Advantages of Exceptions

Now that you know what exceptions are and how to use them, it's time to learn the advantages of using exceptions in your programs.

Advantage 1: Separating Error-Handling Code from "Regular" Code

Exceptions provide the means to separate the details of what to do when something out of the ordinary happens from the main logic of a program. In traditional programming, error detection, reporting, and handling often lead to confusing spaghetti code. For example, consider the pseudocode method here that reads an entire file into memory.

```
readFile {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
}
```

At first glance, this function seems simple enough, but it ignores all the following potential errors.

1. What happens if the file can't be opened?
2. What happens if the length of the file can't be determined?
3. What happens if enough memory can't be allocated?
4. What happens if the read fails?

What happens if the file can't be closed?

To handle such cases, the `readFile` function must have more code to do error detection, reporting, and handling. Here is an example of what the function might look like.

```
errorCodeType readFile {
    initialize errorCode = 0;

    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
```

```
                    if (readFailed) {
                        errorCode = -1;
                    }
                } else {
                    errorCode = -2;
                }
            } else {
                errorCode = -3;
            }
            close the file;
            if (theFileDidntClose && errorCode == 0) {
                errorCode = -4;
            } else {
                errorCode = errorCode and -4;
            }
        } else {
            errorCode = -5;
        }
        return errorCode;
    }
```

There's so much error detection, reporting, and returning here that the original seven lines of code are lost in the clutter. Worse yet, the logical flow of the code has also been lost, thus making it difficult to tell whether the code is doing the right thing: Is the file really being closed if the function fails to allocate enough memory? It's even more difficult to ensure that the code continues to do the right thing when you modify the method three months after writing it. Many programmers solve this problem by simply ignoring it — errors are reported when their programs crash.

Exceptions enable you to write the main flow of your code and to deal with the exceptional cases elsewhere. If the `readFile` function used exceptions instead of traditional error-management techniques, it would look more like the following.

```
readFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
        doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```

Note that exceptions don't spare you the effort of doing the work of detecting, reporting, and handling errors, but they do help you organize the work more effectively.

Advantage 2: Propagating Errors Up the Call Stack

A second advantage of exceptions is the ability to propagate error reporting up the call stack of methods. Suppose that the `readFile` method is the fourth method in a series of nested method calls made by the main program: `method1` calls `method2`, which calls `method3`, which finally calls `readFile`.

```
method1 {
    call method2;
}

method2 {
    call method3;
}

method3 {
```

```
        call readFile;
    }
```

Suppose also that `method1` is the only method interested in the errors that might occur within `readFile`. Traditional error-notification techniques force `method2` and `method3` to propagate the error codes returned by `readFile` up the call stack until the error codes finally reach `method1`—the only method that is interested in them.

```
method1 {
    errorCodeType error;
    error = call method2;
    if (error)
        doErrorProcessing;
    else
        proceed;
}

errorCodeType method2 {
    errorCodeType error;
    error = call method3;
    if (error)
        return error;
    else
        proceed;
}

errorCodeType method3 {
    errorCodeType error;
    error = call readFile;
    if (error)
        return error;
    else
        proceed;
}
```

Recall that the Java runtime environment searches backward through the call stack to find any methods that are interested in handling a particular exception. A method can duck any exceptions thrown within it, thereby allowing a method farther up the call stack to catch it. Hence, only the methods that care about errors have to worry about detecting errors.

```
method1 {
    try {
        call method2;
    } catch (exception e) {
        doErrorProcessing;
    }
}

method2 throws exception {
    call method3;
}

method3 throws exception {
    call readFile;
}
```

However, as the pseudocode shows, ducking an exception requires some effort on the part of the middleman methods. Any checked exceptions that can be thrown within a method must be specified in its `throws` clause.

Advantage 3: Grouping and Differentiating Error Types

Because all exceptions thrown within a program are objects, the grouping or categorizing of exceptions is a natural outcome of the class hierarchy. An example of a group of related exception classes in the Java platform are those defined in `java.io` — `IOException` and its descendants. `IOException` is the most general and represents any type of error that can occur when performing I/O. Its descendants represent more specific errors. For example, `FileNotFoundException` means that a file could not be located on disk.

A method can write specific handlers that can handle a very specific exception. The `FileNotFoundException` class has no descendants, so the following handler can handle only one type of exception.

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

A method can catch an exception based on its group or general type by specifying any of the exception's superclasses in the `catch` statement. For example, to catch all I/O exceptions, regardless of their specific type, an exception handler specifies an `IOException` argument.

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

This handler will be able to catch all I/O exceptions, including `FileNotFoundException`, `EOFException`, and so on. You can find details about what occurred by querying the argument passed to the exception handler. For example, use the following to print the stack trace.

```
catch (IOException e) {
    e.printStackTrace();  //Output goes to System.err.
    e.printStackTrace(System.out);  //Send trace to stdout.
}
```

You could even set up an exception handler that handles any `Exception` with the handler here.

```
catch (Exception e) {    //A (too) general exception handler
    ...
}
```

The `Exception` class is close to the top of the `Throwable` class hierarchy. Therefore, this handler will catch many other exceptions in addition to those that the handler is intended to catch. You may want to handle exceptions this way if all you want your program to do, for example, is print out an error message for the user and then exit.

In most situations, however, you want exception handlers to be as specific as possible. The reason is that the first thing a handler must do is determine what type of exception occurred before it can decide on the best recovery strategy. In effect, by not catching specific errors, the handler must accommodate any possibility. Exception handlers that are too general can make code more error-prone by catching and handling exceptions that weren't anticipated by the programmer and for which the handler was not intended.

As noted, you can create groups of exceptions and handle exceptions in a general fashion, or you can use the specific exception type to differentiate exceptions and handle exceptions in an exact fashion.