# CS 211 Section: Exceptions

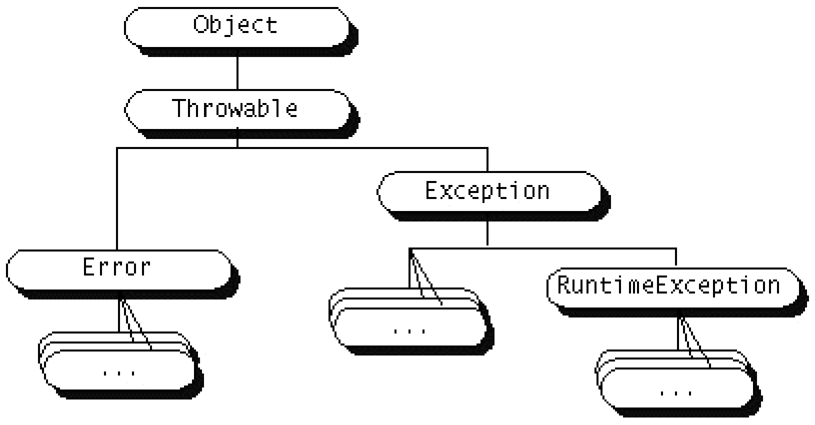
*(by Nicholas Ruozzi with pictures and examples from the Java tutorial)*

Many different types of errors can occur during the construction of a program
1) Compile time errors caused by syntactically or semantically invalid programs
2) Runtime errors such as stack overflows, null pointers exceptions, array index out of bounds, etc.
3) Logically incorrect programs that do not solve the correct problem

Compile time errors are already caught by the compiler, and errors of logic can only be corrected by testing the methods that have been by comparing known results with the actual output of the method. Runtime errors are seemingly not as trivial. The focus of this discussion will be the errors that occur at runtime, which are called exceptions.

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

Java provides an exception handling mechanism that allows programmers to propagate exceptions, "catch" them, and differentiate between different types of exceptions. Java provides two main classes that are used in the handling of exceptions; class Throwable and class Exception. The following diagram (from the Java tutorial):



Notice, class Error also extends class throwable, but this class **should not be used** in the flow of a normal Java program. These types of errors are for what are known as hard errors, i.e. classes of errors that require the termination of the Java virtual machine, etc.

## When to use exception handling:

There are two different types of exceptions in Java: those that require immediate handling and those that do not. I'm sure that most of you have noticed that certain exceptions don't actually cause your program to stop executing, even if the exception is serious (for instance, an ArrayIndexOutOfBoundsException). Why doesn't Java require us to catch these exceptions every time they may potentially occur? The answer is that this would require too many error handling statements, even in simple programs. For

example, you would need to be sure to handle exceptions every time you indexed into an array! These types of exceptions that require no explicit handling extend class RuntimeException (pictured above). Often times we want to alert other methods when a serious exception does in fact occur. So how do we know when an error is serious enough? In Java, the solution is to allow programmers to specify when they believe that an exception will seriously prevent the functioning of any further statements in a particular method (these exceptions do not typically extend RuntimeException).
As a brief example, opening a file (Note that this does ***not*** work):

```
public InputFile(String filename) {
    FileReader  in = new FileReader(filename);
    . . .
}
```

What would happen if the constructor of FileReader was unable to open the correct file? How would we know? Exceptions that’s how. Consider the following excerpt from the Java API:

```
public FileReader(File file) throws FileNotFoundException
```

Throws:

> `FileNotFoundException` - if the file does not exist, is a directory rather than a regular file, or for some other reason cannot be opened for reading.

Notice that the keyword “throws” follows the specification of FileReader’s constructor. This is Java’s technique of alerting programmers that something potentially bad could happen inside of the code for the method and if it does, the code that invokes this method needs to be informed of the problem and take the appropriate action. This technique of forcing a higher method to “listen” for potential errors is called specifying an exception. Any method that invokes another method that has specified an exception is required at compile time to “catch” the exception.

## Throwing Exceptions:

Throwing exceptions is a way to alert other parts of a program that something terrible just happened. Throwing exceptions is easy in Java, simply invoke the **throw** method by passing it the type of exception to throw:

```
throw(NullPointerException);
```

This will throw a null pointer exception (try it).

## Handling Exceptions:

The try-catch-finally block in Java is used to catch, handle, and then cleanup after an exception has occurred. This is required at compile time (the finally is optional) if a method is not declared to throw an exception (discussed below). It possesses the following form:

```
try
{
      // Code that contains at least one specified exception or
      // a throw statement
}
catch(Exception1 e1)
{
      // Code to respond to a particular exception
}
catch(Exception2 e2)
{
      // Code to respond to a different exception
}
.
.
.
finally
{
      // Code to cleanup after the exception has occurred
}
```

Note, a method is not required to handle exceptions that could occur inside of itself. It may be declared with a "throws" following its specification. This will pass the error handling up to a higher method that may actually care about the occurrence of errors. To correct our previous method we could do either of the following:

```
public InputFile(String filename) throws FileNotFoundException{
    FileReader in = new FileReader(filename);
    . . .
}
```

If we don't want to handle the exception explicitly ourselves, or:

```
public InputFile(String filename) {
    try
    {
        FileReader in = new FileReader(filename);
        . . .
    }
    catch(FileNotFoundException e)
    {
        . . .
    }
}
```

If we want to handle the exception ourselves, a try-catch block must be used. Note that eventually some method will be forced to catch the exception thrown by FileReader whether it is done by our code or not. As a suggestion though, only ignore exceptions that are not relevant to your particular method and don't needlessly propagate exceptions.

## What NOT to do with exceptions:

This was just a brief introduction to exceptions and how they are used in Java. I'd like to close with a list of ways that exceptions should never, under no circumstances, be used:

1) Do not throw a runtime exception or extend RuntimeException simply because you don't want to be bothered with specifying the exceptions your methods can throw
2) Do not use exceptions as an excuse to produce unreadable, incomprehensible code:

   ```
   int numArray[];
   .
   .
   .
   int total = 0;
   int i = 0;
   try
   {
         while(true)
         {
               total += numArray[i];
               i++;
         }
   }
   catch(Exception e)
   {
         // just ignore the exception
   }
   ```

   Use the correct exception handling free code which is much more readable:

   ```
   int numArray[];
   .
   .
   .
   int total = 0;
   for(int i = 0; i < numArray.length; i++)
         total += numArray[i];
   ```

3) Do not needlessly throw exceptions when they can be handled by better code construction

It is also possible to create your own exceptions that extend class Exception. This is quite useful when your code has its own special exceptions. For an example of where this might be useful take CS212.