Exceptions

Source	http://www.cs.wisc.edu/~cs302/io/Exceptions.html

Introduction

Exceptions are ***objects*** that store information about the occurrence of an unusual or error condition. They are *thrown* when that error or unusual condition occurs. You have likely experienced the occurrence of an exception occasionally throughout your programming in this course. Examples include: **`NullPointerException`** and **`ArithmeticException`**.

Until now, any exception that occurred ***always*** resulted in a program crash. However, Java and other programming languages have mechanisms for handling exceptions that you can use to keep your program from crashing. In Java, this is known as *catching an exception*.

If any exception occurs and is not caught, the program will crash.

There are two types of exceptions in Java, *unchecked* exceptions and *checked* exceptions.

Unchecked Exceptions

Unchecked exceptions are any class of exception that extends the **`RuntimeException`** class at some point in its inheritance hierarchy. This includes the most common exceptions. An **`ArithmeticException`** occurs when a program tries to divide by zero. A **`NullPointerException`** occurs when you try and access an instance data member or method of a reference variable that does not yet reference an object.

Unchecked exceptions can occur anywhere in a program and in a typical program can be very numerous. Therefore, the cost of checking these exceptions can be greater than the benefit of handling them. Thus, Java compilers do not require that you declare or catch unchecked exceptions in your program code. Unchecked exceptions may be handled as explained for *checked* exceptions in the following section.

All Exceptions that extend the RuntimeException class are unchecked exceptions.

Checked Exceptions

Checked exceptions are exceptions that do ***not*** extend the **`RuntimeException`** class. Checked exceptions ***must*** be *handled* by the programmer to avoid a compile-time error. One example of a checked exception is the **`IOException`** that may occur when the ***`readLine`*** method is called on a **`BufferedReader`** object. Read more about the ***`readLine`*** method in the section on console input in the Java I/O page. All other exceptions you may have experienced were examples of *unchecked* exceptions.

Handling Checked Exceptions

There are two ways to *handle* checked exceptions. You may declare the exception using a *throws clause* or you may *catch the exception*. To declare an exception, you add the keyword ***throws*** followed by the class name of the exception to the method header. See example below.

Any method that calls a method that *throws* a checked exception must also *handle* the checked exception in one of these two ways.

In the example below, either call to the ***readLine*** method may cause an **IOException** to occur. Each method of handling the checked exception is shown.

The throws Clause

The **throws IOException** clause in the method header tells the compiler that we know this exception may occur and if it does, the exception should be thrown to the caller of this method instead of crashing the program. The *throws clause* is placed after the parameter list and before the opening brace of the method. If more than one type of checked exception needs to be declared, separate the class names with commas.

```
/**
 *  Fills a Data object with Strings of data that
 *  are read from a BufferedReader object.
 */
void fillData ( BufferedReader in, Data data )
    throws IOException
{
    String newData = in.readLine();
    while ( newData != null )
    {
        data.addData( newData );
        newData = in.readLine();
    }
}
```

If an exception is always handled using the throws clause, then eventually the exception will propagate all the way back to the ***main*** method of the application. If the ***main*** method *throws the exception* and the exception actually occurs while the program is running, then the program will crash.

The try-catch Statement

An alternative to simply passing the exception back to the caller of your method, is to *catch* the exception yourself. In this case, your method will do something to correct the error or recover and continue the program's execution, if the exception occurs. To *catch* an exception, you must first indicate what code may cause the exception. This is called a *try* block. Together, they are referred to as a **try-catch** statement.

The ***try*** block surrounds the statement and any related statements that may cause the exception to occur. The ***catch*** block follows the try block and defines the code that should be executed, if the exception occurs. If more than one type of checked exception needs to be caught, write a separate catch clause for each type. See How to Catch Multiple Types of Exceptions.

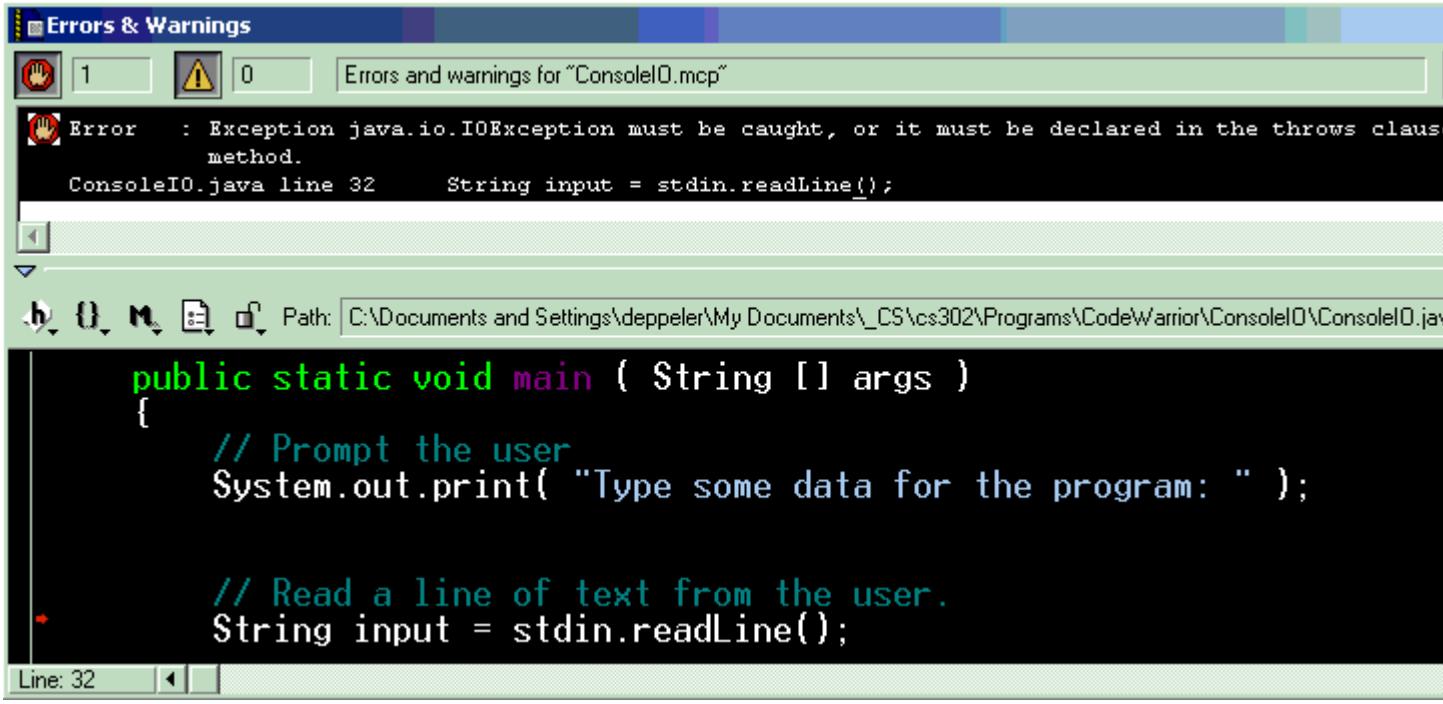
```
/**
 *  Fills a Data object with Strings of data that
 *  are read from a BufferedReader object.
 */
void fillData ( BufferedReader in, Data data )
{
```

```
        try
        {
            String newData = in.readLine();
            while ( newData.equals("") )
            {
                data.addData( newData );
                newData = in.readLine();
            }
        }
        catch ( IOException e )
        {
          System.out.println(e);
          System.out.println("Unable to finish adding data.");
        }
    }
```

All Exceptions that DO NOT extend the `RuntimeException` class are checked exceptions. The compiler requires a throws clause or a try-catch statement for any call to a method that may cause a checked exception to occur. ***If a checked exception occurs and is not caught before it reaches the `main` method of the application, the program will crash.***

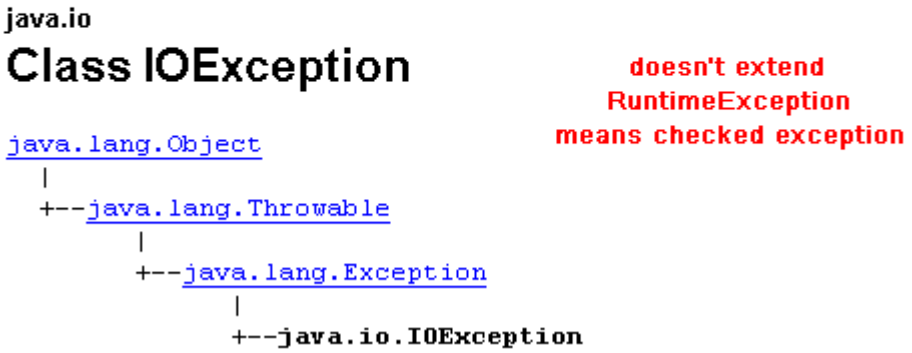
How do I know if it is a checked exception or an unchecked exception?

Experienced programmers seem to just know which exceptions are checked. But, there are two ways that novice programmers can gain this experience. One way that we learn which exceptions are *checked*, is from the compiler. The compiler will produce an error when we call a method that may throw *checked* exceptions. For example, our Console IO program from the Java I/O page will produce the following compile-time error message, if we forget the ***throws IOException*** clause in the method header.

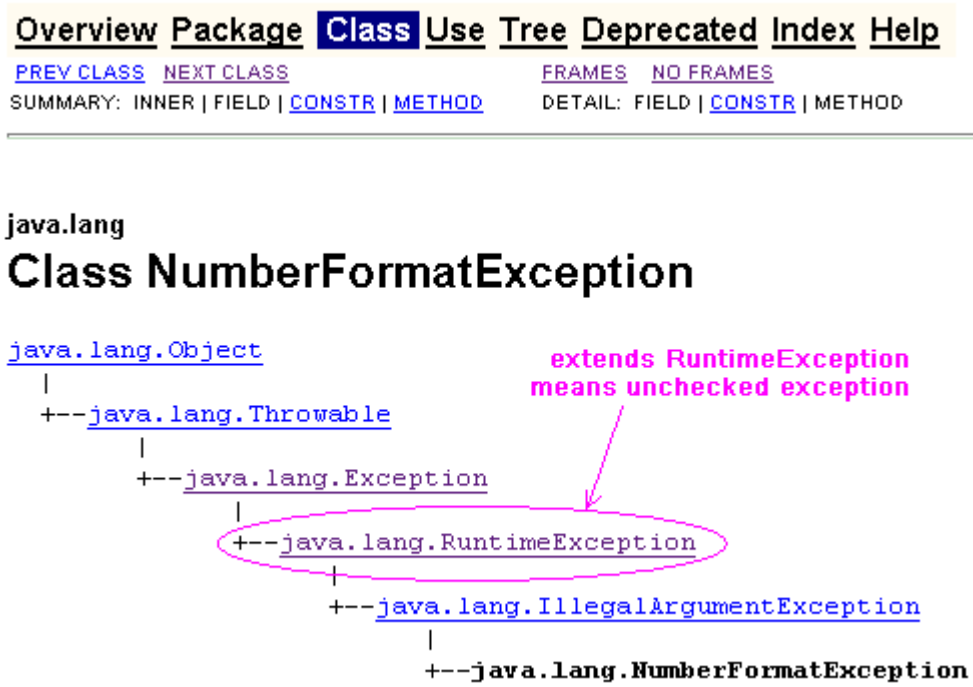


Thus, an **`IOException`** is a checked exception and we now know that the ***`readLine`*** method of the **`BufferedReader`** class may throw such exceptions. From now on, we will always *catch* or include the ***throws IOException*** clause in the method header of any method that calls the ***`readLine`*** method. This also means that any method that calls any of our methods that ***throws IOException*** will have to handle the exception.

A second way we can learn which exceptions are checked exceptions is to view the inheritance hierarchy for the exception class of interest. Use your browser to view the javadoc pages for the **`BufferedReader`** class and navigate to the method detail documentation for the ***`readLine`*** method. The method summary information doesn't show, but the method detail information shows that the ***`readLine`*** method ***throws IOException***. This information tells you what exceptions may be thrown by the ***`readLine`*** method. But, is the **`IOException`** a checked exception? Click on the link to **`IOException`** to find out for sure. You will see the following, which shows that the **`IOException`** class extends the **`Exception`** class, but it does not extend the **`RuntimeException`** class. Since, checked exceptions are those types of exceptions that do not extend the **`RuntimeException`** class, we know this is a checked exception.



Compare and contrast the inheritance hierarchy of the **`IOException`** class (shown above) with that of the **`NumberFormatException`** class (shown below). Notice, that the **`NumberFormatException`** class does have the **`RuntimeException`** class in its inheritance hierarchy and is thus an unchecked exception.



Viewing the javadoc pages will help you identify if a method may cause a *checked* exception. But, there are many *unchecked* exceptions that may occur that are not documented in any method header. They may not be easy to find. Some of the most common unchecked exceptions are: **`ArithmeticException: Divide By Zero`** and **`NullPointerException`**. It is up to the programmer to understand when these type of exceptions can occur and to employ defensive programming to ensure that they do not occur.

When you catch an exception, you [the programmer] have the ability to perform some other action including *fixing* the error condition before continuing with the program's execution. This opportunity to resolve the problem means that the program can continue to perform the operations for which it was intended. However, there are some exceptions that you will not be able to resolve and catching those exceptions will give you the opportunity to save some current state of execution and report the error to the user before exiting the program gracefully. You can always choose not to catch an exception. If that exception occurs, the program will crash.

How to catch a NumberFormatException

To **catch** a **`NumberFormatException`**, put a ***try*** statement block around the code that may cause the exception and a ***catch*** clause after the try block that describes what to do if the exception occurs. Here's a code fragment that shows how to catch a **`NumberFormatException`** that may occur when the ***`parseInt`*** method is called on a **`String`**:

```
System.out.print( "Enter an integer: " );
String input = stdin.readLine();

try
{
    int x = Integer.parseInt( input );
}
```

```
catch ( NumberFormatException e )
{
    System.out.println( e.getMessage()
        + " is not a valid format for an integer." );
}
```

In this example, the user is prompted for an integer. The program then *tries* to parse that string into an `int` value. If the string of characters has a decimal point or letters or some other invalid `int` format, then a **`NumberFormatException`** is thrown. If that happens, the flow of control transfers to the `catch ( NumberFormatException e ) {...}` block of code. In this example, a message is displayed to the user and the program then continues after the catch block without crashing.

The code fragment above will keep your program from crashing if the user enters an invalid `int` format, but it does not give the user another chance to enter a valid integer format. A repetition statement is necessary, if you wish to repeat the prompt and let the user have another chance to enter a valid integer.

A **while loop** that repeats until a valid integer is entered is a good choice. The entire ***try-catch*** statement must be nested inside the loop. The preferred way to control this type of loop is with a `boolean` variable that is set true when a valid integer has been entered. Using a specific integer as a sentinel value for ending the while loop is only okay if there is an integer that can be used that is not otherwise a valid choice for the user.

```
// Code fragment to prompt a user for an integer
// and store that integer in a variable x that can be
// used after the loop has gotten a valid integer from the user.

// used to mark when a valid integer has been entered
boolean valid = false;

// x will hold the integer entered by the user
// It can be initialized to anything, since it will be overwritten
int x = -1;

// Get a valid integer from the user and put into x
while ( ! valid )
{
    try
    {
        System.out.print( "Enter an integer: " );

        x = Integer.parseInt( stdin.readLine() );

        valid = true;   // this statement only executes if the string
                        // entered does not cause a NumberFormatException
    }
    catch ( NumberFormatException e )
    {
        // A NumberFormatException occurred, print an error message
        System.out.println( e.getMessage()
            + " is not a valid format for an integer." );
    }

} // repeats until a valid integer has been entered
```

Notes: The declaration of `valid` must precede its use inside the loop condition expression. The declaration of `x` must also be outside of the loop. This is so that the value of `x` can be used after the loop has finished executing. This is known as the *scope* of the variable. If a variable is declared inside of a statement block, then it will only have *scope (be visible)* to code that is also inside of the block and after the declaration. The statement to set the `boolean` variable named `valid` to `true` must also be inside of the `try` block. If this statement were placed after the try-catch statement, then it would be executed even if a **`NumberFormatException`** occurred.

How to Catch Multiple Types of Exceptions

You can catch more than one type of exception after the same try block. However, the order that the exceptions are listed in the catch clauses is sometimes important. More specific classes of exceptions must be listed before more general classes that could catch the same exception. Here is a code fragment for opening a file for reading.

```
try
{
    // Prompt the user and read a file name
    System.out.print( "Type a file name: " );
    String fileName = stdin.readLine();

     // Create a FileReader object for the specified file
    FileReader fileReader = new FileReader( fileName );

    // Create a BufferedReader object
    BufferedReader inFile = new BufferedReader( fileReader );

    // Intialize line number and read the first line of the file
    int lineNo = 0;
    String line = inFile.readLine();

    // Read entire file line-by-line and echo to screen with line numbers
    while ( line != null )
    {
        System.out.println( ++lineNo + "\t" + line );
        line = inFile.readLine();
    }

    // Close the file reader for safety
    fileReader.close();
}
catch ( FileNotFoundException e )
{
    System.out.println( e.getMessage() + " FILE NOT FOUND" );
}
catch ( IOException e )
{
    System.out.println( e + " IO EXCEPTION" );
}
catch ( Exception e )
{
    System.out.println( e + " EXCEPTION" );
}
```

The class **`Exception`** is the most general type of exception. The **`FileNotFoundException`** class is a more specific type of exception and will not catch ***all*** exceptions that could occur. In the example above, exceptions that are not caught by the **`FileNotFoundException`** catch clause will fall through and possibly be caught by the catch clause for the **`IOException`** class. It they are not caught by the **`IOException`** class, they will be caught by the catch clause for the **`Exception`** class.

It is a compile-time error to place a more general exception clause (for a *super*class) before a more specific exception clause (for a *sub*class). Use the inheritance tree provided by the javadoc to determine which exception class is more specific. The classes that are lower on the inheritance hierarchy are *sub*classes and are more specific than those classes listed higher in the inheritance hierarchy. See the NumberFormatException hierarchy figure and notice that **`NumberFormatException`** is a subclass of **`IllegalArgumentException`**, which is a subclass of **`RuntimeException`** and so on.

Finally

After the ***try-catch*** statement, you may optionally include a *finally* clause. A *finally* clause is used to perform operations that must happen whether or not an exception occurs in the *try* block. If a *finally* clause is used, it must follow all *catch* clauses for a given *try* block. There can only be one *finally* clause for a given *try-catch* statement block.

```
try
{
    ...
}
catch ( IOException e )
{
    ...
}
catch ( Exception e )
{
    ...
}
finally
{
    // any statements that you always want to occur
}
```

Conclusion

Exceptions are a good way to separate the code that handles errors or very special cases from the code that is most commonly executed. This can help the programmer focus on the intended purpose of the code without having to write many nested control statements to verify information at each step of execution. Read Chapter 8, "Exceptions and Assertions" of Wu's "An Introduction to Object-Oriented Programming with Java" textbook for more information about handling exceptions in Java.

~~~ End of Article ~~~