Exceptional Handling

Introduction

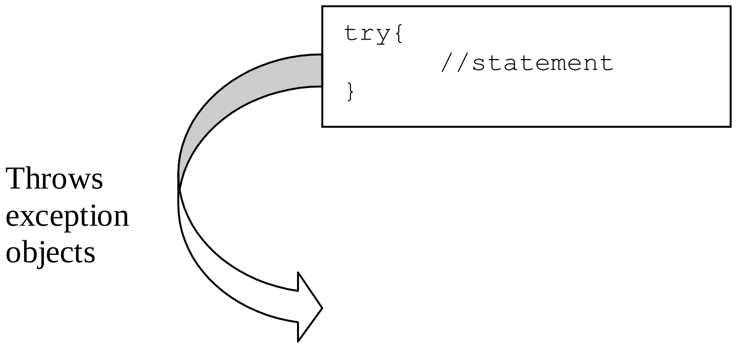
An *exception* is an event that occurs during the execution of a program that disrupts the normal flow of instructions. Many kinds of errors can cause exceptions—problems ranging from serious hardware errors, such as a hard disk crash, to simple programming errors, such as trying to access an out-of- bounds array element. When such an error occurs within a Java method, the method creates an exception object and hands it off to the runtime system. The exception object contains information about the exception, including its type and the state of the program when the error occurred. The runtime system is then responsible for finding some code to handle the error. In Java terminology, creating an exception object and handing it to the runtime system is called *throwing an exception*. The point at which the throw is executed is called the throw point.

Once an exception is thrown, the block in which the exception is thrown expires and control cannot return to the throw point. Thus Java uses the *termination model of exception handling* rather than the *resumption model of exception handling*. It is also not possible to return to the throw point by issuing a return statement ina catch handler. Following are some exceptions that can be generated by java statements.

- ArithmeticException
- NumberFormatException
- ArrayIndexOutOfBoundsException
- FileNotFoundException
- IOException

Syntax of Exceptional Handling

The basic concepts of exception handling are throwing an exception and catching it.



```
try{
        //statement
}
```

```
catch(ExceptionType e)
{
      //statement
}
```

Java uses a keyword **try** to preface a block of code that is likely to case an error condition and throws an exception. A catch block defined by the keyword **catch** catches the exception thrown by the try block. An exception cannot access objects defined within its try block because the try block has expired when the handler begin executing. It is possible to write the multiple catch blocks. But it is to be noted that the handler that catches a subclass object should be placed before a handler that catches a superclass object. If the superclass handler were first, it would catch superclass objects and the objects of subclasses of the superclass as well.

finally Block

The final step in setting up an exception handler is providing a mechanism for cleaning up the state of the method before (possibly) allowing control to be passed to a different part of the program. You do this by enclosing the cleanup code within a finally block. The runtime system always executes the statements within the finally block regardless of what happens within the try block. This block is optional and it is the preferred means for preventing resource leaks.

```
try{
      //statements
}
catch(ExceptionType e)
{
      //statements
}
finally{
      //statement
}
```

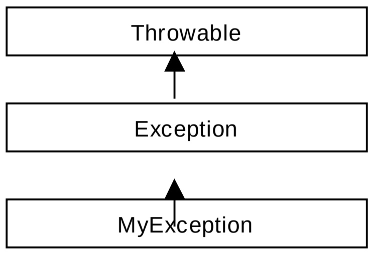
throws Clause

A throws clause lists all the exceptions that can be thrown by a method.

```
void function()throws ExceptionType1, ExceptionType2
{
      //statement
}
```

The types of exception that are thrown by a method are specified in the method definition with a throws clause. A method can throw objects of the indicated classes, or it can throw objects of subclass.

Creating our own Exception



The following program illustrates the creation of own exception.

```
// TestMyException.java
class MyException extends Exception{
  MyException(String msg){
     super(msg);
  }
}
//..............................................................................
class TestMyException{
    public static void main(String[] args){
       int x = 0;
       x = Integer.parseInt(args[0]);
       try{
         if(x<0){
             throw new MyException("Negative Number.");
         }
         else{
             //process with x variable.
         }
       }
       catch(MyException me){
          System.out.println(me.getMessage());
       }
    }
}
```