**Exceptions**

Traditionally, applications used the concept of a global error object. When a piece of code caused an error, it would set the data in this object to indicate the cause of the error and then return to the caller. It was the responsibility of the calling code to examine the error object and determine how to handle it. However, this approach is not robust, because it is too easy for a programmer to forget to handle errors appropriately.

The .NET Framework uses exceptions to help overcome these issues. An exception is an indication of an error or exceptional condition. A method can throw an exception when it detects that something unexpected has happened, for example, the application tries to open a file that does not exist.

**Propagation**

When a method throws an exception, the calling code must be prepared to detect and handle this exception. If the calling code does not detect the exception, the code is aborted and the exception is automatically propagated to the code that invoked the calling code. This process continues until a section of code takes responsibility for handling the exception. Execution continues in this section of code after the exception-handling logic has completed. If no code handles the exception, then the process will terminate and display a message to the user.

**Handling Exceptions**

The try/catch block is the key programming construct that enables you to implement structured exception handling in your applications. You wrap code that may fail and cause an exception in a try block, and add one or more catch blocks to handle any exceptions that may occur.  The recommended strategy to follow with catch blocks is to catch more specific exceptions first, and more general exceptions last.  For example, if you expect to run across a an exception around file access, you would catch the FileNotFoundException in the first catch block and then perhaps create a second catch block that would watch for the generic Exception class to catch any other exception besides the FileNotFoundException.

If the FileNotFoundException is triggered, that catch block will have code to handle that exception.  Otherwise, the other catch block(s) will contain code to handle other exceptions, or to throw the exception back up the stack to the calling application.

The following code example shows the syntax for defining a try/catch block.

try  
{  
    // Try block.  
}  
catch (FileNotFoundException fnfEx)  
{  
    // Catch block 1.  
}  
catch (Exception e)  
{  
    // Catch block n.  
}

**Using Finally**

Some methods may contain critical code that must always be run, even if an unhandled exception occurs. For example, a method may need to ensure that it closes a file that it was writing to or releases some other resources before it terminates. A finally block enables you to handle this situation.

You specify a finally block after any catch handlers in a try/catch block. It specifies code that must be performed when the block finishes, irrespective of whether any exceptions, handled or unhandled, occur. If an exception is caught and handled, the exception handler in the catch block will run before the finally block.

You can also add a finally block to code that has no catch blocks. In this case, all exceptions are unhandled, but the finally block will always run.

The following code example shows how to implement a try/catch/finally block.  
  
  
try  
{  
}  
catch (NullReferenceException ex)  
{  
    // Catch all NullReferenceException exceptions.  
}  
catch (Exception ex)  
{  
    // Catch all other exceptions.  
}  
finally  
{  
   // Code that always runs to close files or release resources.  
}

**Throwing Exceptions**

You can create an instance of an exception class in your code and throw the exception to indicate that an exception has occurred. When you throw an exception, execution of the current block of code terminates and the CLR passes control to the first available exception handler that catches the exception.

To throw an exception, you use the **throw** keyword and specify the exception object to throw.

The following code example shows how to create an instance of the **NullReferenceException** class and then throw the **ex** object.

var ex = new NullReferenceException("The 'Name' parameter is null.");  
throw ex;

A common strategy is for a method or block of code to catch any exceptions and attempt to handle them. If the catch block for an exception cannot resolve the error, it can rethrow the exception to propagate it to the caller.

The following code example shows how to rethrow an exception that has been caught in a catch block.  
  
try  
{  
}  
catch (NullReferenceException ex)  
{  
    // Catch all NullReferenceException exceptions.  
}  
catch (Exception ex)  
{  
    // Attempt to handle the exception  
    ...  
    // If this catch handler cannot resolve the exception,   
    // throw it to the calling code  
    throw;  
}