**C#\_6.1 System.Exception Class**

All exception classes must be derived from the built-in exception class Exception, which is part of the System namespace. i.e. All exceptions are subclasses of Exception.

* SystemException: One very important subclass of Exception is SystemException. This is the exception class from which all exceptions generated by the C# runtime system (that is, the CLR) are derived. SystemException does not add anything to Exception. It simply defines the top of the standard exceptions hierarchy.
* .NET Framework defines several built-in exceptions that are derived from SystemException. Eg: DivideByZeroException exception is generated for zero division.
* Keywords: C# exception handling is managed via four keywords: try, catch, throw, and finally. Program statements that you ***want to monitor*** for exceptions are contained within a try block. If an ***exception occurs within*** the try block, it is thrown. Your code can ***catch this exception*** using catch and handle it in some rational manner. System-generated exceptions are automatically thrown by the runtime system. To ***manually throw an exception***, use the keyword throw. Any code that ***absolutely must be*** ***executed*** ***upon exiting*** from a try block is put in a finally block.

**C#\_6.** **2 try and catch (Same as Java part 6.2, and before "try-with-resource".)**

Note: In C#, specifying exOb is optional. If the exception handler doesn't need access to the exception object (as is often the case), then there is no need to specify exOb.

|  |  |
| --- | --- |
| * C#\_Example 1: Demonstrate exception handling.   **using System**;  **class** ExcDemo1 { **static void Main()** { **int[]** nums = **new** **int**[4];  **try** { **Console.WriteLine**("Before exception is generated.");  nums[7] = 10; *// Generate an index out-of-bounds exception.*  **Console.WriteLine**("this won't be displayed"); }  **catch** (**IndexOutOfRangeException**) {  **Console.WriteLine**("Index out-of-bounds!"); */\* catch the exception \*/* }  **Console.WriteLine**("After catch block."); }} | Description is mainly same as Java part 6.2. Following noticed:   * Notice that no exception variable name is specified in the catch clause. Instead, only the type of the exception (IndexOutOfRangeException in this case) is required. As mentioned, an exception variable is needed only when access to the exception object is required. * In some cases, the value of the exception object can be used by the exception handler to obtain additional information about the error, but in many cases, it is sufficient to simply know that an exception occurred. Thus, it is not unusual for the catch exception variable to be absent (Differ from Java). |

* An exception can be generated by one method and caught by another: Similar as Java part 6.2.

|  |  |
| --- | --- |
| using System;  class ExcTest { public static void GenException() { int[] nums = new int[4];  Console.WriteLine("Before exception is generated.");  nums[7] = 10; Console.WriteLine("this won't be displayed"); } } | class ExcDemo2 { static void Main() {  try { ExcTest.GenException(); }  catch (IndexOutOfRangeException) { Console.WriteLine("Out-of-bounds!"); }  Console.WriteLine("After catch block."); }} |

* Since ***GenException()*** is called from within a try block, the exception that it generates (and does not catch) is caught by the catch in ***Main()***. If ***GenException()*** had caught the exception, it never would have been passed back to ***Main()***.

Note: (Similar to Java part 6.2 notes.)

* Catching one of C#’s standard exceptions prevents abnormal program termination: When an exception is thrown, it must be caught by some piece of code somewhere.
* TYPE specification inside CATCH: The type of the exception must match the type specified in a catch clause. If it doesn’t, the exception won’t be caught.
* Exception handling enables your program to respond to an error and then continue running/ avoiding runtime error: For example, consider the following example that divides the elements of one array by the elements of another. If a division by zero occurs, a DivideByZeroException is generated. In the program, this exception is handled by reporting the error and then continuing with execution (avoiding runtime error).

**using System**;

**class** ExcDemo3 { **static void Main()** { **int[]** numer = { 4, 8, 16, 32, 64, 128 }; **int[]** denom = { 2, 0, 4, 4, 0, 8 };

**for**(**int** i=0; i < numer.**Length**; i++) { **try** { **Console.WriteLine**(numer[i] + " / " + denom[i] + " is " + numer[i]/denom[i]); }

**catch** (**DivideByZeroException**) { **Console.WriteLine**("Can't divide by Zero!"); } } }}

* Once an exception has been handled, it is removed from the system. Therefore, in the program, each pass through the loop enters the try block anew—any prior exceptions have been handled. This enables your program to handle repeated errors.

**C#\_6.3 Try and catch advanced (Same as Java part 6.3)**

|  |  |
| --- | --- |
| * Multiple-Catch: You can associate *more than one* catch clause with a try. For example: | **try** { **Console.WriteLine**(numer[i] + " / " + denom[i] + " is " + numer[i]/denom[i]); }  **catch** (**DivideByZeroException**) { **Console.WriteLine**("Can't divide by Zero!"); }  **catch** (**IndexOutOfRangeException**) { **Console.WriteLine**("No matching element found."); } |

* In general, catch clauses are checked in the order in which they occur in a program. Only a matching clause is executed. All others are ignored.

|  |  |
| --- | --- |
| * Catching All Exceptions: To do this, use a catch clause that specifies no exception type at all. For Example: | **try** { **Console.WriteLine**(numer[i] + " / " + denom[i] + " is " + numer[i]/denom[i]); }  **catch** { **Console.WriteLine**("Some exception occurred."); } */\* This catches all exceptions.\*/* |

* Nested Try Blocks: Same as Java part 6.3. Only Example:

**try** { **for**(**int** i=0; i < numer.**Length**; i++) { **try** { **Console.WriteLine**(numer[i] + " / " + denom[i] + " is " + numer[i]/denom[i]); }

**catch** (**DivideByZeroException**) { **Console.WriteLine**("Can't divide by Zero!"); } } }

**catch** (**IndexOutOfRangeException**) { **Console.WriteLine**("Fatal error -- program terminated."); }

* Throwing an Exception: To manually throw an exception use the throw statement. Its general form is: **throw exceptOb;**
* Here, exceptOb must be an instance of an exception class derived from Exception. Consider following example:

|  |  |
| --- | --- |
| * Note, throw throws an object. Must create an object for it to throw. * Here, the default constructor " ***new DivideByZeroException()*** " is used to create a DivideByZeroException object, but other constructors are available for exceptions. | **try** { **throw** **new** **DivideByZeroException**(); }  **catch** (**DivideByZeroException**) { **Console.WriteLine**("Error caught."); } |

* Rethrowing an Exception: An exception caught by one catch clause can be rethrown so that it can be caught by an outer catch. The most likely reason for rethrowing an exception is to allow multiple handlers access to the exception. To rethrow an exception, you simply specify throw, without specifying an exception. That is, you use this form of throw: **throw ;** *(Example is similar as Java part 6.3)*
* Remember that when you rethrow an exception, it will not be recaught by the same catch clause. It will propagate to an outer catch clause.

**C#\_6.4 Finally (Similar as Java Part 6.5)**

**C#\_6.5 Details on EXCEPTION class**

A catch clause allows you to specify an exception type and a variable. The variable receives a reference to the exception object. Since all exceptions are derived from Exception, all exceptions support the members defined by Exception. We will examine several of Exception's most useful members and constructors, and put the exception variable to use.

* Exception defines several properties. Three of the most interesting are Message, StackTrace, and TargetSite. All are read-only.

|  |  |  |
| --- | --- | --- |
| * Message is a string that describes the nature of the error. | * StackTrace is a string that contains the stack of calls that lead to the exception. | * TargetSite returns an object that specifies the method that generated the exception. |

* Exception also defines several methods. The one that you will most often use is ***ToString()***, which returns a string that describes the exception. ***ToString()*** is automatically called when an exception is displayed via ***WriteLine()***, for example. The following demonstrates the properties and method just mentioned:

|  |  |
| --- | --- |
| using System;  class ExcTest { public static void GenException() {  int[] nums = new int[4];  Console.WriteLine("Before exception.");  nums[7] = 10; /\* index out-of-bounds exception.\*/  Console.WriteLine("this won't be displayed"); }} | class UseExcept { static void Main() {  try { ExcTest.GenException(); }  catch (IndexOutOfRangeException exc) {  Console.WriteLine("Standard message is: " + exc); // calls ToString()  Console.WriteLine("Stack trace: " + exc.StackTrace + "Message: " + exc.Message + "TargetSite: " + exc.TargetSite);  } Console.WriteLine("After catch block."); }} |

|  |  |
| --- | --- |
| * Exception defines the four constructors: | |
| ***public Exception( )*** | This the default constructor. |
| ***public Exception(string str)*** | This specifies the string associated with the Message property associated with the exception. |
| ***public Exception(string str, Exception inner)*** | Specifies what is called an inner exception. It is used when one exception gives rise to another. In this case, inner specifies the first exception, which will be null if no inner exception exists. (The inner exception, if it exists, can be obtained from the InnerException property defined by Exception.) |
| ***protected Exception(***  ***System.Runtime.Serialization.SerializationInfo si,***  ***System.Runtime.Serialization.StreamingContext sc )*** | This constructor handles exceptions that occur remotely and require de-serialization. Notice that the types SerializationInfo and StreamingContext are preceded by System.Runtime.Serialization. This specifies the namespace in which they are contained. |

* Commonly Used Exceptions: The System namespace defines several standard, built-in exceptions. All are derived from SystemException since they are generated by the CLR when runtime errors occur. Commonly used exceptions defined within the System Namespace are given below

|  |  |  |  |
| --- | --- | --- | --- |
| ***ArrayTypeMismatchException*** | Type of value being stored is incompatible with the type of the array | | |
| ***DivideByZeroException*** | Division by zero attempted | ***OutOfMemoryException*** | A call to new fails because insufficient free memory exists |
| ***IndexOutOfRangeException*** | Array index is out of bounds | ***OverflowException*** | An arithmetic overflow occurred |
| ***InvalidCastException*** | A runtime cast is invalid | ***StackOverflowException*** | The stack was overrun |

**C#\_6.6 Custom exceptions: Deriving Exception Classes (Similar to Java part 6.8)**

You can use custom exceptions to handle errors in your own code. To create an exception just define a class derived from Exception. Your derived classes don’t need to actually implement anything, they automatically have the properties and methods defined by Exception available to them. Of course, you can override one or more of these members in exception classes that you create.

* When creating your own exception class, you will generally want your class to support all of the constructors defined by Exception. For simple custom exception classes, this is easy to do because you can ***simply pass along the constructor’s arguments*** to the corresponding Exception constructor via base. Of course, technically, you only need to provide those constructors actually used by your program.
* C#\_Example 2: Here is an example that creates an exception called NonIntResultException. In the program, this exception is thrown when the result of dividing two integer values produces a result with a fractional component.

using System;

class NonIntResultException : Exception { */\* A custom exception \*/*

public NonIntResultException() : base() { }

public NonIntResultException(string str) : base(str) { } */\* Provide the standard constructors \*/*

public NonIntResultException(string str, Exception inner) : base(str, inner) { }

protected NonIntResultException( System.Runtime.Serialization.SerializationInfo si, System.Runtime.Serialization.StreamingContext sc) : base(si, sc) { }

public override string ToString() { return Message;} */\* Override ToString for NonIntResultException. \*/* }

class CustomExceptDemo { static void Main() { int[] numer = { 4, 8, 15, 32, 64, 127, 256, 512 }; *// Here, numer contains some odd values.*

int[] denom = { 2, 0, 4, 4, 0, 8 };

for(int i=0; i < numer.Length; i++) { */\* Throw a custom exception. \*/*

try { if((numer[i] % denom[i]) != 0) throw new NonIntResultException("Outcome of " + numer[i] + " / " + denom[i] + " is not even.");

Console.WriteLine(numer[i] + " / " + denom[i] + " is " + numer[i]/denom[i]); }

catch (DivideByZeroException) { Console.WriteLine("Can't divide by Zero!"); }

catch (IndexOutOfRangeException) { Console.WriteLine("No matching element found."); }

catch (NonIntResultException exc) { Console.WriteLine(exc); } } */\* for lop ends \*/* }}

* Notice that, all of the Exception constructors Implemented. Also notice that the constructors simply execute the base constructor. Because NonIntResultException adds nothing to Exception, there is no need for any further actions. For illustration, NonIntResultException defines all of the standard constructors, even though most are not used by the example. It also overrides the ***ToString()*** method.
* Notice that none of the constructors provide any statements in their body. Instead, they simply pass their arguments along to Exception via base.

**C#\_6.7 Catching Custom/Derived-class Exception**

You need to be careful how you order catch clauses when trying to catch exception types that involve base and derived classes, because a catch clause for a base class will also match any of its derived classes. For example, since the base class of all exceptions is Exception, catching Exception catches all possible exceptions. Of course, using catch without an exception type provides a cleaner way to catch all exceptions.

* If you want to catch exceptions of both a base class type and a derived class type, put the derived class first in the catch sequence. This is necessary because a base class catch will also catch all derived classes. Fortunately, this rule is self-enforcing because putting the base class first causes a compile-time error. For Example:

|  |  |
| --- | --- |
| class ExceptA : Exception {  public ExceptA(string str) : base(str) { }  public override string ToString() { return Message; } }  *// Create an exception derived from ExceptA*  class ExceptB : ExceptA {  public ExceptB(string str) : base(str) { }  public override string ToString() { return Message; } } | class OrderMatters { static void Main() {  for(int x = 0; x < 3; x++) {  try { if(x==0) throw new ExceptA("Caught an ExceptA Error");  else if(x==1) throw new ExceptB("Caught an ExceptB Error ");  else throw new Exception(); }  catch (ExceptB exc) { Console.WriteLine(exc); }  catch (ExceptA exc) { Console.WriteLine(exc); }  catch (Exception exc) { Console.WriteLine(exc); } } */\* for loop ends \*/* }} |

* Notice the order of the catch clauses. This is the only order in which they can occur. Since ExceptB is derived from ExceptA, the catch for ExceptB must be before the one for ExceptA. Similarly, the catch for Exception (which is the base class for all exceptions) must appear last. To prove this point for yourself, try rearranging the catch clauses. Doing so will result in a compile-time error.

A catch that catches a base class exception allows you to catch an entire category of exceptions, possibly handling them with a single catch and avoiding duplicated code.

**C#\_6.8 Catching Using checked and unchecked**

An *arithmetic computation* can cause an overflow. For example, consider the following sequence: **byte** a, b, result; a = 127; b = 127; result = (**byte**)(a \* b);

Here, the product of a and b exceeds the range of a byte value. Thus, the result overflows the type of the result. And gives wrong result, we can handle this kind of error.

* C# allows you to specify whether your code will raise an exception when overflow occurs using the keywords checked and unchecked. To specify that an expression be checked for overflow, used checked. To specify that overflow be ignored, use unchecked. In this case, the result is truncated to fit into the target type of the expression.
* Checked: The checked keyword has these two general forms. One checks a specific expression and is called the operator form of checked. The other checks a block of statements and is called the statement form.

|  |  |
| --- | --- |
| ***checked(expr)*** | ***checked{*** */\* statements to be checked \*/* ***}*** |

* Here, expr is the *expression being checked*. If a ***checked expression*** overflows, then an OverflowException is thrown.
* Unchecked: The unchecked keyword also has two general forms. The first is the operator form, which ignores overflow for a specific expression. The other ignores overflow for a block of statements. They are:

|  |  |
| --- | --- |
| ***unchecked (expr)*** | ***unchecked {*** */\* statements for which overflow is ignored \*/* ***}*** |

* Here, expr is the *expression that is not being checked* for overflow. If an ***unchecked expression*** overflows, then truncation will occur.

|  |
| --- |
| Operator form of checked and unchecked |
| **class** CheckedDemo { **static void Main()** { **byte** a, b, result; a = 127; b = 127;  **try** { result = **unchecked**((**byte**)(a \* b)); */\* The overflow in this expression is truncated. \*/*  result = **checked**((**byte**)(a \* b)); *// The overflow here causes exception*  **Console.WriteLine**("Checked result: " + result); */\* won't execute \*/* }  **catch** (**OverflowException** exc) { **Console.WriteLine**(exc); } }} |

|  |
| --- |
| Statements-block form of checked and unchecked |
| **class** CheckedBlocks { **static void Main()** { **byte** a, b, result; a = 127; b = 127;  **try** { **unchecked** { a = 127; b = 127; result = (**byte**)(a \* b); **Console.WriteLine**("Unchecked result: " + result);  a = 125; b = 5; result = (**byte**)(a \* b); **Console.WriteLine**("Unchecked result: " + result); }  **checked** { a = 2; b = 7; result = (**byte**)(a \* b); **Console.WriteLine**("Checked result: " + result); */\* this is OK \*/*  a = 127; b = 127; result = (**byte**)(a \* b); */\* this causes exception \*/* **Console.WriteLine**("Checked: " + result); */\* won't execute \*/*  }  }  **catch** (**OverflowException** exc) { **Console.WriteLine**(exc); } }} |

* One reason that you may need to use checked or unchecked is that the checked/unchecked status of overflow is determined by the setting of a compiler option and by the ***execution environment*** itself. Thus, for some types of programs, it is best to explicitly specify the ***overflow check status***.

**C#\_6.9 C# I/O System : Predefined Streams**

|  |  |
| --- | --- |
| C# programs perform I/O through streams (similar to Java). C#C also has Byte Streams and Character Streams. There are Three predefined streams, which are exposed by the properties Console.In, Console.Out, and Console.Error, are available to all programs that use the System namespace. | * Console.Out refers to the ***standard output*** stream. By default, this is the console. When you call ***Console.WriteLine()***, for example, it automatically sends information to Console.Out. * Console.In refers to ***standard input***, which is, by default, the keyboard. * Console.Error refers to the standard error stream, which is also the console by default. * these streams can be redirected to any ***compatible I/O device***. The ***standard streams are character streams***. Thus, these streams read and write characters. |

**C#\_6.10 The Stream Classes: Byte Stream, Character Stream and Binary Streams**

The I/O system defines both byte and character stream classes. However, the character stream classes are really just wrappers that convert an underlying byte stream to a character stream, handling any conversion automatically.

* All stream classes are defined within the System.IO namespace. To use these classes, include this statement near the top of a program: ***using System.IO;***
* Console class is defined in the System namespace. So you don’t have to specify System.IO for console input and output.
* The Stream Class: The core stream class is System.IO.Stream. Stream represents a byte stream and is a base class for all other stream classes. It is also abstract, i.e. you cannot instantiate a Stream object directly. Stream defines a set of standard stream operations. Some methods defined by Stream are:

|  |  |
| --- | --- |
| ***A Sampling of the*** Methods ***Defined by*** Stream | |
| Method | Description |
| ***void Close( )*** | Closes the stream. |
| ***void Flush( )*** | Writes the contents of the stream to the physical device. |
| ***int ReadByte( )*** | Returns an integer representation of the next available byte of input. Returns –1 when the end of the file is encountered. |
| ***int Read(byte[ ] buf, int offset, int numBytes)*** | Attempts to read up to numBytes bytes into buf starting at buf [offset], returning the number of bytes successfully read. |
| ***long Seek(long offset, SeekOrigin origin)*** | Sets the current position in the stream to the specified offset from the specified origin. |
| ***void WriteByte(byte b)*** | Writes a single byte to an output stream. |
| ***int Write(byte[ ] buf, int offset, int numBytes)*** | Writes a subrange of numBytes bytes from the array buf, beginning at buf [offset]. The *number of written bytes* returned. |

* Several of the methods shown in ***above table*** will throw an IOException if an I/O error occurs. If an invalid operation is attempted, such as attempting to write to a stream that is readonly, a NotSupportedException is thrown. Other exceptions are possible, depending on the specific method.
* Notice that Stream defines methods that read and write data. However, not all streams will support both of these operations because it is possible to open read-only or write-only streams.
* Not all streams will support position requests via ***Seek()***. To determine the capabilities of a stream, you will use one or more of Stream’s properties. They are shown in ***following table***. Also shown are the Length and Position properties, which contain the length of the stream and its current position.

|  |  |
| --- | --- |
| ***The*** Properties ***Defined by*** Stream | |
| Property | Description |
| ***bool CanRead*** | This property is true if the stream can be read. This property is read-only. |
| ***bool CanSeek*** | This property is true if the stream supports position requests. This property is read-only. |
| ***bool CanTimeout*** | This property is true if the stream can time out. This property is read-only. |
| ***bool CanWrite*** | This property is true if the stream can be written. This property is read-only. |
| ***long Length*** | This property contains the length of the stream. This property is read-only. |
| ***long Position*** | This property represents the current position of the stream. This property is read/write. |
| ***int ReadTimeout*** | This property represents the length of time before a timeout will occur for read operations. This property is read/write. |
| ***int WriteTimeout*** | This property represents the length of time before a timeout will occur for write operations. This property is read/write. |

* The Byte Stream Classes: Several concrete byte streams are derived from Stream. Those that are defined in the System.IO namespace are shown here:

|  |  |  |  |
| --- | --- | --- | --- |
| ***BufferedStream*** | Wraps a byte stream and adds buffering. Buffering provides a performance enhancement in many cases. | ***FileStream*** | A byte stream designed for file I/O. |
| ***UnmanagedMemoryStream*** | A byte stream that uses memory for storage, but is not suitable for mixed-language programming. | ***MemoryStream*** | A byte stream that uses memory for storage. |

* Several other concrete stream classes that provide support for compressed files, sockets, and pipes, among others, are also supported by the .NET Framework. It is also possible for you to derive your own stream classes. However, for the vast majority of applications, the built-in streams will be sufficient.
* Character Stream Wrapper Classes: To create a character stream, you will wrap a byte stream inside one of the character stream wrappers. At the top of the character-stream hierarchy are the abstract classes TextReader and TextWriter. The methods defined by these classes are available to all of their subclasses.
* Following table shows the input methods in TextReader. In general, these methods can throw an IOException on error. (Some also throw other types of exceptions.)

|  |  |  |  |
| --- | --- | --- | --- |
| The Input Methods Defined by TextReader | | | |
| Method | Description | | |
| ***int Peek()*** | Obtains the next character from the input stream, but ***does not remove*** that character. Returns –1 if ***no character*** is available. | | |
| ***int Read()*** | Returns an integer representation of the ***next available character*** from the input stream. Returns –1 when the end of the stream is encountered. | | |
| ***int Read(char[] buf, int offset, int numChars)*** | | | Attempts to read up to numChars characters into buf starting at buf [offset], returning the ***number of characters*** successfully read. |
| ***int ReadBlock(char[] buf, int offset, int numChars)*** | | |
| ***string ReadLine()*** | | Reads the next line of text and returns it as a string. Null is returned if an attempt is made to read at end-of-file. | |
| ***string ReadToEnd()*** | | Returns all of the remaining characters in a stream and returns them as a string. | |

* ***ReadLine()*** reads an entire line of text, returning it as a string. This method is useful when reading input that contains embedded spaces.
* TextWriter defines versions of ***Write()*** and ***WriteLine()*** that output all of the built-in types. For example, here are just a few of their overloaded versions:

|  |  |  |  |
| --- | --- | --- | --- |
| ***void Write(int val)*** | Write an int. | ***void WriteLine(string val)*** | Write a string followed by a new line. |
| ***void Write(double val)*** | Write a double. | ***void WriteLine(uint val)*** | Write a uint followed by a new line. |
| ***void Write(bool val)*** | Write a bool. | ***void WriteLine(char val)*** | Write a character followed by a new line. |

* All throw an IOException if an error occurs while writing.

|  |  |
| --- | --- |
| * In addition to ***Write()*** and ***WriteLine()***, ***TextWriter*** defines the ***Close()*** and ***Flush()*** methods: * ***Flush()*** causes any data remaining in the output buffer to be written to the physical medium. * ***Close()*** closes the stream. | ***virtual void Close()***  ***virtual void Flush()*** |

* The TextReader and TextWriter classes are implemented by *several character-based* *stream classes*, these streams provide the *methods* and *properties* specified by TextReader and TextWriter. These classes and their descriptions are given below

|  |  |  |  |
| --- | --- | --- | --- |
| StreamReader | Read characters from a byte stream. This class wraps a ***byte input stream.*** | StringReader | Read characters from a string. |
| StreamWriter | Write characters to a byte stream. This class wraps a ***byte output stream***. | StringWriter | Write characters to a string. |

* Binary Streams: In addition to the byte and character streams, there are two binary stream classes, which can be used to read and write binary data directly. These streams are called BinaryReader and BinaryWriter.

**C#\_6.11 Console I/O**

Console I/O is accomplished through the standard streams Console.In, Console.Out, and Console.Error.

* Reading Console Input: Console.In is an instance of TextReader, and you can use the methods and properties defined by TextReader to access it. You will usually use the methods provided by Console, which automatically read from Console.In.
* Console defines two input methods: ***Read()*** and ***ReadLine()***.
* Read( ): To read a single character, use the Read( ) method. Its form: ***static int Read()*** It returns the ***next character read*** from the ***console***.
* The character is returned as an int, which must be cast to char. It returns –1 on error. This method will throw an IOException on failure.
* ***Read()*** is line-buffered, so you must press ***ENTER*** before any character that you type will be sent to your program.
* ReadLine( ): To read a string of characters, use the ***ReadLine()*** method. It is: ***static string ReadLine( )***
* ***ReadLine()*** reads characters until you press ***ENTER*** and returns them in a string object. This method will also throw an IOException on failure.
* For example, to reading a *line of characters* from Console.In: **Console.WriteLine**("Enter some characters."); **string** str = **Console.ReadLine**();
* You can call methods on the underlying TextReader, which is available in Console.In. Eg: ***string str = Console.In.ReadLine();***
* Notice how ***ReadLine()*** is now invoked directly on ***Console.In***.
* Writing Console Output: Console.Out and Console.Error are objects of type TextWriter. Console output is most easily accomplished with ***Write()*** and ***WriteLine()***, you can invoke these (and other) methods on the TextWriter that underlies Console.Out and Console.Error if you choose. Eg:

**class** ErrOut { **static void Main()** { **int** a=10, b=0; **int** result;

**Console.Out.WriteLine**("This will generate an exception."); /*\* Write to Console.Out \*/*

**try** { result = a / b; */\* generate an exception \*/* }

**catch**(**DivideByZeroException** exc) { **Console.Error.WriteLine**(exc.Message); */\* Write to Console.Error\*/* } }}

* Both Console.Out and Console.Error default to writing their output to the console, but there are two different streams. Because, the *standard streams can be redirected to other devices*. For example, Console.Error can be redirected to write to a disk file, rather than to the screen.
* Thus, it is possible to direct error output to a log file, for example, ***without affecting*** console ***output***. Conversely, if console output is redirected and error output is not, then *error messages will appear on the console*, where they can be seen.
* C# support an interactive input method that returns as soon as any key is pressed (not line-buffered): ***ReadKey()*** is a Console method, When it is called, it waits until a key is pressed. When a key is pressed, ***ReadKey()*** returns the keystroke immediately. You do not need to press ENTER. Thus, ReadKey( ) allows keystrokes to be read and processed in real time. ***ReadKey()*** has these two forms:

|  |  |
| --- | --- |
| ***static ConsoleKeyInfo ReadKey()*** | ***static ConsoleKeyInfo ReadKey(bool noDisplay)*** |

* The first form waits for a key to be pressed. When that occurs, it returns the key and also displays the key on the screen.
* The second form also waits for and returns a key-press. However, if noDisplay is true, then the key is not displayed. If noDisplay is false, the key is displayed.

**C#\_6.12 File I/O (part 1): FileStream and Byte-Oriented File I/O**

The I/O system provides classes that allow you to read and write files. Of course, the most common type of file is the disk file. At the operating system level, all files are byte-oriented. There are methods that read and write bytes from and to a file. You can also wrap a byte-oriented file stream within a character-based object. Character-based file operations are useful when text is being stored.

* To create a byte-oriented stream attached to a file, you will use the FileStream class. FileStream is derived from Stream and contains all of Stream’s functionality. Remember, the stream classes, including FileStream, are defined in System.IO. Thus, you will usually include: ***using System.IO;***
* Opening and Closing a File: To create a byte stream linked to a file, create a FileStream object. FileStream defines several constructors. Most commonly used one:

***FileStream(string filename, FileMode mode)***

* Here, filename specifies the name of the file to open, which can include a full path specification. The mode parameter specifies how the file will be opened. It must be one of the values defined by the FileMode enumeration. These values are shown in following Table.
* In general, this constructor opens a file for read/write access. The exception is when the file is opened using FileMode.Append. In this case, the file is write-only.
* An exception will be thrown if file opening failed. If the file can't be opened because it doesn't exist, FileNotFoundException will be thrown.
* If the file cannot be opened because of some type of I/O error, IOException will be thrown.
* Other possible exceptions are ArgumentNullException (the filename is null), ArgumentException (the filename is invalid), ArgumentOutOfRangeException (the mode is invalid), SecurityException (user does not have access rights), PathTooLongException (the filename/ path is too long), NotSupportedException (the filename specifies an unsupported device), and DirectoryNotFoundException (specified directory is invalid).
* The exceptions PathTooLongException, DirectoryNotFoundException, and FileNotFoundException are subclasses of IOException. Thus, it is possible to catch all three by catching IOException.

|  |  |  |  |
| --- | --- | --- | --- |
| FileMode.Append | Output is appended to the end of the file. | FileMode.OpenOrCreate | Opens a file if it exists, or creates the file if it does not already exist. |
| FileMode.Open | Opens a preexisting file. | FileMode.CreateNew | Creates a new output file. The file must not *already exist*. |
| FileMode.Truncate | Opens a preexisting file, but reduces its length to zero | FileMode.Create | Creates a new output file. Any preexisting file by the same name will be destroyed. |

* C#\_Example 3: The following shows one way to open the file test.dat for input:

|  |  |
| --- | --- |
| **FileStream** fin;  **try** { fin = **new** FileStream("test", FileMode.Open); }  **catch**(**IOException** exc) { **Console.WriteLine**(exc.Message);  */\* Handle the error.\*/* }  **catch**(**Exception** exc { **Console.WriteLine**(exc.Message); */\* catch any other exception \*/*  */\* Handle the error.\*/* } | * The first catch clause handles situations in which the file is not found, the path is too long, the directory does not exist, or other I/O errors occur. * The second catch, which is a “catch all” clause for all other types of exceptions, handles the other possible errors (possibly by rethrowing the exception). |

* Restrict access: When FileMode.Append is specified, the FileStream constructor just described opens a file with read/write access. If you want to restrict access to just reading or just writing, use this constructor instead: ***FileStream(string filename, FileMode mode, FileAccess how)***
* filename specifies the name of the file to open, and mode specifies how the file will be opened. The value passed in how determines how the file can be accessed. It must be one of the values defined by the FileAccess enumeration: ***FileAccess.Read FileAccess.Write FileAccess.ReadWrite***
* For example, this opens a read-only file: ***FileStream fin = new FileStream("test.dat", FileMode.Open, FileAccess.Read);***
* Closing a File: When you are done with a file, you must close it by calling ***Close()***. Its general form is: ***void Close()***
* Closing a file releases the system resources allocated to the file. ***Close()*** works by calling ***Dispose()***, which actually frees the resources.
* using: using statement, also, offers a way to automatically close a file when it is no longer needed, and this approach is applicable to a variety of situations.
* Reading Bytes from a FileStream: ***FileStream*** defines two methods that read bytes from a file: ***ReadByte()*** and ***Read()***.
* To read a single byte from a file, use ***ReadByte()***, whose general form is: ***int ReadByte()*** Each time it is called, it reads a single byte from the file and returns it as an integer value. It returns –1 when the EOF is encountered. Possible exceptions include NotSupportedException (the stream is not opened for input) and ObjectDisposedException (the stream is closed).
* To read a block of bytes, use ***Read()***, which has this general form: ***int Read(byte[] buf, int offset, int numBytes)***

Read( ) attempts to read up to numBytes bytes into buf starting at buf [offset]. It returns the number of bytes successfully read. An IOException is thrown if an I/O error occurs. Several other types of exceptions are possible, including NotSupportedException, which is thrown if ***reading is not supported*** by the stream.

|  |  |
| --- | --- |
| using System; using System.IO;  class ShowFile { static void Main(string[] args) {int i;  FileStream fin;  if(args.Length != 1) { Console.WriteLine("Usage: ShowFile File"); return; }  try { fin = new FileStream(args[0], FileMode.Open); }  catch(IOException exc) { Console.WriteLine(exc.Message); return; } | do { try { i = fin.ReadByte(); } */\* Read from the file \*/*  catch(IOException exc) { Console.WriteLine(exc.Message); break; }  if(i != -1) Console.Write((char) i);  } while(i != -1); *// Read bytes until EOF is encountered.*  fin.Close(); }} |

* Writing to a File: To write a ***byte*** to a file, use the ***WriteByte( )*** method. Its simplest form is: ***void WriteByte(byte val)*** This method writes the byte specified by val to the file. If the underlying stream is not opened for output, a NotSupportedException is thrown. If the stream is closed, ObjectDisposedException is thrown.
* You can write an array of bytes to a file by calling ***Write()***. It is: ***int Write(byte[] buf, int offset, int numBytes)***

Write( ) writes numBytes bytes from the array buf, beginning at buf[offset], to the file. The number of bytes written is returned. If an error occurs during writing, an IOException is thrown. If the underlying stream is not opened for output, a NotSupportedException is thrown. Other exceptions are possible.

* Buffered Output: When file output is performed, often, that output is not immediately written to the actual physical device. Instead, output is buffered by the operating system until a sizable chunk of data can be written all at once. For example, disk files are organized by sectors, which might be anywhere from 128 bytes long, on up. Output is usually buffered until an entire sector can be written all at once.
* Flush( ): If you want to cause data to be written to the physical device, whether the buffer is full or not, you can call Flush( ): ***void Flush( )***
* An IOException is thrown on failure. If the stream was closed at the time of the call, ObjectDisposedException is thrown.
* Once you are done with an output file, you must remember to closeit using ***Close()***. Doing so ensures that any output remaining in a disk buffer is actually written to the disk. It is not necessary to call ***Flush()*** before closing a file.

|  |  |
| --- | --- |
| using System; using System.IO;  class CopyFile { static void Main(string[] args) { int i;  FileStream fin, fout;  if(args.Length != 2) { Console.WriteLine("Usage: CopyFile From To"); return; }  try { fin = new FileStream(args[0], FileMode.Open); } // Open input file.  catch(IOException exc) { Console.WriteLine(exc.Message); return; } | try { fout = new FileStream(args[1], FileMode.Create); } *// Open output file.*  catch(IOException exc) { Console.WriteLine(exc.Message); fin.Close(); return; }  try { do { i = fin.ReadByte(); */\* Read bytes from one file \*/*  if(i != -1) fout.WriteByte((byte)i); */\* write them to another.\*/*  } while(i != -1); }  catch(IOException exc) { Console.WriteLine(exc.Message); }  fin.Close(); fout.Close(); }} |

**C#\_6.13 File I/O (part 2):Character-Based File I/O**

If you want to store Unicode text, the character streams are certainly your best option. In general, to perform character-based file operations, you will wrap a FileStream inside either a StreamReader or a StreamWriter. These classes automatically convert a byte stream into a character stream, and vice versa.

* At the operating system level, a file consists of a set of bytes. Using a StreamReader or StreamWriter does not alter this fact. StreamReader is derived from TextReader. Thus, StreamWriter and StreamReader have access to the methods and properties defined by their base classes.
* StreamWriter: To create a character-based output stream, wrap a Stream object (such as a FileStream) inside a StreamWriter. StreamWriter defines several constructors. One of its most popular is : ***StreamWriter(Stream stream)***
* Here, stream is the name of an open stream. This constructor throws an ArgumentException if the specified stream is not opened for output and an ArgumentNullException if stream is null. Once created, a StreamWriter automatically handles the conversion of characters to bytes.
* C#\_Example 4: Here is a simple key-to-disk utility that reads lines of text entered at the keyboard and writes them to a file called test.txt. Text is read until the user enters the word “stop.” The utility uses a FileStream wrapped in a StreamWriter to output to the file.

|  |  |
| --- | --- |
| using System;  using System.IO;  class KtoD {  static void Main() {  string str;  FileStream fout; | try { fout = new FileStream("test.txt", FileMode.Create); } catch(IOException exc) { Console.WriteLine(exc.Message); return ; } */\* create file \*/*  Console.WriteLine("Enter text ('stop' to quit)."); StreamWriter fstr\_out = new StreamWriter(fout); */\* Create a* ***StreamWriter****.\*/*  do { Console.Write(": "); str = Console.ReadLine();  if(str != "stop") { str = str + "\r\n"; */\* add newline \*/*  try { fstr\_out.Write(str); } catch(IOException exc) { Console.WriteLine(exc.Message); break; } } */\* if statement ends \*/*  } while(str != "stop"); fstr\_out.Close(); }} |

* Other StreamWriter constructors: In some cases, you can open a file directly using StreamWriter. To do so, use one of these constructors:

|  |  |
| --- | --- |
| ***StreamWriter(string filename)*** | ***StreamWriter(string filename, bool appendFlag)*** |

* Here, filename specifies the name of the file to open, which can include a full path specifier. In the second form, if appendFlag is true, then output is appended to the ***end of an existing file***. Otherwise, output overwrites the specified file. In both cases, if the file does not exist, it is created. Also, both throw an IOException if an I/O error occurs. Other exceptions are also possible. For Example, we can use following line into C#\_Example 4 :

StreamWriter fstr\_out;

try { fstr\_out = new StreamWriter("test.txt"); */\* Open a file using only StreamWriter. \*/* } catch(IOException exc) { Console.WriteLine(exc.Message); return ; }

* StreamReader: To create a character-based input stream, wrap a byte stream inside a StreamReade. StreamReader defines several constructors. A frequently used one is: ***StreamReader(Stream stream)*** Here, stream is the name of an open stream. This constructor throws an ArgumentNullExceptionif stream is null and an ArgumentException if the stream is not opened for input. Once created, a StreamReader will automatically handle the conversion of bytes to characters.
* C#\_Example 5: The following program uses StreamReader to create a simple disk-to-screen utility that reads line-by-line a text file called test.txt and displays its contents on the screen. Thus, it is the complement of the key-to-disk utility shown in the previous section.

|  |  |
| --- | --- |
| using System; using System.IO;  class DtoS { static void Main() {  FileStream fin;  string s; | try { fin = new FileStream("test.txt", FileMode.Open); } catch(IOException exc) { Console.WriteLine(exc.Message); return ; }  StreamReader fstr\_in = new StreamReader(fin);  try { while((s = fstr\_in.ReadLine()) != null) { Console.WriteLine(s); } } catch(IOException exc) { Console.WriteLine(exc.Message); }  fstr\_in.Close(); }} |

* Notice how the ***EOF*** is determined. When the reference returned by ***ReadLine()*** is ***null***, the EOF has been reached.
* Other StreamReader constructors: You can open a file directly using this StreamReader constructor: ***StreamReader(string filename)***
* Here, filename specifies the name of the file to open, which can include a ***full path specifier***. The file must exist. If it doesn’t, a FileNotFoundException is thrown. If ***filename is null***, then an ArgumentNullException is thrown. If filename is an ***empty string***, ArgumentException is thrown. IOException and DirectoryNotFoundException are also possible.
* CHARACTER ENCODING – (UTF-8 ENCODING) when opening a StreamReader or StreamWriter: StreamReader and StreamWriter convert bytes to characters and vice versa based upon a character encoding that specifies how the translation occurs. By default, C# uses the UTF-8 encoding, which is compatible with ASCII. To specify another encoding, you will use overloaded versions of the StreamReader or StreamWriter constructors that include an encoding parameter. In general, you will need to specify a character encoding only under unusual circumstances.

**C#\_6.14 Redirecting the Standard Streams**

The standard streams, such as Console.In, can be redirected. By far, the most common redirection is to a file. When a standard stream is redirected, input or output is automatically directed to the new stream, bypassing the default devices. Redirection of the standard streams can be accomplished in two ways

* By redirecting the standard streams, your program can read commands from a disk file, create log files, or even read input from a network connection.
* Redirection using Command line (Without changing program): When you execute a program on the command line, you can use the **<** and **>** operators to redirect Console.In and Console.Out, respectively. For example:

**using System; class Test { static void Main() { Console.WriteLine("This is a test."); } }**

* Executing the program like this: **Test > log** will cause the line “This is a test.” to be written to a file called log.
* Input can be redirected in the same way. The thing to remember when input is redirected is that you must make sure that what you specify as an input source contains sufficient input to satisfy the demands of the program. If it doesn’t, the program will hang.
* The **<** and **>** command-line redirection operators are not part of C#, but are provided by the operating system. Thus, if your environment supports I/O redirection (as is the case with Windows), you can redirect standard input and standard output without making any changes to your program.
* Redirection from program: To do so, you will use the ***SetIn()***, ***SetOut()***, and ***SetError()*** methods, shown here, which are members of Console:

|  |  |  |
| --- | --- | --- |
| ***static void SetIn(TextReader input)*** | ***static void SetOut(TextWriter output)*** | ***static void SetError(TextWriter output)*** |

* Thus, to redirect input, call ***SetIn()***, specifying the desired stream. You can use any input stream as long as it is derived from TextReader. To redirect output, specify any stream derived from TextWriter. For example, to redirect output to a file, use a StreamWriter.

|  |  |
| --- | --- |
| **using System**; **using System.IO**;  **class** Redirect {  **static void Main()** { **StreamWriter** log\_out;  **try** { log\_out = **new** **StreamWriter**("logfile.txt"); }  **catch**(**IOException** exc) { **Console.WriteLine**(exc.**Message**); **return** ; } | **Console.SetOut**(log\_out); */\* Redirect Console.Out \*/*  **try** { **for**(**int** i=0; i<10; i++) **Console.WriteLine**(i);  **Console.WriteLine**("This is the end of the log file."); }  **catch**(**IOException** exc) { **Console.WriteLine**(exc.**Message**); }  log\_out.**Close**(); }} |

* When you run this program, you won’t see any output on the screen. However, the file logfile.txt will contain the Result:

**C#\_6.15Reading and Writing Binary Data**

To read and write binary values of the C# built-in types, you will use BinaryReader and BinaryWriter. When using these streams, it is important to understand that this data is read and written using its internal, binary format, not its human-readable text form.

* BinaryWriter: A BinaryWriter is a wrapper around a byte stream that manages the writing of binary data. Its most commonly used constructor is:

***BinaryWriter(Stream outputStream)***

* Here, outputStream is the stream to which data is written. To write output to a file, you can use the object created by FileStream for this parameter. If outputStream is null, then an ArgumentNullException is thrown. If outputStream has not been opened for writing, ArgumentException is thrown.
* BinaryWriter defines methods that can write all of C#’s built-in types. Several are shown in Following Table. BinaryWriter also defines the standard ***Close()*** and ***Flush()*** methods that work as described earlier.
* BinaryReader: A BinaryReader is a wrapper around a byte stream that handles the reading of binary data. Its most commonly used constructor is:

***BinaryReader(Stream inputStream)***

* Here, inputStream is the stream from which data is read. To read from a file, you can use the object created by FileStream for this parameter. If inputStream has not been opened for input or is otherwise invalid, an ArgumentException is thrown.
* BinaryReader provides methods for reading all of C#’s built-in types. The most commonly used are shown in Following Table. BinaryReader also defines three versions of ***Read()***, which are:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **int Read()** | | Returns an integer representation of the next available character from the invoking input stream. Returns –1 when attempting to read at the EOF. | | |
| **int Read(byte[] buf, int offset, int num)** | | Attempts to read up to num bytes into buf, starting at buf [offset], and returns the number of bytes successfully read. | | |
| **int Read(char[] buf, int offset, int num)** | | Attempts to read up to num characters into buf, starting at buf [offset], and returns the number of characters successfully read. | | |
| BinaryWriter: Commonly Used Output Methods Defined by BinaryWriter | | | BinaryReader: Commonly Used Input Methods Defined by BinaryReader | |
| Method | Description | | Method | Description |
| ***void Write(sbyte val)*** | Writes a signed byte. | | ***bool ReadBoolean()*** | Reads a bool. |
| ***void Write(byte val)*** | Writes an unsigned byte. | | ***byte ReadByte()*** | Reads a byte. |
| ***void Write(byte[] buf )*** | Writes an array of bytes. | | ***sbyte ReadSByte()*** | Reads an sbyte. |
| ***void Write(short val)*** | Writes a short integer. | | ***byte[] ReadBytes(int num)*** | Reads num bytes and returns them as an array. |
| ***void Write(ushort val)*** | Writes an unsigned short integer. | | ***char ReadChar()*** | Reads a char. |
| ***void Write(int val)*** | Writes an integer. | | ***char[] ReadChars(int num)*** | Reads num characters and returns them as an array. |
| ***void Write(uint val)*** | Writes an unsigned integer. | | ***double ReadDouble()*** | Reads a double. |
| ***void Write(long val)*** | Writes a long integer. | | ***float ReadSingle()*** | Reads a float. |
| ***void Write(ulong val)*** | Writes an unsigned long integer. | | ***short ReadInt16()*** | Reads a short. |
| ***void Write(float val)*** | Writes a float. | | ***int ReadInt32()*** | Reads an int. |
| ***void Write(double val)*** | Writes a double. | | ***long ReadInt64()*** | Reads a long. |
| ***void Write(char val)*** | Writes a character. | | ***ushort ReadUInt16()*** | Reads a ushort. |
| ***void Write(char[] buf )*** | Writes an array of characters. | | ***uint ReadUInt32()*** | Reads a uint. |
| ***void Write(string val)*** | Writes a string. | | ***ulong ReadUInt64()*** | Reads a ulong. |
| * These methods will throw an IOException on failure. | | | ***string ReadString()*** | Reads a string. |

* C#\_Example 6: Here is a program that demonstrates BinaryReader and BinaryWriter. It writes and then reads back various types of data to and from a file.

using System; using System.IO;

class RWData { static void Main() { BinaryWriter dataOut; BinaryReader dataIn; int i = 10; double d = 1023.56; bool b = true;

|  |  |
| --- | --- |
| try { dataOut = new BinaryWriter(new FileStream("testdata", FileMode.Create)); }  catch(IOException exc) { Console.WriteLine(exc.Message); return; }  *// Write binary data to a file.*  try { Console.WriteLine("Writing " + i); dataOut.Write(i);  Console.WriteLine("Writing " + d); dataOut.Write(d);  Console.WriteLine("Writing " + b); dataOut.Write(b);  Console.WriteLine("Writing " + 12.2 \* 7.4); dataOut.Write(12.2 \* 7.4); }  catch(IOException exc) { Console.WriteLine(exc.Message); }  dataOut.Close();  Console.WriteLine(); | *// Now, read the data.*  try { dataIn = new BinaryReader(new FileStream("testdata", FileMode.Open)); }  catch(IOException exc) { Console.WriteLine(exc.Message); return; }  try { i = dataIn.ReadInt32(); Console.WriteLine("Reading " + i);  d = dataIn.ReadDouble(); Console.WriteLine("Reading " + d);  b = dataIn.ReadBoolean(); Console.WriteLine("Reading " + b);  d = dataIn.ReadDouble(); Console.WriteLine("Reading " + d); }  catch(IOException exc) { Console.WriteLine(exc.Message); }  dataIn.Close(); }} |

**C#\_6.16 Random Access Files**

You can also access the contents of a file in random order. To do this, you will use the ***Seek()*** method defined by FileStream. This method allows you to set the file position indicator (also called the file pointer) to any point within a file. The method ***Seek()*** is: ***long Seek(long newPos, SeekOrigin origin)***

* Here, newPos specifies the new position, in bytes, of the file pointer from the location specified by origin. The origin will be one of these values, which are defined by the SeekOrigin *enumeration*:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ***Begin*** | Seek from the beginning of the file. | ***Current*** | Seek from the current location | ***End*** | Seek from the end of the file |

* After a call to ***Seek()***, the next read or write operation will occur at the new file position. If an error occurs while seeking, an IOException is thrown. If the underlying stream does not support position requests, a NotSupportedException is thrown. Other exceptions are possible.
* C#\_Example 7: Here is an example that demonstrates random access I/O. It writes the uppercase alphabet to a file and then reads it back in nonsequential order.

**using System**; **using System.IO**;

**class** RandomAccessDemo { **static void Main()** { **FileStream** f; **char** ch;

**try** { f = **new** **FileStream**("random.dat", **FileMode.Create**); } **catch**(**IOException** exc) { **Console.WriteLine**(exc.**Message**); **return** ; }

*/\* Write the alphabet \*/*

**for**(**int** i=0; i < 26; i++) { **try** { f.**WriteByte**((**byte**)('A'+i)); } **catch**(**IOException** exc) { **Console.WriteLine**(exc.**Message**); f.**Close**(); **return** ; } }

*/\* Use Seek( ) to move the file pointer \*/*

**try** { */\* seek to first byte \*/*  f.**Seek**(0, **SeekOrigin.Begin**); ch = (**char**) f.**ReadByte**(); **Console.WriteLine**("First value is " + ch);

*/\* seek to second byte \*/* f.**Seek**(1, **SeekOrigin.Begin**); ch = (**char**) f.**ReadByte**(); **Console.WriteLine**("Second value is " + ch);

*/\* seek to 5th byte \*/* f.**Seek**(4, **SeekOrigin.Begin**); ch = (**char**) f.**ReadByte**(); **Console.WriteLine**("Fifth value is " + ch);

*/\* Now, read every other value.\*/* **Console.WriteLine**(); **Console.WriteLine**("Here is every other value: ");

for(int i=0; i < 26; i += 2) {

*/\* seek to ith character \*/* f.**Seek**(i, **SeekOrigin.Begin**); ch = (**char**) f.**ReadByte**(); **Console.Write**(ch + " "); } */\** ***for*** *ends \*/* } */\* try ends \*/*

**catch**(**IOException** exc) { **Console.WriteLine**(exc.**Message**); }

**Console.WriteLine**(); f.**Close**(); }}

**C#\_6.17 .NET Structure Name (similar Java TYPE WRAPPER) and Parse( ) :**

Converting Numeric Strings to Their Internal Representation (Recall Java part 6.18 and 8.13)

|  |  |  |  |
| --- | --- | --- | --- |
| * .NET Structure types for C#’s built-in types: The C#’s built-in types, (int and double) and .NET structure type are *indistinguishable*. One is just another name for the other. Because C#’s value types are supported by structures, the value types have members defined for them. * These structures offer a wide array of methods that help *fully integrate* the value types into C#’s object hierarchy. As a side benefit, the numeric structures also define static methods that convert a numeric string into its ***corresponding binary equivalent***. * For the *C# numeric value types*, the .NET structure names and their C# keyword equivalents with corresponding conversion methods are here: * Parse(): To reads and converts strings containing numeric values into their internal, binary format, use ***Parse()*** which is defined for all of the ***built-in numeric types***. | ***.NET structures & C# keyword equivalents with conversion methods:*** | | |
| .NET Structure type | C# type | Conversion Method |
| ***Decimal*** | *decimal* | ***static decimal Parse(string str)*** |
| ***Double*** | *double* | ***static double Parse(string str)*** |
| ***Single*** | *float* | ***static float Parse(string str)*** |
| ***Int16*** | *short* | ***static short Parse(string str)*** |
| ***Int32*** | *int* | ***static int Parse(string str)*** |
| ***Int64*** | *long* | ***static long Parse(string str)*** |
| ***UInt16*** | *ushort* | ***static ushort Parse(string str)*** |
| ***UInt32*** | *uint* | ***static uint Parse(string str)*** |
| ***UInt64*** | *ulong* | ***static ulong Parse(string str)*** |
| ***Byte*** | *byte* | ***static byte Parse(string str)*** |
| ***SByte*** | *sbyte* | ***static sbyte Parse(string str)*** |

* These structures are defined inside the System namespace. Thus, the fully qualified name for Int32 is System.Int32. Conversion methods returns a binary value that corresponds to the string. The Parse() methods can throw FormatException if str does not contain a valid number as defined by the invoking type. ArgumentNullException is thrown if str is null, and OverflowException is thrown if the value in str exceeds the bounds of the invoking type.

str = Console.ReadLine(); try { n = Int32.Parse(str); } */\* Convert a string to an int\*/*

catch(FormatException exc) {Console.WriteLine(exc.Message); return; } catch(OverflowException exc) { Console.WriteLine(exc.Message); return; }

* All of the structures have methods called ***CompareTo()***, which compare the values contained within the wrapper; ***Equals()***, which tests two values for equality; and methods that return the value of the object in various forms. The numeric structures also include the fields MinValue and MaxValue, which contain the minimum and maximum values that can be stored by an object of its type.