Stream

* A stream is an abstraction that either produces or consumes information.
* A stream is linked to a physical device by the I/O system.
* All streams behave in the same manner, even if the actual physical devices they are linked to differ.
* Thus, the I/O classes and methods can be applied to many types of devices.
* The same methods that you use to write to the console can also be used to write to a disk file.
* Two Types,
  + Input Stream
    - if data flows into my application, then we call it as an input stream.
  + Output Stream
    - if the data flows away from my application, then I would call it as output stream.
* The direction of flow determines whether it is an input or output stream.
* Types of Input streams,
  + Byte Stream
  + Character Stream
  + Binary Stream
* Byte Stream and Character Stream
  + Many devices are byte oriented when it comes to I/O operations.
  + At the lowest level, all C# I/O operates on bytes.
  + In C#, char is a 16-bit type, and byte is an 8-bit type.

Conversion between char and byte

* If the ASCII character set is used, then it is easy to convert between char and byte, just ignore the high-order byte of the char value.
* But this won’t work for the rest of the Unicode characters.
* Thus, byte streams are not perfectly suited to handling character-based I/O.
* To solve this problem, the .NET Framework defines several classes that convert a byte stream into a character stream, handling the translation of byte-to-char and char-to-byte automatically.

Stream Classes

* Stream - <<abstract Class>>
* BufferedStream
* FileStream
* MemoryStream
* UnManagedMemoryStream

|  |
| --- |
| Notes\*   * The read method which is not abstract is a method of TextReader (char stream) abstract class. |

MemoryStream

* A MemoryStream represents a Stream in memory.
* There are two ways to create a MemoryStream. You can initialize one from an unsigned byte array, or you can create an empty one. Empty memory streams are resizable, while ones created with a byte array provide a stream “view” of the data.

Stream Classes

* BinaryReader and BinaryWriter – for reading and writing primitive data types as binary values.
* TextReader – Abstract Class; TextWriter – Abstract Class.
  + For reading and writing characters by using an encoding value to convert the characters to and from bytes.
* StreamReader / StreamWriter
  + For reading and writing characters to and from strings.
* StringReader / StringWriter

BinaryStream Classes

* BinaryReader
* BinaryWriter

File Handling

* Has a bunch of static methods some of them are, (see pic)
  + File.ReadAllBytes :
    - Useful for files not stored as plain text.
  + File.ReadAllLines :
    - Opens a file, reads all lines of the file with the specified encoding.
  + File.ReadAllText :
  + File.WriteAllBytes :
  + File.WriteAllLines :
    - Stores a string array in the specified file overwriting the contents.
  + File.WriteAllText :
  + File.AppendAllText :
  + File.AppendText :

Guideline

* **StreamReader** is much faster for large files with 10,000+ lines, but the difference for smaller files is negligible.
* As always, plan for varying sizes of files and use **File.ReadAllLines** only when performance isn’t critical.

System.IO.File Vs StreamWriter

* When a file that is a gigabyte or more is read with **System.io.File**, then there is a guaranteed crash on the 32-bit version.
* No such problem with a **StreamReader** that reads line-by-line, it will use very little memory.

Guidelines for reading an image

* Use **File.ReadAllBytes** to read in an image, PNG, to memory.
* One example usage of this is to cache an image in memory for performance.
* This works very well and gently outperforms reading in the image each time.

Serialization

* Serialization can be defined as the process of storing the state of an object instance to a storage medium.
* During this process, the public and private fields of the object and the name of the class, including the assembly containing the class, is converted to a **stream** **of** **bytes**, which is then written to a **data** **stream**.

Deserialization

* When the object is **deserialized**, an exact clone of the original serialized object is created.

Steps in the Serialization Process

* A check is made to determine if the object is marked with the Serializable attribute. If it is not, a SerializationException is thrown.
* If it is marked appropriately, check if the object implements ISerializable. If it does, GetObjectData() is called on the object.
* If the object does not implement ISerializable, the default serialization policy is used, serializing all fields not marked as NonSerialized.
* A check is made to determine if the formatter has a surrogate selector. If it does, check if the surrogate selector handles objects of the given type. If the selector handles the object type, ISerializable.GetObjectData() is called on the surrogate selector.
* If there is no surrogate selector or if it does not handle the type, a check is made to determine if the object is marked with the Serializable attribute. If it is not, a SerializationException is thrown.

Formatter

* Two types,
  + BinaryFormatter
    - Converts Object -> Byte -> Saves in File System
  + SOAPFormatter
* BinaryFormatter
  + The BinaryFormatter is very efficient and produces a very compact byte stream.
  + All objects serialized with this formatter can also be deserialized with it.
  + It is important to note that constructors are not called when an object is deserialized. (DS obj can be created without a constructor)

Working with Tasks

* Task is an Unit of Work to been done in a separate Thread.
* Wrapper for ThreadPool.
* Demon
  + A thread that runs in background
  + All services.msi in OS runs as Demon.
* ThreadPool threads are not started or stopped they toggle between wait and sleep join and state.
* Starting a Task(see pic)
  + TaskFactory
    - A factory is a place where objects are created
    - Here creates task object and start it.
    - We can recycle the objects threads
  + Prefer 3rd opt
  + M1 function pointer/delegate

Task Hierarchies

* With Task Continuations the Task form a hierarchies.
* When a Task starts a new Task a Parent-Child hierarchy is started.
* When the Parent task is Stopped the Child Task also Stops.

TaskStatus

* Created
* WaitingForActivation
* WaitingForRun
* Running
* WaitingForChildrenToComplete
* RanToCompletion
* Canceled
* Faulted