The StringBuilder Class

- The StringBuilder class allows you to create and modify strings without the overhead of recreating new strings each time.
- It generates a mutable sequence of characters that can change size dynamically as the string is modified, allocating more memory as required.

StringBuilder Classes

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StringBuilder()</td>
<td>Initializes a new default instance with a size of 16</td>
</tr>
<tr>
<td>StringBuilder(int)</td>
<td>Initializes a new instance with a capacity of int</td>
</tr>
<tr>
<td>StringBuilder(string)</td>
<td>Initializes a new instance with a default value of string</td>
</tr>
<tr>
<td>StringBuilder(int1, int2)</td>
<td>Initializes a new instance with a default capacity of int1 and a maximum capacity of int2</td>
</tr>
<tr>
<td>StringBuilder(string, int)</td>
<td>Initializes a new instance with a default value of string and a capacity of int</td>
</tr>
<tr>
<td>StringBuilder(string, int1, int2, int3)</td>
<td>Initializes a new instance with a default value starting at position int1 of string, int2 characters long, with a capacity of int3</td>
</tr>
</tbody>
</table>

StringBuilder Example

```csharp
using System;
using System.Text;
namespace ConsoleApplication10
{
    class Program
    {
        static void Main(string[] args)
        {
            StringBuilder sb = new StringBuilder("test string");
            int length = 0;
            length = sb.Length;
            Console.WriteLine("The result is: '{0} {1}'", sb, length);
            Console.WriteLine("The length is: {0}", length);
            sb.Length = 4;
            length = sb.Length;
            Console.WriteLine("The result is: '{0}'", sb);
            Console.WriteLine("The length is: {0}", length);
            sb.Length = 20;
            length = sb.Length;
            Console.WriteLine("The result is: '{0}'", sb);
            Console.WriteLine("The length is: {0}", length);
        }
    }
}
```

StringBuilder with Append

```csharp
using System;
using System.Text;
namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            StringBuilder sb = new StringBuilder();
            int number = 1;
            sb.AppendFormat("{0}: {1} ", number++, "another string");
            Console.WriteLine("{0}", sb);
        }
    }
}
```
C# Streams

- Data handling is one of the most important jobs of programs.
- The C# language supplies an interface to assist programmers in moving large chunks of data to and from data objects.
- The data stream allows multiple bytes of data to be transferred simultaneously to a data object so that programs can work on blocks of data instead of having to build data elements one byte at a time.

Streams- II

Streams can support three fundamental operations:
- Transferring data from a stream to a memory buffer (reading)
- Transferring data from a memory buffer to a stream (writing)
- Searching the stream for a specific byte pattern (seeking)

C# Exception Programming

- One of the biggest problems for programmers is dealing with abnormal conditions in a program.
- Inexperienced programmers often forget to compensate for error conditions such as dividing by zero.
- This results in ugly and annoying error messages and programs blowing up in customers' faces.
- Such error conditions, or other unexpected behavior occurring when a program executes, are called exceptions.
catch -I
using System;
using System.IO;

class FinallyDemo
{
    static void Main(string[] args)
    {
        FileStream outStream = null;
        FileStream inStream = null;
        try
        {
            outStream = File.OpenWrite("DestinationFile.txt");
            inStream = File.OpenRead("BogusInputFile.txt");
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.ToString());
        }
        finally
        {
            if (outStream != null)
            {
                outStream.Close();
                Console.WriteLine("outStream closed.");
            }
            if (inStream != null)
            {
                inStream.Close();
                Console.WriteLine("inStream closed.");
            }
        }
    }
}

catch -II
using System;
using System.IO;
class CatchError
{
    static void Main()
    {
        int var1 = 1000, var2 = 0, var3;
        try
        {
            var3 = var1 / var2;
        }
        catch (ArithmeticException e)
        {
            Console.WriteLine("Exception: {0}", e.ToString());
            var3 = -1;
        }
        catch (Exception e)
        {
            Console.WriteLine("Exception: {0}", e.ToString());
            var3 = -2;
        }
        Console.WriteLine("The result is: {0}", var3);
    }
}

using System;
namespace Exception3
{
    class X
    {
        int x;
        public X(int a) { x = a; }
        public int add(X o) { return x + o.x; }
    }
    public class NREDemo
    {
        public static void Main()
        {
            X p = new X(10);
            X q = null; // q is explicitly assigned null
            int val;
            try
            {
                val = p.add(q); /*this will lead to an exception*/
            }
            catch (NullReferenceException)
            {
                Console.WriteLine("NullReferenceException!");
                Console.WriteLine("fixing...
                // now, fix it
                q = new X(9);
                val = p.add(q);
                Console.WriteLine("val is {0}", val);";
            }
        }
    }
}

IP Programming Basics
- The Internet Protocol (IP) is at the core of network programming.
- IP is the vehicle that transports data between systems, whether within a local area network (LAN) environment or a wide area network (WAN) environment.
- The two most popular protocols that use IP:
  - The Transmission Control Protocol (TCP),
  - The User Datagram Protocol (UDP).
- The TCP Layer
  - The Transmission Control Protocol (TCP) adds connection information to the data packet.
  - This allows programs to create an end-to-end connection between two network devices, providing a consistent path for data to travel.

The TCP-II
- Because of this feature, TCP is called a connection-oriented protocol.
- Each TCP connection, or session, includes a certain amount of packet overhead related to establishing the connection between the two devices.
- Once the connection is established, data can be sent between the devices without the application having to check for lost or out-of-place data.
To communicate with an application on a remote device, you must know two pieces of information:
- The remote device's IP address
- The TCP port assigned to the remote application

Establishing a TCP Session

- TCP uses connection states to determine the status of a connection between devices.
- A specific handshaking protocol is used to establish these connections and to monitor the status of the connection during the session.
- The TCP session has three phases:
  - Opening handshake
  - Session communication
  - Closing handshake

The UDP Layer

- The User Datagram Protocol (UDP) is another popular high-level protocol used in IP communications.
- After studying TCP, UDP will seem like a snap.
- Unlike TCP, UDP provides a connectionless path between network devices to transmit data, and thus does not need all of the overhead of session establishment flags and connection states.

The UDP header fields are pretty straightforward:
- Source Port
- Destination Port
- Message Length
- Checksum
- Next Level Protocol

Programming with TCP and UDP

- Now that you have looked under the hood of TCP and UDP
- These two protocols move data between network devices in very different ways.

TCP Programming Features

- The most important thing to remember about using TCP is that it is a connection-oriented protocol.
- Once a connection exists between two devices, a reliable data stream is established to ensure that data is accurately moved from one device to the other.
TCP Programming Features - II

- TCP must ensure the integrity of the data, it keeps all sent data in a local buffer until a positive acknowledgement of reception is received from the remote device.
- TCP must keep a local buffer of received data to ensure that all of the pieces are received in order before passing the data to the application program.

TCP must ensure the integrity of the data, it keeps all sent data in a local buffer until a positive acknowledgement of reception is received from the remote device. TCP must keep a local buffer of received data to ensure that all of the pieces are received in order before passing the data to the application program.

Finding IP Address Information

System.Net Namespace

One of the biggest advantages you will notice in the .NET network library is the way IP address/port pairs are handled. .NET defines two classes in the System.Net namespace to handle various types of IP address information:

- IPAddress
- IPEndPoint

IPAddress Methods

- Equals: Compares two IP addresses
- GetHashCode: Returns a hash value for an IPAddress object
- GetType: Returns the type of the IP address instance
- HostToNetworkOrder: Converts an IP address from host byte order to network byte order
- IsLoopBack: Indicates whether the IP address is considered the loopback address
- NetworkToHostOrder: Converts an IP address from network byte order to host byte order
- Parse: Converts a string to an IPAddress instance
- ToString: Converts an IPAddress to a string representation of the dotted decimal format of the IP address

The IPAddress class also provides four read-only fields that represent special IP addresses for use in programs:

- Any: Used to represent any IP address available on the local system
- Broadcast: Used to represent the IP broadcast address for the local network
- Loopback: Used to represent the loopback address of the system
- None: Used to represent no network interface on the system

Example

```csharp
using System;
using System.Net;

class AddressSample
{
    public static void Main()
    {
        IPAddress test1 = IPAddress.Parse("192.168.1.10");
        IPAddress test2 = IPAddress.Loopback;
        IPAddress test3 = IPAddress.Broadcast;
        IPAddress test4 = IPAddress.Any;
        IPAddress test5 = IPAddress.None;

        IPHostEntry ihe = Dns.GetHostByName(Dns.GetHostName());
        IPAddress myself = ihe.AddressList[0];

        if (IPAddress.IsLoopback(test2))
            Console.WriteLine("The Loopback address is: {0}", test2.ToString());
        else
            Console.WriteLine("Error obtaining the loopback address");

        Console.WriteLine("The Local IP address is: {0}", myself.ToString());

        if (myself == test2)
            Console.WriteLine("The loopback address is the same as local address.");
        else
            Console.WriteLine("The loopback address is not the local address.");

        Console.WriteLine("The test address is: {0}", test1.ToString());
        Console.WriteLine("Broadcast address: {0}", test3.ToString());
        Console.WriteLine("The ANY address is: {0}", test4.ToString());
        Console.WriteLine("The NONE address is: {0}", test5.ToString());
    }
}
```
The results
- C:\>AddressSample
  - The Loopback address is: 127.0.0.1
  - The Local IP address is: 192.168.1.6
  - The loopback address is not the local address.
  - The test address is: 192.168.1.1
  - Broadcast address: 255.255.255.255
  - The ANY address is: 0.0.0.0
  - The NONE address is: 255.255.255.255

IPEndPoint
- The .NET Framework uses the **IPEndPoint** object to represent a specific IP address/port combination.
- An **IPEndPoint** object is used when binding sockets to local addresses, or when connecting sockets to remote addresses.

**IPEndPoint** Methods
- **Create**: Creates an EndPoint object from a SocketAddress object
- **Equals**: Compares two **IPEndPoint** objects
- **GetHashCode**: Returns a hash value for an **IPEndPoint** object
- **GetType**: Returns the type of the **IPEndPoint** instance
- **Serialize**: Creates a SocketAddress instance of the **IPEndPoint** instance
- **ToString**: Creates a string representation of the **IPEndPoint** instance

Example **IPEndPoint**
```csharp
using System;
using System.Net;

class IPEndPointSample
{
    public static void Main()
    {
        IPAddress test1 = IPAddress.Parse("192.168.1.1");
        IPEndPoint ie = new IPEndPoint(test1, 8000);
        Console.WriteLine("The IPEndPoint is: {0}", ie.ToString());
        Console.WriteLine("The AddressFamily is: {0}", ie.AddressFamily);
        Console.WriteLine("The address is: {0}, and the port is: {1}
", ie.Address, ie.Port);
        Console.WriteLine("The min port number is: {0}", IPEndPoint.MinPort);
        Console.WriteLine("The max port number is: {0}
", IPEndPoint.MaxPort);
        ie.Port = 80;
        Console.WriteLine("The changed IPEndPoint value is: {0}", ie.ToString());
        SocketAddress sa = ie.Serialize();
        Console.WriteLine("The SocketAddress is: {0}", sa.ToString());
    }
}
```

What are Sockets?
- Sockets provide a common interface to the various protocols supported by networks.
- They allow you to establish connections between machines to send and receive data.
- Sockets support the simultaneous connection of multiple clients to a single server machine.
Socket Logical Structure

How do Sockets Function?
- There are several modes of operation available for sockets.
- A very common mode is to establish a socket listener that listens on some port, say 4040, for connection requests.
- When a socket client, from another process or a remote computer, requests a connection on port 4040, the listener spawns a new thread that starts up a socket server on a new port, say 5051.
- From that time on the socket client and socket server communicate on port 5051. Either one can send data, in the form of a group of bytes, to the other.
- Meanwhile the listener goes back to listening for connection requests on port 4040.

Socket Client, Server, and Listener

How do Sockets Function?
- There are several modes of operation available for sockets.
- A very common mode is to establish a socket listener that listens on some port, say 4040, for connection requests.
- When a socket client, from another process or a remote computer, requests a connection on port 4040, the listener spawns a new thread that starts up a socket server on a new port, say 5051.
- From that time on the socket client and socket server communicate on port 5051. Either one can send data, in the form of a group of bytes, to the other.
- Meanwhile the listener goes back to listening for connection requests on port 4040.

Client/Server Configuration

Socket Data Transfer
- The receiving socket, either client or server, has a buffer that stores bytes of data until the receiver thread reads them.
- If the receiver buffer is full, the sender thread will block on a send call until the receiver reads some of the data out of the buffer.
- For this reason, it is a good idea to assign a thread to the receiver to empty the buffer and enqueue the data for a worker thread to digest.
- If the receiver buffer becomes full during a send, the send request will return having sent less than the requested number of bytes.
- If the receiving buffer is empty, a read request will block.
- If the receiving buffer has data, but less than the number of bytes requested by a read, the call will return with the bytes available.

Non-Blocking Communication
Socket Programming: Synchronous Clients

- The steps for creating a simple synchronous client are as follows.
- Create a Socket instance.
- Connect the above socket instance to an endpoint.
- Send or Receive information.
- Shutdown the socket.
- Close the socket.
- The Socket class provides a constructor for creating a Socket instance.

Basic .Net Network Objects

- **TCPEndPoint**
  - TCPEndPoint(port)
  - AcceptTcpClient()
  - AcceptSocket()
  - Start()
  - Stop()
- **Socket**
  - Send(byte[], size, socketFlags)
  - Receive(byte[], size, socketFlags)
  - Close()
  - Shutdown(SocketShutDown)

More Network Programming Objects

- **TCPClient**
  - TCPClient()
  - Connect(IPAddress, port)
  - GetStream()
  - Close()
- **NetworkStream**
  - NetworkStream(Socket)
  - Read(byte[], offset, size)
  - Write(byte[], offset, size)

Simple Socket Client

```csharp
TcpClient tcpc = new TcpClient();
Byte[] read = new Byte[32]; // read buffer
String server = args[0]; // server name
// Try to connect to the server
tcpc.Connect(server, 2048);
// Get a NetworkStream object
Stream s;
  s = tcpc.GetStream();
// Read the stream and convert it to ASCII
int bytes = s.Read(read, 0, read.Length);
  String Time = Encoding.ASCII.GetString(read);
// Display the data
  Console.WriteLine("Received {0} bytes", bytes);
  Console.WriteLine("Current date and time is: {0}", Time);
  tcpc.Close();
```

Simple Socket Server

Examples of Socket

What About Thread Synchronization?

- **Threads**
  - 1) is the main GUI thread that starts when you start the Server application. The thread labeled 1.
  - 2) starts whenever any client tries to connect to the socket. The thread labeled 2.
  - 3) spawns when there is any write activity by any one of the connected clients.
Multi-threaded Server

- If we want to support concurrent clients, the server must spawn a thread for each new client.
- C# Thread class makes that fairly simple.
  - Create a class that provides a non-static processing function. This is the code that serves each client.
  - Each time the TCPListener accepts a client it returns a socket. Pass that to the thread when it is constructed, and start the thread.

Multi Thread

- Example "Thread"