University of Technology
Computer Sciences
Dr. Mohammad Natiq

Network Programming
Lecture One

Working with Variables, Operators, and Expressions using C Sharp Language

Understanding Statements

A *statement* is a command that performs an action, such as calculating a value and storing the result, or displaying a message to a user. You combine statements to create methods. You'll learn more about methods in Chapter 3, "Writing Methods and Applying Scope," but for now, think of a method as a named sequence of statements. *Main*, which was introduced in the previous chapter, is an example of a method.

Statements in C# follow a well-defined set of rules describing their format and construction. These rules are collectively known as *syntax*. (In contrast, the specification of what statements *do* is collectively known as *semantics*.) One of the simplest and most important C# syntax rules states that you must terminate all statements with a semicolon. For example, you saw in Chapter 1 that without its terminating semicolon, the following statement won't compile:

```
Console.WriteLine("Hello World!");
```

Tip C# is a "free format" language, which means that white space, such as a space character or a new line, is not significant except as a separator. In other words, you are free to lay out your statements in any style you choose. However, you should adopt a simple, consistent layout style to make your programs easier to read and understand.

The trick to programming well in any language is learning the syntax and semantics of the language and then using the language in a natural and idiomatic way. This approach makes your programs more easily maintainable. As you progress through this book, you'll see examples of the most important C# statements.

Using Identifiers

Identifiers are the names you use to identify the elements in your programs, such as namespaces, classes, methods, and variables. (You will learn about variables shortly.) In C#, you must adhere to the following syntax rules when choosing identifiers:

- You can use only letters (uppercase and lowercase), digits, and underscore characters.
- An identifier must start with a letter or an underscore.

For example, *result*, *_score*, *footballTeam*, and *plan9* are all valid identifiers, whereas *result%*, *footballTeam$*, and *9plan* are not.

Important C# is a case-sensitive language: *footballTeam* and *FootballTeam* are not the same identifier.

Identifying Keywords

The C# language reserves 77 identifiers for its own use, and you cannot reuse these identifiers for your own purposes. These identifiers are called *keywords*, and each has a particular meaning. Examples of keywords are *class*, *namespace*, and *using*. You'll learn the meaning of most of the C# keywords as you proceed through this book. The following table lists the keywords:

abstract	do	in	protected	true
as	double	int	public	try
base	else	interface	readonly	typeof
bool	enum	internal	ref	uint
break	event	is	return	ulong
byte	explicit	lock	sbyte	unchecked
case	extern	long	sealed	unsafe
catch	false	namespace	short	ushort
char	finally	new	sizeof	using
checked	fixed	null	stackalloc	virtual
class	float	object	static	void
const	for	operator	string	volatile
continue	foreach	out	struct	while
decimal	goto	override	switch	
default	if	params	this	
delegate	implicit	private	throw	

Tip In the Visual Studio 2012 Code and Text Editor window, keywords are colored blue when you type them.

C# also uses the following identifiers. These identifiers are not reserved by C#, which means that you can use these names as identifiers for your own methods, variables, and classes, but you should avoid doing so if at all possible.

add	get	remove
alias	global	select
ascending	group	set
async	into	value
await	join	var
descending	let	where
dynamic	orderby	yield
from	partial	

Using Variables

A *variable* is a storage location that holds a value. You can think of a variable as a box in the computer's memory holding temporary information. You must give each variable in a program an unambiguous name that uniquely identifies it in the context in which it is used. You use a variable's name to refer to the value it holds. For example, if you want to store the value of the cost of an item in a store, you might create a variable simply called *cost* and store the item's cost in this variable. Later on, if you refer to the *cost* variable, the value retrieved will be the item's cost that you stored there earlier.

Naming Variables

You should adopt a naming convention for variables that helps you avoid confusion concerning the variables you have defined. This is especially important if you are part of a project team with several developers working on different parts of an application; a consistent naming convention helps to avoid confusion and can reduce the scope for bugs. The following list contains some general recommendations:

- Don't start an identifier with an underscore. Although this is legal in C#, it can limit the interoperability of your code with applications built by using other languages, such as Microsoft Visual Basic.
- Don't create identifiers that differ only by case. For example, do not create one variable named *myVariable* and another named *MyVariable* for use at the same time, because it is too easy to get them confused. Also, defining identifiers that differ only by case can limit the ability to reuse classes in applications developed by using other languages that are not case sensitive, such as Microsoft Visual Basic.
- Start the name with a lowercase letter.

Declaring Variables

Variables hold values. C# has many different types of values that it can store and process—integers, floating-point numbers, and strings of characters, to name three. When you declare a variable, you must specify the type of data it will hold.

You declare the type and name of a variable in a declaration statement. For example, the following statement declares that the variable named *age* holds *int* (integer) values. As always, the statement must be terminated with a semicolon.

```
int age;
```

The variable type *int* is the name of one of the *primitive* C# types, *integer*, which is a whole number. (You'll learn about several primitive data types later in this chapter.)

After you've declared your variable, you can assign it a value. The following statement assigns *age* the value 42. Again, you'll see that the semicolon is required.

```
age = 42;
```

The equal sign (=) is the *assignment* operator, which assigns the value on its right to the variable on its left. After this assignment, you can use the *age* variable in your code to refer to the value it holds. The next statement writes the value of the *age* variable, 42, to the console:

```
Console.WriteLine(age);
```

Working with Primitive Data Types

C# has a number of built-in types called *primitive data types*. The following table lists the most commonly used primitive data types in C# and the range of values that you can store in each.

Data type	Description	Size (bits)	Range	Sample usage

int	Whole numbers (integers)	32	–231 through 231 – 1	int count;
				count = 42;

long	Whole numbers (bigger range)	64	–263 through 263 – 1	long wait;
				wait = 42L;

float	Floating-point numbers	32	±1.5 x 10-45 through ±3.4 x 1038	float away;
				away = 0.42F;

double	Double-precision (more accurate) floating-point numbers	64	±5.0 x 10-324 through ±1.7 x 10308	double trouble;
				trouble = 0.42;

decimal	Monetary values	128	28 significant figures	decimal coin;
				coin = 0.42M;

string	Sequence of characters	16 bits per character	Not applicable	string vest;
				vest = "fortytwo";

char	Single character	16	0 through 216 – 1	char grill;
				grill = 'x';

bool	Boolean	8	true or false	bool teeth;
				teeth = false;

University of Technology
Computer Sciences
Dr. Mohammad Natiq

Network Programming
Lecture Two

Writing Methods and Applying Scope using C Sharp Language

Creating Methods

A *method* is a named sequence of statements. If you have previously programmed using a language such as C, C++, or Microsoft Visual Basic, you will see that a method is similar to a function or a subroutine. A method has a name and a body. The method name should be a meaningful identifier that indicates the overall purpose of the method (*calculateIncomeTax*, for example). The method body contains the actual statements to be run when the method is called. Additionally, methods can be given some data for processing and can return information, which is usually the result of the processing. Methods are a fundamental and powerful mechanism.

Declaring a Method

The syntax for declaring a C# method is as follows:

```
returnType methodName ( parameterList )
{
    // method body statements go here
}
```

- The *returnType* is the name of a type and specifies the kind of information the method returns as a result of its processing. This can be any type, such as *int* or *string*. If you're writing a method that does not return a value, you must use the keyword *void* in place of the return type.

- The *methodName* is the name used to call the method. Method names follow the same identifier rules as variable names. For example, *addValues* is a valid method name, whereas *add$Values* is not. For now, you should follow the camelCase convention for method names—for example, *displayCustomer*.

- The *parameterList* is optional and describes the types and names of the information that you can pass into the method for it to process. You write the parameters between opening and closing parentheses, ( ), as though you're declaring variables, with the name of the type followed by the name of the parameter. If the method you're writing has two or more parameters, you must separate them with commas.

- The method body statements are the lines of code that are run when the method is called. They are enclosed between opening and closing braces, { }.

Here's the definition of a method called *addValues* that returns an *int* result and has two *int* parameters, *leftHandSide* and *rightHandSide*:

```
int addValues(int leftHandSide, int rightHandSide)
{


     // ...
     // method body statements go here
     // ...

}
```

Here's the definition of a method called *showResult* that does not return a value and has a single *int* parameter, called *answer*:

```
void showResult(int answer)
{
    // ...
}
```

Notice the use of the keyword *void* to indicate that the method does not return anything.

Important Visual Basic programmers should notice that C# does not use different keywords to distinguish between a method that returns a value (a function) and a method that does not return a value (a procedure or subroutine). You must always specify either a return type or *void*.

Returning Data from a Method

If you want a method to return information (that is, its return type is not *void*), you must include a *return* statement at the end of the processing in the method body. A *return* statement consists of the keyword *return* followed by an expression that specifies the returned value, and a semicolon. The type of the expression must be the same as the type specified by the method declaration. For example, if a method returns an *int*, the *return* statement must return an *int*; otherwise, your program will not compile. Here is an example of a method with a *return* statement:

```
int addValues(int leftHandSide, int rightHandSide)
{
    // ...
    return leftHandSide + rightHandSide;
}
```

The *return* statement is usually positioned at the end of the method because it causes the method to finish, and control returns to the statement that called the method, as described later in this chapter. Any statements that occur after the *return* statement are not executed (although the compiler warns you about this problem if you place statements after the *return* statement).

If you don't want your method to return information (that is, its return type is *void*), you can use a variation of the *return* statement to cause an immediate exit from the method. You write the keyword *return* immediately followed by a semicolon. For example:

```
void showResult(int answer)
{
    // display the answer
    ...
    return;
}
```

If your method does not return anything, you can also omit the *return* statement because the method finishes automatically when execution arrives at the closing brace at the end of the method. Although this practice is common, it is not always considered good style.

In the following exercise, you will examine another version of the MathsOperators project from Chapter 2. This version has been improved by the careful use of some small methods. Dividing code in this way helps to make it easier to understand and more maintainable.

Examine method definitions الجزء العملي

1. Start Visual Studio 2012 if it is not already running.
2. Open the Methods project in the \Microsoft Press\Visual CSharp Step By Step\Chapter 3\Windows *X*\Methods folder in your Documents folder.
3. On the DEBUG menu, click Start Debugging.

   Visual Studio 2012 builds and runs the application. It should look the same as the application from Chapter 2.
4. Refamiliarize yourself with the application and how it works, and then return to Visual Studio. On the DEBUG menu, click Stop Debugging (or click Quit in the Methods window if you are using Windows 7).
5. Display the code for MainWindow.xaml.cs in the Code and Text Editor window (in Solution Explorer, expand the MainWindow.xaml file and then double-click MainWindow.xaml.cs).
6. In the Code and Text Editor window, locate the *addValues* method.

   The method looks like this:

```
private int addValues(int leftHandSide, int rightHandSide)
{
    expression.Text = leftHandSide.ToString() + " + " + rightHandSide.ToString();
    return leftHandSide + rightHandSide;
}
```

Note Don't worry about the *private* keyword at the start of the definition of this method for the moment; you will learn what this keyword means in Chapter 7, "Creating and Managing Classes and Objects."

The *addValues* method contains two statements. The first statement displays the calculation being performed in the *expression* text box on the form. The values of the parameters *leftHandSide* and *rightHandSide* are converted to strings (using the *ToString* method you met in Chapter 2) and concatenated together using the string version of the plus operator (+).

The second statement uses the *int* version of the + operator to add the values of the *leftHandSide* and *rightHandSide int* variables together, and then returns the result of this operation. Remember that adding two *int* values together creates another *int* value, so the return type of the *addValues* method is *int*.

If you look at the methods *subtractValues, multiplyValues, divideValues*, and *remainderValues*, you will see that they follow a similar pattern.

7. In the Code and Text Editor window, locate the *showResult* method.

The *showResult* method looks like this:

```
private void showResult(int answer)
{
    result.Text = answer.ToString();
}
```

This method contains one statement that displays a string representation of the *answer* parameter in the *result* text box. It does not return a value, so the type of this method is *void*.

Tip There is no minimum length for a method. If a method helps to avoid repetition and makes your program easier to understand, the method is useful regardless of how small it is.

There is also no maximum length for a method, but usually you want to keep your method code small enough to get the job done. If your method is more than one screen in length, consider breaking it into smaller methods for readability.

Calling Methods

Methods exist to be called! You call a method by name to ask it to perform its task. If the method requires information (as specified by its parameters), you must supply the information requested. If the method returns information (as specified by its return type), you should arrange to capture this information somehow.

Specifying the Method Call Syntax

The syntax of a C# method call is as follows:

```
result = methodName ( argumentList )
```

- The *methodName* must exactly match the name of the method you're calling. Remember, C# is a case-sensitive language.
- The *result* = clause is optional. If specified, the variable identified by *result* contains the value returned by the method. If the method is *void* (that is, it does not return a value), you must

omit the *result* = clause of the statement. If you don't specify the *result* = clause and the method does return a value, the method runs but the return value is discarded.

- The *argumentList* supplies the information that the method accepts. You must supply an argument for each parameter, and the value of each argument must be compatible with the type of its corresponding parameter. If the method you're calling has two or more parameters, you must separate the arguments with commas.

Important You must include the parentheses in every method call, even when calling a method that has no arguments.

To clarify these points, take a look at the *addValues* method again:

```
int addValues(int leftHandSide, int rightHandSide)
{
    // ...
}
```

The *addValues* method has two *int* parameters, so you must call it with two comma-separated *int* arguments, such as this:

```
addValues(39, 3);     // okay
```

You can also replace the literal values 39 and 3 with the names of *int* variables. The values in those variables are then passed to the method as its arguments, like this:

```
int arg1 = 99;
int arg2 = 1;
addValues(arg1, arg2);
```

If you try to call *addValues* in some other way, you will probably not succeed for the reasons described in the following examples:

```
addValues;            // compile-time error, no parentheses
addValues();          // compile-time error, not enough arguments
addValues(39);        // compile-time error, not enough arguments
addValues("39", "3"); // compile-time error, wrong types for arguments
```

The *addValues* method returns an *int* value. This *int* value can be used wherever an *int* value can be used. Consider these examples:

```
int result = addValues(39, 3);    // on right-hand side of an assignment
showResult(addValues(39, 3));     // as argument to another method call
```

The following exercise continues with the Methods application. This time, you will examine some method calls.

Console Class

Represents the standard input, output, and error streams for console applications.

Examples

The following example demonstrates how to read data from, and write data to, the standard input and output streams. Note that these streams can be redirected by using the SetIn and SetOut methods.

```
using System;

public class Example {
    public static void Main()
    {
        Console.Write("Hello ");
        Console.WriteLine("World!");
        Console.Write("Enter your name: ");
        String name = Console.ReadLine();
        Console.Write("Good day, ");
        Console.Write(name);
        Console.WriteLine("!");
    }
}
// The example displays output similar to the following:
//        Hello World!
//        Enter your name: James
//        Good day, James!
```

University of Technology
Computer Sciences
Dr. Mohammad Natiq

Network Programming
Lecture Three

Object-Oriented Programming

OOP stands for Object-Oriented Programming. Procedural programming is about writing procedures or methods that perform operations on the data, while object-oriented programming is about creating objects that contain both data and methods.

Object-oriented programming has several advantages over procedural programming:

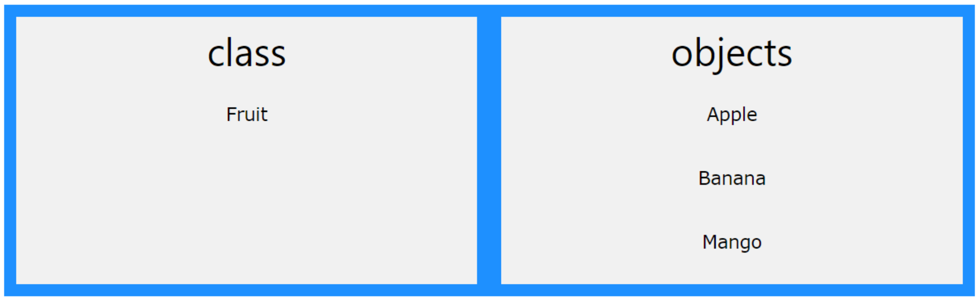
- OOP is faster and easier to execute.
- OOP provides a clear structure for the programs.
- OOP helps to keep the C# code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug.
- OOP makes it possible to create full reusable applications with less code and shorter development time.

Tip: The "Don't Repeat Yourself" (DRY) principle is about reducing the repetition of code. You should extract out the codes that are common for the application, and place them at a single place and reuse them instead of repeating it.

What are Classes and Objects?

Classes and objects are the two main aspects of object-oriented programming.

Look at the following illustration to see the difference between class and objects:



So, a class is a template for objects, and an object is an instance of a class.

Classes and Objects

You learned from the previous chapter that C# is an object-oriented programming language.

Everything in C# is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an object. The car has attributes, such as weight and color, and methods, such as drive and brake.

A Class is like an object constructor, or a "blueprint" for creating objects.

Create a Class

To create a class, use the class keyword:

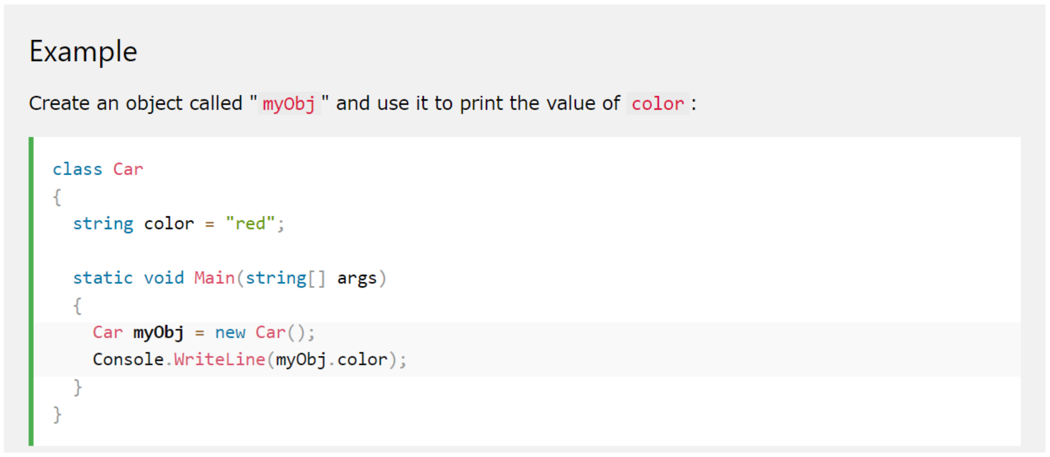
Create a class named "Car" with a variable color:

```
class Car
{
  string color = "red";
}
```

Create an Object

An object is created from a class. We have already created the class named Car, so now we can use this to create objects.

To create an object of Car, specify the class name, followed by the object name, and use the keyword new:


Example

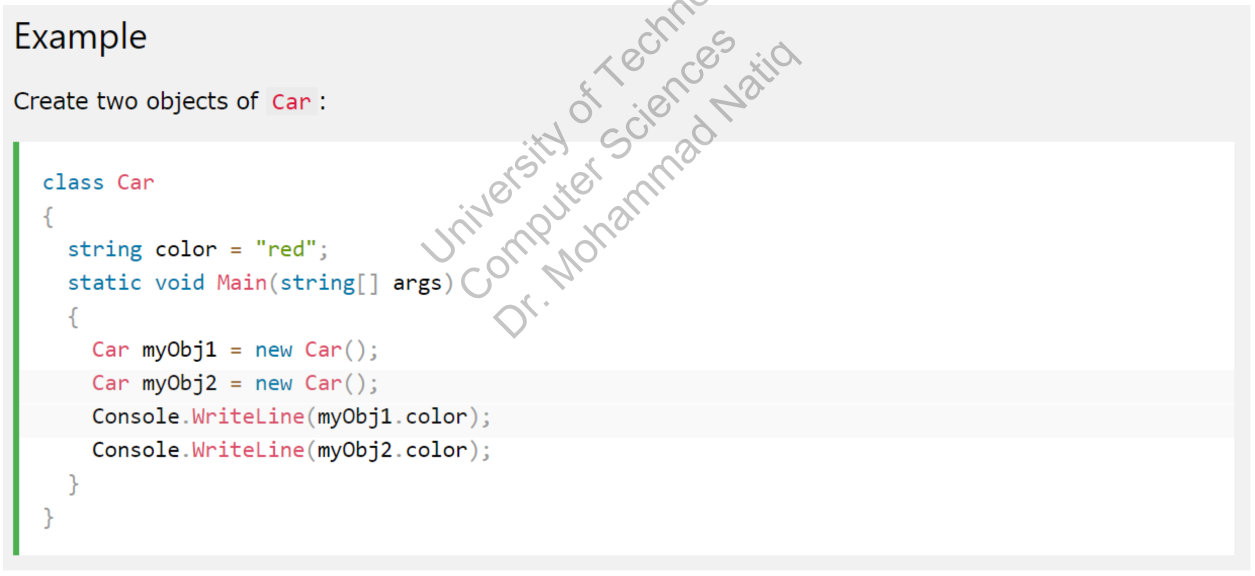
Create an object called "myObj" and use it to print the value of color:

```
class Car
{
  string color = "red";

  static void Main(string[] args)
  {
    Car myObj = new Car();
    Console.WriteLine(myObj.color);
  }
}
```

Multiple Objects

You can create multiple objects of one class:


Example

Create two objects of Car:

```
class Car
{
  string color = "red";
  static void Main(string[] args)
  {
    Car myObj1 = new Car();
    Car myObj2 = new Car();
    Console.WriteLine(myObj1.color);
    Console.WriteLine(myObj2.color);
  }
}
```

Using Multiple Classes

You can also create an object of a class and access it in another class. This is often used for better organization of classes (one class has all the fields and methods, while the other class holds the Main() method (code to be executed)).

prog2.cs

prog.cs

prog2.cs

```
class Car
{
  public string color = "red";
}
```

prog.cs

```
class Program
{
  static void Main(string[] args)
  {
    Car myObj = new Car();
    Console.WriteLine(myObj.color);
  }
}
```

Class Members

Fields and methods inside classes are often referred to as "Class Members":

Example

Create a `Car` class with three class members: **two fields** and **one method**.

```
// The class
class MyClass
{
  // Class members
  string color = "red";        // field
  int maxSpeed = 200;          // field
  public void fullThrottle()   // method
  {
    Console.WriteLine("The car is going as fast as it can!");
  }
}
```

Fields

In the previous chapter, you learned that variables inside a class are called fields, and that you can access them by creating an object of the class, and by using the dot syntax (.).

The following example will create an object of the Car class, with the name myObj. Then we print the value of the fields color and maxSpeed:

Example

```
class Car
{
  string color = "red";
  int maxSpeed = 200;

  static void Main(string[] args)
  {
    Car myObj = new Car();
    Console.WriteLine(myObj.color);
    Console.WriteLine(myObj.maxSpeed);
  }
}
```

You can also leave the fields blank, and modify them when creating the object:

Example

```
class Car
{
  string color;
  int maxSpeed;

  static void Main(string[] args)
  {
    Car myObj = new Car();
    myObj.color = "red";
    myObj.maxSpeed = 200;
    Console.WriteLine(myObj.color);
    Console.WriteLine(myObj.maxSpeed);
  }
}
```

Object Methods

Methods normally belongs to a class, and they define how an object of a class behaves.

Just like with fields, you can access methods with the dot syntax. However, note that the method must be `public`. And remember that we use the name of the method followed by two parantheses `()` and a semicolon `;` to call (execute) the method:

Example

```
class Car
{
  string color;                 // field
  int maxSpeed;                 // field
  public void fullThrottle()    // method
  {
    Console.WriteLine("The car is going as fast as it can!");
  }

  static void Main(string[] args)
  {
    Car myObj = new Car();
    myObj.fullThrottle();  // Call the method
  }
}
```

Access Modifiers

By now, you are quite familiar with the public keyword that appears in many of our examples:

C# has the following access modifiers:

Modifier	Description
`public`	The code is accessible for all classes
`private`	The code is only accessible within the same class
`protected`	The code is accessible within the same class, or in a class that is inherited from that class. You will learn more about inheritance in a later chapter
`internal`	The code is only accessible within its own assembly, but not from another assembly. You will learn more about this in a later chapter

Private Modifier

If you declare a field with a private access modifier, it can only be accessed within the same class:

Example

```
class Car
{
  private string model;

  static void Main(string[] args)
  {
    Car Ford = new Car("Mustang");
    Console.WriteLine(Ford.model);
  }
}
```

If you try to access it outside the class, an error will occur:

Example

```
class Car
{
  private string model = "Mustang";
}

class Program
{
  static void Main(string[] args)
  {
    Car myObj = new Car();
    Console.WriteLine(myObj.model);
  }
}
```

Network Programming APIs

Working with Sockets

A socket is an object that represents a low-level access point to the IP stack. This socket can be open or closed or one of a set number of intermediate states. A socket can send and receive data down this connection. Data is generally sent in blocks of a few kilobytes at a time for efficiency; each of these blocks is called a packet.

All packets that travel on the Internet must use the Internet protocol. This means that the source IP address, destination address must be included in the packet. Most packets also contain a port number. A port is simply a number between 1 and 65,535 that is used to differentiate higher protocols, such as email or FTP (Table 3.1). Ports are important when it comes to programming your own network applications because no two applications can use the same port. It is recommended that experimental programs use port numbers above 1024.

Packets that contain port numbers come in two flavors: UDP and TCP/IP. UDP has lower latency than TCP/IP, especially on startup. Where data integrity is not of the utmost concern, UDP can prove easier to use than TCP, but it should never be used where data integrity is more important than performance; however, data sent via UDP can sometimes arrive in the wrong order and be effectively useless to the receiver. TCP/IP is more com-plex than UDP and has generally longer latencies, but it does guarantee that data does not become corrupted when traveling over the Internet. TCP is ideal for file transfer, where a corrupt file is more unacceptable than a slow download; however, it is unsuited to Internet radio, where the odd sound out of place is more acceptable than long gaps of silence.

Creating a simple “hello world” application

This program will send the words “hello world” over a network. It consists of two executables, one a client, the other a server. These two programs could be physically separated by thousands of kilometers, but as long as the IP addresses of both computers are known, the principle still works.

In this example, the data will be sent using UDP. This means that the words “hello world” will be bundled up with information that will be used by IP routers to ensure that the data can travel anywhere it wishes in the world. UDP data is not bundled with headers that track message integrity or security. Furthermore, the receiving end is not obliged to reply to the sender with acknowledgments as each packet arrives. The elimination of this requirement allows UDP data to travel with much lower latency than TCP. UDP is useful for small payload transfers, where all of the data to be sent can be contained within one network packet. If there is only one packet, the out-of-sequence problems associated with UDP do not apply; therefore, UDP is the underlying protocol behind DNS.

ASCIIEncoding Class

Encoding is the process of transforming a set of Unicode characters into a sequence of bytes. Decoding is the process of transforming a sequence of encoded bytes into a set of Unicode characters. *Some protocols require ASCII or a subset of ASCII. In these cases ASCII encoding is appropriate*.

The **GetByteCount** method determines how many bytes result in encoding a set of Unicode characters, and the **GetBytes** method performs the actual encoding.

Likewise, the **GetCharCount** method determines how many characters result in decoding a sequence of bytes, and the **GetChars** and **GetString** methods perform the actual decoding.

```
ASCIIEncoding ascii = new ASCIIEncoding();
String unicodeString ="This Unicode string contains two characters "
Byte[] encodedBytes = ascii.GetBytes(unicodeString);
foreach (Byte b in encodedBytes)
        {
          Console.Write("[{0}]", b);
        }
```

IPEndPoint Class

The IPEndPoint class contains the host and local or remote port information needed by an application to connect to a service on a host. By combining the host's IP address and port number of a service, the IPEndPoint class forms a connection point to a service.

public class IPEndPoint : System.Net.EndPoint

PROPERTIES

Address Gets or sets the IP address of the endpoint.
AddressFamily Gets the Internet Protocol (IP) address family.
Port Gets or sets the port number of the endpoint.

METHODS

Create(SocketAddress) Creates an endpoint from a socket address.
Parse(String) Converts an IP network endpoint (address and port) represented as a string to an IPEndPoint instance.
ToString() Returns the IP address and port number of the specified endpoint.

Example

```
string serverIP = "192.168.0.1";
int port = 8000;
IPEndPoint remoteEP = new IPEndPoint(IPAddress.Parse(serverIP), port);
```

Writing a simple UDP client

To get started, open Visual Studio .NET, click New Project, then click Visual C# projects, and then Windows Application. Set the name to "UDP Client" and press OK.

Now, design the form as shown in Figure 3.1. Name the button button1 and the textbox tbHost.

```
private void button1_Click(object sender, System.EventArgs e)
{
  UdpClient udpClient = new UdpClient();
  udpClient.Connect(tbHost.Text, 8080);
  Byte[] sendBytes = Encoding.ASCII.GetBytes("Hello World?");
    udpClient.Send(sendBytes, sendBytes.Length);
}
```

You also need to include some assemblies by adding these lines to just

under the lock of the using statements at the top of the code:

```
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.IO;
```

Writing a simple UDP server

The purpose of the UDP server is to detect incoming data sent from the UDP client.

Table: Significant members of the UdpClient class.

Method or Property	Purpose
`Constructor`	Initializes a new instance of the `UdpClient` class. For client UDP applications, this is used as `new UdpClient (string,int);` for servers use `new UdpClient(int).`
`Close()`	Closes the UDP connection.
`DropMulticastGroup()`	Leaves a multicast group.
`JoinMulticastGroup()`	Adds a `UdpClient` to a multicast group. This may be invoked thus: `JoinMulticastGroup(IPAddress).`
`Receive()`	Returns a UDP datagram that was sent by a remote host. This may be invoked thus: `Receive(ref IPEndPoint).` Returns `Byte[].`
`Send()`	Sends a UDP datagram to a remote host. This may be invoked thus `Send(byte[], int).`
`Active`	Gets or sets a value indicating whether a connection to a remote host has been made. Returns `Bool`
`Client`	Gets or sets the underlying network sockets. Returns `Socket.`

As before, create a new C# project, but with a new user interface, as shown below. The list box should be named lbConnections.

A key feature of servers is multithreading (i.e., they can handle hundreds of simultaneous requests). In this case, our server must have at least two threads: one handles incoming UDP data, and the main thread of execution may continue to maintain the user interface, so that it does not appear hung.

```
public void serverThread()
{
 UdpClient udpClient = new UdpClient(8080);
 while(true)
 {
  IPEndPoint RemoteIpEndPoint = new IPEndPoint(IPAddress.Any,
    0);
  Byte[] receiveBytes = udpClient.Receive(ref
RemoteIpEndPoint);
  string returnData = Encoding.ASCII.GetString(receiveBytes);
  lbConnections.Items.Add(
   RemoteIpEndPoint.Address.ToString() + ":" +
returnData.ToString()
  );
 }
```

Again, we use the UdpClient object. Its constructor indicates that it should be bound to port 8080, like in the client. The Receive method is blocking (i.e., the thread does not continue until UDP data is received). In a real-world application, suitable timeout mechanisms should be in place because UDP does not guarantee packet delivery. Once received, the data is in byte array format, which is then converted to a string and displayed on-screen in the form source address: data.

There is then the matter of actually invoking the serverThread method asynchronously, such that the blocking method, Receive, does not hang the application. This is solved using threads as follows:

```
 private void Form1_Load(object sender, System.EventArgs e)
 {
  Thread thdUDPServer = new Thread(new
 ThreadStart(serverThread));
  thdUDPServer.Start();
 }
```

To finish off, the following assemblies are to be added:

```
using System.Threading;
using System.Net;
using System.Net.Sockets;
using System.Text;
```

University of Technology
Computer Sciences
Dr. Mohammad Natiq

System.IO

(StreamReader and StreamWriter)

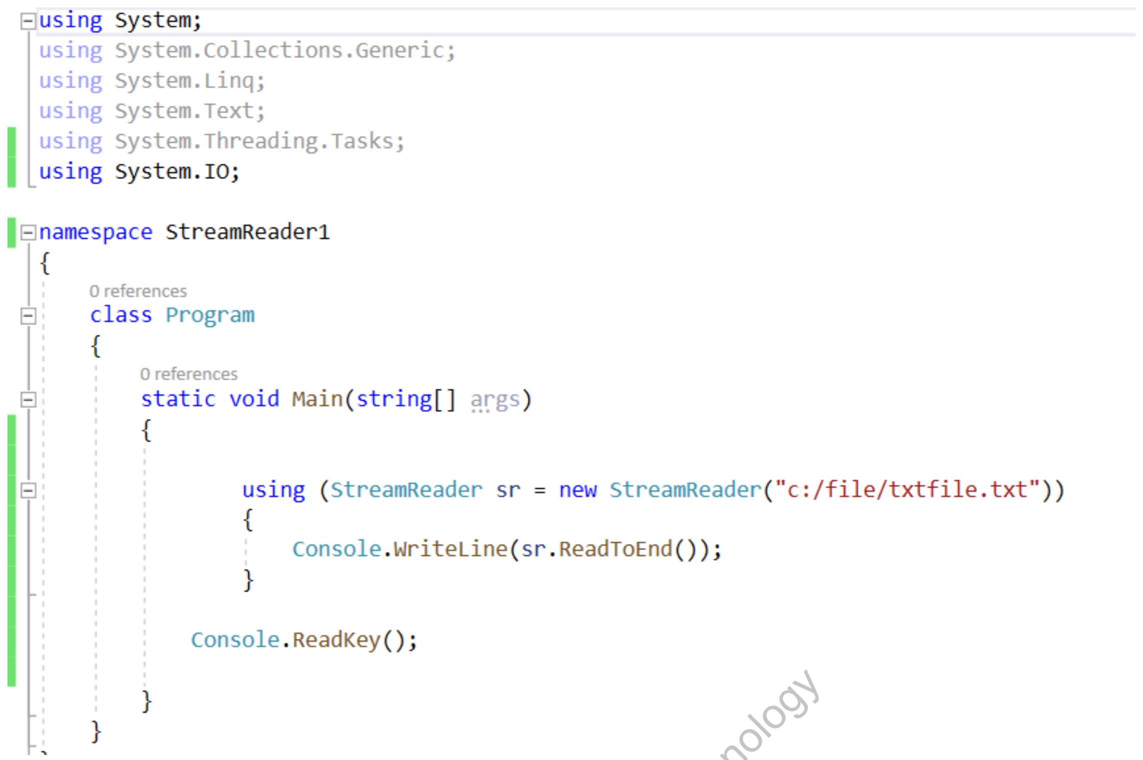
C# StreamReader Class

StreamReader is used to read characters to a stream in a specified encoding. StreamReader.Read method reads the next character or next set of characters from the input stream. StreamReader is inherited from TextReader that provides methods to read a character, block, line, or all content.

StreamReader is defined in the System.IO namespace. StreamReader provides the following methods:

1. Peak – Returns if there is a character or not.
2. Read - Reads the next character or next set of characters from the input stream.
3. ReadBlock - Reads a specified maximum number of characters from the current stream and writes the data to a buffer, beginning at the specified index.
4. ReadLine - Reads a line of characters from the current stream and returns the data as a string.
5. ReadToEnd - Reads all characters from the current position to the end of the stream.

The following example uses an instance of StreamReader to read text from a file.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.IO;

namespace StreamReader1
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {

                using (StreamReader sr = new StreamReader("c:/file/txtfile.txt"))
                {
                    Console.WriteLine(sr.ReadToEnd());
                }

            Console.ReadKey();

        }
    }
```

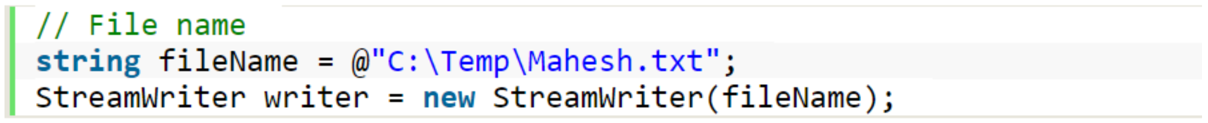
C# StreamWriter Class

StreamWriter class in C# writes characters to a stream in a specified encoding. StreamWriter.Write() method is responsible for writing text to a stream.

StreamWriter is defined in the System.IO namespace. StreamWriter provides the following Write methods:

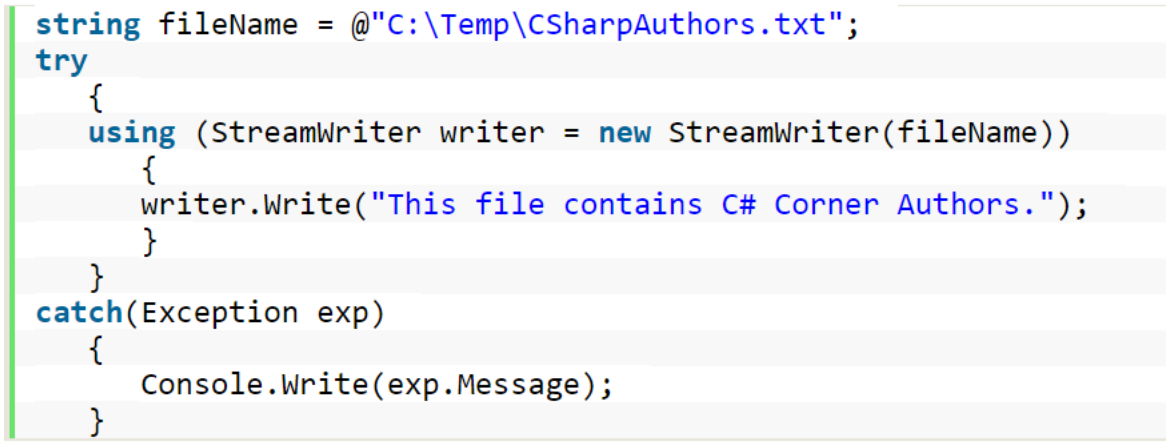
1. Write – Writes data to the stream.
2. WriteLine – Writes a line terminator to the text string or stream.
3. Creating a StreamWriter using a Filename

The following code snippet creates a StreamWriter from a filename with default encoding and buffer size.



```
// File name
string fileName = @"C:\Temp\Mahesh.txt";
StreamWriter writer = new StreamWriter(fileName);
```

The following code snippet creates a StreamWriter and adds some text to the writer using StreamWriter.Write method. If the file does not exist, the writer will create a new file. If the file already exists, the writer will override its content.



```
string fileName = @"C:\Temp\CSharpAuthors.txt";
try
    {
    using (StreamWriter writer = new StreamWriter(fileName))
        {
        writer.Write("This file contains C# Corner Authors.");
        }
    }
catch(Exception exp)
    {
        Console.Write(exp.Message);
    }
```

A good practice is to use these objects in a using statement so that the unmanaged resources are correctly disposed. The using statement automatically calls Dispose on the object when the code that is using it has completed.

C# using statement

C# and .NET provide resource management for managed objects through the garbage collector - You do not have to explicitly allocate and release memory for managed objects. Clean-up operations for any unmanaged resources should be performed in the destructor in C#.

To allow the programmer to explicitly perform these clean-up activities, objects can provide a Dispose method that can be invoked when the object is no longer needed. The C# using statement defines a boundary for the object outside of which, the object is automatically destroyed. The using statement in C# is exited when the end of the "using" statement block or the execution exits the "using" statement block indirectly, for example- an exception is thrown.

The "using" statement allows you to specify multiple resources in a single statement. The object could also be created outside the "using" statement. The objects specified within the using block must implement the IDisposable interface. The framework invokes the Dispose method of objects specified within the "using" statement when the block is exited.

StreamWriter.Write() method

StreamWriter.Write() method writes a char, string of chars, or a string to the steam. The following code snippet creates and writes different content to the stream

```
using (StreamWriter writer = new StreamWriter(stream, Encoding.UTF8 ))
{
    // Write a char
    writer.Write('y');
    // Write a char[]
    char[] arr = { '1', '3', '5', '7' };
    writer.Write(arr);
    string author = "Mahesh Chand";
    // Write a string
    writer.Write(author);
}
```

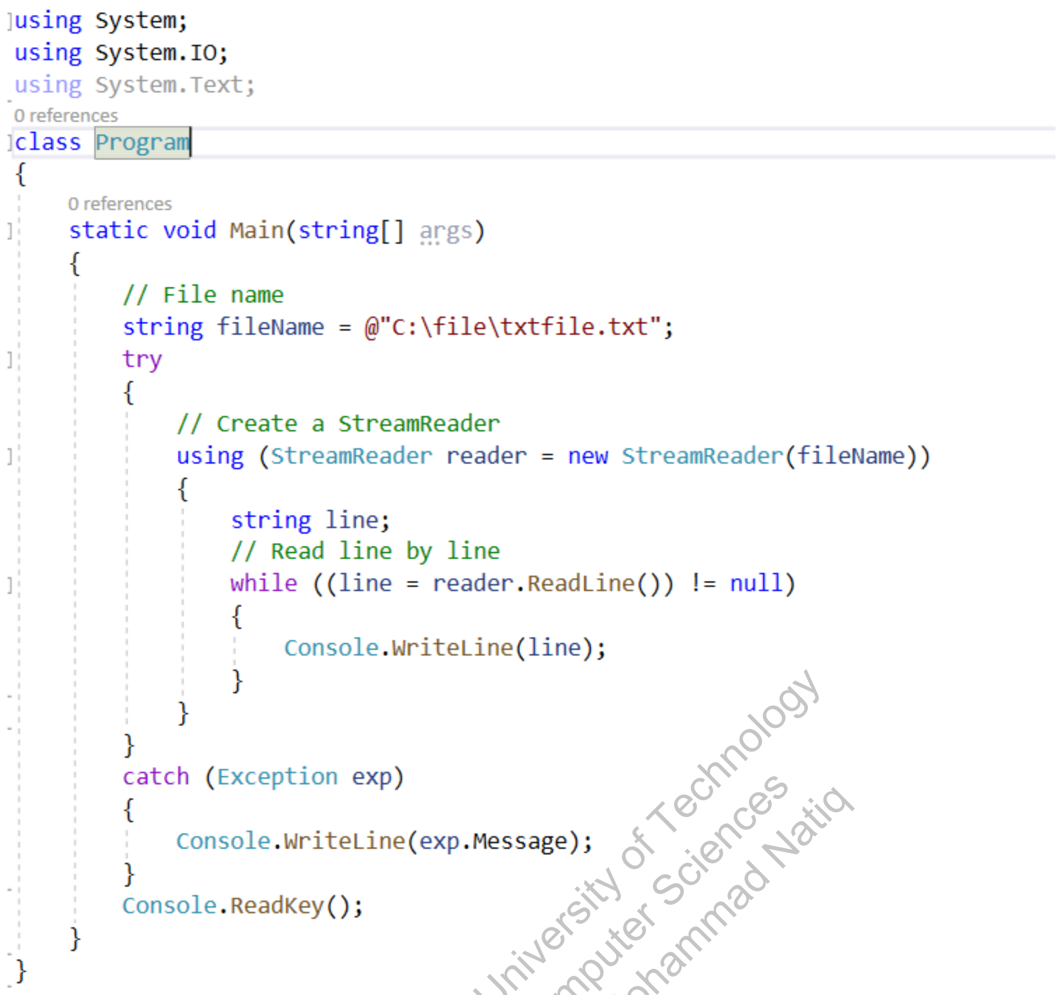
File.AppendText(String) Method

Creates a StreamWriter that appends UTF-8 encoded text to an existing file, or to a new file if the specified file does not exist.

The following example appends text to a file.

```
public void AppendTxt(string txt)
{
    using (StreamWriter sp = File.AppendText(path))
    {
        sp.WriteLine(txt);
    }
}
```

The following code example creates a StreamReader and reads a file content one line at a time and displays to the console. If there is an exception, the exception is displayed on the console.



```
using System;
using System.IO;
using System.Text;
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        // File name
        string fileName = @"C:\file\txtfile.txt";
        try
        {
            // Create a StreamReader
            using (StreamReader reader = new StreamReader(fileName))
            {
                string line;
                // Read line by line
                while ((line = reader.ReadLine()) != null)
                {
                    Console.WriteLine(line);
                }
            }
        }
        catch (Exception exp)
        {
            Console.WriteLine(exp.Message);
        }
        Console.ReadKey();
    }
}
```