

From <http://www.onlineprogramminglessons.com>

These C# mini lessons will teach you all the C# Programming statements you need to know, so you can write 90% of any C# Program.

- Lesson 1 Input and Output**
- Lesson 2 Methods**
- Lesson 3 Classes and Inheritance**
- Lesson 4 Operators**
- Lesson 5 Programming Statements**
- Lesson 6 Arrays, Lists, HashSets and Dictionaries**
- Lesson 7 Overloading, Overriding and Interfaces**
- Lesson 8 Generic Interfaces and Classes**
- Lesson 9 Overloaded Operators**
- Lesson 10 File Access**
- Lesson 11 Polymorphism and C# Objects**
- Lesson 12 Recursion**
- Lesson 13 Delegates and Lambda Expressions**
- Lesson 14 LINQ**
- Lesson 15 Expressions and Expression Trees**
- Lesson 16 Events,Asynchronous methods and Threading**
- Lesson 17 Project**

Conventions used in these lessons:

bold - headings, keywords, code

italics - code syntax

underline - important words

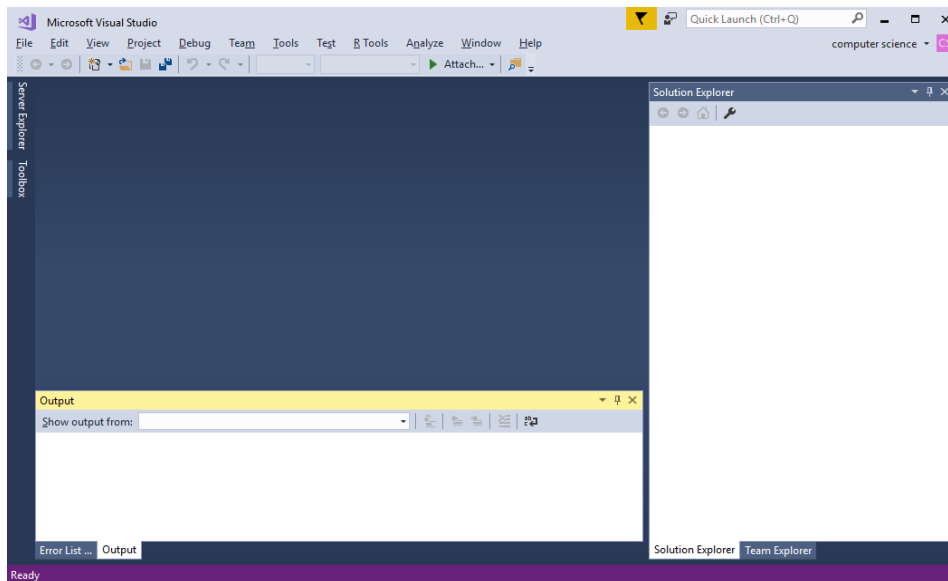
- (open round bracket
-) close round bracket
- { open curly bracket
- } close curly bracket
- [open square bracket
-] close square bracket

Let's get started!

You first need to download Visual Studio to edit, compile and run C# programs. Download the community version, its free.

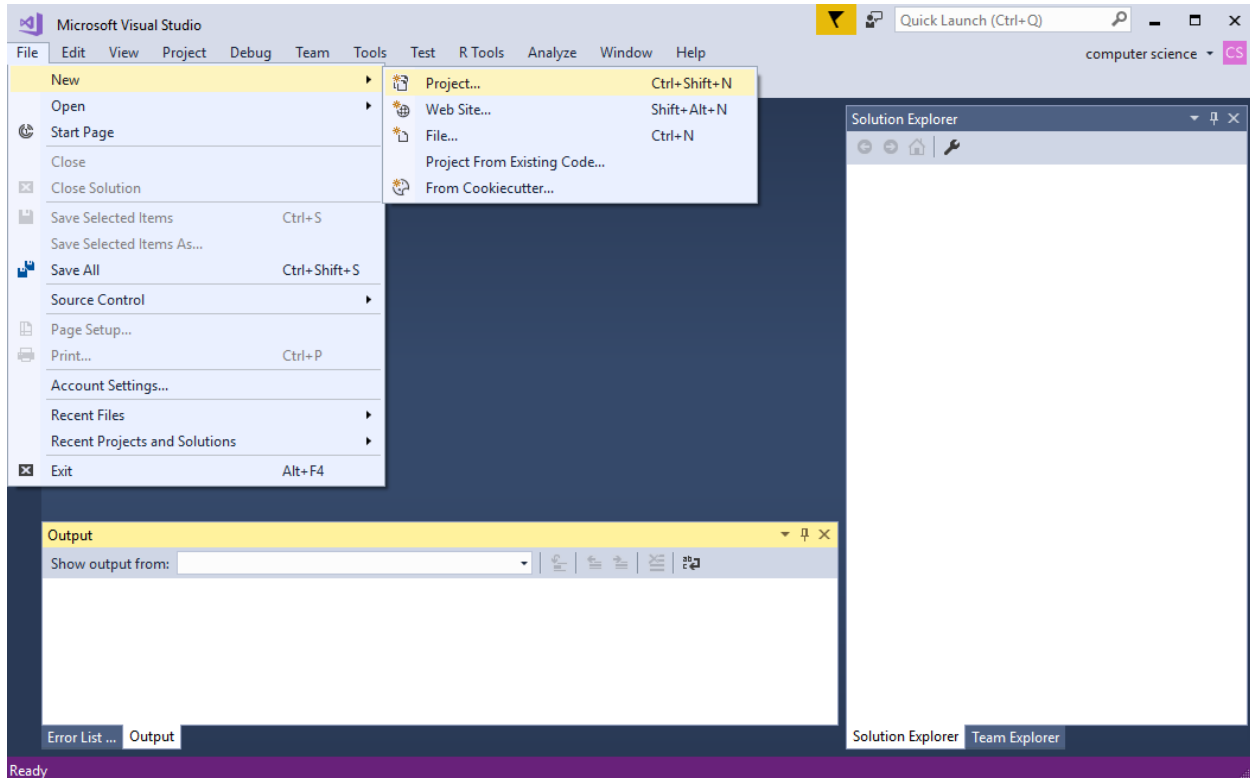
<https://www.visualstudio.com/thank-you-downloading-visual-studio/?sku=Community&rel=15&page=defaultinstall>

Once you installed and run visual studio you will get this screen.

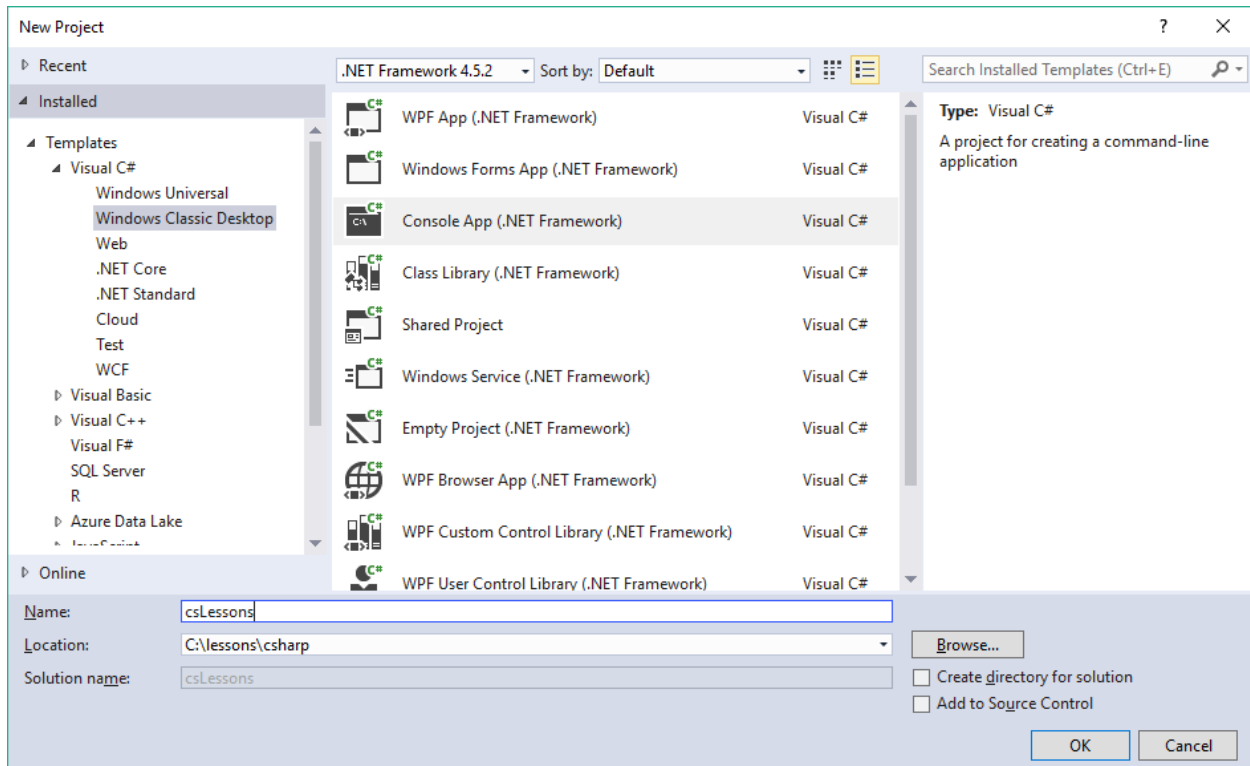


Lesson 1 Input and Output

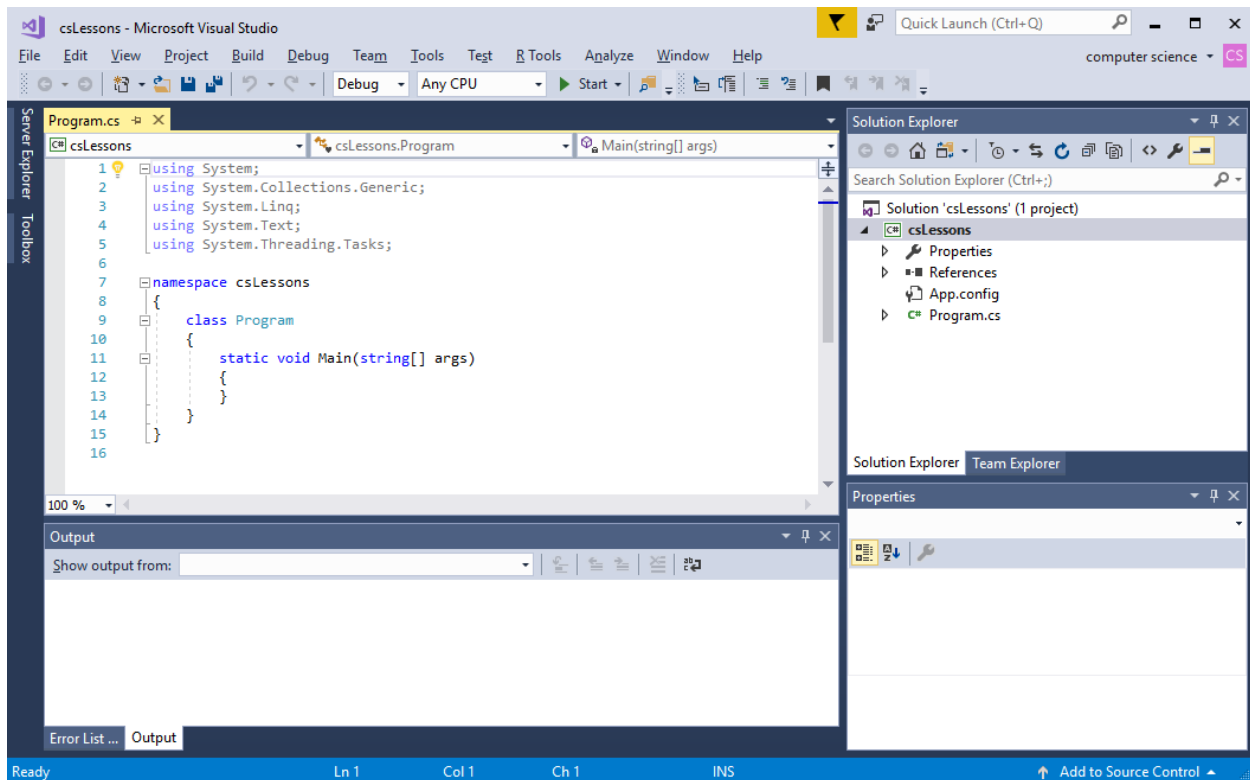
You first need to make a Project to store all your C# lesson files..
From the File menu select New Project



Select the template C# Windows Classic Desktop Console App (Net Framework)
We will use the project name csLessons. You need to change the location of the project to the location of where your C# lesson files will be stored on your computer. Using Windows File Explorer, make an additional folder called cSharp that will store your csLessons project files. When you make a project in visula studio a Solution is automatically made. Projects are contained in Solutions. In this case the Project name and Solution name are the same.

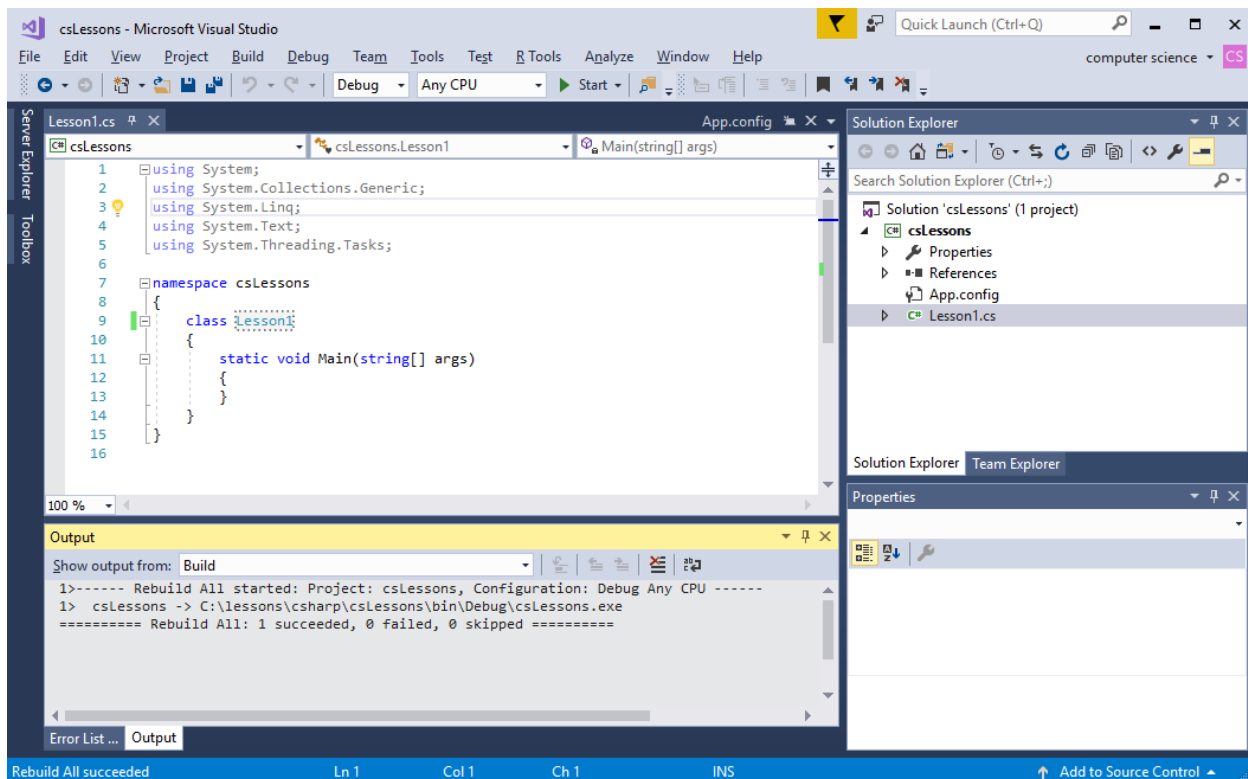


Press OK then select. You will get something like this:



A default program file called Program.cs is created. You will have to rename this file to Lesson1.cs

Right click on Program.cs in the Solution Window and rename to Lesson1.cs. You will also need to change the class name from Program to Lesson1. You should now have something like this.

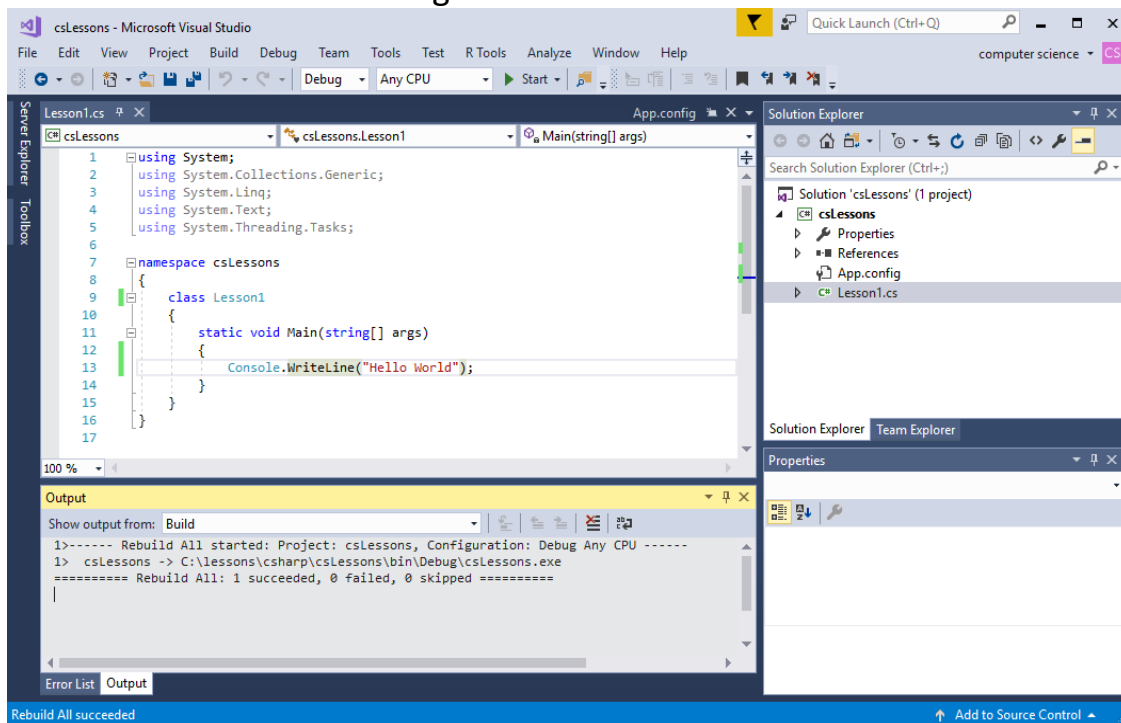


All C# programs are contained in a C# Class. A class contains variables that store information and methods that do operations on these variables. This is a big step in programming evolution as well as for someone just learning programming. You will also notice that the class Lesson2 is contained in the name space csLessons which happens to be the same name as the Project we created.

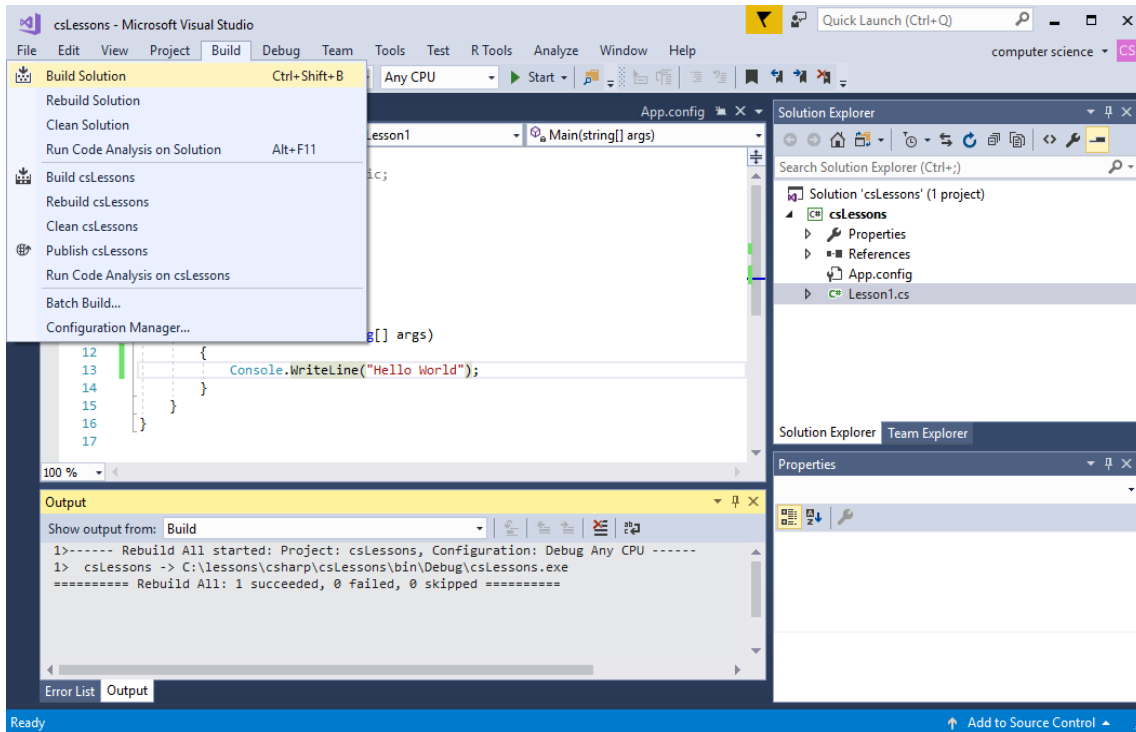
It's now time to write your first C# Program. You will print Hello World on the screen. In the C# Editor inside the **static void Main(string[] args)** statement type

Console.WriteLine ("Hello World");

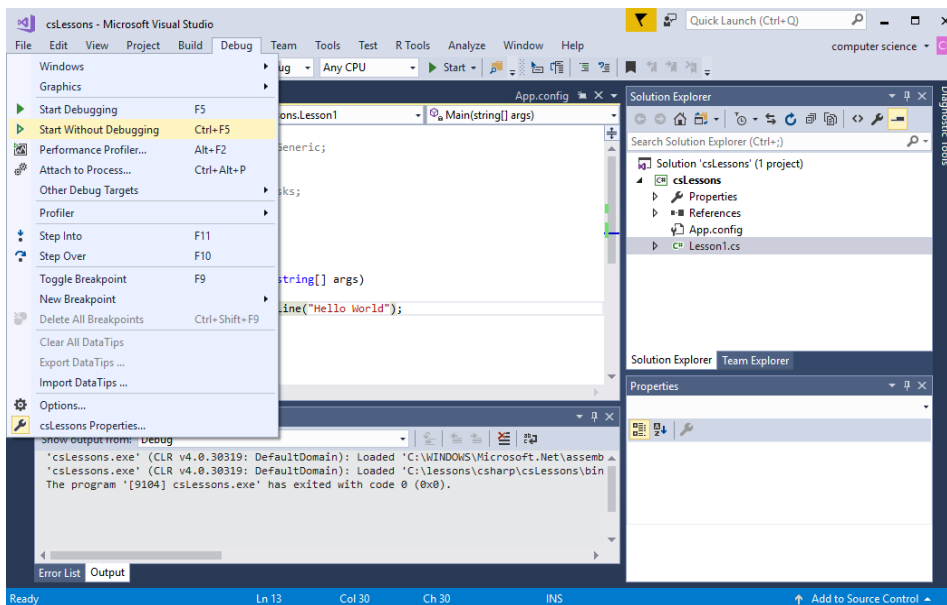
You will now have something like this:

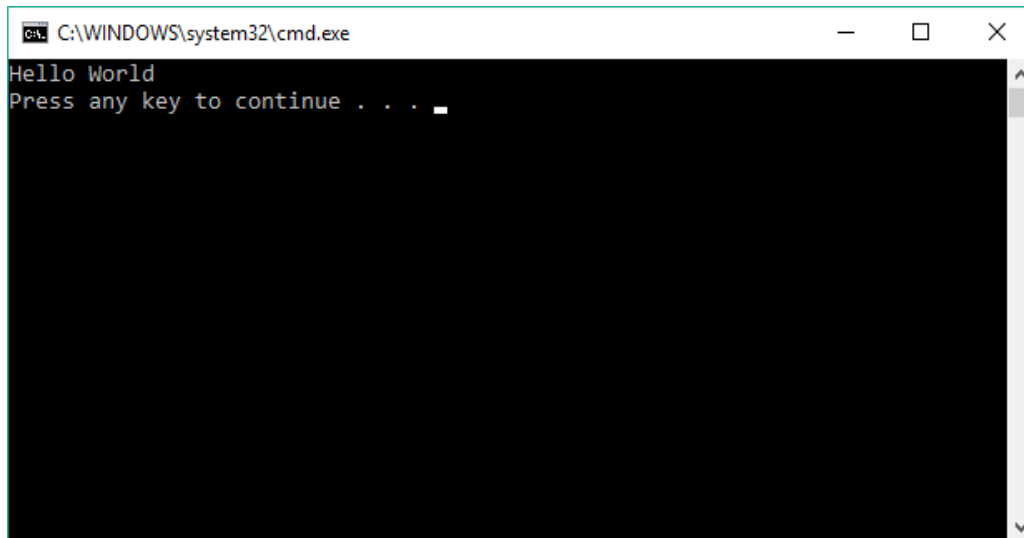


Next you need to build your C# Lesson1.cs file so you can check for any errors, before you can run the program file. From the Build menu select Build Solution.



If you do get some errors, then you must correct them and then rebuild the file. Errors are typing mistakes that can be easily corrected but may be difficult to find. Errors are always the unexpected and the overlooked. If you do not have any errors, then you can run your file. From the Project menu select Start Without Debugging Hello World is now printed in the Build Output window.





Recapping: A C# program is enclosed in a namespace. The namespace is usually the project name. Inside the namespace are the classes. The class contains variables that contain values and methods that contain programming statements.

The Methods contain programming statements that tell the computer what to do. In side the Lesson1 class we a have a main method. The main method is executed when the program first runs.

```
namespace csLessons
{
    class Lesson1
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World");
        }
    }
}
```

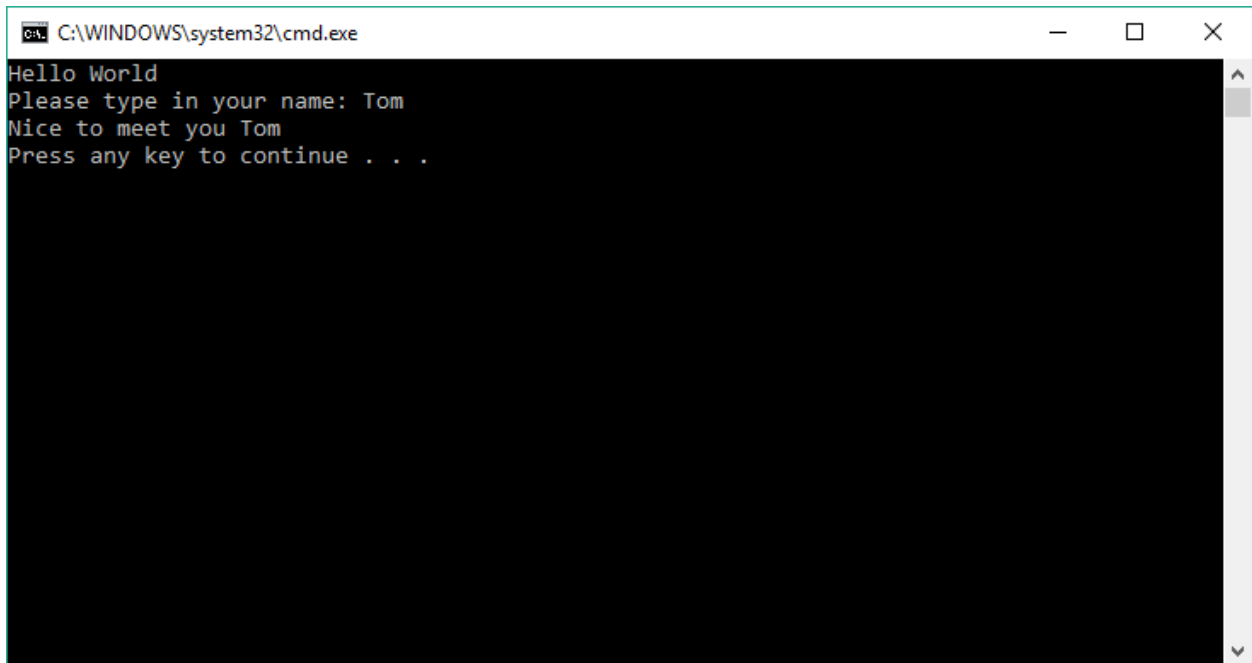
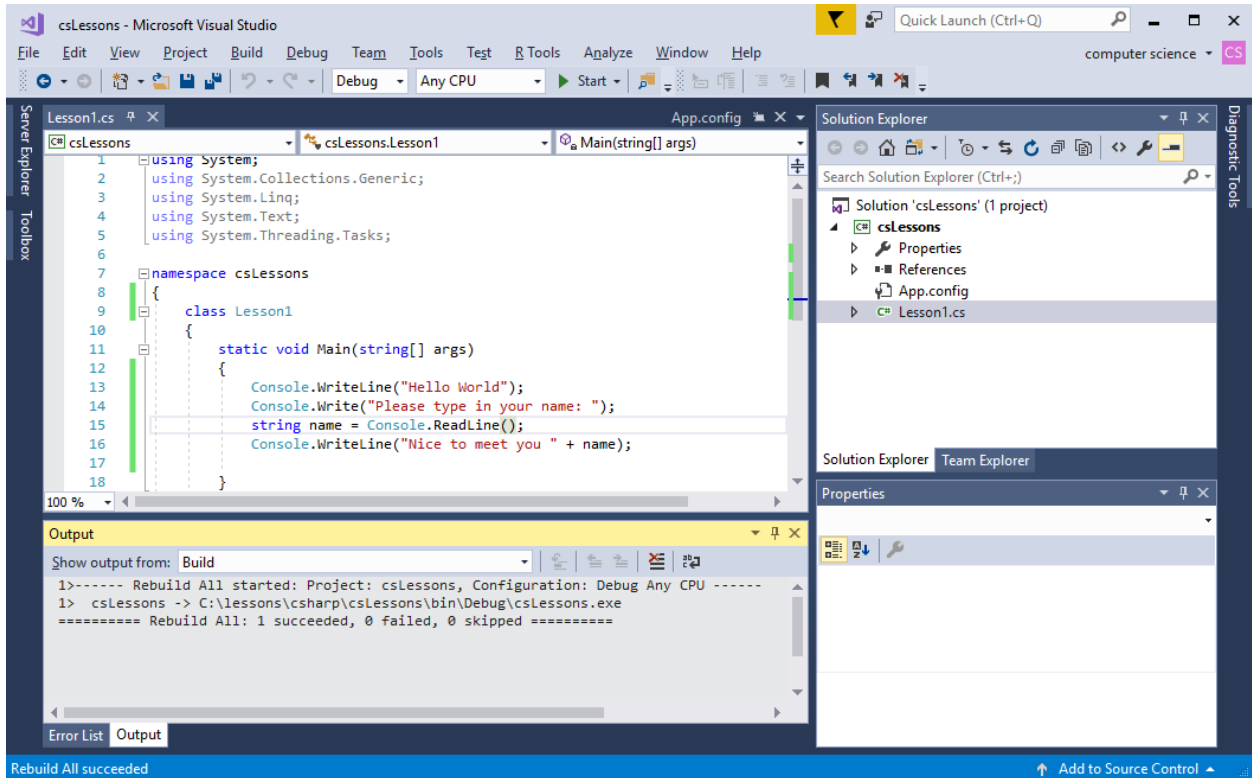
The main method contains the **Console.WriteLine** statement that prints the string message “Hello World” on the screen. Before we proceed it is important to understand the terminology, classes, methods, programming statements and objects.

data type	What type of data is to be represented
Variable	Stores a string or numeric value. All variables have a data type
programming statement	is an instruction containing commands to perform a desired action, like printing a value on the screen, get a value from the key board or calculate a certain value.
Method	contains programming statements that tell the computer what to do and performs operations on variables using these programming statements
Class	defines values and methods
Namespace	Usually the project name, that contains all the classes
Object	Computer memory allocated for a variables defined in a class when a program runs

The next thing we need to do is get values from the keyboard. We will ask the user to type in their name and then greet them. Type in the following statements in the C# editor right after the Hello World statement.

```
Console.Write("Please type in your name: ");
string name = Console.ReadLine();
Console.WriteLine("Nice to meet you " + name);
```

Now run your program, and enter the name "Tom". You will get something like this:



Recapping:

We first ask the user to type in their name using the **Console.Write** statement.

```
Console.Write ("Please type in your name: ");
```

Then we obtain the user name from the keyboard using the **ReadLine** method from the Console class

```
string name = Console.ReadLine();
```

The entered name is placed in the string variable name. The **Console.WriteLine** statement prints out the string message "Nice to meet you" and the name of the user stored in the variable name.

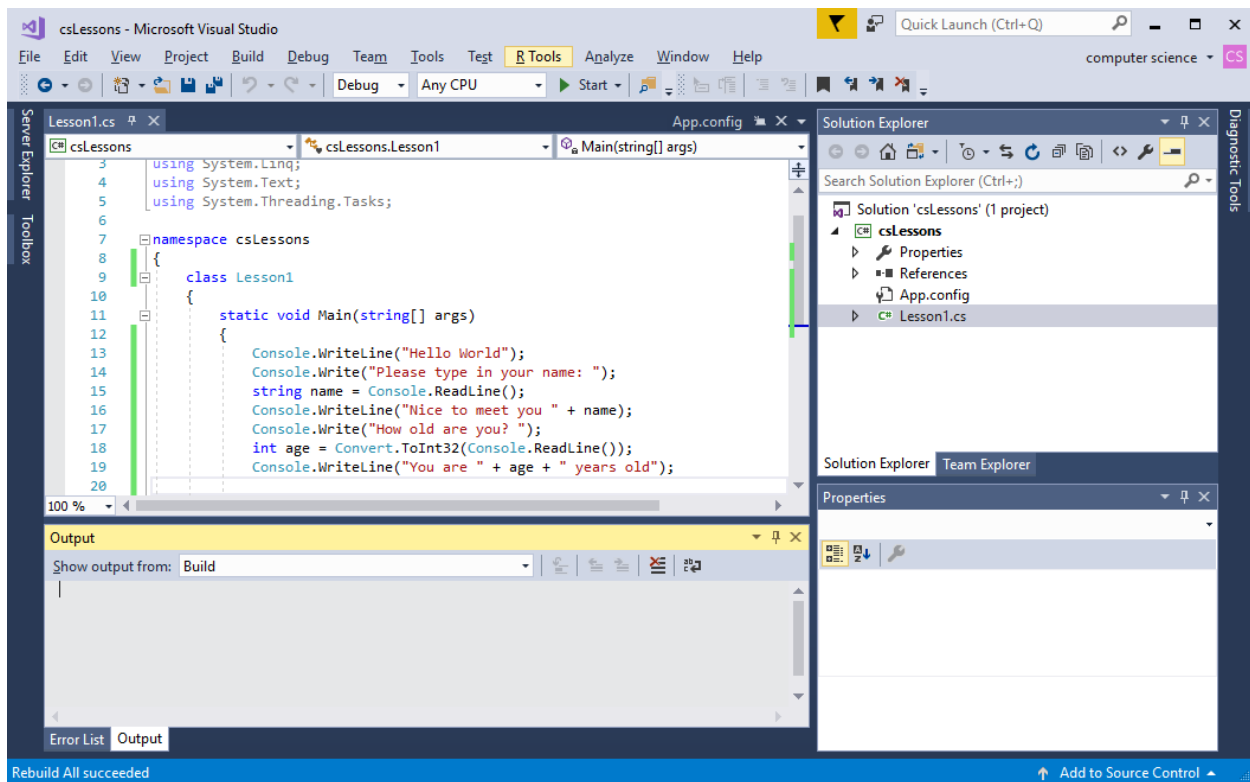
```
Console.WriteLine ("Nice to meet you " + name)
```

Note inside the **Console.WriteLine** statement the string message and variable name are joined by a '+'.

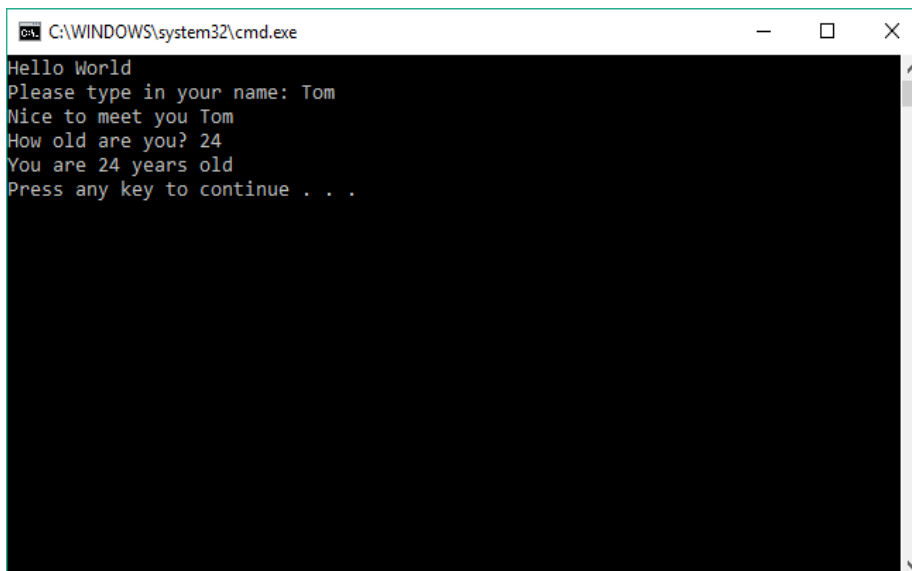
C# has two types of values **string** values and **numeric** values. String values are messages enclosed in double quotes like "Hello World" where as numeric values are numbers like 5 and 10.5 Numeric values without decimal points like 5 are known as an **int** and numbers with decimal points like 10.5 are known as a **float** or **double**. Variable's store string or numeric values that can be used later in your program.

We now continue our program to ask the user how old they are. Type in the following statements at the end of your program and then run the program.

```
Console.Write("How old are you? ");  
int age = Convert.ToInt32(Console.ReadLine());  
Console.WriteLine("You are " + age + " years old");
```



Run the program and enter Tom for name and 24 for age, you should get something like this.



Recapping:

The **Console.Write("How old are you? ")** statement asks the user to enter their age. The **int age = Convert.ToInt32(Console.ReadLine())** statement receives a string number from the keyboard using **Console.ReadLine()** and converts the string to a numeric int using **Convert.ToInt()** and then assigns the numeric int to the variable age. The **Console.WriteLine ("You are " + age + " years old");** statement is used to print out the message, the persons name and age. Again the '+' operator is used to join the age numeric value to the string messages.

If you have got this far then you will be a great C# programmer soon.

Most people find Programming difficult to learn. The secret of learning program is to figure out what you need to do and then choose the right program statement to use.

If you want to print messages and values to the screen you use a **Console.WriteLine** statement.

If you want to get values from the user, you use an **Console.ReadLine()** statement.

If you need a numeric value, you use an **Convert.ToInt()** statement to convert String numbers to int numbers. You can also use **Convert.ToFloat()** statement to convert String numbers to decimal float numbers or the **Convert.ToDouble()** statement to convert string numbers to decimal double numbers.

The difference between float decimal numbers and double decimal numbers is just accuracy.

You should concentrate on getting your programs running rather than understand how they work. Once you get your programs running and you execute them understanding will be come much easier. Understanding is much easier now because you can now make an association connection to the program statement that is running that produces the desired input or output action.

C# Data Types

C# has many data types that can be used to represent different kinds of numbers as follows:

Data Type	Size	Min value	Max Value	Example
byte	8	-128	127	byte x = 100;
short	16	-32768	32767	short x = 1000;
int	32	-2 ³¹	2 ³¹ -1	int x = 10000;
long	64	-2 ⁶³	2 ⁶³ -1	long x = 10000;
float	32	-1.4E-45	3.4028235E38	float f = 10.5;
double	64	-4.9E-324	4.9E-324	double d = 10.5;
decimal	128	-7.9E28	7.9E28	Decimal d = 10.5m;
bool	1	False	True	bool x = true;
char (unicode)	16	'\u0000' (0)	'\uffff' (65535)	char x = 'A';

Note: C# also has unsigned data types `ubyte`, `ushort`, `uint` and `ulong`.

Data Type	Size	Min value	Max Value	Example
ubyte	8	0	256	ubyte x = 100;
ushort	16	0	65535	ushort x = 1000;
uint	32	0	2 ³² -1	uint x = 10000;
ulong	64	0	2 ⁶⁴ -1	ulong x = 10000;

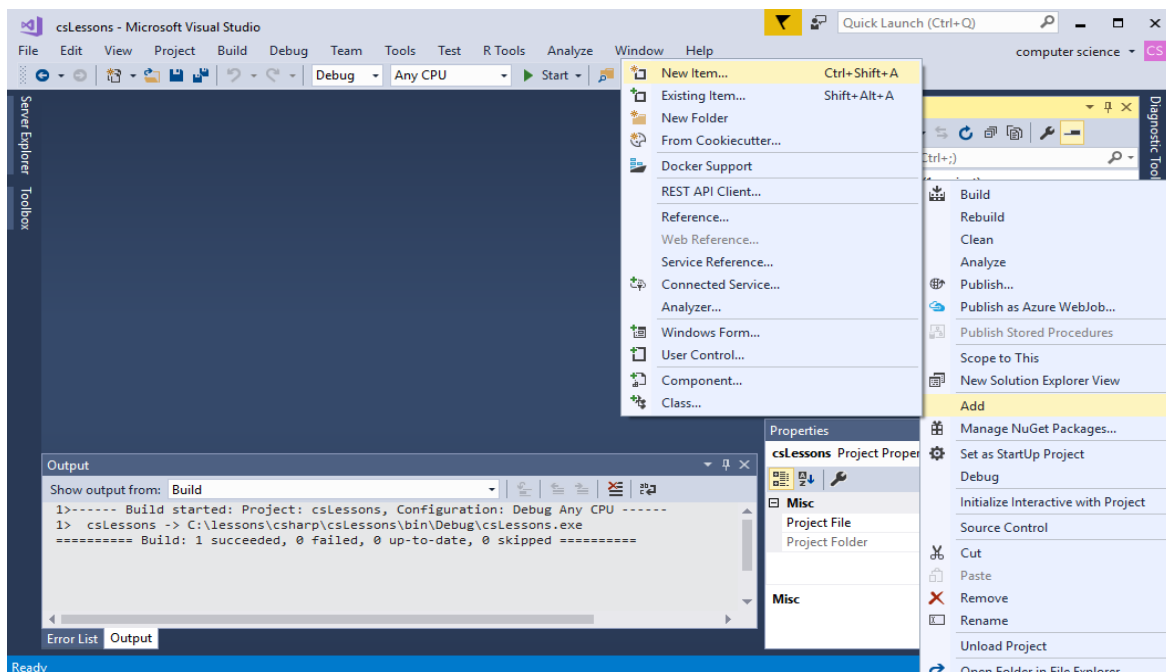
Lesson 1 Homework

Write a C# program that asks someone what their profession **title** is and what their **salary** is. Their title could be a doctor, lawyer etc. Next print out the details to the screen, what their title is and how much money they make. For the profession **title** use a String variable. For the **salary** variable you can use **float** or **double** data type. You will need to use the **float.Parse** statement to convert a String number to float number or the **double.Parse** statement to convert String numbers to double number. Call your java program Homework1.cs and class Homework1.

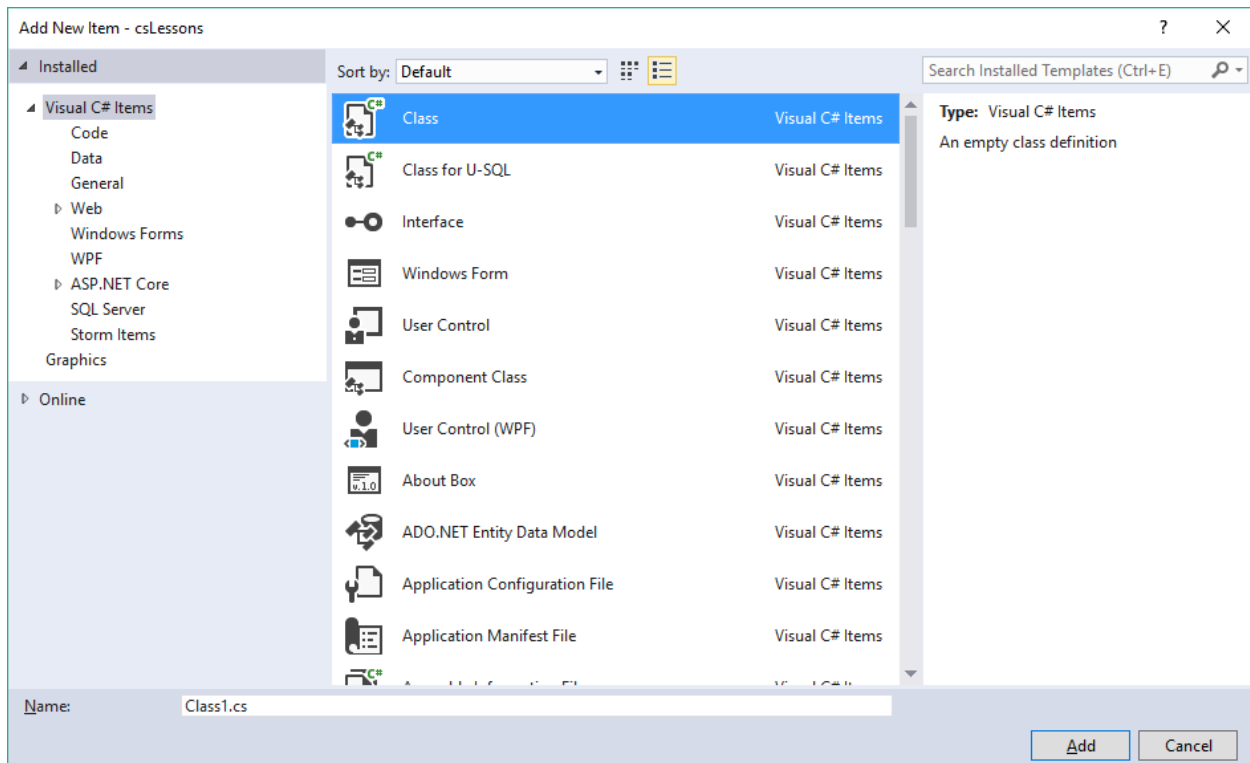
Lesson 2 **Methods**

Methods allow you to group many programming statements together so that you can reuse them repeatedly in your C# Program. Methods are analogous to functions in other programming languages. They are called methods because they are contained in a class and cannot be used by themselves. The most common method is the Main method that starts a C# program, which we used previously in Lesson 1. A class may have many methods. Each method has a dedicated purpose, some action to perform. Methods usually are defined at the top of the program in order as they are used. The main function is the last one because it will call all the preceding methods. When a method is called in a programming statement it means it is executed. C# also has many built in methods that you can use, that make C# programming easier to do. You already used some of them in Lesson 1, **Write, WriteLine, ReadLine**. As we proceed with these lessons you will learn and use many more methods. It is now time to add more methods to our previous Lesson 1 program. We will make a welcome(), getName() and getAge() methods. Make a new C# file called Lesson2.cs

Right click on csLesson project in Solution Explorer window and select Add then New Item in the sub menu.



Then select Class make sure it says Lesson2.cs at the bottom for the class name.



and then type in the following code.

```
public class Lesson2  
{  
  
    public static void welcome()  
    {  
        Console.WriteLine("Hello World");  
    }  
  
    public static String enterName()  
    {  
        Console.Write("Please type in your name: ");  
        string name = Console.ReadLine();  
        return name;  
    }
```



```

public static int enterAge()
{
    Console.Write("How old are you? ");
    int age = Convert.ToInt32(Console.ReadLine());
    return age;
}

public static void printPerson(String name, int age)
{
    Console.WriteLine("Nice to meet you " + name);
    Console.WriteLine(name + " You are " + age + " years old");
}

public static void Main(string[] args)
{
    welcome();
    String name = enterName();
    int age = enterAge();
    printPerson(name, age);
}
}

```

You will need to exclude the Lesson1.cs file in the Solution Explorer window. Right click on Lesson1.cs in the Solution Explorer window and select exclude from Project. Then build and run Lesson2.cs. You will get the same results as in Lesson1.

Methods make your program more organized and manageable to use. Methods have three different purposes. Methods can receive values, return values, receive and return values or receive or return nothing.

The syntax for a method is as follows:

*access_modifier non-access_modifier return_datatype method_name
(parameter_list)*

Parameter list = data_type parameter_name [,data_type parameter_name]

Access modifiers allow who can access the method. In this lesson we were introduced to the **public** and **private** access modifiers. Non-access modifiers indicate how the method can be used. In this lesson we have used the **static** non-access modifier. Static refers to the code that is loaded when a program runs, and available to be used right away. Methods return values using the return statement and receive values through the parameter list. The data type specifies what kind of data is returned or received. In Lesson 1 we were introduced to the int, float, double and string data types.

The welcome() method just prints a statement and receives no values or returns no value. The void data type indicates no value is returned or received.

```
public static void welcome()  
    {  
        Console.WriteLine("Hello World");  
    }
```

The get Name() and getAge() methods both return a value using the return statement. The getName() method returns a string value where as the getAge method returns an int value.

```
public static String getName()  
    {  
        Console.Write("Please type in your name: ");  
        string name = Console.ReadLine();  
        return name;  
    }
```

```
public static int getAge()  
    {  
        Console.Write("How old are you? ");  
        int age = Convert.ToInt32(Console.ReadLine());  
        return age;  
    }
```

The `printDetails` function receive a name and age value to print out, but return's no value. The `printDetails` Method receives the name and age through the parameter list.

```
public static void printDetails(String name, int age)
{
    Console.WriteLine("Nice to meet you " + name);
    Console.WriteLine(name + " You are " + age + " years old");
}
```

The **name** and **age** inside the round brackets of the **printDetails** method definition statement are known as **parameters** and contain values to be used by the method. The parameters just store values from the calling function and are not the same variables that are in the calling function. Although the parameter names and values may be same as in the calling function variable names, but they are at different memory locations. The main purpose of the parameters is to receive values for the methods. The main method call's the preceding methods to run them and store the values in variables and pass the stored variable values to the methods. Calling a method means to execute the method. The values that are passed to the called method from the calling method is known as **arguments**.

Variables inside a method are known as **local variables** and are known to that method only. Name and age are local variables in the main function but are also arguments to the `printDetails` method.

```
public static void Main(string[] args)
{
    welcome();
    String name = getName();
    int age = getAge();
    printDetails(name, age);
}
```

The main purpose of our sample program is to gather and print out information therefore all methods should be static. Static means the compiler will generate the code for our class and make it readily available to be used. This is what we want.

It's now time to comment your program All programs need to be commented so that the user knows what the program is about. Just by reading the comments in your program you will know exactly what the program is supposed to do. We have two types of comments in C#. Header comments that are at the start of a program or a method. They start with `/*` and end with a `*/` and can span multiple lines like this.

```
/*  
Program to read a name and age from a user and  
print the details to the screen  
*/
```

Other comments are for one line only and explain what the current or proceeding program statement it is to do. The one-line comment starts with a `//` like this:

```
// method to read a name from the key board are return the value
```

We now comment the program as follow. Please add all these comments to your program.

```
/*  
Program to read a name and age from a user and print  
the details on the screen  
*/  
  

```

```

// method to read a name from the key board are return the value
public static String enterName()
{
    Console.Write("Please type in your name: ");
    string name = Console.ReadLine();
    return name;
}

// method to read an age from the key board are return the value
public static int enterAge()
{
    Console.Write("How old are you? ");
    int age = Convert.ToInt32(Console.ReadLine());
    return age;
}

// method to print out a user name and age
public static void printPerson(String name, int age)
{
    Console.WriteLine("Nice to meet you " + name);
    Console.WriteLine(name + " You are " + age + " years old");
}

// run program
public static void Main(string[] args)
{
    welcome();
    String name = enterName();
    int age = enterAge();
    printPerson(name, age);
}
}

```

Lesson 2 Homework

Take your homework program from Lesson 1 and add methods to it. Make methods `welcome`, `enterTitle`, `enterSalary`, `displayInfo`. Call all these methods from the main method. Call your C# program `Homework2.cs` and class `Homework2`.

Lesson 3 Classes and Inheritance

Classes represent another level in program organization. They represent programming units that contain variables to store values and methods to do operations on these variables. This concept is known as **Object Oriented Programming**, and is a very powerful concept. It allows programming units to be used over again in other programs. The main benefit of a class is to store values and do operations on them transparent from the user of the class. It is very convenient for the programmers to use classes. They are like building blocks that are used to create many sophisticated programs with little effort.

A class starts with the keyword **class** and the class name. Following the private class definition header are the constants and variables. Constants store values once initialized never change and variables are used to store values. Following the variables are the constructor's properties and methods. Constructors are used to initialize the variables defined in the class, properties are used to access the variables, and method are used to do operations on the variables. These operations may include addition, incrementing values etc. The syntax for a class definition is as follows.

```
class class_name  
{  
constant definitions  
variable definitions  
constructors  
properties  
methods  
ToString method  
}
```

We have already used a class in our previous lesson. We will now write a Person class that has variables to store a name and age and methods to do operations on them, like initializing retrieval, assignment and output. Make a new C# class file called Person.cs, and type the following code into it.

```

/*
Person Class to store a person's name and age
*/

// define a class Person
public class Person {
    private String name;
    private int age;

    // initialize Person
    public Person(String name, int age){
        this.name = name;
        this.age = age;
    }

    // return and assign name
    public String Name(){
        get { this.name;}
        set (this.name=value;}
    }

    // return and assign age
    public int Age{
        get{return this.age;}
        set {this.age = value;}
    }

    // return person info as a string
    public override string ToString() {
        string sout = "Nice to meet you " + this.name + "\n";
        sout += this.name + " You are " + this.age + " years old";
        return sout;
    }
}

```


The Person class definition starts with the class key word and class name Person. We use the public access modifier because we want others to use our class.

```
public class Person{
```

Our Person class has 2 private variables to store person name and age.

```
private string name;  
private int age;
```

We make the variables private because we want them to be only access by our class methods, nobody else. A class contains a **Person** method that initializes the class. This **Person** method is also known as a **constructor**. The constructor has the same name as the class.

```
// initialize Person  
public Person(string name, int age){  
    this.name = name;  
    this.age = age;  
}
```

The programming statements inside the constructor assign values to the variables name and age from the parameters name and age.

```
this.name = name  
this.age = age
```

The keyword **this** specifies which variables belongs to the Person class. The parameter name and age just store values to be assigned to the Person class variables and are not the same ones in the Person class.

C# has Properties rather than getters and setters, they basically do the same thing but more compact. A property can do both returning and assigning depending on the way you do it.

```
// Name property
```

```

// return/assign name
public string Name
{
    get{ return this.name; }
    set{ this.name = value; }
}

// Age property
// return/assign age
public int Age
{
    get{ return this.age; }
    set{ this.age = value; }
}

```

You will notice each properties has get and set's. The get returns a value. The set assigns a value. The value assigned comes from the keyword value. The **this** keyword indicate the variables that are defined in the class.

All classes should have a **ToString()** function so that it can easily return the class info as a string message.

```

// return person info as a string
public override string ToString() {
    String sout = "Nice to meet you " + this.name + "\n";
    sout += this.name + " You are " + this.age + " years old";
    return sout;
}

```

Our ToString method has the **override** keyword meaning it overrides the Object class ToString method. The Object class is the super class of all Classes here all classes come from. Override means we want our ToString method to be called not the Object ToString method to be called. Notice we have no print statement in our **ToString()** method.

We assign information to the local variable `sout` and return the `sout` value. A local variable is just known to the function it resides in. The `sout` variable uses the `+` operator to join values together as a message.

Usually a class definition must not contain any input or output statements. A class must be a reusable program unit not dependent on any input or output print statements. The purpose of the class is to contain information that can be easily accessed. Therefore, our main function must provide all the input and output print statements. We will use the input and output method from our previous program.

Make a new file called `Lesson3.java` and type in the following code. Exclude `Lesson2.cs` from the Solution Explorer since a solution can only have one Main method.

```
public class Lesson3
{
    // method to print welcome message
    public static void welcome()
    {
        Console.WriteLine("Hello World");
    }

    // method to read a name from the key board are return the value
    public static string getName()
    {
        Console.Write("Please type in your name: ");
        String name = Console.ReadLine();
        return name;
    }

    // method to read an age from the key board are return the value
    public static int getAge()
    {
        Console.Write("How old are you? ");
        int age = Convert.ToInt32(Console.ReadLine());
        return age;
    }
}
```

```
// main method to run program
public static void Main(string[] args)
{

    // print welcome message
    welcome();

    // get person info from keyboard
    String name = getName();
    int age = getAge();

    // make a Person object
    Person p = new Person(name, age);

    // print out person details
    Console.WriteLine(p.ToString());
}
}
```

Build and run the program, type Tom for name and 34 for age. You will get the following output.

```
Hello World
Please type in your name: Tom
How old are you? 24
Nice to meet you Tom
Tom You are 24 years old
```

Notice we create the Person class with the following statement:

```
Person p = new Person(name, age);
```

This calls the **Person** constructor of the person class to create the person object and initialized with the values name and age. The mechanism that allocates memory in the computer for the variables and function code defined in the class, is known as **instantiation**.

When a class is instantiated in computer memory it is known as an **object**. when a class is written in a program then it is still known as a class not an object. Objects are made from class definitions. An object is the memory allocated for the variables defined in the class when the program runs.

The **WriteLine** statement calls the **ToString()** method to print out the Student info.

```
Console.WriteLine(p.toString());
```

You can automatically call the ToString() method just by using the Person variable

```
Console.WriteLine (p);
```

Once you got this program running, add the following to the main method of the Lesson3 class. Make a new person called p2. Copy the values from person p using the properties, then using the properties again assign a new name Sue and age 36 to the second person and print out the person p2 details. You should get something like this:

```
Hello World
Please type in your name: Tom
How old are you? 24
Nice to meet you Tom
Tom You are 24 years old
Nice to meet you Sue
Sue You are 36 years old
```

If you can do this, you are almost great C# programmer!

Default constructors have no parameters and are mainly used to initialize the variables defined in the class to default values.

```
public Person(){
    this.name = "";
    this.age = 0;
}
```

Auto properties

Auto properties combine defining variables and properties at the same time, may be a little confusing or redundant but they can make your class quite compact. Here are the Person class auto properties for Name and Age.

```
public string Name { get; set;}
public int Age { get; set;}
```

You use auto properties in the same way as you use properties. The only difference is auto properties do not require instance variables.

Lesson 3 Homework Question 1:

Make a Profession class that stores somebody's profession **title** and amount of **money** they make. Make default and initializing constructors, getters and setters and a ToString methods. Call your class Profession and put your class in a file called Profession.java. Make a C# program file that has a main method that instantiates a Profession class. The main method has additional methods to get the professions title and amount of money that they make from the keyboard. You can use some of the methods from Lesson2. Call your java main program file Homework3.jcs.

INHEIRITANCE

The beauty of classes is that they can be extended to increase their functionality. We can make a Student class that uses the public variables and methods from the Person class. This is known as **inheritance**. The Student class can only access the non-private variables and methods from the Person class. We now have additional access modifiers public, protected, package and private.

access modifier	description	example
public	Access by anybody	public int age;
protected	Access by derived class	Protected int age;
package	Access by classes in project	int age;
private	Access by its own class only	private int age;

The Student class will have an additional variable called **idnum** that will represent a string student id number. Using inheritance, the student class will be able to use the public variables and methods of the Person class. The Person class is known as the **super** or **base** class and the Student class is known as the **derived** or **sub** class. The Person class knows nothing about the Student class where as the Student class knows all about the Person class.

Create a new C# file called Student.cs. Create a class called Student that inherits from the Person class using this statement.

```
// define a class Student that inherits the Person class  
public class Student: Person {
```

The colon : specifier is used to define the inheritance relationship. This means the Student class can use the public variables and methods from the Person class. We also need to define a student id number variable for the Student class.

```
// student id number  
private string idnum;
```

```
// default Student  
public Student(string name, int age, String idnum)
```

```

    : base(name, age)
    {
        this.idnum = idnum;
    }

```

We now make a constructor that will initialize the student name, age and idnum.

```

// initialize Student
public Student(string name, int age, String idnum)
    : base(name, age)
    {
        this.idnum = idnum;
    }

```

The colon : operator calls the base constructor the Person class to create a Person object and initialize the name and age variables in the Person object from the input parameters name and age. The **idnum** variable is initialized in the Student constructor.

Student ID Property

```

// return/assign idnum
public string IdNum
{
    get{ return this.idnum; }
    set{ this.idnum = value; }
}

```

The last thing you need to make the **ToString()** method. By using the **base** keyword you can call functions directly from the super Person class inside the Student derived class. Here is the Student **ToString()** method

```

// return student info as a string
public override string ToString() {

```



```
string sout = base.ToString();
sout += " Your student id number is " + this.idnum;
return sout;
}
```

Once you got the Student class made then add programming statements to the lessons3.cs file to obtain a student name, age and idnum. You will have to make an additional getID() function to obtain a student id number from the key board.

Instantiate a student object and use the keyboard methods to obtained a name, age and idnum then print out the student details. You should get something like this:

```
Hello World
Please type in your name: Tom
How old are you? 24
Nice to meet you Tom
Tom You are 24 years old
Nice to meet you Sue
Sue You are 36 years old
Please type in your name: Bill
How old are you? 18
What is your student ID? S1234
Nice to meet you Bill
Bill You are 18 years old Your student id number is S1234
```

Auto properties

Auto properties combine defining variables and properties at the same time, may be a little confusing or redundant but they can make your class quite compact. Here is a auto property for the Student IdNum.

```
public string IdNum { get; set;}
```

You use auto properties in the same way as you use properties. The only difference is auto properties do not require instance variables.

Lesson 3 Homework Question 2:

Make a Payroll class that inherits your previous homework Profession class that stores a boolean or a String value to indicate if a worker works full time or part time. Use your C# main program file Homework3.cs from Question 2.

private boolean fullTime;

You will need default and initializing constructors, getters, setters and a ToString methods. The initializing constructor would receive a profession title, salary and a boolean indicating full time or part time, A true value would indicate full time employment, where false value would indicate part time employment.

For the part time variable getter use:

public boolean isFullTime();

For the part time variable setter use:

public void setFullTime(boolean partTime);

Store the Payroll class in a java file called Payroll.java.

In the main method of your Homework3.java program make an additional **public static boolean enterFullTime()** method asking if a worker is full time or part time, that returns a Boolean true or false value. If they enter 'Y' return true, if they enter 'N' return false.

boolean fullTime = enterFullTime();

Then instantiate a Payroll class object and pass this **fullTime** value to your Payroll class constructor as well as the profession and salary of the worker.

Use the Payroll class ToString method to print out the workers profession, pay amount and indicate if it is a full time or part time worker.

You can still use file Homework3.cs

Lesson 4 Operators

Operators

Operators do operations on variables like addition + , subtraction – and comparisons > etc. We now present all the C# operators with examples. Make a new C# class file called Operators.cs. In the main method type out the examples and use Console.WriteLine statements to print out the results. You can type in the operation right inside the Console.WriteLine statement just like this Console.WriteLine (3+2) or Console.WriteLine (3 > 2). Alternatively, you can use variables instead.

```
int x = 3;  
int y = 2;  
int r = x + y;  
Console.WriteLine (r);
```

Arithmetic Operators

Arithmetic operators are used to do operations on numbers like addition and subtraction.

```
int x = 3 + 2;  
Console.WriteLine (x); // would print out 5
```

Operator	Description	Example	Result
+	Add two operands or unary plus	3 +2	5
-	Subtract right operand from the left or unary minus	3 - 2 -2	1 -2
*	Multiply two operands	3 * 2	6
/	Divide left operand by the right one	5 / 2	2
%	Modulus - remainder of the division of left operand by the right	5 % 2	3

Comparison Operators

Comparison operators are used to compare values. It either returns true or false according to the condition. True and false variables and values are known as bool.

```
bool b = 3 < 5;  
Console.WriteLine (b); // world print out false
```

Operator	Description	Example	Result
>	Greater than - true if left operand is greater than the right	5 > 3	true
<	Less than - true if left operand is less than the right	3 < 5	true
==	Equal to - true if both operands are equal	5 == 5	true
!=	Not equal to - true if operands are not equal	5 != 5	true
>=	Greater than or equal to - true if left operand is greater than or equal to the right	5 >= 3	true
<=	Less than or equal to - true if left operand is less than or equal to the right	5 <= 3	true

Logical Operators

Logical operators are the **and**, **or**, **not** bool operators.

```
Bool b = 3 < 5 && 2 < 6  
Console.WriteLine (b); // would print out true
```

Operator	Description	Example	Result
&&	true if both the operands are true	true && true	true
	true if either of the operands is true	true false	true
!	true if operand is false (complements the operand)	! false	true

Bitwise Operators

Bitwise operators act on operands as if they were binary digits. It operates bit by bit. Binary numbers are base 2 and contain only 0 and 1's. Every decimal number has a binary equivalent. Every binary number has a decimal equivalent. For example, decimal 2 is 0010 in binary and decimal 7 is binary 0111.

In the table below: 10 = (0000 1010 in binary) 4 = (0000 0100 in binary)

Console.WriteLine by itself does not print binary number just decimal numbers.

```
int x = 10 | 4
```

```
Console.WriteLine (b); // world print out 14
```

Operator	Description	Example	Result
&	Bitwise AND	10 & 4	0 (0000 0000 in binary)
	Bitwise OR	10 4	14 (0000 1110 in binary)
~	Bitwise NOT	~10	-11 (1111 0101 in binary)
^	Bitwise XOR	10 ^ 4	14 (0000 1110 in binary)
>>	Signed Bitwise right shift	10 >> 2	2 (0000 0010 in binary)
<<	Bitwise left shift	10 << 2	40 (0010 1000 in binary)

Shift operators

The shift operators multiply by powers of 2 or divide by powers of 2.

Multiplying is accomplished by shifting the bits left. Dividing by 2 is accomplished by shifting the bits right.

```
int x = 10; (0000 1010 in binary)
```

```
int y = x << 2; y = 10 * 2 * 2 = 40 (0010 1000 in binary)
```

```
int z = y >> 2; z = 40 / 2 / 2 = 10 (0000 1010 in binary)
```

You can shift by any power of 2

Multiplying by powers of 2:

```
x = 4;  
Console.WriteLine(x); // 4  
x = x << 2;  
Console.WriteLine(x); // 16 because 4 * 2 * 2 = 16
```

Dividing by powers of 2:

```
Console.WriteLine(x); // 8  
x = x >> 2;  
Console.WriteLine(x); // 16 / 2 / 2 = 2
```

<<	Bitwise left shift	10 << 2	10 * 2 * 2 = 40 (0010 1000 in binary)
>>	Signed Bitwise right shift	40 >> 2	40 / 2 / 2 = 10 (0000 1010 in binary)

Increment/Decrement Operators ++ --

Increment operators ++ increment a variable value by 1 and decrement operators -- decrement a value by 1.

They come in two versions, **prefix** increment/decrement value **before** or **postfix** increment/decrement value **after**.

Prefix Increment before : $y = ++x$;

x is incremented then value of y is assigned the value of x

	x	y
int x = 5	5	?
y = ++x	6	6

```

x = 5;
Console.WriteLine(x); // 5
y = ++x;
Console.WriteLine(y); // 6
Console.WriteLine(x); // 6

```

postfix increment after y = x++

The value of y is assigned the value of x and then x is incremented

	x	y
	-----	-----
int x = 5	5	?
y = x++	6	5

```

x = 5;
Console.WriteLine(x); // 5
y = x++;
Console.WriteLine(y); // 5
Console.WriteLine(x); // 6

```

prefix Decrement before y = --x

x is decremented then value of y is assigned the value of x

	x	y
	-----	-----
int x = 5	5	?
y = --x	4	4

```

x = 5;
Console.WriteLine(x); // 5
y = --x;
Console.WriteLine(y); // 4
Console.WriteLine(x); // 4

```


postfix decrement after y = x--

The value of y is assigned the value of x and then x is decremented

	x	y
	-----	-----
int x = 5	5	?
y = x--	4	5

```
x = 5;  
Console.WriteLine(x); // 5  
y = x--;  
Console.WriteLine(y); // 5  
Console.WriteLine(x); // 4
```

Increment decrement operators are usually used stand alone to increment or decrement a variable value by 1.

Assignment Operators

Assignment operators are used to assign values to variables.

x = 5 is a simple assignment operator that assigns the value 5 on the right to the variable **x** on the left. There are various compound operators in C# like **x += 5** that adds a value to a variable. It is equivalent to **x = x + 5**.

Operator	Compound	Equivalent
=	x = 5	x = 5
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
%=	x %= 5	x = x % 5
&=	x &= 5	x = x & 5
=	x = 5	x = x 5
^=	x ^= 5	x = x ^ 5
>>=	x >>= 5	x = x >> 5
<<=	x <<= 5	x = x << 5

String Operators

String operators operate on string objects, String objects contain string data and are **immutable** meaning they cannot be changed internally.

There are many string operation's, most of them are method calls. Here are just a few of them:

```
// declare and assign String
string s1 = "hello";
string s2 = "there";

// join two strings together
string s3 = s1 + s2;
Console.WriteLine (s3); // hello there

// get a character from string
char c = s3[0];
Console.WriteLine (c); // h

//get a sub string (start, length)
string s4 = s3.Substring(0,5);
Console.WriteLine (s4); // hello

// add a character to a string
string s5 = s3.Substring(0,5)+ 'X' + s3.Substring(5);
Console.WriteLine (s5); // hello there

// make string lower case
string s6 = s5.ToLower();
Console.WriteLine (s6); // hello there

// make string upper case
string s7 = s5.ToUpper();
Console.WriteLine (s7); // HELLOXTHERE
```

```
// test if 2 strings are equal
Console.WriteLine (s1.Equals(s2)); // false

// test if 2 strings have the same memory location
Console.WriteLine (s1 == s2); // false

// test if 2 strings are less greater or equal
// -1 = less 0 = equal 1 = greater
Console.WriteLine (s1.CompareTo( s2)); // -1
```

As stated earlier you should type in all the above examples and try them out in a C# file called Lesson4.cs.

LESSON 4 HOMEWORK

1. Print out if a number is even, using just a print statement and a arithmetic operator
2. Print out of a number is odd, using just a print statement and a arithmetic operator
3. Swap 2 number using a temporary variable
4. Swap 2 numbers using operators + and -
5. Multiply a number by 8 using a shift operator
6. Divide a number by 8 using a shift operator
7. Using a print statement, add 2 numbers together and check if they are less than multiplying them together
8. Using a print statement, add 2 numbers together and check if they are less than multiplying them together and greater then multiplying them together.

9. Using a print statement, add 2 numbers together and check if they are less than multiplying them together or greater than multiplying them together

10. Make a string of your favourite word and replace the first letter with another letter, hint use substring.
Example : change "hello" to "jell"

11. Make a string of your favourite word and replace the last letter with another letter, hint use substring.
Example : change "jell" to "jelly"

12. Make a string of your favourite word and remove the middle letter, hint use substring.
Example : change "jelly" to "jelly"

13. Make a string of your favorite word, Split it in the middle, make the first part lower case and the second part upper case, hint use substring.
Example : change "jelly" to "jelly"

14. Using substring replace the last letter with the first letter in a word
Example : change "jelly" to "Yell"

15. Split the word in the middle put the last part at the first and the first part at the last
Example : change "Yell" to "Lye"

Call your C# file homework4.cs

Lesson 5 Programming Statements

Programming statements allow you to write complete C# Program. We have already looked at simple input, print and assignment statements. We now present you with branch and loop programming statements. Start a new C# class lesson5.cs to test all these programming statements.

```
/*  
Lesson5.cs  
Programming statements  
*/
```

Branch Control Statements

Branch control statements allow certain program statements to execute and other not.

The **if** branch **control** statements contain a condition using conditional operators from the previous lesson to direct program flow.

```
If (condition)  
Statement(s)
```

When the condition is evaluated to be true the statements belonging to the if statement execute. An if statement is a one-way branch operation.

```
// if statement  
x = 5;  
if (x == 5)  
{  
    Console.WriteLine ("x is 5");  
}
```

x is 5

We now add an else statement. An if-else control construct is a two-way branch operation.

```
if (condition)
    statements
else
    statements

// if else statement
x = 2
if (x == 5)
{
    Console.WriteLine ("x is 5");
}
else
{
    Console.WriteLine ("x is not 5");
}
```

x is not 5 I like C# Programming

We can also have additional else if statements to make a multi-branch.

```
// multi if else
x = 10
if (x == 5)
{
    console. WriteLine ("x is 5");
}
else if (x < 5)
{
    console. WriteLine ("x less than 5");
}
```

```
else if (x > 5)
{
    console.WriteLine ("x greater than 5");
}
```

x greater than 5

Our multi branch if-else can also end with an else statement.

```
// multi if-else else
x = 5
if (x < 5)
{
    Console.WriteLine ("x less than 5");
}
else if (x > 5)
{
    Console.WriteLine ("x greater than 5");
}
else
{
    Console.WriteLine ("x is 5");
}
```

x is 5

switch statement

A switch statement is an organized if-else statement. It is a little limited since it can only handle equals.

// switch statement

```
x = 2;
switch(x)
{
    case 1:
        Console.WriteLine ("x is 1");
        break;
    case 2:
        Console.WriteLine ("x is 2");
        break;
    case 3:
        Console.WriteLine ("x is 3");
        break;
    default:
        Console.WriteLine ("x is " + x);
        break;
}
```

x is 2

nested if statements

if statements can also be nested to make complicated conditions simpler.

```
// nested if statement
x = 5;
if (x >= 0)
{
    if (x > 5)
        Console.WriteLine ("x greater than 5");
}
```



```

else
    Console.WriteLine ("x less than equal 5");
}
Console.WriteLine ("I like C# Programming");

```

<pre> x less than equal 5 I like C# Programming </pre>
--

Loop Control Statements

Loop control statements allow program statements to repeat themselves.

while loop

The while loop allows you to repeat programming statements repeatedly until some condition is satisfied. While loops are used when you do not know how many items you have.

The while loop requires an initialized counter, a condition, program statements and then increment or decrement a counter.

```

Initialize counter
while condition:
    statement(s)
    increment/decrement counter

```

Here is a while loop you can try out that prints out the number 5 to 1 backwards.

```

// while loop
x = 5
while (x > 0)
{
    Console.WriteLine (x)
    x-=1
}

```

<pre> 5 4 3 2 1 </pre>

for Loop

Another loop is the **for** loop. It is much more automatic than the while loop but more difficult to use. All loops must have a counter mechanism. The for loop needs a start count value, condition, increment/decrement counter. For loops are used when you know how many items you have.

```
for (start_count_value,condition, increment/decrement_counter):  
    Statement(s)
```

Here we have a for loop to print out values 1 to 5, try it out.

```
// for loop  
for (int i=1;i<=5;i++)  
    Console.WriteLine (i);
```

```
1  
2  
3  
4  
5
```

Here is a for loop that counts backwards using a negative increment

```
// for loop counting backward  
for (int i=5;i>0;i--)  
    Console.WriteLine (i)
```

```
5  
4  
3  
2  
1
```

Nested for loops

Nested for loops are used to print out 2 dimensional grids by row and column.

```
// nested for loop  
for (int r=0;r<5;r++)  
{  
    print(r + " : ")  
    for (int c =0; c < 5; c++)  
        Console.WriteLine ( c + " ")  
    Console.WriteLine ("")  
}
```

```
1 : 1 2 3 4 5  
2 : 1 2 3 4 5  
3 : 1 2 3 4 5  
4 : 1 2 3 4 5  
5 : 1 2 3 4 5
```

Loops can also be used to print out characters in a string variable

```
// print out characters in a string
s = "Hello";
for (int i=0;i<s.Length();i++)
    Console.WriteLine (s[i]);
Console.WriteLine ("")
```

H
e
l
l
o

LESSON5 HOMEWORK

Question 1 Exam Grader

Ask someone to enter an exam mark between 0 and 100. If they enter 90 or above printout an “A”, 80 or above print out a “B”, 70 or above print out a “C”, 60 or above print out a “D” and “F” if below 60. Hint: use if else statements.

You can visualize a grade chart like this:

Mark Range	Exam Grade
90 to 100	A
80 to 89	B
70 to 79	C
60 to 69	D
0 to 59	F

Question 2 Mini Calculator

Make a mini calculator that takes two numbers and a operation like -, +, * and /. Prompt to enter two number's and a operation like this:

Enter first number: 3
Enter second number: 4
Enter (+, -, *, /) operation: +

Then print out the answer like this:

3 + 4 = 7

Hint: use a switch statement.

Use a while or do while loop so that they can repeatedly enter many calculations. Terminate the program when they enter a letter like 'X' for the first number.

Question 3 Triangle Generator:

Use nested for loops to print out a triangle using '*' like this:

```
      *
     * *
    * * *
   * * * *
  * * * * *
```

Ask the user how many rows they want.

Hint: use 2 nested for loops, start with a square of stars

Question 4 Enhanced Triangle Generator:

Use nested for loops to print out a triangle using '*' like this:

```
      *
     ***
    *****
   ********
  **********
 **********
```

Ask the user how many rows they want.

Hint: use 2 nested for loops, start with a square of stars

Question 5 Reverse a String

Reverse a String using a **while** loop or a **for** loop. You will have to put the reversed characters in a second string, since you cannot in Java directly change the characters in the original String.

Question 6 Test if a number is prime

Make a function called **isPrime(x)** that tests if a number is prime. In a loop divide the number between 2 to number-1 (or 2 to square root of number+1. For square root use:

```
x = (int)Math.Sqrt(n);
```

If the number can be divided by any of the divisors then the number is not prime, else it is prime. Print out the first 100 prime numbers.

The first 10 prime numbers are: 2, 3, 5, 7, 11, 13, 17, 19, 23, and 29

Question 7 Print out all factors of a number

Make a function call **factors(x)** that will print out all the factors of a number. The factors of a number is all the divisors divided by the number evenly.

Example:

The Factors of 50 are:

1
2
5
10
25
50

Question 8 Print out all prime factors of a number

Make a function call **prime_factors(x)** that will print out all the prime factors of a number. The prime factors of a number is all the prime number divisors divided by the number evenly.

Example: $12 = 2 \times 2 \times 3$

Following are the steps to find all prime factors.

- 0) Enter a number n
- 1) While n is divisible by 2, print 2 and integer divide n by 2
- 2) In a **for** loop from i = 3 to square root of n + 1 increment by 2
in a **while** loop while n is divisible by i
print i
integer divide n by integer i
- 3) print n if it is greater than 2.

For square root use:

```
x = Math.Sqrt(n);
```

Question 9 Make a Guessing game

Ask the user to guess a number between 1 and 100. If they guess too high tell them "Too High". If they guess too low tell them they guess "Too Low". If they guess correct tell them "Congratulations you are Correct! ". Keep track how many tries they took to guess. At the end of each game ask the user if they want to play another game. After all games have been played print out the average number of tries for all games played. Store the sum of the tries per game and divide by the number of games played. Play up to a maximum of 10 games.

Make a C# class called **GuessingGame** with a main function.

Use a constant for the maximum number to guess 100.

```
public static final int MAX_NUMBER = 100;
```

Place the constant right after you declare your class.

You will also need to generate random numbers to guess.
You can use this code to generate a random number:

```
Random rgen = new Random();  
int number = rgen.Next(MAX_NUMBER)+ 1;
```

Where **MAX_NUMBER** is a constant to represent the maximum number to guess 100

Also use a constant for the maximum number of tries to play 10.

```
public static final int MAX_TRIES = 10;
```

You should have functions to print a welcome message, explaining how to play the game, generate a random number, get a guess from the keyboard, check if a guess is correct and print out the number of tries per game and average score for all games. The main function should just call your functions in a loop. Call your C# file `GuessingGame.cs`.

Question 10 Object Oriented Guessing Game

Make a **GuessGame** class that store the random number to guess, and the number of tries to guess the random number. The constructor should generate the random number, and you should have a method to check if a guess is correct, too low or too high. Return a 0 if they are correct a negative number id too low and a positive number if too high. Also make a method to return the number of tries. The main function should just handle inputs from the keyboard and printing output to the console. The `GuessGame` class should not handle any input and output, and is used, mainly to store data.

The main function would instantiate a new **GuessingGame** object per game. The guessing game class should also have a static variable to store the sum of the tries per game and the number of games.

```
public static in TotalTries = 0.  
public static int NumGames = 0;
```

Call your C# file with the main function `GuessingGame2.cs`

LESSON 6 ARRAYS, HASHSETS and DICTIONARIES

Arrays store many sequential values together accessed by a common variable name. We have one dimensional arrays and multi dimensional arrays. One dimensional arrays are considered a single row of values having multiple columns. You can visualize a one-dimensional array as follows.

Value1	Value2	Value3	Value4	Value5
--------	--------	--------	--------	--------

We declare and initialize 1 dimensional array with values as follows:

```
int[] a = {1,2,3,4,5};
```

1	2	3	4	5
---	---	---	---	---

We access the values of the array using an index. In C# index start at 0. To retrieve the first value we use the index 0 enclosed in square brackets as follows:

```
int x = a[0];
```

To assign a value to a column location we also use an index and the square brackets.

```
a[0] = 2;
```

Print number of elements in an 1 dimensional array

```
int numElements = a.Length;  
Console.WriteLine(numElements); // 5
```

we next declare a one-dimensional array of size 5 without initializing the column values, all values are set to 0.

```
int[] a2 = new int[5];
```


In this situation you need to assign values separately. Array column locations are assigned by an index starting at 0.

```
a2[0] = 1;  
a2[1] = 2;  
a2[2] = 3;  
a2[3] = 4;  
a2[4] = 5;
```

1	2	3	4	5
---	---	---	---	---

Array location columns are retrieved by an index

```
int x = a[0];  
Console.WriteLine(x); // 1
```

The column index of a one-dimensional array can be visualized as follows.

[0]	[1]	[2]	[3]	[4]
-----	-----	-----	-----	-----

We use for loops to print out values in array. C# does not do automatically for you.

```
// print out values in a 1 dimensional array  
int[] a = {1,2,3,4,5};  
for (int i=0;i<a.Length;i++)  
    Console.WriteLine (a[i] + " ");  
Console.WriteLine ("");
```

1 2 3 4 5

Two-dimensional arrays

Two-dimensional arrays have grid of rows and columns. A 3 by 4 two-dimensional array is visualized as follows, (3 rows and 4 columns).

1	2	3	4
5	6	7	8
9	10	11	12

Here we declare and initialize a two-dimensional array with values:

```
int [,] b = {{1,2,3,4},{5,6,7,8},{9,10,11,12}};
```

Here we just declare a two-dimensional array without initializing.

```
int[,] b2 = new int[3,4];
```

We **assign** values to the two-dimensional array by row index 2 and column index 3. The row index first and the column index second.

```
b2[2,3] = 11;
```

We **retrieve** values from the two-dimensional array also by row index 2 and column index 3. The row index is first and the column index second.

```
x = b2[2,3];  
Console.WriteLine(x); // 11
```

The row index and column index of a two-dimensional array can be visualized as follows. The row index is first and the column index second. We have 3 rows and 4 columns.

[0,0]	[0,1]	[0,2]	[0,3]
[1,0]	[1,1]	[1,2]	[1,3]
[2,0]	[2,1]	[2,2]	[2,3]

Print number of rows in an 2 dimensional array

```
int rows = a.GetLength(0);  
Console.WriteLine(rows); // 3
```

Print number of columns in an 2 dimensional array

```
int columns = a.GetLength(1);  
Console.WriteLine(columns); // 4  
// print out values in a two dimensional array
```

```

int[,] b = {{1,2,3},{4,5,6},{7,8,9}};

for (int r=0;r < b.GetLength(0); r++)
{
    for (int c=0;c < b.GetLength(1); c++)
    {
        Console.Write (b[r,c] + " ");
    }
    Console.WriteLine ("");
}

```

```

1 2 3
4 5 6
7 8 9

```

We use **b.GetLength(0)** to get the length of the first dimensions which is 3 rows. We use **b.GetLength(1)** to get the length of the second dimensions which is 3 columns.

Lists

Lists are expandable arrays. A List needs to know what type of data type it will use so you need to specify the data type to be used inside triangle brackets <int>.

```

// make an List of values
List<int> list1 = new List<int>();
list1.Add(1);
list1.Add(2);
list1.Add(3);
list1.Add(4);
list1.Add(5);

```

```

1
2
3
4
5

```

Alternately you can use an initialization list:

```

List<int> list1 = new List<int>{1,2,3,4,5};

```

C# does not print out the list as nicely as is done in other programming languages, we must do it this awkward way.

```

// Print out array list
list1.ForEach(Console.WriteLine); // 1,2,3,4,5

// get and print the number of elements in an list
Int x = list1.Count();

Console.WriteLine (x) // 5

// Get and print a value from a list at a specified location
Console.WriteLine ( list1[0]) // 1

// remove a value from a list
list1.Remove (3)
list1.ForEach(Console.WriteLine); // 1,2,4,5

// remove a list item by index
list1.RemoveAt(0)
list1.ForEach(Console.WriteLine); // [2,5]

// get index of a value in an array
int index = list1.IndexOf((5));
Console.WriteLine(index); // 1

// test if a value is in a list returns true or false
Console.WriteLine (list1.Contains(5)) // false

```

Printing out values from a List

For loops can also print out values from lists. We can do this 3 different ways. Index **for loop** inside an index counter, **foreach loop** that can travel through each item in the list using a variable value and a loop that uses an **Enumerator** to traverse through the list by extracting the next value per iteration of the loop. You need have **using System.Collections.Generic;** and **using System.Linq;** at the top of your Lesson6.cs file.

There are 3 ways to print out a List using a list;

(1) Using an **for** index loop:

```
// using an index loop
for (int i=0;i<list1.sCount();i++)
    Console.WriteLine (list1[i]);
Console.WriteLine ("");
```

(2) Using a **foreach** loop:

```
// use an foreach loop
foreach(int v in list1)
    Console.WriteLine (v);
Console.WriteLine ("");
```

```
1
2
3
4
5
```

A **foreach** loop uses a variable and the keyword **in** to traverse the list.

(3) Using an enumeration loop

```
// use an enumeration
var e = list1.GetEnumerator();
while(e.MoveNext())
{
    x = e.Current;
    Console.WriteLine (x);
}

Console.WriteLine ("");
```

```
1
2
3
4
5
```

The Enumerator obtained from the list method GetEnumerator is assigned to the variable e. The variable e has data type **var** that can represent any data type. We

use the **MoveNext()** to traverse through the loop and the **Current** property to get the value at the current position.

HashSets

HashSets are like a List but just store unique values, there are no duplicates allowed. Hashsets store values into an array using a calculated index which results in faster execution time than Lists. You use a HashSet when you want to store unique items with no duplication. A HashSet also needs to know what type of data type it will use. The data type to be used is specified inside Triangle brackets <int>.

```
// make HashSet  
HashSet<int> set1 = new HashSet<int>();  
  
// add values to hash set  
set1.Add(1);  
set1.Add(2);  
set1.Add(2);  
set1.Add(3);  
set1.Add(4);  
set1.Add(4);  
set1.Add(4);  
set1.Add(4);  
set1.Add(5);
```

Alternately you can use an initialization list:

```
HashSet <int> set1 = new HashSet <int>{1,2,2,3,4,4,4,4,5};
```

C# does not print out the list as easy as other programming languages, we have to do it this awkward way.

```

// print out set
Console.WriteLine(String.Join<int>(", ", set1)); // 1,2,3,4,5

// return true if set contains item
Console.WriteLine (set1.contains(5)); // true

// remove item by value from hash set
set1.remove(5);
Console.WriteLine(string.Join<int>(", ", set1)); // 1,2,3,4

```

Printing out values from a HashSet using a loop

We can also use a **for loop** to print out the items stored in a Hash Set. Printing values in a HashSet is more difficult since the HashSet values are not stored in sequential memory locations. The Hash set values are not always in order. We only use a **foreach** loop that can travel through each item in the list using a variable value and a loop that uses an Enumerator to traverse through the list by extracting the next value per iteration of the loop.

There are 2 ways to print out a List using a list;

(1) Using an **foreach** loop:

```

foreach(int v in set1)
    Console.WriteLine (v);
Console.WriteLine ("");

```

```

1
2
3
4
5

```

(2) Using an enumeration loop

```

// use an enumeration loop
var e2 = set1.GetEnumeration();
while(e2.MoveNext())
{
    x = e2.Current;

```

```

1
2
3
4
5

```

```
        Console.WriteLine (x);
    }
    Console.WriteLine ("");
```

Using a HashSet to print out unique letters in a word.

```
String s2 = "tomorrow:";
HashSet<char> set3 = new HashSet<char>();
foreach(char c in s2)
{
    set3.Add(c);
}

foreach(char c in set3)
{
    Console.Write(c);
}
Console.WriteLine("");
```



tomrw

Dictionary's

A Dictionary contains a key and a value. A Dictionary can have many keys and corresponding values. Think of a Dictionary like a telephone book with the name as the key and the telephone number as the value.

A Dictionary need to know what type of data type it will use, the data type is specified inside Triangle brackets <string, string>. The first data type is for the key and the second data type is for the value.

```
// make empty map
Dictionary<String, String> map1 = new Dictionary<String, String>();

// add values to Dictionary
map1["name"] = "Tom";
map1["age"] = "24";
map1["idnum"] = "S1234";
```


alternately you can use an initialization list:

```
Dictionary<String, String> map2 = new Dictionary<String, String>
{
    { "name", "Tom" },
    { "age", "24" },
    { "idnum", "S1234" }
};

// get values From a Dictionary
Console.WriteLine (map1["name"]); // Tom
Console.WriteLine (map1["age"]); // 24
Console.WriteLine (map1["idnum"]); // S1234
```

C# does not print out the list as easy as other programming languages, we have to do it this awkward way.

```
// print out Dictionary
map1.ToList().ForEach(m => Console.WriteLine(m.Key + ":" + m.Value));

// name:Tom
// age:24
// idnum:S1234

// get a list of keys from the Dictionary
List<string> keys = new List<string>(map1.Keys);

// print keys of a Dictionary
keys.ForEach(Console.WriteLine); // name idnum age]

// get values of a Dictionary
List<string> values = new List<string>(map1.Values);

// print values of a Dictionary
values.ForEach(Console.WriteLine); // Tom S1234 24
```

Printing out values from a Dictionary using loops

We also use for loops can to print out Dictionary. The main purpose of loops is to print out the Dictionary in order by key or order by values.

Print out a dictionary using a for each loop.

```
// print dictionary by key and value
foreach (var kv in map1)
{
    Console.WriteLine(kv.Key + " : " + kv.Value);
}
```

```
name : Tom
age : 24
idnum : S1234
```

Print out dictionary by keys unsorted.

```
// print Dictionary by key
foreach(String key in map1.Keys)
{
    String value = map1.get(key);
    Console.WriteLine (key + " : " + value);
}
Console.WriteLine ("");
```

```
name : Tom
age : 24
idnum : S1234
```

Print out dictionary by values unsorted.

```
// print Dictionary by value
```

```
Tom : name
24 : age
S1234 : idnum
```

```

foreach(string value in map1.values())
{
    foreach(String key in map1.keySet())
    {
        if(map1.get(key).equals(value))
        {
            Console.WriteLine (value + " : " + key );
        }
    }
}
Console.WriteLine ("");

```

Print out dictionary sorted by key.

```

// print Dictionary sorted by key
List<string> keys = new List<string>(map1.Keys);
keys.Sort();

foreach(string key in keys)
{
    string value = map1.get(key);
    Console.WriteLine (key + ":" + value);
}
Console.WriteLine ("");

```

age: 24 idnum: S1234 name: tom

Print HashMap sorted by value

To print a Dictionary sorted by value we first obtain a **KeyValuePair** list. An **KeyValuePair** is an object containing both the key and value. We then sort the list by value. We use a compare internal method that sorts the **KeyValuePair** objects by values. This internal method is known as a **delegate**.

```

// print Dictionary sorted by value
List<KeyValuePair<string, string>> kvList = map1.ToList();

```

```
kvList.Sort(delegate
    (KeyValuePair<string, string> p1, KeyValuePair<string, string> p2)
    {
        return p1.Value.CompareTo(p2.Value);
    }
);
```

Lastly we print out the sorted list.

```
foreach (KeyValuePair<string, string> p in kvList)
{
    Console.WriteLine(p.Key + ":" + p.Value);
}
```

```
age: 24
idnum: S1234
name: tom
```

Type all the above examples in your file lesson6.cs and, make sure you get the same results.

LESSON 6 HOMEWORK

Question 1

Make an array to store 5 numbers 1 to 5.
Swap two values in an array. Print array before and after using a for loop.

Question 2

Make an array of 10 numbers 1 to 10, print out the numbers in the array, then add up all the numbers and print out the sum.

Question 3

Make an array of 10 numbers 1 to 10, print out the numbers in the array. Ask the user of your program to enter a number in the array. Search for the number in the array and report if it is found or not found.

Question 4

Make an array of 10 numbers 1 to 10, print out the numbers in the array. Ask the user of your program to enter a number in the array. Search for the number in the array and report the array index where the number was found otherwise print -1 meaning no index found.

Question 5

Make an array of 10 numbers 1 to 10, print out the numbers in the array. Reverse all the numbers in the array in-place using a loop. Hint: use swap and 2 indexes i and j. Index i starts at the beginning of the array and index j starts at the end of the array. The i's increment and the j's decrement. Print out the reversed array.

Question 6

Make a 2 dimensional array of 3 rows and 3 columns. Fill the 2 dimensional array with numbers 1 to 9. Add up the sum of all rows, and print the sum at the end of each row. Add up the sums of all columns, and print the sums at the end of each column. Your output should look like this.

```
1 2 3 : 6
4 5 6 : 15
7 8 9 : 24
--- --- ---
11 15 18
```

Question 7

Make an array to hold 10 numbers 1 to 10.
Generate 1000 random numbers between 1 and 10.
Keep track of the random numbers generated in your array.
Print out all the numbers and their counts from the array.
Print out the numbers with the smallest and largest count.
Print out the number of even and odd number counts.
You can make a random number like this:

```
int x = (int)(Math.random()*10) + 1;
```

Question 8

Make an List called animals of your 5 favorite animals like: cat, dog, tiger, monkey and mouse.

Make an List called sounds of your 5 favorite animals sounds:
“meow”, ”bark”, ”roar”, “ee ee ee” and ”squeak squeak”

In a loop ask what sound each animal make?

EXAMPLE: What sound does a cat make?

Using the values in the sound List, Tell them if that are right or wrong,
and keep track of the correct answers.
At the end of the program tell them their score.

Question 9

Make an List called list1 of your favorite animals like: elephant, cat and dog.
Print out the list of animals.

Ask the user of your program to type in name of one of the animal names from
your list, that they don't like.

Remove the animal from this list and put into another list called list2.

Then ask them to type in the name of an animal they do like. Add this name to list1 and to list2.

Then print out the animals list1 and list2

Question 10

From question5 put all the animals from animal list1 and list2 into a HashSet called set1. Then take all the animals from list1 and list2 that are common between them and put into another set called set2. Print out both sets.

Question 11

Make a Dictionary called map1 of your favorite animal kinds like (cat, dog, tiger). Give each animal a name (Tom, Sally, Rudolf) . Use the animal name as the key and the animal kind as the value.

Example: fluffy cat

Then make another Dictionary called map2 , use the animal kind (cat, dog, tiger) as the key and the animal sound (meow,bark,roar) as the value.

Example: cat meow

Next print out the keys ((Tom, Sally, Rudolf) of the first Dictionary.

Ask the user to type in one of the animal names.

Get the animal kind ((cat, dog, tiger) from the first Dictionary map1 using the animal name that the user typed in.

Ask the user what kind of animal it is.

If they are correct tell them they are correct

Else

Print to the screen the name of the animal and what kind of animal it is like:

Fluffy is a cat and end the program

Next ask the user what kind of sound the animal makes?

From the second Dictionary get the sound that animal makes. If they guess the correct sound then tell them correct or tell them what sound the animal make's like:
Cat's meow

Put your code in c# file Lesson6Homework.cs

Question 12

Sentence Generator

A Sentence is composed of the following:

<article><adjective><noun><adverb><verb><article><adjective><noun>

Make an List<String> of articles like: "a", "an" and "the"

Then make an List<String> of adjectives like: "fat", "big", "small"

Then make an List<String> of nouns like: "cat", "rat", "house"

Then make an List<String> of adverbs like: "slowly", "gently", "quickly"

Then make an List<String> of verbs like "ate", "sat on", "pushed"

Make a Dictionary<String,List<String>> called words to hold all the lists:

```
words.Add( "articles",articles);  
words.Add( "adjectives",adjectives);  
words.Add( "nouns",nouns);  
words.Add( "adverbs",adverbs);  
words.Add( "verbs",verbs);
```

Next make an array of Strings or a List<String> called keys

Containing "articles", "adjectives", "nouns", "adverbs", "verbs", "articles", "adjectives", "nouns" which are Dictionary keys used to make a sentence:

Finally make a sentence using the Dictionary entries, using the parts of speech as the Dictionary keys and by selecting random words from the Dictionary values.

```
String sentence = "";  
Random random = new Random();  
foreach(String key in keys)  
{
```



```
int r = random.next(words[key].Count());  
sentence += words[key][r] + " ";  
}
```

Then print out the sentence.

```
Console.WriteLine(sentence);
```

You should get something like this:

The big cat slowly ate the small rat

Which has picked random words from the dictionary sentence structure:

<article><adjective><noun><adverb><verb><article><adjective><noun>

You can put your code in your homework6.cs file.

LESSON 7 OVERLOADING, OVERRIDING AND INTERFACES

OVERLOADING

Overloading means same method name but different parameter list signature. When you make different constructors, this is overloading as follows:

```
public Person(){
    this.name = "";
    this.age = 0;
}
```

```
public Person(String name, int age){
    this.name = name;
    this.age = age;
}
```

Do not confuse overriding with overloading. **Overriding** means same method name and same parameter list where as **Overloading** means same method name but different parameter list.

OVERRIDING

Overriding allows you to call the methods of a derived class over the methods of a super base class. The derived method has the same name and parameter list signature of a super base class method. You have all ready used an overridden method's in your Person and Student classes the ToString method. Notice we

have used the **override** keyword to indicate this method is to be override the base class ToString method.

```
public override String ToString() {
    String sout = "Nice to meet you " + this.name + "\n";
    sout += this.name + " You are " + this.age + " years old";
    return;
}
```

There is another important overridden method you should know about, the **Equals** method. It returns **true** if two Objects are equal by value.

```
// return true if two objects equal
public bool Equals(Object obj) {
    return (this == obj);
}
```

In C# when you override the **Equals** method you must also override the **GetHashCode** method. The GetHashCode method must return a unique value for the data store l your object. Fortunately, each value has its own hash code value that you can use. A object is considered equal if both the equals method and hashcode value are the same as the other object you are equating.

Here is an example **GetHashCode** method:

```
// return unique number code
public override int GetHashCode()
{
    return this.data.GetHashCode();
}
```

Todo:

Add an equals method to your Person class to test if the Person objects are equal by name. You will need to use the **is** operator to test if the obj parameter is a Person Object.

```
public override bool Equals(Object obj)
```

```

{
    if(obj is Person)
    {
        Person p = (Person) obj;
        // return if person names are equal
        return this.name.Equals(p.name);
    }
    return false;
}

```

To do:

Add a **GetHashCode()** method to your Person class. Return the hash code of the name of the Person.

```

public override int GetHashCode()
{
    return this.name.GetHashCode();
}

```

An Equals method that takes a Person parameter

For convenience you can also make an Equals method that takes a Person instead of an Object. This Equals method does not override the Object Equals method. The Person Equals method is called instead of the Equals object method when a Person object argument is supplied.

```

public bool Equals(Person p)
{
    // return true if person names are equal
    if (p != null)
        return this.name.Equals(p.name);
    else return false;
}

```

Todo: Add the Equals Person method to your Person class.

The Person parameter Equals method will return true if both the names are equal.

To do:

Make a new class file called Lesson5.cs with a main method. Make some Person Objects with same name and some with different names. Using two Person objects call the Person Objects Equals method. Print out the results using Console.WriteLine();

Overriding the Equals method in a Derived class

When we override the Equals method in a derived class we can use the Equals method of the base class to check if the base class object is also equal. We call the Equals method from the Person base class like this:

```
base.Equals(s)
```

Although we give it a Student variable the compiler extracts the Person object from the Student object.

Here is the Student Equals method:

```
public override bool Equals(Object obj)
{
    if(obj != null)
    {
        if (obj is Student)
        {
            Student s = (Student)obj;
            // return if person names and student id are equal
            return base.Equals(s) && this.idnum.Equals(s.idnum);
        }
    }
    return false;
}
```

The Student Equals method overrides the Person Equals method and the Person Equals method and Person Equals method overrides the Object Equals method.

Todo:

Add the Student equals methods to your Student class.

You also need to include a **GetHashCode()** method to complement the Equals method. We will return the hashcode of the student id.

```
// return unique number code
public override int GetHashCode()
{
    return this.name.GetHashCode();
}
```

A Equals method that takes a Student parameter

For convenience you can also make an Equals method that takes a Student object instead of an object. This Student Equals method does not override the Student Object Equals method. The Equals Student method is called when a Student is supplied as an argument to the Equals method.

```
// return true if two students are equal
public bool Equals(Student s)
{
    if (s != null)
        // return if person names are equal
        return base.Equals(s) && this.idnum.Equals(s.idnum);
    else return false;
}
```

Todo: Also add this Equals method to your Student class.

The Student parameter Equals method will return true if the name and student number s are equal.

INTERFACES

An interface just contains method declarations with no code body. Methods without a code body is known as abstract methods. Methods with code body are known as concrete methods. A class with abstract methods are known as an abstract class. A class with all abstract methods are known as an interface. The purpose of the interface is to specify what methods a class should have. The interface also becomes the super base class for the class it implements. An interface can represent many other classes that implement the interface. The

abstract and public keywords are not necessary when defining an interface because they are already understood to be there.

The syntax for an interface is as follows:

```
interface interface_name  
    {  
        method_definition_headers  
    }
```

You have already used C# classes that implement interfaces. The string class implements the IComparable Interface having the CompareTo method as follows:

```
public interface IComparable  
{  
    int CompareTo(Object obj2);  
}
```

The **CompareTo** method returns a negative number if the object value is less than the parameter obj2, returns a positive number if the object value is greater than the parameter obj2 value and returns 0 if the object value is equal to the parameter obj2 value

To use the CompareTo method your class must implement the IComparable interface **by placing** IComparable at the end of the class declaration separated by a colon :

```
public class MyClass: IComparable
```

to do:

Add a CompareTo method to your Person class, so you can compare person names. Your **Person** class must now implement the IComparable interface using the colon operator and the IComparable name as follows:

```
public class Person: IComparable{
```

Here is the **CompareTo** to method for the Person class. You will also need to use the **is** operator to test if the obj parameter is a Person Object.

```
// compare 2 person names  
public int CompareTo(object obj)  
{  
    If p != null)  
    {  
        If (p is Person)  
        {  
            Person p = (Person)obj;  
            return this.name.CompareTo( p.name);  
        }  
    }  
    return false;
```

Compare two Person objects and report if they are less than, greater than or equal.

Adding a CompareTo method to a Derived class

When adding a CompareTo method to a derived class, we put the IComparable declaration at the end of the class declaration separated by a comma.

```
class derived_class_name: base_class_name, interface_name
```

When we override the CompareTo method in a derived class we can also use the CompareTo method of the base class to help us compare the derived class. We call the CompareTo method from the a base class like this:

```
base.CompareTo(s)
```


Although we give it a derived class variable the compiler extracts the base class object from it. A derived class object always has a base class object inside it.

We first add the `Comparable` to our `Student` class declaration

```
public class Student : Person,Comparable{
```

We also need to add the `new` key word to the `CompareTo` method because it has the same signature as the `CompareTo` method of the `Person` class.

```
public new int CompareTo(object obj)
```

We want `Student.CompareTo(object)` method to hide the inherited `Person.CompareTo(object)` method. The `new` keyword is used to hide the `Person.CompareTo(object)` method. Hiding allows the `Student.CompareTo(object)` method to be used over the `Person.CompareTo(object)` method.

Here is the `Student` class `CompareTo` method:

```
// compare 2 person names
public int CompareTo(Object obj)
{
    if(obj != null)
    {

        Student s = (Student)obj;

        if (s is Student)
        {
            if (base.CompareTo(s) == 0)
                return this.idnum.CompareTo(s.idnum);
            else return base.CompareTo(s);
        }
    }
    return 0;
}
```

In the `Student` is equal method we first check if the `Person` base class is equal, if so we compare the student number, else we just return the person comparison

result. Although we pass a Student variable to the Student equals method the compiler extracts the Person object from it.

To do

Add the student CompareTo method to your Student class, In your Lesson5.java main method, make some persons with different names. Using the CompareTo method print out the results of comparing different and same Person objects, using Console.WriteLine()

Making your Own Interface

ICalculator Interface

Here is an interface that defines methods that can be used in a Calculator. You can put the Calculator interface into a file called ICalculator.cs

```
/*
 * ICalculator interface
 * specifies the method a Calculator should have
 */

interface ICalculator {

    double add(double a, double b);
    double sub(double a, double b);
    double mult(double a, double b);
    double divide(double a, double b);
}
```

A MyCalculator class implementing the ICalculator interface is as follows:

```
/*
 * class MyCalculator implements Calculator interface
 * implements the methods of the Calculator interface
 */

public class MyCalculator: ICalculator{

    public double add(double a, double b){
        return a + b;
    }
    public double sub(double a, double b){
        return a - b;
    }
    public double mult(double a, double b){
        return a * b;
    }
    public double divide(double a, double b){
        return a/b;
    }
}
```

Add a main method to the MyCalculator.cs file. In a loop asking the user to type in 2 numbers and what operation they want, and display the results. You will have to instantiate a MyCalculator object in your main class because the MyCalculator class methods are not static.

```
ICalculator calc = new MyCalculator();

int a = 3;
int b = 5;

int result = calc.add(a,b);
Console.WriteLine(result);
```

LESSON 7 HOMEWORK

Question 1

Add the copy constructor to your **Profession** class and to your derived **Payroll** class from previous lesson 3. The Payroll derived class would have to pass the Payroll object to the Profession class using the super keyword. In your main method of Homework7.cs make a few Professions and Payrolls and print them out, then make a copy of each one using the copy constructor.

Question 2

Add the **equals** and **compare To** methods to your **Profession** and **Payroll** class from Lesson 3. The **Profession** class should compare the profession and/or salary, and the **Payroll** class can compare the bonus and optionally the result from the **Profession** class. In your main method of Homework7.cs make a few **Profession** and **Payroll**'s and print them out, then check if they are equal using the equals method and compare the objects using the **compareTo** method, then print out the results.

Question 3

Make a Interface called **IAnimal** that has a void method **sound()**.

```
interface IAnimal {  
  
    public void sound();  
}
```

Put the interface in a C# file called IAnimal.java.

Make a class called Cat that implements the IAnimal interface. It has a constructor that receives a name, a sound method that prints out “meow” and a toString() method the prints out the name of the animal and type. Put this in a java file called Cat.java.

Make a class called Dog that implements the IAnimal interface. It has a constructor that receives a name, a sound method that prints out “bark” and a toString() method the prints out the name of the animal and type. Put this in a java file called Dog.java.

Make a class called Tiger that implements the IAnimal interface. It has a constructor that receives a name, a sound method that prints out “roar” and a ToString() method the prints out the name of the animal and type. Put this in a java file called Tiger.java.

Make a class of your favorite Animal that implements the IAnimal interface. It has a constructor that receives a name, a sound method that prints out “????” and a ToString() method the prints out the name of the animal and type. Put this in x c# file called Animal.cs file.

Your toString method should return the animal name what kind of animal it is like:

“I am Fluffy and I am a cat”

In your main method of Homework7.cs make the 4 four animals and print out what each animal says by first calling the toString method and then the sound method. You may want to put all the animals in a IAnimal array and print in a loop.

```
IAnimal[] animals = new IAnimal[4];
```

This is possible because an interface can represent any class that implements it.

An example output is:

“I am Fluffy I am a cat”
“meow”

Question 4

Add the **compareTo** methods to your **Profession** and **Payroll** class from Question2 , using the Generic Comparable interface **Comparable<Profession>**. Each will receive a **Profession** and **Payroll** parameter respectively. The **Profession** class should compare the profession and/or salary and the **Payroll** class can compare the bonus and optionally the result from the **Profession** class. The derived class Payroll must implement **Comparable<Profession>** not **Comparable<Payroll>** to avoid a name clash. Check if the homework 6 program still work's the same.

LESSON 8 GENERIC Interfaces and Classes

Generics allow a class to have a specified data type when it is declared and instantiated. A Generic class store an object class as it's data type. The object class is the super base class of all classes, and can represent any other class. The object class has very few methods available to do things with, just methods like equals, ToString() etc.

We have already used Generics with the List, HashSet and Dictionary when you specified the data types that the List would use.

```
List<int> list1 = new List<int>();
```

This means the List will hold an **int** data type that allows you to use int numbers.

You define a Generic class like this:

```
access_modifier class_name<T>  
{  
    T variable name;  
}
```

T can represent any class data type that the generic class will use.

You instantiate a generic class the same way as any other class except you must specify the data type it will use in diamond <> brackets like this

```
List<Integer> list1 = new List<Integer>();
```

Here is a simple Generic class you can put into a class file called TItem.java

// Simple Generic class to hold a generic data type

```
public class TItem<T>  
{  
    private T data; // generic variable  
  
    // construct TItem  
    public TItem(T data)  
    {  
        this.data = data;  
    }  
  
    // Data property  
    public T Data  
    {  
        get { return data;}  
        set { data = value;}  
    }  
  
    // return a information string  
    public override string ToString()  
    {  
        return data.ToString();  
    }  
  
}
```

You instantiate the TItem class like this

```
TItem<int> tItem = new TItem<int>(5);
```

To do:

Instantiate a TItem object with a int data type and any value you like. Call the Data property and print out the return value. Using the Data property assign a

new value to the TItem object like 8. Call the Data property again to print out the return value. Finally print out item with the ToString method.

Adding an Equals method to the TItem class.

We first add Equals method that takes an Object.

```
// return true if equal
public override bool Equals(Object other)
{
    if(other != null)
    {
        if (other is TItem<T>)
        {
            return ((TItem<T>)other).data.Equals(data);
        }
    }
    return false;
}
```

We also need to add a GetHashCode() method

```
// return unique number code
public override int GetHashCode()
{
    return this.data.GetHashCode();
}
```

Next we add the Generic Equals method that just takes a T data type.

```
// return true if equal
public bool Equals(T other)
{
    return data.Equals(other);
}
```

```
}
```

Notice we have 2 equals method's one taking an object and the other one taking a T data type.

```
// return true if equal
public override bool Equals(Object other)
{
    If(other != null)
    {
        if (other is TItem<T>)
        {
            TItem<T> totem = (TItem<T>)other;
            return.data.Equals(other.data);
        }
        return false;
    }
}
```

```
// return true if equal
public bool Equals(TItem<T> titem)
{
    If(titem != null)
        return this.data.equals(titem.data);
    else return false;
}
```

The first one overrides the Equals method of the Object class the other one just takes a TItem object for convenience but does not override the Equals method in the object class type. Probably the second Equals methods are more convenient to use.

To do:

Make two `int Titem` objects with same or different values Print out if the two item objects are equal using the `Equals` method.

Next call the `equals` method with a `int` data type. The second `Equals` method that takes a `T` data type should be called.

Repeat the above with different data types like `Double`, `string` even a `Person`.

Equals method that takes a Titem class.

We can make another `equals` method the receives a `Titem` object. This method will just be called from a `Titem` object. The object `Equals` method will now be called with non `Titem` objects.

```
// compare Titem objects
public bool Equals(Titem<T> other)
{
    If(other != null)
        return other.Equals(other.data);
    else return false;
}
```

Todo

Add the `Titem Equals` method too your `Titem` generic class. Make two `int Titem` objects with same or different values. Print out if the two item objects are equal using the `Equals` method.

GENERIC INTERFACES

Interfaces can also be Generic to. The `IComparator` interface also has generic version

```
public interface IComparable<T>
{
    int CompareTo(T obj);
}
```

```
}
```

In the class declaration we would add the T data type to the IComparable

```
class class_name: interface_name<T>
```

Adding the Generic Interface to the Person class

With the use of Generics we can add a CompareTo method to our Person class that takes a Person rather than an Object.

```
// compare 2 person names  
public int CompareTo(Person p)  
{  
    if(p != null)  
        return String.Compare(this.name, p.name);  
    else return 0;  
}
```

Our Person class definition would then look like this:

```
public class Person: IComparable<Person>
```

to do:

In your Person class add the Person type to your IComparable class declaration. Add this CompareTo method to your Person class. Then in your Lesson6.cs file try it out. The Person compare to method should now be called rather than the object CompareTo method.

Adding the Generic Interface to a Derived class

To add a Generic Interface to a Derived class we just add the <T> to the interface name.

```
class derived_class_name: base_class_name, interface_name<T>
```

Adding the Generic Interface to the Student class

We just add the Student type to the IComparable tag.

```
public class Student : Person, IComparable<Student>
```

Our Student Equals method will now look like this:

```
// compare 2 students  
public int CompareTo(Student s)  
{  
    if (s != 0)  
    {  
        if (base.CompareTo(s) == 0)  
            return this.idnum.CompareTo(s.idnum);  
        else return base.CompareTo(s);  
    }  
    else return 0;  
}
```

to do:

In your Student class add the Student type to your IComparable class declaration. Add this CompareTo method to your Student class. Then in your Lesson6.cs file try it out. The Student compare to method should now be called rather than the object CompareTo method.

Generic class implementing the Comparable interface

A Generic class implementing the Comparable interface would be like this

```
public class MyGenericClass<T>:Comparable<T> {
```

This does not work properly because the compiler does not know the data type of T at compile time. So we need to tell the compiler in advance what data type T is using the **where** keyword and a specified data type.

```
public class MyGenericClass <T>:IComparable<T> where T: IComparable<T>
```

Our CompareTo function will now look like this.

```
public int CompareTo(T data)
{
    if (this.data.CompareTo(data) < 0) return -1;
    if (this.data.CompareTo(data) > 0) return -1;
    else return 0;
}
```

Todo

Have your TItem class implement the IComparable interface just to compare the T data only.

```
public class TItem<T>: IComparable<T> where T : IComparable<T>
```

Add a CompareTo method to your TItem class that will compare T data.

```
public int CompareTo(T data)
{
    // compare the 2 data elements and return the result
    if (this.data.CompareTo(data) < 0) return -1;
    if (this.data.CompareTo(data) > 0) return -1;
    else return 0;
}
```

In your Lesson7.cs file use your TItem objects to compare values. Print out the results using Console.WriteLine statements.

If you want to compare TItem objects you can do something like this.

```
public class TItem<T> : IComparable<TItem<T>> where T : IComparable<T>
```

The CompareTo method would look like this:

```
public int CompareTo(TItem<T> item)  
{  
    If (item != 0)  
    {  
        if (this.data.CompareTo(item.data) < 0) return -1;  
        if (this.data.CompareTo(item.data) > 0) return -1;  
        else return 0;  
    }  
    else return 0;  
  
}
```

Add the CompareTo method to TItem class to compare 2 TItem Objects.
In your Lesson6.cs file use your TItem objects to compare other TItem objects.
Print out the results.

GENERIC CALCULATOR

We can make our Calculator Generic too. The Generic calculator will take any data type. We first define a Generic Calculator Interface.

```
/*  
* Generic ICalculator interface  
* specifies the methods a ICalculator should have  
*/
```

```
public interface TCalculator<T> {  
  
    public T add(T a, T b);  
    public T sub(T a, T b);  
  
}
```

```

    public T mult(T a, T b);
    public T divide(T a, T b);
}

```

We can also make our MyCalculator class to be generic and implement the generic TCalculator interface. By making our calculator generic then we can use any data type.

A Generic is a data type with no methods except ToString and Equals() methods. There are no operators available like + - < > etc. The problem is the compiler does not know what a T is yet because T data type is specified at **run time** not **compile time**. There are many solutions to this problem. Making separate int, double char calculators etc.

We will present one that will let the compiler which method to call depending on the generic data type specified.

We will just present the solution for the some of the methods, you will do the rest on your own.

We have made separate add, sub, mult and divide functions for the data types we are interested in. In run time these methods would be called over the generic ones

```

// generic calculator
public class MyTCalculator<T> : TCalculator<T>
{
    public int add(int a, int b)
    {
        return a + b;
    }

    public double add(double a, double b)
    {
        return a + b;
    }

    public int sub(int a, int b)
    {

```



```

    return a - b;
}

public double sub(double a, double b)
{
    return a - b;
}

public int mult(int a, int b)
{
    return a * b;
}

public double mult(double a, double b)
{
    return a * b;
}

public int divide(int a, int b)
{
    return a / b;
}

public double divide(double a, double b)
{
    return a / b;
}

public T add(T a, T b)
{
    return add(a, b);
}
public T sub(T a, T b)
{
    return sub(a, b);
}
public T mult(T a, T b)
{
    return mult(a, b);
}
public T divide(T a, T b)

```

```

    {
        return divide(a, b);
    }
}

```

When the calculator runs the correct method is called for the data type specified.

To do:

To run the MyTCalculator you will do something like this:

```

MyTCalculator<int> tcalc = new MyTCalculator<int>();
int x = tcalc.add(3, 4);
Console.WriteLine(x);

```

7

Try running different types of numbers int and double. Add some more methods to handle char and strings.

LESSON8 Question 1

Make generic Animal class called TAnimal that can hold any type of the animals Cat, Dog, Tiger, etc. from Question 3 that implements the **IAnimal** interface also from Question 3

```

public class TAnimal<T extends IAnimal> implements IAnimal

```

The TAnimal class should have a generic instance variable T animal and T Animal getters and setters. We need to use **<T extends IAnimal>** because we want the compiler to know that T will represent a IAnimal so we can call the sound method from the animal instance variable. The IAnimal interface specifies the sound method() print out a animal sound.

```
interface IAnimal {  
    public void sound();  
}
```

In the main method of Homework 7.java make some animals and print them out with a sound like this:

```
TAnimal<Cat> cat = new TAnimal<Cat>(new Cat("Tom"));  
cat.sound();
```

Make an array of Generic TAnimals. When you make a Generic array you do not specify the data type but have to suppress any warnings.

```
TAnimal[] animals = new TAnimal[3];
```

Put some IAnimals in the animals array. You can use the animals you made above or make some new ones

```
animals[0] = new TAnimal<Cat>(new Cat("tom"));  
animals[1] = new TAnimal<Dog>(new Dog("bill"));  
animals[2] = new TAnimal<Tiger>(new Tiger("george"));
```

Print out the animal and then call the sound method.

```
for(int i=0;i<animals.length;i++)  
    {  
        System.out.println(animals[i].toString());  
        animals[i].sound();  
    }
```

You should get something like this:

```
"I am Fluffy I am a cat"  
"meow"
```

LESSON 8 Question 2

Make a generic TIAAnimal interface that has an additional method called talk that takes a Generic parameter that let's two animals to talk to each other. TIAAnimal extends IAnimal interface from Question 5.

```
interface TIAAnimal<T> extends IAnimal{  
  
    public void talk(T animal2);  
}
```

where the IAnimal interface from Question 3 is:

```
interface IAnimal {  
  
    public void sound();  
}
```

Make a generic class called TalkingTAnimal that extends TAnimal that can hold any type of the animals Cat, Dog, Tiger, etc. from Question 3 that implements the IAnimal interface.

```
public class TalkingTAnimal<T extends IAnimal> extends TAnimal<T>  
    implements TIAAnimal<TalkingTAnimal<T>>
```

<T extends IAnimal> means we want the compiler that **T** represents a **IAnimal** so we can call the sound method from it.

extends TAnimal<T> means we want to use all the methods from the super class TAnimal

TIAAnimal<TalkingTAnimal<T>> means we want the talk method parameter to be a TalkingTAnimal of Animal type **T**, so we can call the sound from it.

The TalkingTAnimal class should have a generic instance variable **T animal** and T getters and setters.

The talk method just lets each animal talk to each other by each calling the sound method(). It could say

“meow”

“ my name is Tony”

You need to make two talk methods, one that receives a T and a the other one receives TalkingTAnimal<T> .

The T would represent a IAnimal like Cat or Dog where as the TalkingTAnimal<T> would represent a TalkingAnimal that could be a Cat or Dog.

```
public void talk(T animal2)  
    {  
        sound();  
        animal2.sound();  
    }
```

```
public void talk(TalkingTAnimal<T> animal2)  
    {  
        sound();  
        animal2.sound();  
    }
```

In the main method of Homework 7.java make some animals and print them out talking to each other like this:

```
TalkingTAnimal<Cat> cat = new TalkingT<Cat>(new Cat("tom"));  
TalkingTAnimal<Cat> cat2= new TalkingT<Cat>(new Cat("sue"));  
  
cat.talk(cat2);  
cat2.talk(cat);
```

You should get something like this

```
Meow  
My name is tom and I am a Cat  
Meow  
My name is sue and I am a Cat
```

Lesson 9 C# OVERLOADED OPERATOR METHODS

Overloaded Operator methods let you use operators like +, -, *, / etc. for methods in your objects. The only draw back they must be declared static. Not all operators can be overloaded. The following table indicates which operators can be overloaded

Operators	Overloadability
+, -, !, ~, ++, --, true, false	These unary operators can be overloaded.
+, -, *, /, %, &, , ^, <<, >>	These binary operators can be overloaded
==, !=, <, >, <=, >=	The comparison operators can be overloaded
&&,	The conditional logical operators cannot be overloaded, but they are evaluated using & and , which can be overloaded.
[]	The array indexing operator cannot be overloaded, but you can define indexers.
+=, -=, *=, /=, %=, &=, =, ^=, <<=, >>=	Assignment operators cannot be overloaded, but +=, for example, is evaluated using +, which can be overloaded.

A overloaded operator method is public and static and uses the keyword operator and one of the above listed operators.

public static MyClass operator +(MyClass a, MyClass b)

We can make an MyArray class than stores values that can has an overloaded operator + methods to add two MyArray objects together.

```

// add two arrays
public static MyArray operator +(MyArray a, MyArray b)
{
    MyArray c = new MyArray(a.Count);

    for (int i = 0; i < a.Count; i++)
    {
        c.insert(a.items[i] + b.items[i]);
    }

    return c;
}

```

Our MyArray class will look like this.

```

public class MyArray
{
    private double[] items;
    private int numItems = 0;

    // make set of size
    public MyArray(int size)
    {
        items = new double[size];
    }

    // insert item to set
    public void insert(double item)
    {
        items[numItems++] = item;
    }

    // retrieve item from set

```

```

public double retrieve(int i)
{
    return items[i];
}

// property to return number of items in this set
public int Count
{
    get
    {
        return numItems;
    }
}

// add two arrays
public static MyArray operator +(MyArray a, MyArray b)
{
    MyArray c = new MyArray(a.Count);

    for (int i = 0; i < a.Count; i++)
    {
        c.insert(a.items[i] + b.items[i]);
    }
    return c;
}

// print out set
public override string ToString()
{
    String sout = "[ ";
    for (int i = 0; i < numItems; i++)
    {
        sout += items[i] + " ";
    }

    sout += "];";
}

```



```
        return sout;
    }
}
```

Todo:

Put the MyArray class in a file called MyArray.cs, then in your Lesson6.cs file inside your Main method make a small test program like this.

```
public static void Main(string[] args)
{
    MyArray a = new MyArray(5);
    a.insert(1);
    a.insert(2);
    a.insert(3);

    Console.WriteLine(a.ToString());

    MyArray b = new MyArray(5);
    b.insert(4);
    b.insert(5);
    b.insert(6);

    Console.WriteLine(b.ToString());
    MyArray c = a + b;
    Console.WriteLine(c.ToString());
}
```

Your output will look like this:

```
[ 1 2 3 ]
[ 4 5 6 ]
[ 5 7 9 ]
```

Once you got this working make operator overloaded methods for -, * and /

INDEXERS

Indexers allow your object to use index square bracket[] just like as an array. It is assumed your object has arrays that you want to be accessed.

An indexer returns a value and uses this keyword and square brackets enclosing an array index. The index is used to assign or retrieve values from the object's internal array using the get and set accessors like this:

```
public int this[int index]
{
    get
        {
            return a[index];
        }

    set
        {
            a[index] = value;
        }
}
```

We can add indexers to our MyArray class as follows:

```
public double this[int index]
{
    get
    {
        return items[index];
    }
}
```

```

    }

    set
    {
        // adjust num items
        If(numItems < index)
            numItems = index + 1;
        items[index] = value;
    }

}

```

In your Lesson6.cs main function we can add some test code.

```

a[0]=1;
a[1]=2;
a[2]=3;

for(int i=0;i<a.Count;i++)
{
    Console.WriteLine(a[i]);
}

```

Our output would now be like this:

```

[ 1 2 3 ]
[ 4 5 6 ]
[ 5 7 9 ]
1
2
3

```

Todo:

Add the indexers to your myArray class and the indexer test code to your Lesson 6 main method.

Lesson9 Homework Question 1

Make a Calculator using Operators +,-,*,/,%,^

Lesson9 Homework Question 2

Make a generic Calculator using Operators +,-,*,/,%,^

LESSON10 File Access

File Access

C# has extensive file objects for reading and writing to file. We concentrate on the most used. You need to have **using System.IO** at the top of your Lesson10.cs to use the file stream classes. A **try** block is used to catch any errors like file not found, corrupt file etc. The catch statement is used to catch the errors. When a program encounters abnormal operation, an Exception is thrown. The try block initiates the operation so that the catch block can catch the exception and report the exception. Failure to include a try catch block would result in immediate program termination.

Todo:

Make a file call Lesson10.cs to try out all the following code.
You need to have **using System.IO** at the top of your Lesson10.cs.

Write character to a file

The **StreamWriter** class is used to write characters one by one sequentially to a file using the **Write** method.

```
// write characters to a file
try
{
```

```

// write chars to a file
StreamWriter sw = new StreamWriter("data.txt");

string s = "Hello";

for (int i = 0; i < s.Length; i++)
{
    char c = s[i];
    sw.Write(c);
}

sw.Write('\n');
sw.Close();
}

catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}

```

Read characters from a file

The **StreamReader** class is used to read characters one by one sequentially from a file.

```

// read characters to file
try
{
    StreamReader sr = new StreamReader("data.txt");
    int ch = sr.Read();

    while (ch != -1)
    {

```

```
    Console.Write((char)ch);
    ch = sr.Read(); // don't forget to read again
}
```

```
sr.Close();
}
```

```
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
```

write lines to a file

The **StreamWriter** class is used to write lines one by one sequentially to a file.

```
// write lines to a file
try
{
    StreamWriter sw = new StreamWriter("data.txt");
    sw.WriteLine("hello");
    sw.WriteLine("goodbye");
    sw.Close();
}

catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
```

Hello goodbye

Read line by line from a file

The **StreamReader** class is used to read lines one by one sequentially from a file.

```
try
{
    // Open the file for read
    StreamReader sr = new StreamReader("data.txt");
    string line = sr.ReadLine();
    // for each line in file
    while (line != null)
    {
        Console.WriteLine(line);
        line = sr.ReadLine();
    }

    // close the file
    sr.Close();
}

catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
```

```
happy days are here again
we will see you tomorrow
hello
hello
```

Append line to end of file

We can also write lines to the end of a file (append) using the **StreamWriter** class set to append with the true argument.

```
// write lines to end of a file
```

```
try
{
    StreamWriter sw2 = new StreamWriter("data.txt", true);
    sw2.Write("tomorrow");
    sw2.Close();
}

catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
```

Todo

Read the file and print it out again.
You should get something like this:

```
hello
goodbye
tomorrow
```

write to a csv file

A csv file write lines to a file where all the words are separated by commas.

```
// write lines to a file
try
{
    StreamWriter sw = new StreamWriter("data.csv");
    sw.WriteLine("Happy, days, are, here, again");
    sw.WriteLine("See, you, tomorrow");
    sw.Close();
}
```



```
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
```

Read a csv file.

A csv file is a file where data is stored row by row in columns separated by commas. The **StreamReader** class is used again to read lines one by one sequentially from a file. We use the **Split** method from the **string** class to separate each line into an array of words. Each word is called a Token.

```
// read csv file
// Open the file for read
try
{
    StreamReader sr = new StreamReader("data.csv");
    string line = sr.ReadLine();

    // read line one at a time
    // till the file is empty
    while (line != null)
    {
        String[] tokens = line.Split(', ');

        for (int i = 0; i < tokens.Length; i++)
        {
            Console.WriteLine(tokens[i] + " ");
        }

        line = sr.ReadLine();
    }
}
```

```
Console.WriteLine("");  
sr.Close();  
}
```

Output token words

```
Happy  
Days  
Are  
Here  
again  
See  
you  
tomorrow
```

Reading and Writing objects to a file

Step1: Make a book class

All objects written to a binary file must be tagged as **[Serializable]**

```
[Serializable]  
public class Book  
{  
    public string Title{get;set;}  
    public string Author{get;set;}  
  
    public Book(String title, String author)  
    {  
        Title = title;  
        Author = author;  
    }  
}
```

```
public String toString()
{
    return "Book " + title + " written by " + author;
}
}
```

Step 2: Make book object and print out

```
// read/write objects
Book book = new Book("Alice in Wonderland", "Lewis Carroll");
Console.WriteLine(book);
```

Book Alice in Wonderland written by Lewis Carroll

Step3: Write Book object to a binary file “book.bin”

We use the `System.Runtime.Serialization.Formatters.Binary.BinaryFormatter` method to write an object to a binary file.

```
//serialize
try
{
    StreamWriter sw3 = new StreamWriter("book.bin");

    var bformatter = new
        System.Runtime.Serialization.Formatters.Binary.BinaryFormatter();

    bformatter.Serialize(sw3.BaseStream, book);

    sw3.Close();
}

catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
```

```
}
```

Step 4: Read book object from binary file “book.bin”

We also use the `System.Runtime.Serialization.Formatters.Binary.BinaryFormatter` method to read an object from a binary file.

```
//deserialize
try
{
    StreamReader sr5 = new StreamReader("book.bin");

    var bformatter2 = new
        System.Runtime.Serialization.Formatters.Binary.BinaryFormatter();

    Book book2 = (Book)bformatter2.Deserialize(sr5.BaseStream);

    sr5.Close();

}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
```

Step 5: print out book read from binary file “book.bin”

```
Console.WriteLine(book2);
```

Book Alice in Wonderland written by Lewis Carroll

LESSON 10 HOMEWORK

Question 1

Write a small 5 line story to a file called “story.txt” then close the file. Then open the story text file that has the small story in it, and count the number of letters, words, sentences and lines. Words are separated by spaces and new lines.

Sentences are separated by periods “.” or other punctuation like “?”.

Lines are separated by ‘\n’. Words may contain numbers and punctuation like apple80 and don’t. Print a report to the screen: the number of letters, words, sentences and lines. Also write the report to a file called report.txt. Open the report file and display the report file lines to the screen. Call your cs file Homework10.cs

Question 2

Write a program that writes out another C# program to a file, then open up the file you wrote that contains the C# program and execute it.

Algorithm:

1. Using StreamWriter write C# statement to a file called “../question2.cs”
2. Write lines to a file with WriteLine statements like:

```
sw.WriteLine("using System;");
sw.WriteLine("public class Question2")
sw.WriteLine("{}");
sw.WriteLine("public static void Main(String[] args)");
sw.WriteLine("{}");
sw.WriteLine("Console.WriteLine(\"I like programming\")");
sw.WriteLine("{}");
sw.WriteLine("{}");
```

you need to alternate between hardcoded double quotes and specified \" double quotes

3. close the file
4. open the Test.cs file in Visula Studio as add an existing item and run the C# file. It should print out

I like programming

You can put your code in your C# file Homework10.cs or make a new C# File called Homework10Question2.cs

LESSON 11 POLYPHORISM and C# Objects

Polymorphism is another powerful concept in Object Oriented Programming. Polymorphisms lets one super class represent many other different types of derived classes and then execute a method for each derived class having the same When each method executed, each method will produce a different

behavior. We will have an Employee super class to represent different kinds of Employees. Managers, Secretaries, Salesman and Workers derived classes.

Make a new C# file called Employee.cs You will need private variables: name, employee ID number and salary. Make constructors, properties and ToString method.

We make our Employee class abstract meaning it can never be instantiated. An abstract class contains abstract methods that have no body. We will need an abstract method to calculate the employee's wages for the week. An abstract method is a method, that is defined in the super class that has a method header definition but no programming statements. The programming statements will be defined in the derived class. This known as method **overriding** since it has the same method name and identical parameter data types. We can also have method **overloading** where we have same method name but different parameter data types.

Here is our Employee super abstract class.

```
/*
 * Employee abstract class
 *
 */

public abstract class Employee {

    private String name;
    private String id;
    private double salary;
    public const int WEEKS_IN_YEAR = 52;

    // construct Employee with name, id and salary
    public Employee(String name,String id,double salary)
    {
        this.name=name;
        this.id=id;
    }
}
```

```

        this.salary = salary;
    }

    // return/assign employee name
    public string Name
    {
        get{ return this.name; }
        set{ this.name=value; }
    }

    // return/assign employee id
    public string ID
    {
        get{ return this.id; }
        set{ this.id = value; }
    }

    // return employee yearly salary
    public double Salary
    {
        get { return this.salary; }
        set { this.salary = value; }
    }

    // calculate weekly pay
    public abstract double weeklyPay();

    // return employee info
    public override string ToString()
    {
        return name + " " + id + " $" + salary;
    }

```



```
}  
}
```

We now need to make the derived classes. You can put them in the same file as the Employee class or in separate files. We will have 4 derived classes. Each derived class will calculate the pay for the week differently, calculated from the yearly salary. Each derived class will calculate the weekly pay separately as follows:

Derived class	How to calculate weekly pay
Manager	Divide yearly salary by number of weeks in year
Secretary	Divide yearly salary by number of weeks in year plus \$100 bonus
Salesman	Divide yearly salary by number of weeks in year plus sales Commission rate
Worker	Divide yearly salary by number of weeks plus any overtime time and a half

It is probably best to put each derived class in a separate file. This way it makes thing easier these classes to use in other projects.

```
/*  
 * Manager class  
 * Manager.cs  
 */  
  
public class Manager:Employee {  
  
    // construct Manager with name, id and salary  
    public Manager(string name,string id,double salary)  
    :base(name,id,salary){  
    }  
  
    // calculate weekly pay  
    public override double weeklyPay()  
    {
```

```

        double pay = Salary / WEEKS_IN_YEAR;
        return pay;
    }

    // return manager info
    public override string ToString()
    {
        return "Manager " + base.ToString();
    }
}

/*
 * Secretary derived class
 * Secretary.cs
 */

public class Secretary:Employee {

    // Constants
    public const double BONUS = 100;

    // construct Secretary with name, id, salary
    public Secretary(string name,string id,double salary)
    : base(name, id, salary)
    {
    }

    // calculate weekly pay
    public override double weeklyPay()
    {
        double pay = Salary/ WEEKS_IN_YEAR + BONUS;
        return pay;
    }
    // return employee info
    public override string ToString()

```

```

        {
            return "Secretary " + base.ToString();
        }
    }

    /*
    * Salesman derived class
    * Salesman.cs
    */

    public class Salesman:Employee {

        // Constants
        public const double COMMISSION_RATE = .25;

        // weekly sales
        private double sales;

        // construct Salesman with name, id, salary and sales
        public Salesman(string name,string id,double salary, double sales)
        : base(name, id, salary)
        {
            this.sales = sales;
        }

        // calculate weekly pay
        public override double weeklyPay()
        {
            double pay = Salary/WEEKS_IN_YEAR + sales * COMMISSION_RATE;
            return pay;
        }

        // return employee info
        public override string ToString()

```

```

        {
            return "Salesman " + base.ToString() + " Sales: " + sales;
        }
    }

    /*
    * Worker derived class
    * Worker.cs
    */
    public class Worker:Employee {

        // Constants
        public const double OVERTIME_RATE = 1.5;
        public const int HOURS_IN_WEEK = 40;

        // hours overtime
        private int overtime;

        // construct Secretary with name, id, salary
        public Worker(string name,string id,double salary,      int overtime)
        :base(name, id, salary)
        {
            this.overtime = overtime;
        }

        // calculate weekly pay
        public override double weeklyPay()
        {
            double pay_rate = Salary/ WEEKS_IN_YEAR / HOURS_IN_WEEK;
            double pay = Salary/ WEEKS_IN_YEAR
                + overtime * pay_rate * OVERTIME_RATE;
            return pay;
        }
        // return employee info
        public override string ToString()

```

```

    {
        return "Worker " + base.ToString();
    }
}

```

Polymorphism is like a giant else-if statement. If it is a Salesman derived class object, then calculate weekly wage using sales and commission.

Our first step is to make an array to hold Employee derived objects.

```

// make an array of employees
Employee[] employees = new Employee[4];

```

Next, we will add with the derived objects to the Array. Each derived object gets a name, employee id, yearly salary, the salesman gets the sales for the week and the worker get the number of over time hours for the week.

```

// fill array with derived objects
employees[0] = new Manager("Tom Smith", "M1234", 100000);
employees[1] = new Secretary("Mary Smith", "S5678", 40000);
employees[2] = new Salesman("Bob Smith", "SM1111", 20000, 10000);
employees[3] = new Worker("Joe Smith", "W2222", 30000, 5);

```

Next, we loop through the array printing out the employee info and the calculated weekly pay. Notice the weekly pay is different for each employee type, this is what we want, automatic selection. This is polymorphism in action.

```

// loop through employee array
// print out employee info
// calculate weekly pay.

```

```

foreach (Employee emp in employees)

```

```

{
    // print out employee info
    Console.WriteLine(e.ToString());

    // calculate weekly pay
    double pay = e.weeklyPay();

    // print weekly pay
    Console.WriteLine("My weekly pay is: $" + pay.ToString("c2"));
}

```

To do:

Add an abstract **raise** method to the Employee class to give each employee different types of raises. This method will return a double. Some could be percentages, others can be a fixed value, how many sales or how many over time hours worked..

In another loop in the main method give each employee a raise and then recalculate their weekly pay and print out the results. Use a for each loop.

Abstract classes implementing interfaces

We can make an abstract class called **Animal** that implements a **IAnimal** interface. The IAnimal interface contains the sound method, what sound a animal makes.

```

public interface IAnimal
{
    string sound();
}

```

The abstract Animal class will implement the IAnimal interface. The sound method specified in the IAnimal interface will be implemented in the classes derived from the abstract Animal class. The abstract Animal class will also have an abstract method called getType that will return the type of Animal. The abstract method getType() will also be implemented by the derived class of the Animal class.

```

public abstract class Animal:IAnimal
{

```

```

    public string Name{get;set};

    public Animal(string name)
    {
        Name = name;
    }

    public abstract string sound();

    public abstract string getType();

    public override string ToString()
    {
        return "my name is: " + name;
    }
}

```

We then can make a derived Cat class that inherits the Animal class.

```

public class Cat:Animal
{
    public Cat(string name):base(name)
    {
    }

    public override string sound()
    {
        return "meow";
    }

    public override string getType()
    {
        return "cat";
    }

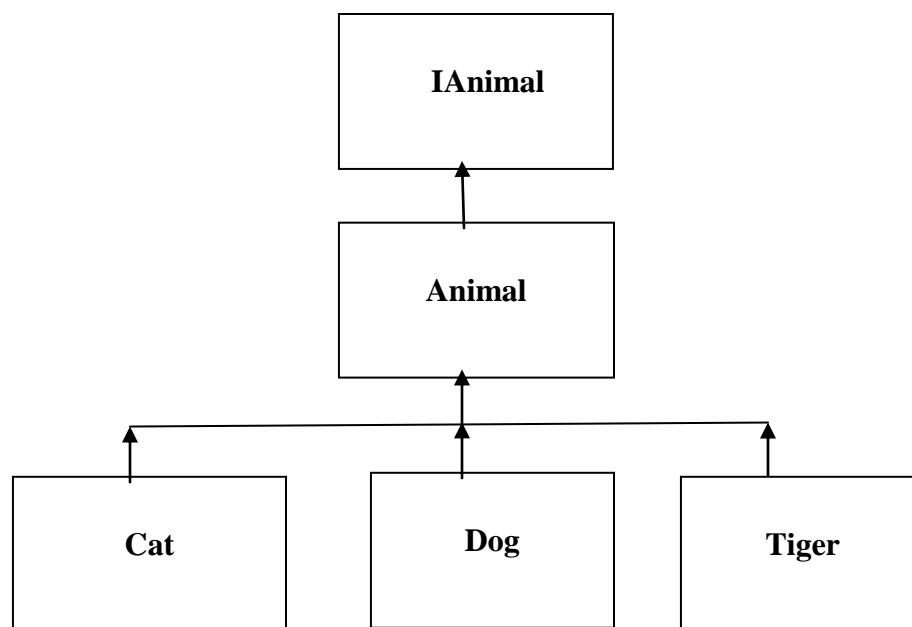
    public override string ToString()
    {
        return "I am a cat " + base.ToString();
    }
}

```

To do:

Make derived classes Dog and Tiger, and in a main method make an array of IAnimal interfaces and print out what animal it is what name, type of animal and what sound it makes in a foreach loop. Put the main method in a java class called Animals and in a file called Animals.java.

```
IAnimal[] animals = new IAnimal[3];  
animals[0] = new Cat("Toby");
```



C# BUILT IN CLASSES AND OBJECTS.

C# has lots of built in objects, meaning classes already written for you that you can use right away. We cannot cover them all, but we can cover a few of them. We have already covered the String object , all of these object have lots of methods, to do many different operations. The Math class is used allot for

mathematical calculations. The Math class is not considered an Object but just a class because all its methods are static.

Here are some of the math class common used fields and methods

Following are the fields for Math class

E – the base of the natural logarithms.

PI – the ratio of the circumference of a circle to its diameter.

Here are the Math Class methods

Method	Description
static double Abs(double a)	returns the absolute value of a double value.
static double Acos(double a)	returns the arc cosine of a value; the returned angle is in the range 0.0 through pi
static double Asin(double a)	returns the arc sine of a value; the returned angle is in the range -pi/2 through pi/2.
static double Atan(double a)	returns the arc tangent of a value; the returned angle is in the range -pi/2 through pi/2
static double Atan2(double y, double x)	returns the angle theta from the conversion of rectangular coordinates (x, y) to polar coordinates (r, theta).
static double Ceil(double a)	returns the smallest double value that is greater than or equal to the argument and is equal to a mathematical integer.
static double Cos(double a)	returns the trigonometric cosine of an angle
static double Floor(double a)	returns the largest double value that is less than or equal to the argument and is equal to a mathematical integer
static double Log(double a)	returns the natural logarithm (base e) of a double value.
static double Log10(double a)	returns the base 10 logarithm of a double value
static double Max(double a, double b)	returns the greater of two double values
static double Min(double a, double b)	returns the smaller of two double values
static double Pow(double a, double b)	returns the value of the first argument raised to the power of the second argument
static long Round(double a)	returns the double value that is closest in value to the argument and is equal to a mathematical

	integer
static double Sqrt(double a)	returns the correctly rounded positive square root of a double value.
static double Tan(double a)	returns the trigonometric tangent of an angle
static double ToDegrees(double angrad)	converts an angle measured in radians to an approximately equivalent angle measured in degrees
static double ToRadians(double angdeg)	converts an angle measured in degrees to an approximately equivalent angle measured in radians.

Random Class

The Random class is also used to generate Random numbers. The Random class has many methods to generate Random numbers. You must instantiate a Random object before you can use it because the Random class methods are not static.

```
Random rgen = new Random();
```

You can generate double random number using the NextDouble method

```
double d = rgen.NextDouble();
```

This generates a double number between 0 and 1.0 like 0.6549805947125389

You can generate integer random number using the Next method. This method allows you to specify the upper bound -1

```
int r = rgen.Next(10)
```

Generates a random number between 0 and 9 like 7

You can also generate a random number with a lower and upper bound-1

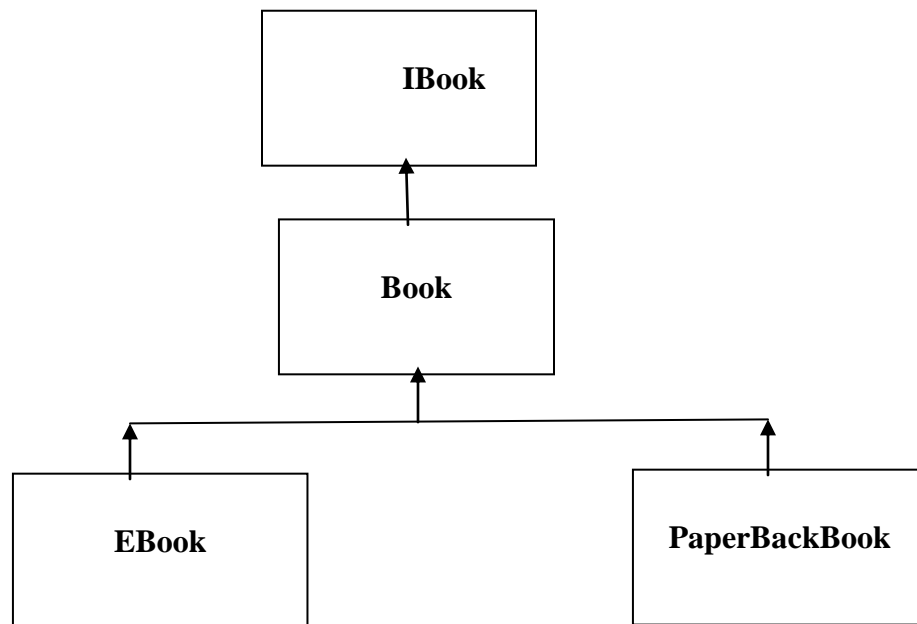
```
int r2 = rgen.Next(5,10);  
Console.WriteLine(r2);
```

Generates a random number between 5 and 9 like 8

LESSON 11 HOMEWORK

Question 1

Make a book interface call **IBook** that has a method to calculateTotalCost to purchase a book. Make a Abstract Book class called **Book** that implements the **IBook** interface and has the abstract method calculateTax and calculateShipping. A Book has a title, cost and a toString() method. Make derived classes **EBook** and **PaperBackBook**. A EBook has no shipping cost but a PaperBackBook has a shipping cost of \$2. A EBook has 10% tax rate but a PaperBackBook has no tax. Use constants of variables passed to each constructor. Make a class called Homework11 with a main method. In the main method make an array of IBook interfaces. Populate the array with one EBook and one PaperBook. In a loop print out the book's detail, calculate the shipping and tax and add to the cost as final total cost. Print out the cost, tax, shipping charges and final cost using the calculateTotalCost method. Put every thing in one file called Homework11.cs



LESSON 12 RECURSION

When a function calls itself it is known as **recursion**. Recursion is analogous to a while loop. Most while loop statements can be converted to recursion, most recursion can also be converted back to a while loop.

The simplest recursion is a function calling itself printing out a message.

```
public static void printMessage()  
{  
    Console.WriteLine("I like programming");  
    printMessage();  
}
```

```
I like programming  
I like programming  
I like programming  
I like programming  
I like programming  
...
```

Unfortunately this program will run forever, so you will need to stop the program somehow while it is running.

We can add a counter **n** to it so it can terminate at some point.

```
public static void printMessage(int n)  
{  
    if(n > 0)  
    {  
        Console.WriteLine ("I like programming");  
        PrintMessage(n-1);  
    }  
}
```

Now the program will print the message **n** times

Every time the `printMessage` function is called **n** decrements by 1. When **n** is 0 the recursion stops. You may place the statement `Console.WriteLine("I like programming\n")` before or after the recursive call. If you put it before than the message is printed first before each recursive call.

If you put after than the message is printed after all the recursive calls are made. There is quite a difference in program execution.

The operation is very similar to the following while loop:

```
n = 5  
while(n > 0)
```

```
{  
    Console.WriteLine ("I like programming\n");  
    n--;  
}
```

You should now run the recursion function

You would call the function like this:

```
printMessage(5);
```

It will print I like programming 5 times.

```
I like programming  
I like programming  
I like programming  
I like programming  
I like programming
```

Recursion is quite powerful, a few lines of code can do so much.

Our next example will count of numbers between 1 and n. This example may be more difficult to understand, since recursion seems to work like magic, and operation runs in invisible to the programmer.

```
public static int countn(int n)  
{  
    if(n == 0)  
    {  
        return 0;  
    }  
    else  
    {  
        return countn(n-1) + 1;  
    }  
}
```

count(5) would return 5 because $1 + 1 + 1 + 1 + 1 = 5$

You can run it in a program like this:

```
Console.WriteLine (count(5) ); // 5
```

When $(n == 0)$ this is known as the base case. When $n == 0$ the recursion stops and 0 is returned to the last recursive call. Otherwise the **countn** function is called and n is decremented by 1.

It works like this:

```
main calls countn(5) with n = 5
countn(5) calls countn(4) with n=4
countn(4) calls countn(3) with n=3
countn(3) calls countn(2) with n = 2
countn(2) calls countn(1) with n = 1
countn(1) calls countn(0) with n = 0
```

```
countn(0) returns 0 to count(1) since n == 0
countn(1) add's 1 to the return value 0 and then returns 1 (0 + 1) to count(2)
countn(2) add's 1 to the return value 1 and then returns 2 (1 + 1) to count(3)
countn(3) add's 1 to the return value 2 and then returns 3 (2 + 1) to count(4)
countn(4) add's 1 to the return value 3 and then returns 4 (3 + 1) to count(5)
countn(5) add's 1 to the return value 4 and then returns 5 (4 + 1) to main()
```

```
main() receives 5 from count(5) and prints out 5
```

The statement **return countn(n-1) + 1** is used to call the function recursively and also acts as a place holder for the value returned by the called function **countn**. The returned value is then added to 1 and then returned.

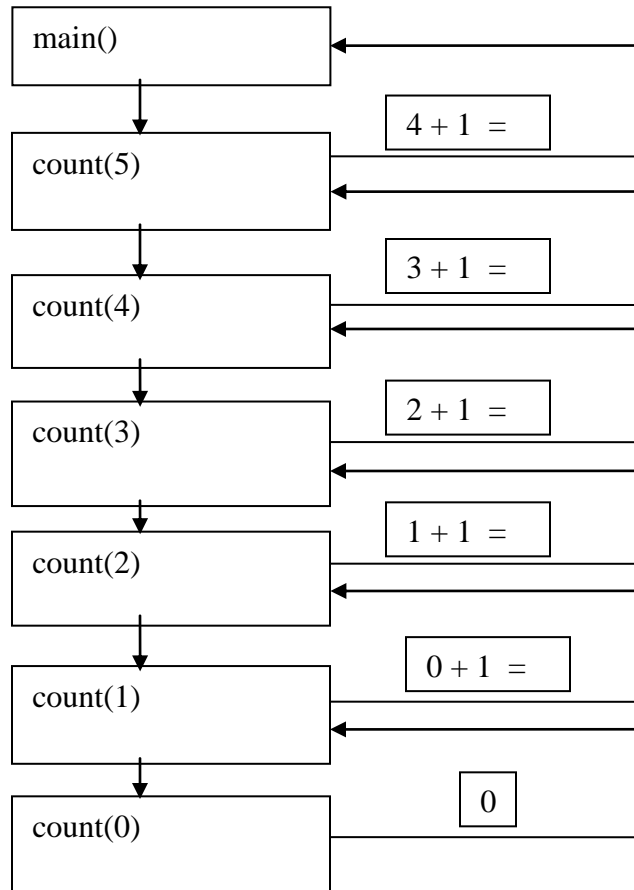
The value returned from the recursive **countn** function is the previous value and the 1 is the present value to be added to the previous value.

We could rewrite the recursive part as follows:

```
int x = countn(n-1);  
return x + 1;
```

x will now receive the return value from the **countn** recursive function call and 1 will be added to the return value and this new value will be returned to the caller.

If you can understand the above then you understand recursion. If you cannot then maybe the following diagram will help you understand.



You probably don't need to understand how recursion works right away. Sometime you just need to accept things for now then understand later. One day it will hit you when you are thinking about something else.

Basically recursion works like this:

For every recursive function call the parameter and local variables are stored.

Technically they are stored in temporary memory called a stack.

Every time the function returns the previous numbers that were stored are restored and now become the current number, to be used to do a calculation. The numbers are restored in **reverse** order.

Function call/ return	N
call count(n-1)	5
call count(n-1)	4

call count(n-1)	3
call count(n-1)	2
call count(n-1)	1
count(n-1) returns 0	0
count(n-1) returns 0 + 1	1
count(n-1) returns 1 + 1	2
count(n-1) returns 2 + 1	3
count(n-1) returns 3 + 1	4
count(n-1) returns 4 + 1	5

The thing to remember about recursion is it always return's back where it is called. A recursive function call behaves the same way as a non-recursive function, program execution resumes right after the recursive function call.

Here are some more recursive function examples:

```
// Sum of numbers between 1 to n
public static int sumn(int n)
{
    if(n ==0)
    {
        return 0;
    }
    else
    {
        return sumn(n-1) + n;
    }
}
sumn(5) would return 15
```

You can run it in a program like this:

Console.WriteLine (sumn(5)); // 15

It works similar to countn instead of adding 1 its adds n's.

$$0+1+2+3+4+5 = 15$$

Our counter n serves 2 purposes a recursive counter and a number to add.

The previous value is returned from the recursive sumn function and the present value is n. The value n is restored in reverse order that is saved in. Every time the recursive sumn function is called it is saved. Every time the recursive sumn function is returned n is restored in traverse order.

Main calls sumn(5)

N	Call sumn(n-1)	Previous	present	Calculated	Returned value
5	Call sumn(4) (5-1)				
4	Call sumn(3) (4-1)				
3	Call sumn(2) (3-1)				
2	Call sumn(1) (2-1)				
1	Call sumn(0) (1-1)				
0	Return 0				
1	Return sumn(0) +1	0	1	0 + 1	1
2	Return sumn(1)+2	1	2	1+2	3
3	Return sumn(2)+3	3	3	3+3	6
4	Return sumn(3)+4	6	4	6+4	10
5	Return sumn(4)+5	10	5	10+5	15 to main

Main receives 15 from sum(5)

The sumn function just returns the value 0 when n is 0 and other times just calls itself then adds the value n to the returned value. Basically sumn is just calling itself and adding n to the returned values. For each recursive call sumn(n-1) is called first after each recursive call the n is added and a value is returned,. N is added in reverse order because sumn(n-1) is called first n times.

Multiply numbers 1 to n (factorial n)

We can also make a **multn** function which multiples n rather than adding n. This is basically factorial n.

```
public static int multn(int n)
{
```

```

if(n ==0)
{
    return 1;
}

else
{
    return multn(n-1) * n;
}
}

```

multn(5) would return 120

since $1*2*3*4*5 = 120$

Our base case returns 1 rather than 0 or else our result would be 0;

You can run it in a program like this:

```
Console.WriteLine (multn(5) ); // 120
```

Power x^n

Another example is to calculate the power of a number x^n
 In this case we need a base parameter b and an exponent parameter n .

```

public static int pown(int b, int n)
{
    if(n ==0)
    {
        return 1;
    }

    else
    {

```

```

    return pown(b,n-1) * b;
}
}

```

pown(2,3) would return 8 because $2*2*2=8$ since $2^3=8$

You can run it in a program like this:

```

Console.WriteLine (pown(2,3) ); // 8

```

Every time a recursive call is made the program stores the local variables in a call stack. Every time recursive call finishes executing, the save local variables disappear and the previous local variables are available. These are the ones present before the recursive function was called. These save variables may now be used in the present calculations.

For the above example $2^3=8$ the call stack would look like this.

			n=0			
			b=2	1		
			n=1	n=1		
			b=2	b=2	2	
	n=4	n=2	n=2	n=2		
	b=2	b=2	b=2	b=2	4	
n=5	n=5	n=3	n=3	n=3	n=3	
b=2	b=2	b=2	b=2	b=2	b=2	8

Every time the recursive function finished executing it returns a value. Each returning value is multiplied by the base b. In the above case the returning values are 1,2,4 and 8

The return value is the value from the previous function multiplied by b (2)

```

return pown(b,n-1) * b;

```

the function first returns 1 then $1 * b = 1 * 2 = 2$ then $2 * 2 = 4$ and finally $4 * 2 = 8$

efficient power x^n

A more efficient version of pown can be made relying on the fact that even n can return $b * b$ rather than just return $* b$ for odd n

```
public static int pown2(int b,int n)
{
    if (n == 0)
    {
        return 1;
    }
    if (n % 2 == 0)
    {
        return pown2(b, n-2) * b * b;
    }

    else
    {
        return pown2(b, n-1) * b;
    }
}
```

Operation is now much more efficient $1 * 2 * 4 = 8$

You can run it in a program like this:

```
Console.WriteLine (pown2(2,3) ); // 8
```

Summing a sequence

Adding up all the numbers in a sequence

$$n \quad (n * (n + 2)) / 2$$

0	0
1	1
2	4
3	7
4	12
5	17

Total:	42

```
public static int seqn(int n)
{
    if(n == 0)
    {
        return 0;
    }

    else
    {
        int x = (n * (n + 1)) / 2;
        Console.WriteLine ( x );
        return x + seqn(n-1);
    }
}
```

seqn(5) would return 35 because $0 + 1 + 4 + 7 + 12 + 17 = 42$

You can run it in a program like this:

```
System.out.println(seqn(5) ); // 42
```

You can print out the sequence by modifying the **seqn** function like this:

```

public static int seqn2(int n)
{
    if(n == 0)
    {
        return 0;
    }
    else
    {
        int x = (n * (n + 1))/ 2;
        Console.WriteLine ( x );
        return x + seqn2(n-1);
    }
}

```

You can run it in a program like this:

```

Console.WriteLine (seqn2(5) );

```

You will get an output like this:

```

17
12
7
4
1
42

```

The sequence printed backwards and the final sum is 42

To do:

Try this formula: $f(n-1) + 2 * (n-1)$

Fibonacci sequence

Recursion is ideal to directly execute recurrence relations like Fibonacci sequence. The Fibonacci numbers are the numbers in the following integer sequence.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,

In mathematical terms, the sequence f_n of Fibonacci numbers is defined by the recurrence relation.

$$f_n = f_{n-1} + f_{n-2}$$

with seed values

$$f_0 = 0 \text{ and } f_1 = 1.$$

A recurrence relation is an equation that defines a sequence based on a rule that gives the next term as a function of the previous term(s).

```
public static int fib(int n)  
{  
    if (n == 0)  
    {  
        return 0;  
    }  
  
    else if (n == 1)  
    {  
        return 1;  
    }  
  
    else  
    {  
        return fib(n-1) + fib(n-2);  
    }  
}
```

Notice The recursive statement is identical to the recurrence relation

fib(10) would return 55 because $21 + 34 = 55$

You can run it in a program like this:

```
Console.WriteLine (fib (10) ); // 55
```

Combinations

We can also calculate combinations using recursion.

Combinations are how many ways can you pick r items from a set of n distinct elements.

Call it nCr (n choose r)

$5C2$ (5 choose 2) would be

Pick two letters from set $S = \{A, B, C, D, E\}$

Answer: $\{A, B\}, \{B, C\}, \{B, D\}, \{B, E\}, \{A, C\}, \{A, D\}, \{A, E\}, \{C, D\}, \{C, E\}, \{D, E\}$

There are 10 ways to choose 2 letters from a set of 5 letters. The combination formula is

$$nCr = \frac{n!}{r!(n-r)!}$$

The Recurrence relation for calculated combinations is:

base cases:

$$nCn = 1$$

$$nC0 = 1$$

recursive case:

$$nCn = n-1Cr + n-1Cr-1 \text{ for } n > r > 0$$

Our recursive function for calculating combinations is:

```
public static int combinations(int n, int r)
{
    if (r == 0 || n == r)
    {
```



```

    return 1;
}

else
{
    return combinations(n-1, r) + combinations(n-1, r-1);
}
}

```

`combinations(5,2)` would return 10

You can run like this:

```
System.out.println(combinations(5,2) ); // 10
```

Print a string out backwards

With recursion printing out a string backwards is easy, it all depends where you put the print statement. If you put **before** the recursive call then the function prints out the characters in reverse, since `n` goes from `n-1` to `0`. If you put the print statement **after** the recursive call then the characters are printed not in reverse since `n` goes from `0` to `n-1`.

```

// reverse a string
public static void printReverse(String s, int n)
{
    if(n == 0)
    {
        Console.WriteLine (s.charAt(0));
    }

    else
    {
        Console.Write(s.charAt(n));
        printReverse(s, n-1);
    }
}

```

You would call the `printReverse` function like this

```

string s = "hello";
printReverse(s, s.length()-1);

```

olleh

Check if a string is a palindrome

A palindrome is a word that is spelled the same forward as well as backwards:
Like "radar" and "racecar"

```
// return true if string is a palindrome otherwise return false
public static boolean isPalindrome( String s, int i, int j)
{
    if (i >= j)
    {
        return true;
    }

    else
    {
        if (s.charAt(i) != s.charAt(j))
            return false;
        else
            return isPalindrome(s,i+1, j-1);
    }
}
```

You would call the **isPalindrome** function like this:

```
string s2 = "radar";
Console.WriteLine (s2 + " " + (isPalindrome(s2, 0,s2.length()-1)) );
```

Radar true

```
string s3 = "apple";
Console.WriteLine (s3 + " " + (isPalindrome(s3, 0,s3.length()-2)));
```

apple false

Permutations

Permutations are how many ways you can rearrange a group of numbers or letters. For example for the string “ABC” the letters can be rearranges as follows:

ABC
ACB
BAC
BCA
CBA
CAB

Basically we are swapping character and then print them out
We start with ABC if we swap B and C we end up with ACB

```
// print permutations of string s  
public static void printPermutations(char[] s, int i, int j)  
{  
    int k;  
    char c;  
  
    // print out permutation  
    if (i == j)  
    {  
        Console.WriteLine ( new String(s) );  
    }  
  
    else  
    {  
        for (k = i; k <= j; k++) {  
  
            // swap i and k  
            c = s[i];  
            s[i] = s[k];  
            s[k] = c;  
  
        }  
    }
```

```

// recursive call
printPermutations(s, i + 1, j);

// put back, swap i and k
c = s[i];
s[i] = s[k];
s[k] = c;
}
}
}

```

You would call the printReverse function like this:

```

char[] ca = {'A','B','C'};
printPermutations(ca, 0,ca.length-1);

```

ABC
ACB
BAC
BCA
CBA
CAB

Combination sets

We have looked at combinations previously where we wrote a function to calculate how many ways you can choose r letters from a set of n letters.

nCr = n choose r

Combinations allow you to pick r letters from set $S = \{A, B, C, D, E\}$

$$n = 5 \quad r = 2 \quad nCr = 5C2$$

Answer: $\{A, B\}, \{B, C\}, \{B, D\}, \{B, E\}, \{A, C\}, \{A, D\}, \{A, E\}, \{C, D\}, \{C, E\}, \{D, E\}$

We are basically filling a seconded character array with all possible letters up to r . Start with ABCDE we would choose AB then AC then AD then AE etc.

We use a loop to traverse the letters starting at n =0, and fill the comb string. When n = r we then print out the letters stored in the comb string

```
public static void printCombinations(char[] s, char[] combs,
    int start, int end, int n, int r)
{
    int i = 0;
    int j = 0;

    // current combination is ready to be printed
    if (n == r)
    {
        for (j = 0; j < r; j++)
        {
            Console.Write(combs[j]);
        }
        Console.WriteLine ("");
        return;
    }

    // replace n with all possible elements.
    for (i = start; i <= end && end - i + 1 >= r - n; i++)
    {
        combs[n] = s[i];
        printCombinations(s, combs, i+1, end, n+1, r);
    }
}
```

You would call the printCombinations function like this:

```
char[] ca2 = {'A','B','C','D','E'};
char[] combs = new char[ca2.length+1] ;
int r = 2
printCombinations(ca2, combs,0,ca2.length-1,0,r);
```

```
AB
AC
AD
AE
BC
BD
BE
CD
CE
DE
```

The difference between combinations and permutations is that in a combination you can have different lengths within the set, where as in permutations they are the same length as the set but different arrangements.

Determinant of a matrix using recursion

In linear algebra, the determinant is a useful value that can be computed from the elements of a square matrix. The determinant of a matrix A is denoted $\det(A)$, $\det A$, or $|A|$

In the case of a 2×2 matrix, the formula for the determinant is:

$$|A| = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$$

For a 3×3 matrix A, and we want the formula for its determinant $|A|$ is

$$|A| = \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = a \begin{vmatrix} e & f \\ h & i \end{vmatrix} - b \begin{vmatrix} d & f \\ g & i \end{vmatrix} + c \begin{vmatrix} d & e \\ g & h \end{vmatrix}$$

$$= aei + bgf - ceg - bdi - afh$$

Each determinant of a 2×2 matrix in this equation is called a "minor" of the matrix A. The same sort of procedure can be used to find the determinant of a 4×4 matrix, the determinant of a 5×5 matrix, and so forth.

Our code actually follows the above formula, calculating and summing the minors.

// calculate determinant of a matrix

```

public static float determinant(float matrix[][], int size)
{
    int c;
    float det=0;
    int sign=1;
    float[][] b=new float[3][3];
    int i,j;
    int m,n;

    // base case
    if(size == 1)
    {
        return (matrix[0][0]);
    }
    else
    {
        det=0;
        for(c=0; c<size; c++)
        {
            m=0;
            n=0;
            for(i=0; i<size; i++)
            {
                for(j=0; j<size; j++)
                {
                    b[i][j] = 0;
                    if(i!=0 && j!=c)
                    {
                        b[m][n] = matrix[i][j];
                        if(n<(size-2))
                        {
                            n++;
                        }

                        else
                        {
                            n=0;
                            m++;
                        }
                    }
                }
            }
        }
    }
}

```

-306

```

        }
    }
}
det = det + sign*(matrix[0][c]*determinant(b,size-1));
sign = -1*sign; // toggle sign
}
}
return (det);
}

```

You call and run the determinant function like this:

```
float m[3][3] = {{6,1,1},{4,-2,5},{2,8,7}};
```

```
Console.WriteLine ("det = " + determinant(m,m.length) );
```

There are many more recursive examples, too numerous to present.
If you do all the following to do questions you will be a recursive expert.

LESSON 12 HOMEWORK

Question 1

Write a recursive function **int addNumbers(int[] a, int n)** adds up all numbers in an array and returns the sum.

Question 2

Write a recursive function **int searchNumber(int[] a, int n, int x)** that searches for a number(x) in an array and return the index (n) of the number if found otherwise returns -1 if not found.

Question 3

Write a recursive function **int largestNumber(int[] a, int n)** that returns the largest number in an array,

Question 4

Write a recursive function **int smallestNumber(int[] a, int n)** that returns the smallest number in an array,

Question 5

Write a recursive function **boolean searchDigit(int number, int x)** that searches for a digit in an positive number and return true if the number is found otherwise returns false if not found.

Question 6

Write a recursive function called **int sumDigits (int number)** that adds up all the digits in a positive number of any lengths. The recursive function receives an int number and returns the sum of all the digits.

Question 7

Write a recursive function called **void printArrayReverse(int[] a, int n)** that prints an array backwards. The recursive **printArrayReverse** method receives the array and the array length – 1 prints the array in reverse. Make sure your method prints a new line at the end

Question 8

Write a recursive function called **void printArray(int a[], int n)** that prints an array. The recursive **printArray** method receives the array and the array length – 1 and prints the array. Make sure your method prints a new line at the end.

Question 9

Write a recursive function called **void reverseString(String s)** that reverses a string in place. The recursive **reverseString** method receives the string and returns the string in reverse. No printing is allowed. You need to use the substring method since you cannot replace individual letters in a string.

Question 10

Write a recursive function called **void formatNumber(String s)** that can insert commas in a number. For example 1234567890 becomes 1,234,567,890

Question 11

Write a recursive function **boolean isEven(int n)** that return true if a number has even count of digits or false if the number of digits is odd. Hint: use $/10$ and $\%2$

Question 12

Write a recursive function **boolean isOdd(int n)** that return true if a number has odd count of digits or false if the number of digits is even. Hint: use $/10$ and $\%2$

Question 13

Write a recursive function **void printBinary(int d)** that would print a decimal number as a binary number. A binary number just has digits 0 to 1. Where as a decimal number has digits 0 to 9. The decimal number 5 would be 0101 in binary, since $1*1 + 0*2 + 1*4 + 0*8$ is 10. We are going right to left. To convert a decimal number to binary You just need to take mod 2 of a digit and then divide the number by 2

$$5\%2 = 1 : 1$$

$$5/2 = 2$$

$$2\%2 = 0 : 0$$

$$2/2 = 1$$

$$1\%2 = 1 : 1$$

$$1/2 = 0$$

$$0\%2 = 0 : 0$$

We are done so going backwards

5 in binary is 0 1 0 1

Question 14

Write a recursive function **bool isPrime(int number, int n)** that returns true if a number is prime otherwise false.

A prime number can only be divided evenly by itself. 2,3,5,7, are prime numbers. You can use the mod operator % to test if a number can be divided evenly by itself. $6 \% 3 = 0$, so 6 can be divided evenly by 3 so 6 is not a prime number.

Algorithm:

If the number is less than 2 then it is not a prime number.

If the number is 2 then it is a prime number.

If the number can be divided evenly by n then it is not a prime number

If n reached the number then it is a prime number.

Start n with 2 and increment to the number recursively.

Examples:

isPrime(10,2)	false	10 is not a prime number
isPrime(7,2)	true	7 is a prime number

Question 15

Make a recursive function called **partition(int[] a,int n)** that will partition an array in place into odd and even numbers

Example: **partition([1, 2, 3, 4, 5, 6], 0)**

Array before:

[1, 2, 3, 4, 5, 6]

Array After:

[1, 3, 5, 2, 4, 6]

Use your **printArray** method to print out the array before and after the partition method is called.

Put all your functions in a cs file called Homework12.cs Test all the recursive functions in the main function.

Lesson 13 Delegates and Lambda Expressions

A delegate is a type that represents a reference to a method that includes a parameter list and return type.

Delegates are used to call methods and pass methods as arguments to other methods. Delegate can also be use in callback's where one function calls another one and in event handling when the program needs to respond to an event generated by another section of code. Here is how you make and use a delegate:

A Delegate that receives a parameter:

Step 1: Declare a delegate

A delegate is declared with parameter data type and return data type to represent a method that you want to refer to.

Delegate Syntax:

```
public delegate return_data_type delegate_name(paramater_list);
```

Here we declare a delegate to represent a method that print's out a message.

```
// declare delegates  
public delegate void PrintDelegate(string message);
```

Step 2: Write the method to be represented by the delegate

```
// function to print a message  
public static void printMessage(string message)  
{  
    Console.WriteLine(message);  
}
```

Step 3: Assign the method to the delegate

```
// assign printMessage to delegate
PrintDelegate dprint = printMessage;
```

Step 4 : Call the printMethod using the delegate

```
// call printMessage function using delegate
dprint("I Like C#");
```

```
I Like C#
```

The **dprint** delegate variable is actually a reference to the printMessage method.

```
dprint("I Like C#");
```

really means:

```
printMessage("I Like C#");
```

Here is the complete program:

```
using System;
namespace csLessons
{
    public class Lesson13
    {
        // declare delegate to print message
        public delegate void PrintDelegate(string message);

        // function to print a message
        public static void printMessage(string message)
        {
            Console.WriteLine(message);
        }

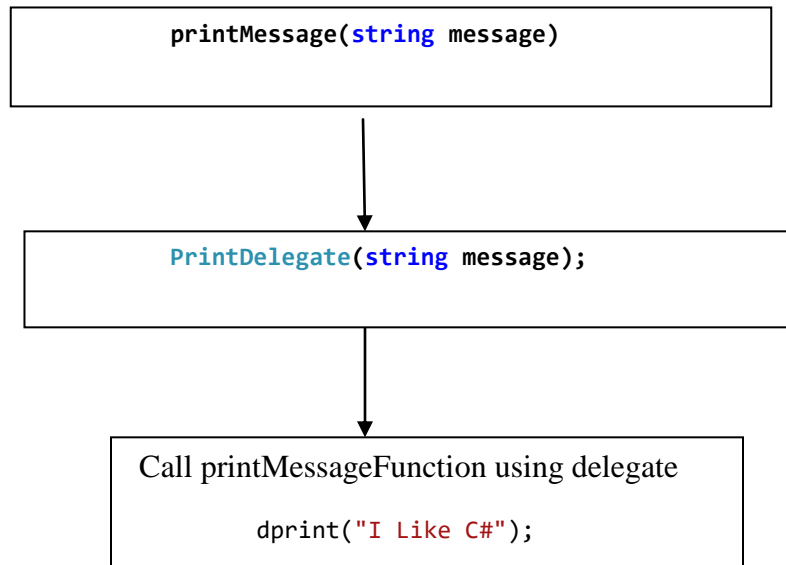
        static void Main(string[] args)
        {
            // assign printMessage function to delegate
            PrintDelegate dprint = printMessage;
```

```

// call printMessage function using delegate
dprint("I Like C#");
}
}
}

```

Here is the program flow:



A Delegate that receives and returns a squared value:

Step 1: declare delegate to square a number

```

// declare square delegate
public delegate void SquareDelegate(int x);

```

Step 2: write the square method to be represented by the delegate

```

// function to square a value
public static int square(int x)
{
    return x * x;
}

```

Step 3: assign the method to the delegate

```
// assign square to square delegate
SquareDelegate dsquare = square;
```

Step4 : call square method using delegate

```
// call square function using delegate
int result = dsquare(5);
```

Step5: print out the results

```
Console.WriteLine(result);
```

```
25
```

Here is the complete program:

```
using System;

namespace csLessons
{
    public class Lesson13
    {
        // declare delegates
        public delegate int SquareDelegate(int x);

        // function to square a value
        public static void square(int x)
        {
            return x * x;
        }

        static void Main(string[] args)
        {
            // assign square to square delegate
            SquareDelegate dsquare = square;
        }
    }
}
```



```

        // call square function using delegate
        int result = dsquare(5);
        Console.WriteLine(x); // 25
    }
}
}

```

A delegate that returns true or false

Step 1: declare delegate to return true or false

```

// declare equals delegate
public delegate bool EqualsDelegate(int x);

```

Step 2: write the equals5 method to be represented by the delegate

```

// function to check if a number equal to 5
public static bool equals5(int x)
{
    return x == 5;
}

```

Step 3: assign the method to the delegate

```

// assign equals5 method to square delegate
EqualsDelegate dequals = equals4;

```

Step4 : call square method using delegate

```

// call equals5 function using delegate and print results
bool bresult = dequals(5);

```

Step5: print out the results

```

Console.WriteLine(bresult);

```

```

true

```

Here is the complete program:

```
using System;

namespace csLessons
{
    public class Lesson13
    {
        // declare equals delegate
        public delegate bool EqualsDelegate(int x);

        // function to check if a number equal to 5
        public static bool equals5(int x)
        {
            return x == 5;
        }

        static void Main(string[] args)
        {
            // assign eqlals5 method to square delegate
            EqualsDelegate dequals = equals4;

            // call square function using delegate
            int result = dsquare(5);
            Console.WriteLine(x); // 25
        }
    }
}
```

Passing a delegate to a method

Passing a delegate to a method is like passing a method to another method and executing the method inside a method.

Step 1: Declare a delegate to represent the method you want to pass

```
// declare calculate delegate
public static delegate int CalculateDelegate(int a, int b);
```

Step 2: Write the method to be represented by the delegate

```
// function to add two numbers
public int add(int a, int b)
{
    return a + b
}
```

Step 3: Write the method to receive a delegate parameter

```
// function that has a delegate as a parameter and invokes delegate
public static void calculate(CalculateDelegate d)
{
    int result = d.Invoke(3, 4);
    Console.WriteLine(result);
}
```

The Invoke method of the delegate will execute the method represented by the delegate and pass the arguments 3 and 4 to it.

Step 4: Assign the delegate to the method to be passed

```
CalculateDelegate d3 = add;
```

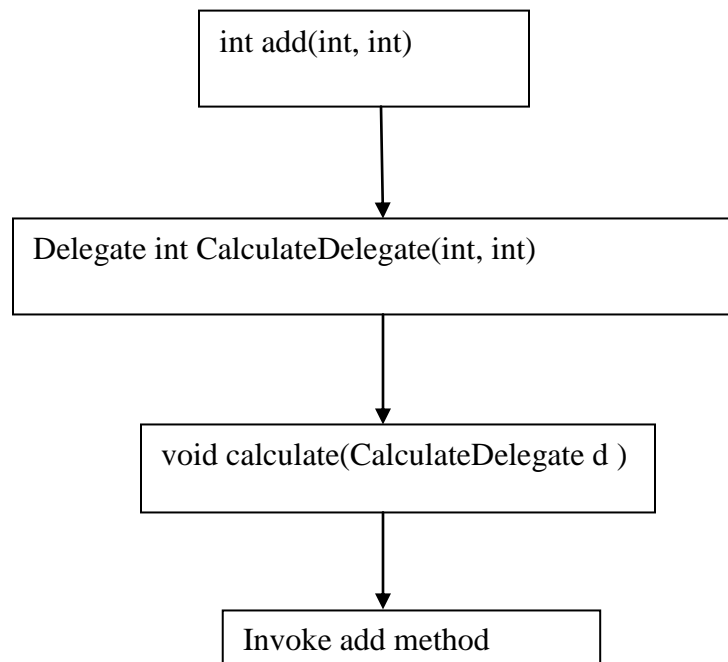
Step 5: Pass the delegate to the calculate method

```
calculate(d3);
```

The calculate method receives the add method through the delegate. The calculate method pass 3 and 4 to the add method. The add method adds $3 + 4 = 7$ and return 7 to the calculate method which the displays the result on the screen.

7

Here is the program flow:



Here is the complete program:

```
using System;
namespace csLessons
{
    public class Lesson13
    {
        // declare calculate delegate
        public delegate int CalculateDelegate(int a, int b);

        // function to add 2 numbers
        public static int add(int a, int b)
```

```

    {
        return a + b;
    }

    // function that has a delegate as a parameter
    public static void calculate(CalculateDelegate d)
    {
        int result = d.Invoke(3, 4);
        Console.WriteLine(result);
    }
    static void Main(string[] args)
    {

        // assign the delegate to the method to be passed
        CalculateDelegate d3 = add;

        // pass the delegate to the calculate method
        calculate(d3);

    }
}

```

Using Action delegate

The Action is a generic delegate supplied by C# for you that refers to a method that accepts one or more arguments but returns no value. You use an Action delegate, when you want to refer to a method that returns void.

Action delegate syntax:

```
public delegate Action<TParameter>
```

We can assign our printMessage to Action delegate.

```
// using Action delegate
Action<String> action = printMessage;
```

We then call the printMessage method from the action delegate

```
action("Called from an Action delegate");
```

The output as follows:

```
Called from an Action delegate
```

Here is the complete program:

```
using System;
namespace csLessons
{
    public class Lesson13
    {
        // function to print a message
        public static void printMessage(string message)
        {
            Console.WriteLine(message);
        }

        static void Main(string[] args)
        {
            // using Action delegate
            Action<String> action = printMessage;

            // call printMesage function using Action delegate
            action("Called from an Action delegate");
        }
    }
}
```

Using Func delegate

Func is a generic delegate supplied by C# for you, it requires zero or more input parameters and one output parameter. The last parameter is considered to be the output parameter.

Func delegate syntax:

```
public delegate TResult Func<in T, out TResult>(T arg);
```

The last parameter in the angle brackets <> is considered the return type, and the remaining parameters are considered input parameter types

Assign the square method to a Func delegate:

Step1: Assign the square method to the Func delegate

```
Func<int, int> fsquare = square;
```

Step 2: Call the square method using the delegate and pass the argument 5

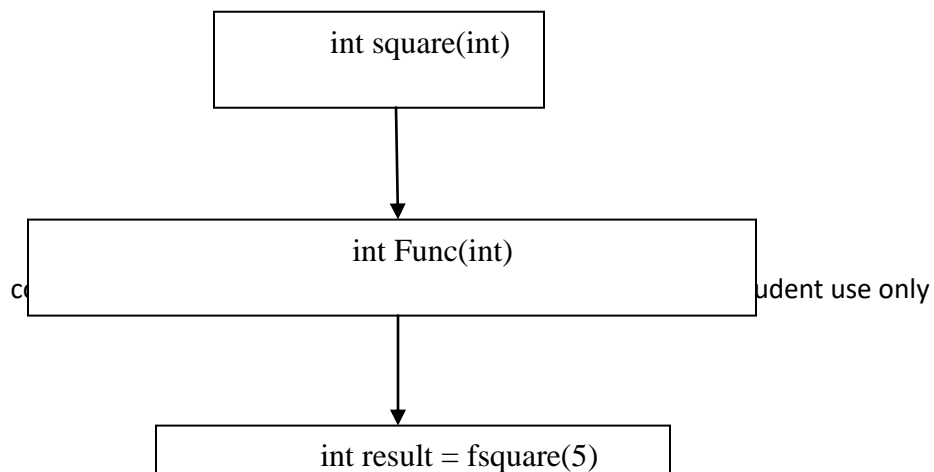
```
int result = fsquare(5);
```

Step 3: print out the result

```
Console.WriteLine(result);
```

```
25
```

Here is the Program flow:



Here is the complete program:

```
using System;

namespace csLessons
{
    public class Lesson13
    {
        // function to square a value
        public static void square(int x)
        {
            return x * x;
        }

        static void Main(string[] args)
        {
            // assign square function to func square delegate
            Func<int, int> fsquare = square;

            // call square function using func square delegate
            int result = fsquare(5);
            Console.WriteLine(x); // 25
        }
    }
}
```


Assign the add method to the Func delegate:

Step1: Assign the add method to the Func delegate

```
Func<int, int, int> fadd = add;
```

Step 2: Call the add methods using the delegate and pass the arguments 3 and 4

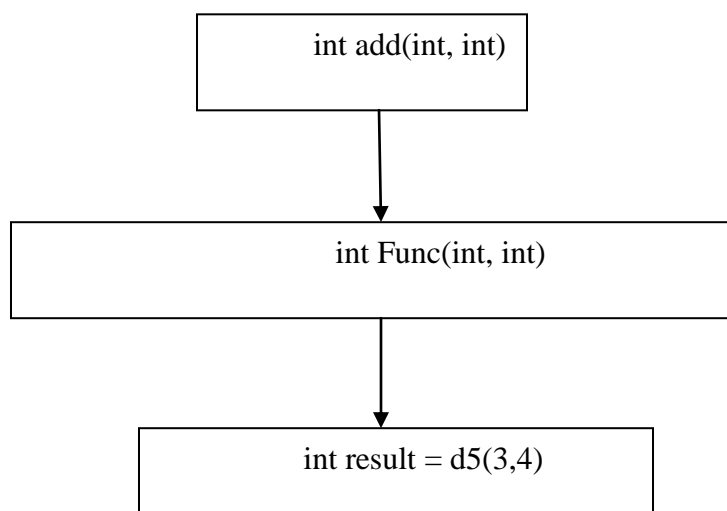
```
Int result = fadd(3, 4);
```

Step 3: Print out the result

```
Console.WriteLine(result);
```

```
7
```

Here is the Program flow:



Here is the complete program:

```
using System;
namespace csLessons
{
    public class Lesson13
    {
        // function to add 2 numbers
        public static int add(int a, int b)
        {
            return a + b;
        }

        static void Main(string[] args)
        {
            Assign the add method to the Func delegate
            Func<int, int, int> fadd = add;

            Call the add method using the func delegate
            int result = fadd(3, 4);

            Print out the result
            Console.WriteLine(result);
        }
    }
}
```

Using the Predicate delegate

The Predicate delegate returns a true or false value and is usually used to search items in a data collection.

Here is the syntax for a Predicate delegate.

```
Predicate<T>
```

Note: The Predicate<T> is basically equivalent to Func<T,bool>.

Here are the steps to make and use a **Predicate** delegate using a **equals** method:

Step 1: Write the equals method to be represented by the delegate

```
// function to check if a number is equal to another number
public static bool equals5(int x)
{
    return x == 5
}
```

Step 3: Assign the method to the Predicate delegate

```
// assign equals5 to delegate
Predicate<int> pred = equals5;
```

Step 4 : Call equals method using delegate

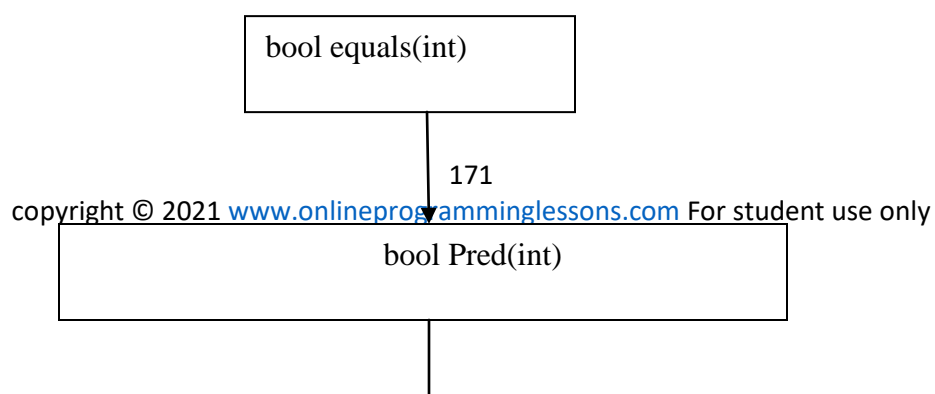
```
// call equals5 function using delegate
bresult = pred(5);
```

Step 5: Print out the result

```
Console.WriteLine(bresult);
```

true

Here is the program flow:



Here is the complete program:

```
using System;
namespace csLessons
{
    public class Lesson13
    {

        // function to check if a number is equal to another number
        public static bool equals5(int x)
        {
            return x == 5
        }

        static void Main(string[] args)
        {

            // assign printMessage to delegate
            Predicate<int> pred = equals5;

            // call equals5 function using delegate
            bool result = pred(5);

            Print out the result
        }
    }
}
```

```
        Console.WriteLine(result);
    }
}
```

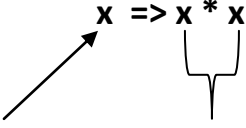
Lambda Expressions

A Lambda expression is used to create an anonymous function. An anonymous function is a function with no name that can be created instantaneously inside another function.

Lambda Expression syntax:

(input-parameters) => expression

The inputs are on the left side and the program expression statement(s) are on the right side and the lambda operator => in the middle.


input-parameter => expression

The above lambda expression really means:

```
int f(int x)
{
    return x * x;
}
```

A lambda expression is just a short compact syntax to make a function with no name to make programming easier.

using a lambda expression

To use a lambda expression you need to assign it to a delegate. We can use the delegates from previous section.

lambda expression to print a message

We will convert our printMessage function to a lambda expression:

```
// function to print a message
public static void printMessage(string message)
{
    Console.WriteLine(message);
}
```

Our lambda expression would be:

```
message => Console.WriteLine(message);
```



input-parameter => expression

step 1: assign lambda expression to the PrintDelegate

```
// assign lambda function to PrintDelegate
PrintDelegate d6 = message => Console.WriteLine(message);
```

Step 2 : call lambda expression method using delegate that prints a message

```
// call lambda function using delegate
d6("I am a Lambda expression");
```

```
I am a Lambda expression
```

Here is the complete program:

```
using System;
namespace csLessons
{
    public class Lesson13
    {
        static void Main(string[] args)
        {

            // assign lambda function to PrintDelegate
            PrintDelegate d6 = message => Console.WriteLine(message);

            // call lambda function using delegate
            d6("I am a Lambda expression");
        }
    }
}
```

lambda expression to square a value

We will convert our square methods to a lambda expression

```
// function to square a number
public static int square(int x)
{
    return x * x;
}
```

Our lambda expression would be:

$x \Rightarrow x * x$

input-parameter => expression

Step 1: assign the square lambda expression to the SquareDelegate

```
// assign square lambda to delegate  
SquareDelegate dlsquare = x => x * x;
```

Step 2 : call square lambda expression using delegate

```
// call square lambda function using delegate  
int result = dlsquare(5);
```

Step 3: Print out the result

```
Console.WriteLine(result);
```



25

Here is the complete program:

```
using System;  
namespace csLessons  
{  
    public class Lesson13  
    {  
        static void Main(string[] args)  
        {  
  
            // assign printMessage to delegate  
            SquareDelegate dlsquare = x => x * x;  
  
            // call square lambda function using delegate  
            int result = dlsquare(5);  
            Console.WriteLine(result); // 25  
        }  
    }  
}
```



```
}  
}  
}
```

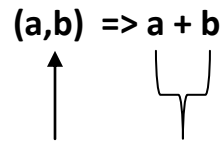
lambda expression to add two numbers

We will convert our add method to a lambda expression:

```
// add 2 numbers  
public static int add(int a, int b)  
{  
    return a + b;  
}
```

Our lambda expression would be:

(a,b) => a + b



input-parameters => expression

Step 1: assign the add lambda expression to the CalculateDelegate

```
// assign lambda add function to delegate  
CalculateDelegate dladd = (a,b) => a + b;
```

Step 2 : call add lambda expression using delegate

```
// call lambda add function using delegate  
int result = dladd(3,4);
```

Step 3: Print out the result

```
Console.WriteLine(result);
```

```
7
```

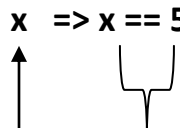
Here is the complete program:

```
using System;
namespace csLessons
{
    public class Lesson13
    {
        static void Main(string[] args)
        {
            // assign lambda add function to delegate
            CalculateDelegate dladd = (a,b) => a + b;

            // call lambda add function using delegate
            int result = dladd(3,4);
            Console.WriteLine(result);
        }
    }
}
```

Assigning a equals lambda expression to the EqualsDelegate:

$x \Rightarrow x == 5$



input-parameters => expression

To run this lambda function we assign it to a **EqualsDelegate** delegate with 1 input value and 1 bool output value

Step1:

```
// assign lambda equals5 function to delegate
EqualsDelegate dlequals = x => x == 5;
```

Step 2:

```
// call lambda equals function using delegate
bool bresult = dlequals(5);
```

Step 3: Print out the result

```
Console.WriteLine(bresult);
```

```
true
```

Here is the complete program:

```
using System;
namespace csLessons
{
    public class Lesson13
    {
        static void Main(string[] args)
        {
            // assign lambda add function to delegate
            EqualsDelegate dlequals = x => x == 5;

            // call lambda add function using delegate
            int result = dlequals(5);
            Console.WriteLine(result);
        }
    }
}
```

Assigning lambdas to Action, Func and Pred delegates:

Assigning the printMessage lambda expression to the Action delegate:

```
message => Console.WriteLine(message);
```



input-parameter => expression

```
Action<string> action1 = message => Console.WriteLine(message);
```

```
// call printMessage function using Action delegate  
Action1 ("I am a Lambda expression");
```

I am a Lambda expression

Assigning the square lambda expression to the Func delegate

```
x => x * x
```



input-parameter => expression

```
Func<int, int> flsquare = x => x * x;  
int result = flsquare(5);  
Console.WriteLine(result);
```

25

Assigning the add lambda expression to the Func delegate

```
(a,b) => a + b
```



input-parameters => expression


To run this lambda function we assign it to a Func delegate with 2 int inputs and 1 int output.

```
Func<int, int, int> fladd = (a,b) => a + b;  
int sum = fladd(3, 4);  
Console.WriteLine(sum);
```

7

Assigning the equals lambda expression to the Predicate delegate

x => x == 5



input-parameters => expression

To run this lambda function we assign it to a Pred delegate with 1 input and 1 bool output.

```
Pred<int> plequals = x => x == 5;  
bool bresult = plequals(5);  
Console.WriteLine(bresult);
```

true

lambda Statement expressions

Lambda Statement expression may have more than 1 expression statement, The statements are to be enclosed by curly brackets { } and each statement ends with a semi colon ; . A return statement is needed if the Statement lambda

Is suppose to return a value. Local variables must be declared with their expected data types.

The lambda Statement expressions syntax is:

(input-parameters) => { <sequence-of-statements> }

Here is a 2 input Statement lambda expression that return a int value.

`(a, b) => { int x = (a - b); int y = (a - b); return x*x + y*y; };`

(input-parameters) => { <sequence-of-statements> }

We assign the lambda Statement expressions to a **Func** delegate

```
Func<int, int, int> distance = (a, b) =>
{
    int x = (a - b);
    int y = (a - b);
    return x*x + y*y;
};
```

```
result = distance(3, 4);
Console.WriteLine(result);
```

2

Using delegate operator

The **delegate** operator creates an anonymous method from method code that can be converted to a delegate type.

```
Delegate_type delegate_name = delegate_operator anonymous_method
```

In the following example we create a anonymous method the that adds two received values and return them.

```
Func<int, int, int> sum = delegate (int a, int b) { return a + b; };
```

In this example we create a anonymous method that receives a number and return true if it is within a certain range.

```
FindDelegate(int x) d = delegate (int n) {return n > 3 && n < 8};
```

Here is an example to return a sublist of numbers from an array of numbers within a certain range using an anonymous function and delegate operator.

Here are the steps:

Step 1: Make a Find delegate

```
// declare Find delegate  
public delegate bool FindDelegate(int x);
```

Step 2: Make a where method that receives a array of numbers and a delegate to compare the numbers with a range of values using the delegate and return an array of numbers within the range.

```
// where method that receives a array of numbers  
public static int[] where(int[] numbers, FindDelegate d)  
{  
    int i = 0;  
    List<int> results = new List<int>();  
    foreach (int x in numbers)
```

```

        if (d(x))
        {
            results.Add(x);
            i++;
        }
    return results.ToArray();
}

```

The **where** function receives an array of numbers and a delegate and uses the delegate to select numbers in the array by comparing the values. The selected values are placed into a list. When all numbers have been examined the list of number values are returned as an array.

Step 3: Make an array of numbers and call the **where** method with the array of numbers and an inline delegate anonymous method.

```

int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

// call where methods with numbers array and anonymous method
int[] results = where(numbers, delegate (int n)
{
    return n > 3 && n < 8;
});

```

The above statement uses the delegate operator creates an anonymous method that is converted to a delegate type as an argument to the where function.

This is equivalent to:

We first make the delegate:

```

FindDelegate(int x) d = delegate (int n) {return n > 3 && n < 8;};

```

Then call the where method with the array and delegate variable:

```

int[] results = where(numbers, d);

```

Step 4: Print out the results

```

// print results

```



```

foreach (int n in results)
{
    Console.Write(n + " ");
}

Console.WriteLine();

```

4 5 6 7

Here is the complete program:

```

using System;
namespace csLessons
{
    public class Lesson13
    {

        // declare Find delegate
        public delegate bool FindDelegate(int x);

        // where method that receives a array of numbers
        public static int[] where(int[] numbers, FindDelegate d)
        {
            int i = 0;
            List<int> results = new List<int>();
            foreach (int x in numbers)
            {
                if (d(x))
                {
                    results.Add(x);
                    i++;
                }
            }

            return results.ToArray();
        }
    }
}

```

```

static void Main(string[] args)
{
    int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    // call where methods with numbers array and anonymous method
    int[] results = where(numbers, delegate (int n)
    {
        return n > 3 && n < 8;
    });

    // print results
    foreach (int n in results)
    {
        Console.Write(n + " "); // 4 5 6 7
    }

    Console.WriteLine();
}
}
}

```

CALLBACKS

A Callback calls another function. Delegate make good call backs. We will make a function that will call another method using a delegate when a task is completed.

Step 1: Make call back delegate

```

// make call back delegate
public delegate void CallbackDelegate(string message);

```

Step 2: Make the call back method

```

// make call back method
public static void Callback(string result)

```

```
{  
    Console.WriteLine(result);  
}
```

This method prints out a message when the task is completed. When the task is completed this callback method is called.

Step 3: Make start task function

```
// make start task method  
public static void StartNewTask(CallBackDelegate taskCompletedCallBack)  
{  
    Console.WriteLine("I have started new Task.");  
    taskCompletedCallBack("I have completed Task.");  
}
```

When this function is called it starts a task and when the task is completed the callback function is called using the CallBackDelegate.

Step 4: Test the call back

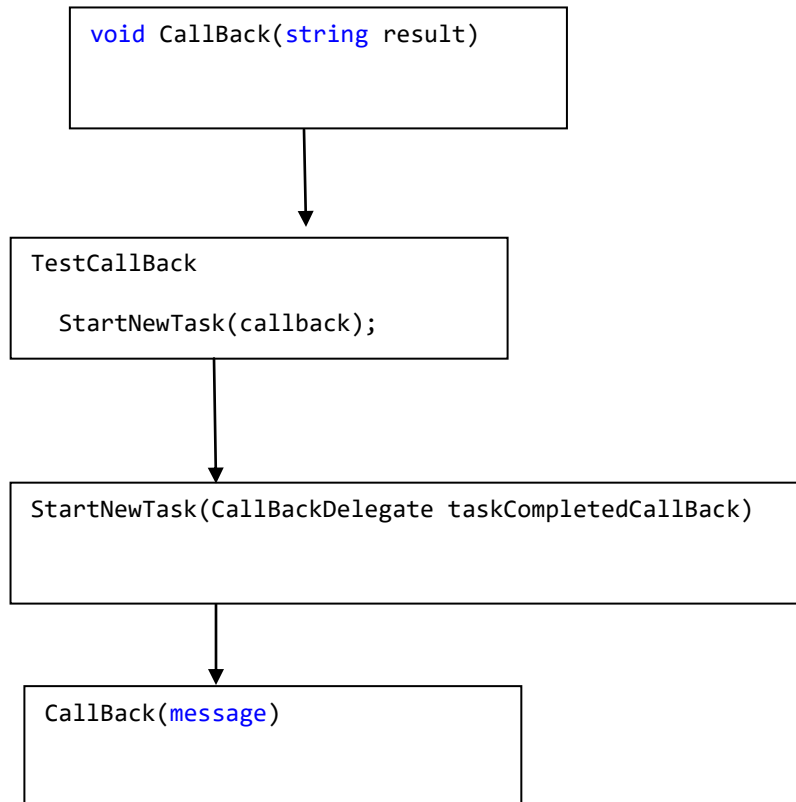
```
// method to test call back  
public static void TestCallBack()  
{  
    callBackDelegate callback = CallBack;  
    startNewTask(callback);  
}
```

The TestCallBack function assigns the callback method to a delegate that is sent to the startNewTask method that starts a task.

Here is the output:

```
I have started new Task.  
I have completed Task.
```

Here is the program flow:



Here is the complete program:

```
using System;  
namespace csLessons  
{  
    public class Lesson13  
    {  
  
        // make call back delegate  
        public delegate void CallBackDelegate(string message);  
  
        // make call back method  
        public static void Callback(string result)
```

```

        {
            Console.WriteLine(result);
        }

// make start task method
public static void StartNewTask(CallBackDelegate taskCompletedCallBack)
{
    Console.WriteLine("I have started new Task.");
    taskCompletedCallBack("I have completed Task.");
}

// method to test call back
public static void TestCallBack()
{
    callBackDelegate callback = CallBack;
    startNewTask(callback);
}

    static void Main(string[] args)
    {
        // test callback
        TestCallBack();
    }
}
}

```

LESSON 13 Homework

Question 1

Write a 2 input lambda expression that receives two inputs that return true if the 2 inputs are equal or otherwise false if they are not equal. Execute the lambda expression in a Func delegate.

Question 2

Make a calculate method that receives two values and a func delegate. Feed the calculate method with 2 numbers and a lambda expression that adds two numbers. Printout the results.

Question 3

Write a lambda statement that receives a name as first name and last name separated by a comma. Print out the first name and last name on separate lines. Use an Action delegate to execute the lambda statement expression.

Question 4

Write a lambda expression that does not receive any parameter but prints out a message of your choice. You can use this syntax to create a lambda expression with no inputs:

```
() => lamda_expression
```

Question 5

Make a Student class that has a name, student number and a age. Make a delegate called **FindStudent** that will return true if a Student age is between 20 and 30 years old. Make a array of Students. Make a **where** method that receives the array of students and the **FindStudent** delegate. The where method will then return a list of students between 20 and 30 years old.

In your main method call the **where** method with an array of Student objects and a method using the delegate operator that will return true if the students ages

are between 20 and 30 years old. Print out the returned list of students. Run the program, you should get something like this:

```
1 John 18  
6 Chris 17  
7 Rob 19
```

Question 6

Make a program that has a function called `enterNumbers` that receives a `CalculateDelgate` that could add two numbers together and returns the result. The `enterNumbers` function also receives a `CallBackDelegate` that receives a result to print out a value using the call back method.

In the `enterNumbers` function ask the user to enter 2 numbers, then call the `CalculateDelegate` to add the two numbers and return the result. Once you have the result call the `CallBackDelegate` to print out the result.

In your main method call the `enterNumbers` with the 2 delegates, you can assign them to functions, lambdas, or delegate functions.

Put all your Lesson13 homework in a file called `Homework13.cs`

Lesson14 LINQ

LINQ lets you do queries on Collections list arrays, Lists etc. LINQ stands for Language Integrated Query. LINQ is similar to the SQL Language that is used to query data bases. LINQ allows users to make queries on Collections instead.

LINQ has two types of syntax, LINQ Query syntax and LINQ Method syntax. We will start with LINQ Query syntax. We can write LINQ queries for the classes that implement IEnumerable<T> or IQueryable<T> interface.

The Enumerable class includes extension methods for the classes that implement IEnumerable<T> interface. ALL the built-in collection classes like List implement IEnumerable<T> interface and so we can write LINQ queries to retrieve data from the built-in collections.

The Queryable class includes extension methods for classes that implement IQueryable<t> interface. The IQueryable<T> interface is used to provide querying capabilities against a specific data source where the type of the data is known. For example, Entity Framework API implements IQueryable<T> interface to support LINQ queries with underlying databases such as MS SQL Server. You can do LINQ query syntax two ways, using a lambda expression or a Conditional expression and using a Conditional method. Both will generate the same results.

LINQ Query syntax using lambda expression

```
var result = from <variable> in <Collection>  
<Query Operators> <lambda expression>  
<select or group operator> <format result>
```

LINQ Query syntax using an conditional expression:

```
var result = from <variable> in <Collection>  
<Query Operators> <variable>.Conditional(argument)  
<select or group operator> <format result>  
<Query Operators> are select, join etc
```


Each LINQ keyword has a dedicated purpose:

Keyword	Purpose
From	Specifies a data source and a range variable
Variable	represents each successive element in the source sequence
Collection	data source
Let	store the result of an expression, such as a method call, in a new range variable.
Where	filter out elements from the source data based on one or more predicate expressions.
Group	produce a sequence of groups organized by a key that you specify.
Into	Provides an identifier that can serve as a reference to the results of a join, group or select clause.
Select	produces a sequence of items of the data source
Orderby	sort the results in either ascending or descending order.
Join	to associate and/or combine elements from one data source with elements from another data source based on an equality comparison between specified keys in each element

SELECTING

Here is our first LINQ expression to select all the numbers in an array

```
var result = from <variable> in <collection> <select> <format the results>
```

```
IEnumerable<int> result = from n in numbers select n;
```



collection

The LINQ expression will return an IEnumerable object that allows you to enumerate over the result of the LINQ expression. The IEnumerable object has code to iterate over the selected number results. Instead of var we use **IEnumerable<int>** to specify the return data type so you know what it is. var means any data type and is used for convenience.

using select

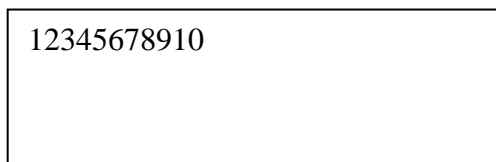
Our first example is to select all the numbers in an array:

```
// make an array of number
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

// make LINQ expression
IEnumerable<int> result = from n in numbers select n;

// execute Linq query and print out the results
foreach (var item in result)
{
    Console.Write(item);
}

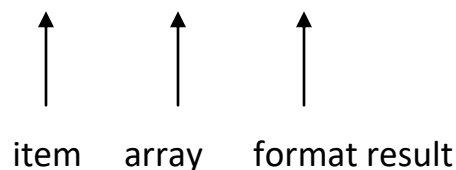
Console.WriteLine();
```



12345678910

This example is easy to understand:

```
IEnumerable<int> result = from n in numbers select n;
```



↑ ↑ ↑
item array format result

n is an item in the array that we want to select from the array **numbers**
A LINQ query must query to some kind of data sources whether it can be array, collections, XML or other databases. After writing LINQ query, it must be executed to get the result. The **foreach** loop executes the query on the data source and gets the result and then iterates over the result set.

Using Where

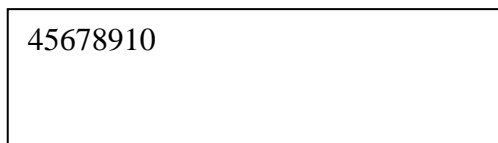
Here is another example that uses a **where** clause that provides a condition.

```
// make an array of number
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

// make LINQ expression
result = from n in numbers where n > 3 select n;

// execute Linq query and print out the results
foreach (var item in result)
{
    Console.Write(item);
}

Console.WriteLine();
```

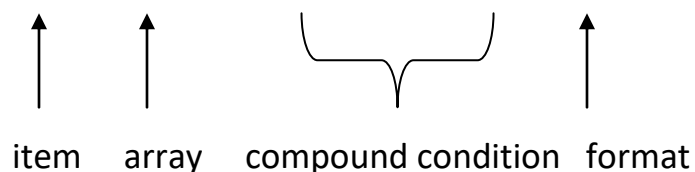


45678910

Using where with compound condition

Here is a **where clause** with a compound condition:

```
IEnumerable<int> result = from n in numbers where n > 3 && n < 9 select n;
```



item array compound condition format

The result is formatted to include all numbers between 3 and 8.

```
// make an array of number
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

// make LINQ expression
IEnumerable<int> result = from n in numbers where n > 3 && n < 9 select n;

// execute Linq query and print out the results
foreach (var item in result)
{
    Console.Write(item);
}

Console.WriteLine();
```

45678

Here is the complete program:

```
/*
 * Lesson14.cs
 * LINQ
 */

using System;
using System.Collections.Generic;
using System.Linq;

public class Lesson14
{
    static void Main(string[] args)
    {
        // array of numbers
        int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

        // return selected numbers
```

```

IEnumerable<int> result = from n in numbers select n;

// print out selected numbers
foreach (var item in result)
{
    Console.Write(item);
}

Console.WriteLine();

// select numbers > 3
result = from n in numbers where n > 3 select n;

// print out selected numbers
foreach (var item in result)
{
    Console.Write(item);
}

Console.WriteLine();

// select numbers > 3 and numbers less than 9
result = from n in numbers where n > 3 && n < 9 select n;

// print out selected numbers
foreach (var item in result)
{
    Console.Write(item);
}

Console.WriteLine();

// pause
Console.ReadLine();
}
}

```

IEnumerable methods

The IEnumerable returned result has many methods to do operations like Count, Min, Max, Sum, Average. These methods are called from the **returned result** not on the LINQ expression part.

```
// make an array of number
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

IEnumerable<int> result = from n in numbers select n;
```

You can call inline on the result variable:

```
int count = result.Count()
```

Or you can call on the equation:

```
int count = (from n in numbers select n).Count();
```

Using the returned result:

Here are some of them:

```
Console.WriteLine("Count: " + result.Count());
Console.WriteLine("Max: " + result.Max());
Console.WriteLine("Min: " + result.Min());
Console.WriteLine("Sum:" + result.Sum());
Console.WriteLine("Average:" + result.Average());
Console.WriteLine("Contains 7:" + result.Contains(7));
```

```
Count: 5
Max: 8
Min: 4
Sum:30
Average:6
```

SORTING

Using orderby

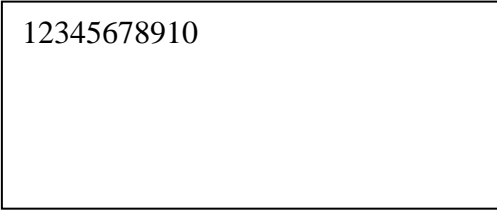
The **orderby** keyword is used to sort selected items. Ascending order is the default, the descending keyword is used to specify descending order.

```
// array of unsorted numbers
int[] numbers2 = { 10,4,3,1,6,7,8,5,2,9 };

// sort ascending
result = from n in numbers2 orderby n select n;

// print out selected numbers
foreach (var item in result)
{
    Console.Write(item);
}

Console.WriteLine();
```



12345678910

Using orderby descending

```
// array of unsorted numbers
int[] numbers2 = { 10,4,3,1,6,7,8,5,2,9 };

// sort descending
result = from n in numbers2 orderby n descending select n;
```

```
// print out selected numbers
foreach (var item in result)
{
    Console.Write(item);
}
```

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

```
10987654321
```

GROUPING

The GroupBy operator returns groups of elements based on some key value. The result of GroupBy operators is a collection of groups.

We will make a group of numbers between 4 and 8 (true) and not 4 and 8 (false)

Using group

```
// array of numbers
int[] numbers2 = { 10,4,3,1,6,7,8,5,2,9 };

// group numbers
var groupedResult = from n in numbers
                    group n by n > 3 && n < 9;

//iterate each group
foreach (var group in groupedResult)
{
    //Each group has a key False and True
    Console.WriteLine("Group: {0}", group.Key);

    // Each group has inner collection
    foreach (int n in group)
        Console.WriteLine("Number: {0}", n);
}
```


We iterate for each group in the returned collection of groups. Then we iterate through the groups of numbers. We use var because it is quite convenient to do. You do not need to worry what object is being returned.

Here is the output:

```
Group: False
Number: 1
Number: 2
Number: 3
Number: 9
Number: 10
Group: True
Number: 4
Number: 5
Number: 6
Number: 7
Number: 8
```

The result is two groups **true** and **false**.

JOIN

The joining operators join the two sequences (collections) and produce a result. The Join operator joins two sequences (collections) based on a key and returns a resulting sequence. The Join operator operates on two collections, inner collection & outer collection. It returns a new collection that contains elements from both the collections which satisfies specified expression. It is the same as **inner join** of SQL query.

The join syntax takes 5 parameters:

```
public static IEnumerable<TResult> Join<TOuter, TInner, TKey, TResult>(this  
IEnumerable<TOuter> outer,  
    IEnumerable<TInner> inner, Func<TOuter, TKey> outerKeySelector,  
    Func<TInner, TKey> innerKeySelector,  
    Func<TOuter, TInner, TResult> resultSelector);
```

Here we will join 2 numbers arrays on common numbers:

```
// numbers1
int[] nums1 = { 1, 2, 3, 4, 5,6,7,8 };

// numbers2
int[] nums2 = { 3,4,5,6,7,8,9,10};

// join common numbers
var results = from n1 in nums1
              join n2 in nums2
              on n1 equals n2
              select n1;

// print out selected numbers
foreach (var item in results)
{
    Console.Write(item + " ");
}
Console.WriteLine();
```

Here is the output:

```
3 4 5 6 7 8
```

Using a Equals Conditional expression

The Equals method returns true if the LINQ variable is equal enclosed condition is evaluated to be true. The Equals method is called from the LINQ variable.

Here is the syntax:

```
var result = from <variable> in <Collection>
              <Query Operators> <variable>.Condition(argument)
              <select or group operator> <format result>
```

Here is the code:

```
// make an array of number
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

// print out equal numbers
result = from s in numbers
        where s.Equals(5)
        select s;

// print out selected numbers
foreach (var item in result)
{
    Console.Write(item);
}

Console.WriteLine();
```

A rectangular box with a black border containing the number 5 in the top-left corner.

Using a Contains Conditional Expression:

The Contains method returns true if the LINQ variable contains the provided value. In our example we check if one of the fruits contain the provided letter.

```
// using a Contains method
String[] fruits = { "apple", "pear", "orange", "peach", "grape" };

// get fruits that contain 'r'
IEnumerable<String> sresult = from f in fruits where f.Contains('r') select f;

// print out selected numbers
foreach (var item in sresult)
{
    Console.Write(item + " ");
}
}
```

Console.WriteLine();

```
pear orange grape
```

We have selected only the fruits that contain the letter 'r'

LINQ Method Syntax

Method syntax uses extension methods included in the [Enumerable](#) or [Queryable](#) classes.

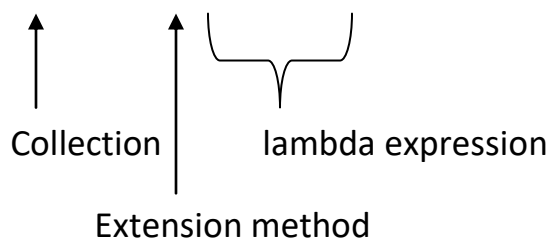
The compiler converts query syntax into method syntax at compile time.

Here is the LINQ Method Syntax:

```
IEnumerable<T> result= collection.extension_method( lambda expression);
```

The **LINQ method syntax** comprises of extension methods and Lambda expression. Example:

```
IEnumerable<int> result = numbers.Where(n => n > 3);
```



The Where method defined in the Enumerable class accepts a [predicate](#) delegate as `Func<int, bool>`. This means you can pass any delegate function that accepts a `int` object as an input parameter and returns a Boolean value. The lambda expression works as a delegate passed in the Where clause.

Note: The **LINQ Method Syntax** is different from the **LINQ Query syntax** using an conditional expression. In the **LINQ Query syntax** the conditional method is part of the lambda expression variable.

```
result = from s in numbers where s.Equals(5) select s;
```

Where as in the **LINQ Method Syntax** the extension methods is attached to the LINQ method syntax variable.

```
result = numbers.Where(n => n > 3);
```

We have converted our **query syntax** to **method syntax**:

Select numbers > 3

```
// make an array of numbers
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

// select numbers > 3
result = numbers.Where(n => n > 3);

// print out selected numbers
foreach (var item in result)
{
    Console.Write(item);
}

Console.WriteLine();
```

```
45678910
```

Select numbers between 3 and 9

```
// select numbers > 3 and numbers less than 9
result = numbers.Where ( n => n > 3 && n < 9);

// print out selected numbers
foreach (var item in result)
{
    Console.Write(item);
}

Console.WriteLine();
```

45678

OrderBy ascending

```
// unsorted numbers
int[] numbers3 = { 10, 4, 3, 1, 6, 7, 8, 5, 2, 9 };

// sort ascending
result = numbers3.OrderBy(n => n);

// print out selected numbers
foreach (var item in result)
{
    Console.Write(item);
}

Console.WriteLine();
```

12345678910

OrderBy descending

```
// unsorted numbers
int[] numbers3 = { 10, 4, 3, 1, 6, 7, 8, 5, 2, 9 };

// sort descending
result = numbers3.OrderBy(n => -n);

// print out selected numbers
foreach (var item in result)
{
    Console.WriteLine(item);
}

Console.WriteLine();
```

```
10987654321
```

GroupBy

```
// make an array of numbers
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

// group by numbers
var groupedResult = numbers.GroupBy( n => n > 3 && n < 9);

//iterate each group
foreach (var group in groupedResult)
{
    //Each group has a key
    Console.WriteLine("Group: {0}", group.Key);

    foreach (int n in group) // Each group has inner collection
        Console.WriteLine("Number: {0}", n);
}
```

```
Group: False
Number: 1
Number: 2
Number: 3
Number: 9
Number: 10
Group: True
Number: 4
Number: 5
Number: 6
Number: 7
Number: 8
```

Join

```
// numbers1
int[] nums1 = { 1, 2, 3, 4, 5, 6, 7, 8 };

// numbers2
int[] nums2 = { 3, 4, 5, 6, 7, 8, 9, 10 };

// join common numbers
var results = nums1.Join(nums2,
    n1 => n1,
    n2 => n2,
    (n1, n2) => n1);

// print out selected numbers
foreach (var item in results)
{
    Console.Write(item + " ");
}

Console.WriteLine();
```

Here is the join output:

```
3 4 5 6 7 8
```


Standard Query Operators

Standard Query Operators in LINQ are actually extension methods for the `IEnumerable<T>` and `IQueryable<T>` types. They are defined in the `System.Linq.Enumerable` and `System.Linq.Queryable` classes. There are over 50 standard query operators available in LINQ that provide different functionalities like filtering, sorting, grouping, aggregation, concatenation, etc.

We can just go over a few of them.

Standard Query Operators in Query Syntax:

```
var result = from n in number where n > 3 && n < 8 select n
```

Standard Query Operators in Method Syntax:

```
var result = numbers.where(n > 3 && n < 8).ToList()
```

The **ToList** method converts the result to a List object.

Standard query operators in query syntax is converted into extension methods at compile time. So both are same.

Standard Query Operators can be classified based on the functionality they provide. The following table lists all the classification of Standard Query Operators:

Classification	Standard Query Operators
Filtering	Where, OfType
Sorting	OrderBy, OrderByDescending, ThenBy, ThenByDescending, Reverse
Grouping	GroupBy, ToLookup
Join	GroupJoin, Join
Projection	Select, SelectMany
Aggregation	Aggregate, Average, Count, LongCount, Max, Min, Sum

Quantifiers	All, Any, Contains
Elements	ElementAt, ElementAtOrDefault, First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault
Set	Distinct, Except, Intersect, Union
Partitioning	Skip, SkipWhile, Take, TakeWhile
Concatenation	Concat
Equality	SequenceEqual
Generation	DefaultEmpty, Empty, Range, Repeat
Conversion	AsEnumerable, AsQueryable, Cast, ToArray, ToDictionary, ToList
Filtering	Where, OfType
Sorting	OrderBy, OrderByDescending, ThenBy, ThenByDescending, Reverse
Grouping	GroupBy, ToLookup
Join	GroupJoin, Join

Examples using Standard Query Operators on Objects

We will first make a Person class.

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int BirthYear { get; set; }

    public override String ToString()
    {
        return FirstName + ", " + LastName + " " + BirthYear;
    }
}
```

Then store some Persons in a List of Persons:

```
List<Person> persons = new List<Person>()
{
    new Person()
    {
        FirstName = "Terrance", LastName = "Johnson", BirthYear = 2005
    },
}
```

```

new Person()
{
    FirstName = "John", LastName = "Smith", BirthYear = 1966
},
new Person()
{
    FirstName = "Eva", LastName = "Birch", BirthYear = 2002
}
}

```

Using Select New

Select all people born on or after 1910

```

//Get the full combined name for people born in 1990 or later
var fullNames = from p in persons
    where p.BirthYear >= 1990
    select new { p.FirstName, p.LastName };

```

Same example using method syntax

```

//Get the full combined name for people born in 1990 or later
var fullNames = persons.Where(x => x.BirthYear >= 1990)
    .Select(x => new { x.FirstName, x.LastName });

```

The **select new** clause create a new data structure. This structure is composed of compound objects. With the select new clause, we create new objects of an AnonymousType as the result of a query. An Anonymous type still allows you to access the member variable.

Print out results using Anonymous type fullNames

```

// print out fullnames
foreach (var item in fullNames)
{
    Console.WriteLine(item.FirstName + " " + item.LastName);
}

```

Terrance Johnson Eva Birch

Find First Element in a Collection

Return the first element in a collection.

Return the first person in the persons collection.

```
var first = persons.First(); //First element in the collection
```

Terrance, Johnson 2005

In this situation the first Person in the persons list is returned. A Person object is returned.

Find first Element that matches a Condition

Return the first element in a collection that matches a condition

Return the first person that has birth year greater than 2001

```
//First element that matches a condition  
first = persons.First(p => p.BirthYear > 2001);  
Console.WriteLine(firstWithCondition);
```

Terrance, Johnson 2005

The First() method throws an exception if no items are found. We can have it instead return a default value of null by using FirstOrDefault()

```
//First element that matches a condition OR default value  
first = persons.FirstOrDefault(x => x.BirthYear > 2005);
```

The result is null because there are no birthYear greater then 2005

```
null
```

You can also use `firstOrDefault` with no condition.

```
//First element in collection or default value  
var firstOrDefault = users.FirstOrDefault();
```

Obtaining a Single Result

We can also get exactly one item using `Single()` or `SingleOrDefault()`:

```
var singleUser = users.Single(x => x.FirstName == "John");  
var singleUser = users.SingleOrDefault(x => x.LastName == "Johnson");
```

Both `Single()` and `SingleOrDefault()` will throw an exception if more than one item matches the condition.

```
Console.WriteLine(singleUser)
```

```
Terrance, Johnson 2005
```

Distinct

`Distinct` return all distinct items in a collection (unique).

```
var distinct = persons.Distinct();  
  
foreach (var item in distinct)  
{  
    Console.WriteLine(item);  
}
```

```
Terrance, Johnson 2005  
John, Smith 1966  
Eva, Birch 2002
```

Ordering Example

We can order results from a LINQ query by their properties using the methods **OrderBy()** and **ThenBy()**.

We first order by first name, if both first names are equal then we order by last name.

Ascending Order

```
//Alphabetical order by first name then last name
var orderedUsers = persons.OrderBy(p => p.FirstName)
    .ThenBy(p => p.LastName);

// print out fullnames
foreach (var item in orderedUsers)
{
    Console.WriteLine(item.FirstName + " " + item.LastName);
}
```

Eva Birch John Smith Terrance Johnson

Descending Order

We can specify descending order using `OrderByDescending`

```
// Reverse alphabetical order by first name, then by last name
var descendingOrder
    = persons.OrderByDescending(x => x.FirstName)
    .ThenByDescending(x => x.LastName);
```

```
//We can also use the orderby and descending keywords in the query syntax:
descendingOrder = from x in persons
```

```
orderby x.BirthYear descending, x.FirstName descending
select x;
```

```
// print out fullnames
foreach (var item in descendingOrder)
{
    Console.WriteLine(item.FirstName + " " + item.LastName);
}
```

```
Terrance Johnson
Eva Birch
John Smith
```

Lesson14 Homework

Question 1

Use one of your classes from previous lesson and put into a list of objects. Use the syntax query syntax to search for one of your items on the list of objects. Then use syntax method query to search for one of your items on the list of objects. Use the Contains extension method.

LESSON 15 Expression and Expression Trees

Review of lambda Expression

A lambda expression is a short form of a function with no name and is known as a **anonymous** function.

The following is a lambda expression that adds 1 to a number.

```
x => x + 1
```

It is short form for the function:

```
int inc(int x)
{
    return x + 1;
}
```

The following lambda expression adds 2 numbers together.

```
(a,b) => a + b
```

It is short form for the function :

```
int add(int a, int b)
{
    return a + b;
}
```

Lambdas' need delegates to execute them.

Review of delegate

A **delegate** is a **data type** that is used to reference a function by assigning a function to it. You can use a delegate to execute a function using the delegate variable. A delegate must be declared before it can be used, just like a class that must be declared before it can be used.

Here is a delegate that can be used to reference the above **add** function.

```
// declare sum delegate
delegate int SumDelegate(int a, int b);
```

Here we assign the add function to our delegate.

```
// assign add function to delegate
SumDelegate sum1 = add;
```

Here we execute the add function using the delegate variable **sum1**

```
// execute delegate
int x = sum1(3, 4);
Console.WriteLine(x);
```

We can also assign a lambda function to a delegate and execute it using the delegate variable.

```
// assign lambda to delegate
SumDelegate sum2 = (a, b) => a + b;
x = sum2(3, 4);
Console.WriteLine(x);
```

Review of Func delegate

A **Func** delegate is a generic delegate that has zero or more input parameters and one output parameter. The last parameter is considered as an output parameter.

Here we assigning our add function to a func delegate and execute the function using the delegate.

```
// assign a function to a func delegate
Func<int, int, int> sum3 = add;
x = sum3(3, 4);
```

```
Console.WriteLine(x);
```

Here we assigning a lambda function to a func delegate and execute the function using the delegate.

```
// assign a lambda to a func delegate
Func<int, int, int> sum4 = (a, b) => a + b;
x = sum2(3, 4);
Console.WriteLine(x);
```

Assigning void functions to delegates

A void function does not return a value.
Here is a void function that prints out a message

```
// print a message
public static void printMsg(string msg)
{
    Console.WriteLine(msg);
}
```

Here is a void lambda function that prints out a message.

```
msg => Console.WriteLine(msg);
```

Here is a delegate that represents a void function.

```
// declare msg delegate
delegate void MsgDelegate(string s);
```

Here we assign the void function to the delegate and execute the function using the delegate.

```
// assign a void function to a action delegate
MsgDelegate msg1 = printMsg;
msg1("I like c#");
```

Here we assign a void lambda function to the delegate and execute the function using the delegate.

```
// assign a void lambda to a action delegate
MsgDelegate msg1 = msg => Console.WriteLine(msg);
msg4("I like c#");
```

Review of Action delegate

Action delegate is a generic delegate that has zero or more input parameters and one output parameter. The Action delegate does not return a value.

Here we assign a void function to an Action delegate and execute the function using the delegate.

```
// assign a void function to a action delegate
Action<string> msg3 = printMsg;
msg3("I like c#");
```

Here we assign a void lambda function to an Action delegate and execute the function using the delegate.

```
// assign a void lambda to a action delegate
Action<string> msg4 = msg => Console.WriteLine(msg);
msg4("I like c#");
```

Expression

An Expression represents a strongly typed lambda expression.

Expression<TDelegate>

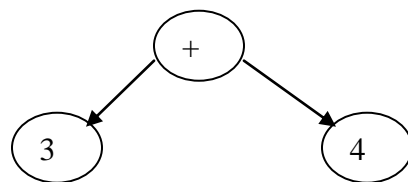
The Expression class is located in the System.Linq.Expressions namespace. The Expression<TDelegate> requires a delegate type **Func** or **Action**.

When a lambda Expression is assigned to the Func or Action type delegate, the C# compiler converts the lambda expression into **executable code** at compile time.

When a lambda expression is assigned to a `Expression<TDelegate>` type the C# compiler converts the lambda expression into an **Expression tree** instead of executable code. The expression tree must be compiled into a **Func** or a **Action** delegate in run time. Once compiled the function reference from the delegate can be executed.

An **Expression Tree** stores an arithmetic expression into a tree structure made up of nodes, The nodes may represent a mathematical operation or a value.

Here is an Expression Tree that's adds two numbers `sum = a + b`



Summary:

`Func<T>` = lambda expression = Executable code

`Expression<TDelegate>` = lambda expression = Expression Tree

Here is an expression that takes a Func delegate where a lambda sum function is assigned to it. The result is an expression tree `exp1`.

```
Expression<Func<int, int, int>> exp1 = (a, b) => a + b;
```

We then compile the Expression and a Func delegate is returned.

```
Func<int, int, int> sum5 = exp1.Compile();
```

We then execute the Func delegate and print out the result.

```
x = sum5(3, 4);  
Console.WriteLine(x);
```

Recapping:

Expression tree are expressions arranged in a tree-like data structure. Each node in an expression tree is an expression. For example, an expression tree can be used to represent mathematical formula $x + y$ where x , $+$ and y will be represented as an expression and arranged in the tree like structure.

Expression tree is an in-memory representation of a lambda expression. It holds the actual elements of the query, not the result of the query.

The expression tree makes the structure of the lambda expression transparent and explicit. You can interact with the data in the expression tree just as you can with any other data structure.

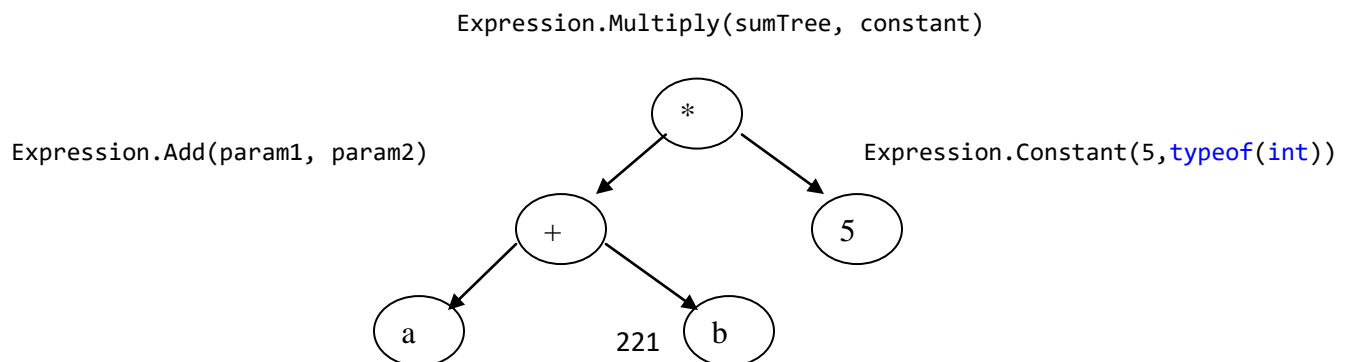
Creating an expression tree manually

The Expression class has methods to create parameters, constants and for mathematical operations. Here are some of them:

Expression	Use
<code>Expression.Parameter(typeof(int));</code>	int parameter
<code>Expression.Constant(5, typeof(int));</code>	constant expression
<code>Expression.Add(param1, param2);</code>	Add 2 expressions
<code>Expression.Multiply(param1, param2);</code>	Multiply 2 expressions
<code>Expression.LessThan(param1, param2);</code>	Compare 2 parameters less than
<code>Expression.GreaterThan()</code>	Compare 2 parameters greater than
<code>BinaryExpression</code>	Root of Expression tree (or sub)

In the following example we will manually create the Expression tree from the lambda expression:

$$(a,b) => (a + b) * 5$$



```
Expression.Parameter(typeof(int), "a")
```

```
Expression.Parameter(typeof(int), "b")
```

```
// building an expression tree manually
```

```
//defining expression (a + 5) * 5
```

```
ParameterExpression param1 = Expression.Parameter(typeof(int), "a");
```

```
ParameterExpression param2 = Expression.Parameter(typeof(int), "b");
```

```
BinaryExpression sumTree = Expression.Add(param1, param2);
```

```
ConstantExpression param3 = Expression.Constant(5, typeof(int));
```

```
BinaryExpression productTree = Expression.Multiply(sumTree, param3);
```

```
// create expression tree
```

```
Expression<Func<int, int, int>> sumLambda =
```

```
    Expression.Lambda<Func<int, int, int>>(productTree, param1, param2);
```

```
//compiling expression to create Func delegate
```

```
Func<int, int, int> compiledSum2 = sumLambda.Compile();
```

```
// use delegate to add two numbers
```

```
x = compiledSum2(3, 4);
```

```
// print result
```

```
Console.WriteLine(productTree + " = " + x);
```

```
((a + b) * 5) = 35
```

Todo: type in or copy/paste in the above code and run it.

Parsing Expression Tree

We can extract the individual components of the expression tree

We first get the input parameters from the above expression tree

```
// Decompose the expression tree.
```

```
ParameterExpression params1 = sumLambda.Parameters[0];
```

```
ParameterExpression params2 = sumLambda.Parameters[1];
```

From the expression tree we can get the body

```
BinaryExpression operation = (BinaryExpression)sumLambda.Body;
```

From the body we can get the left sub tree and the right constant

```
BinaryExpression sum = (BinaryExpression)operation.Left;  
ConstantExpression constant = (ConstantExpression)operation.Right;
```

From the left sub tree we can get the a and b parameters

```
ParameterExpression sumleft = (ParameterExpression)sum.Left;  
ParameterExpression sumright = (ParameterExpression)sum.Right;
```

We can now print out the expression tree components

```
Console.WriteLine("Decomposed expression: ({0},{1}) => ({2} {3} {4}) {5} {6}",  
    params1.Name,params2.Name,sumleft.Name, sum.NodeType, sumright.Name,  
    operation.NodeType,constant.Value);
```

Decomposed expression: (a,b) => (a Add b) Multiply 5

Todo: type in or copy/paste in the above code and run it.

Class to printout an Expression tree

The ExpressionVisitor class can traverse an Expression tree. We override the class so we can print out the traversed tree.

```
class MyExpressionVisitor : ExpressionVisitor  
{  
    // traverse expression tree  
    protected override Expression VisitBinary(BinaryExpression node)  
    {  
        Console.Write("(");  
  
        this.Visit(node.Left);  
  
        switch (node.NodeType)  
        {  
            case ExpressionType.Add:  
                Console.Write(" + ");  
                break;  
        }  
    }  
}
```

```

        case ExpressionType.Subtract:
            Console.WriteLine(" - ");
            break;

        case ExpressionType.Multiply:
            Console.WriteLine(" * ");
            break;

        case ExpressionType.Divide:
            Console.WriteLine(" / ");
            break;
    }

    this.Visit(node.Right);

    Console.WriteLine(")");

    return node;
}

// print constant value
protected override Expression VisitConstant(ConstantExpression node)
{
    Console.WriteLine(node.Value);
    return node;
}

// print parameter name
protected override Expression VisitParameter(ParameterExpression node)
{
    Console.WriteLine(node.Name);
    return node;
}
}

```

Todo:

Put the visitor class in your Lesson 15cs file.

We then make an Expression tree with our lambda equation from above

```
Expression<Func<int, int, int, double>> expr = (a,b) => (a + b) * 5;
```

We then make the MyExpressionVisitor object

```
MyExpressionVisitor myVisitor = new MyExpressionVisitor();
```

We then visit our expression tree


```
myVisitor.Visit(expr.Body);
Console.WriteLine();
```

Here is the output:

```
((a + b) * 5)
```

Lesson15 Homework Question 1

Make an Expression Tree using a func delegate and a lambda that has three parameters and divide by 3 like this, and print out the result.

```
(x,y,z) => ((x + y) * z) / 3.0;
```

Lesson15 Homework Question 2

Manually make the expression tree, and execute it.

Lesson15 Homework Question 3

Extract all the components from the expression tree and print them out

Lesson15 Homework Question 5

Visit the expression tree, and print out the nodes.

Lesson15 Homework Question 6

Repeat questions 1 to 4 with this equation

```
(x, y, z) => x + y > x * y
```

Hint use:

```
Expression.LessThan()
Expression.GreaterThan()
```

You will also need to update the MyExpressionVisitor class to handle less than < and greater than >

Lesson 16 Events, Asynchronous methods and Threading

An event is a notification sent by an object to signal the occurrence of an action.

The class who raises events is called the EventGenerator.

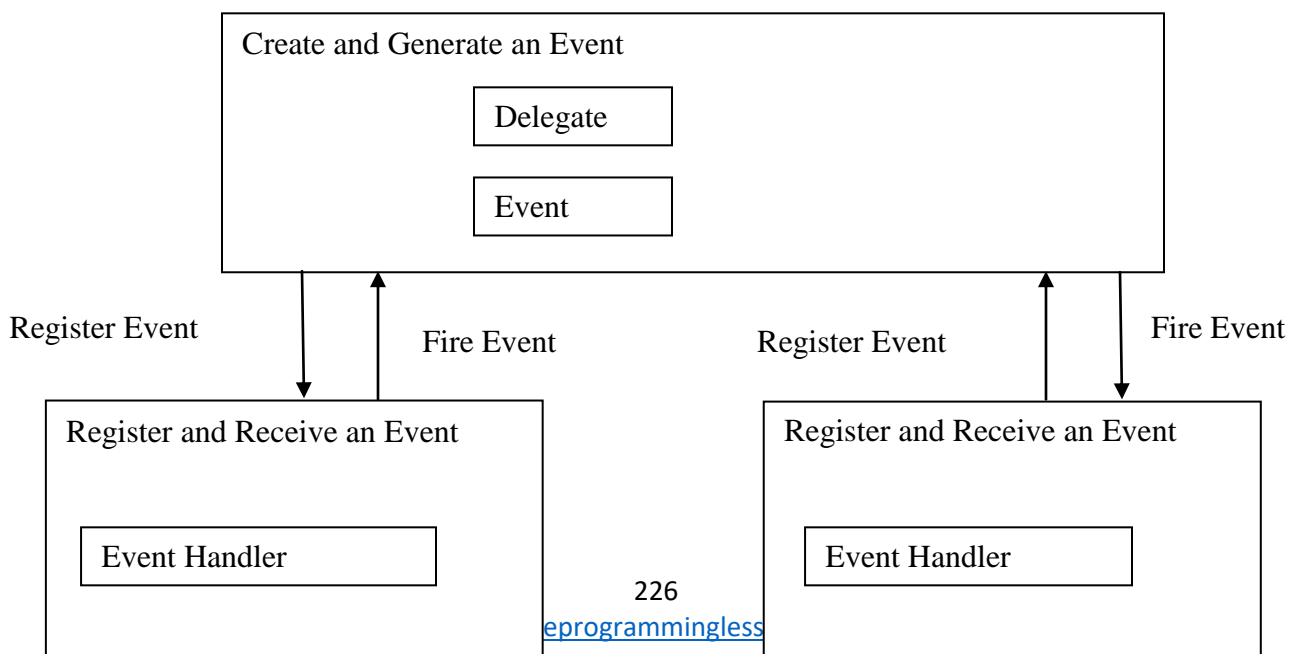
The class who receives the event notification is called the EventReceiver

There can be multiple subscribers of a single event.

A EventGenerator raises an event (fires event) when some action occurred.

EventReceivers who are interested in getting a notification when an action occurs, registers with the EventGenerator and handles the event when the event occurs.

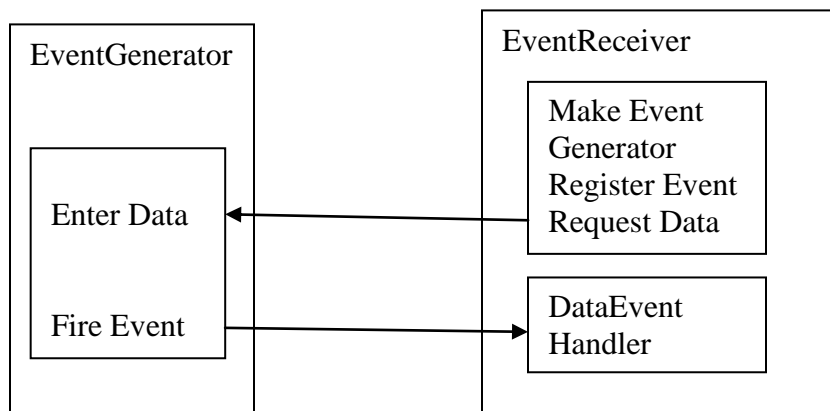
In C#, an event is an encapsulated delegate. The event is dependent on the delegate. The delegate defines the signature for the event handler method of the EventReceiver class.



Simple Event Example

To do: Make a C# file called Lesson16.cs and enter the code for each step.

We are making a EventGenerator to send a DataEvent that contains a Name to a EventReceiver's event handler for display.



Declaring an Event

Step 1: Declare a delegate to represent a event

```
// declare delegate to represent an event
public delegate void DataEventHandler();
```

Step 2: Declare a class to generate an Event

```
class Data
{
```

Step 3: Declare a variable of the delegate with event keyword.

```
public event DataEventHandler dataEvent;
```

Step 4: make a method to generate an event

```
public void enterData()  
{
```

Step 5: do some action: entering a name

```
    Console.WriteLine("Enter your Name: ");  
    string name = Console.ReadLine();
```

step 6: generate the event stored in the delegate (? means if dataEvent is not null)

```
    dataEvent?.Invoke();  
}  
}
```

Step 7 Make a class to register the event and receive the event

```
public class Lesson16  
{
```

Step 8 in a method make the data object

```
public static void Main(string[] args)  
{  
    // make data object  
    Data data = new Data();
```

Step 9 in the method register the event

```
    // register event  
    data.dataEvent += processDataEvent;
```

Step 10 request data, and wait for data ready

```
    // get some data  
    data.enterData();
```

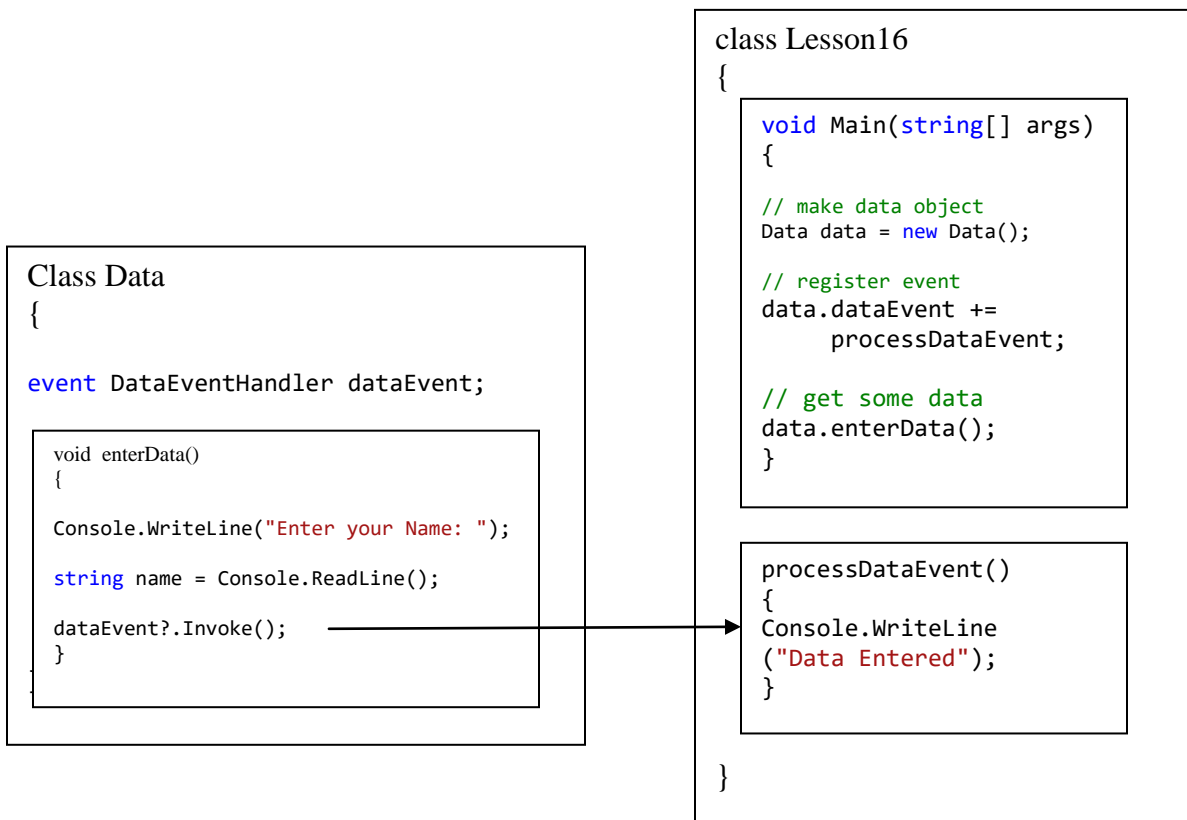
step 11 make event handler method

```

public static void processDataEvent()
{
    Console.WriteLine("Data Entered");
}
}

```

Step 12 user enters data and the processDataEvent is called and Data Entered is printed on the screen



Run the program and enter a name.

You should get something like this:

```

Enter your Name:
tom
Data Entered

```

Here is the complete program:

```

// declaring an Event

// declare delegate to represent an event
public delegate void DataEventHandler();

// class to create an event,store event handlers
// and fire the event
class Data
{
    // make event using delegate
    public event DataEventHandler dataEvent;

    // get a name and fire event
    public void enterData()
    {
        // get a name
        Console.WriteLine("Enter your Name: ");
        string name = Console.ReadLine();

        // fire event
        dataEvent?.Invoke();
    }
}

// class to register an event and receive the event
public class Lesson16
{
    public static void Main(string[] args)
    {
        // make data object
        Data data = new Data();

        // register event
        data.dataEvent += processDataEvent;

        // get some data
        data.enterData();
    }
}

```

```

// data event handler
public static void processDataEvent()
{
    Console.WriteLine("Data Entered");
}
}

```

Using EventHandler Delegate

C# has built-in delegate types **EventHandler** and **EventHandler<TEventArgs>** for events. Any event should need two parameters: the source of the event and event data. Use the **EventHandler** delegate for all events that do not include event data. Use **EventHandler<TEventArgs>** delegate for events that include data to be sent to handlers.

We re-write our example using the **EventHandler**

```

// class to create an event,store event handlers and fire the event
class Data
{
    // make event using delegate
    public event EventHandler dataEvent;

    // get a name and fire event
    public void enterData()
    {
        // get a name
        Console.WriteLine("Enter your Name: ");
        string name = Console.ReadLine();

        // fire event
        dataEvent?.Invoke(this, EventArgs.Empty);
    }
}

// class to register an event and receive the event
public class Lesson16
{
    public static void Main(string[] args)
    {
        // make data object
    }
}

```

```

Data data = new Data();

// register event
data.dataEvent += processDataEvent;

// get some data
data.enterData();
}
// data event handler
public static void processDataEvent(object sender, EventArgs e)
{
    Console.WriteLine("Data Entered");
}
}
}

```

Todo:

Type in the above code and run it
You should get something like this:

```

Enter your Name:
tom
Data Entered

```

Sending and Receiving Data

We can use the **EventHandler<EventArgs>** to send and receive data to the event handler. We are using `EventHandler<string>` so we can send and receive a name.

We can now send the name to the eventHandler when we Invoke it.

```

// fire event
dataEvent?.Invoke(this, name);

```

The event handler now receives the name for display


```

// data event handler
public static void processDataEvent(object sender, string name)
{
    Console.WriteLine("Data Entered: " + name);
}

```

Here is the complete code using the using EventHandler<string> argument

```

// using EventHandler delegate passing data
// class to create an event,store event handlers
// and fire the event
class Data
{
    // make event using delegate
    public event EventHandler<string> dataEvent;

    // get a name and fire event
    public void enterData()
    {
        // get a name
        Console.WriteLine("Enter your Name: ");
        string name = Console.ReadLine();

        // fire event
        dataEvent?.Invoke(this, name);
    }
}

// class to register an event and receive the event
public class Lesson16
{
    public static void Main(string[] args)
    {
        // make data object
        Data data = new Data();

        // register event
        data.dataEvent += processDataEvent;
    }
}

```

```

    // get some data
    data.enterData();
}

// data event handler
public static void processDataEvent(object sender, string name)
{
    Console.WriteLine("Data Entered: " + name);
}
}

```

Todo:

Type in the above code and run it
You should get something like this:

```

Enter your Name:
tom
Data Entered: tom

```

Custom EventArgs

We can make our own EventArgs to pass custom data through custom methods or properties

// using custom EventHandler delegate passing data

```

// custom EventArgs for Name
public class ProcessEventArgs : EventArgs
{
    public string Name { get; set; }

    public ProcessEventArgs(string name)
    {
        Name = name;
    }
}

```

We can now invoke the eventHandler by creating a `ProcessEventArgs` with a entered Name

```
// fire event
dataEvent?.Invoke(this, new ProcessEventArgs(name));
```

Now the event handler can return the name from the `ProcessEventArgs` object.

Here is the complete code using the `ProcessEventArgs` object.

```
// class to create an event,store event handlers
// and fire the event
class Data
{
    // make event using delegate
    public event EventHandler<ProcessEventArgs> dataEvent;

    // get a name and fire event
    public void enterData()
    {
        // get a name
        Console.WriteLine("Enter your Name: ");
        string name = Console.ReadLine();

        // fire event
        dataEvent?.Invoke(this, new ProcessEventArgs(name));
    }
}

// class to register an event and receive the event
public class Lesson16
{

    public static void Main(string[] args)
    {

        // make data object
        Data data = new Data();

        // register event
```

```
data.dataEvent += processDataEvent;

// get some data
data.enterData();
}

// data event handler
public static void processDataEvent(object sender, ProcessEventArgs e)
{
    Console.WriteLine("Data Entered: " + e.Name);
}
}
```

Todo:

Type in the above code and run it

You should get something like this:

```
Enter your Name:
tom
Data Entered: tom
```

Lesson16 Question 1

Make a program that displays the time every second on the screen. Use A delay loop and your own Timer event.

Asynchronous programming with async and await

Asynchronous means to wait for an operation to complete

Synchronous means do not wait.

Async is a keyword to indicate a method will be asynchronous

Await keyword instructs the code to wait for an operation to complete

The await keyword provides a non-blocking way to start a task, then continue executing other code until that task completes.

Task

A Task is a piece of code that you can execute
The Task class represents a single operation that does not return a value and that usually executes asynchronously.

Creating a task

Step 1: make a Action delegate that prints out a message

```
// make action delegate  
Action action1 = () => Console.WriteLine("Task 1");
```

Step2 : create a task object

The task constructor gets an action delegate.

```
// make task  
Task task1 = new Task(action1);
```

Step 3: start task

```
// start task  
task1.Start();
```

Step 4: wait for task to complete

```
// wait for task to compete  
task1.Wait();
```

todo:

type in the above code and run it
you should get something like this:

```
Task 1
```

Remove task1.wait() statement and then run again

What has happen?

Executing more than 1 task

Todo:

Make 4 actions:

```
Action action1 = () => Console.WriteLine("Task 1");  
Action action2 = () => Console.WriteLine("Task 2");  
Action action3 = () => Console.WriteLine("Task 3");  
Action action4 = () => Console.WriteLine("Task 4");
```

Make 4 tasks:

```
Task task1 = new Task(action1);  
Task task2 = new Task(action2);  
Task task3 = new Task(action3);  
Task task4 = new Task(action4);
```

run action using Task.Factory.StartNew(action2);

run action 3 using Task.Run(action3);


run action 4 synchronously using task4.RunSynchronously();

```
task1.Start();  
Task task2 = Task.Factory.StartNew(action2);  
Task task3 = Task.Run(action3);  
task4.RunSynchronously();
```

wait for all tasks to complete

```
task1.Wait();
task2.Wait();
task3.Wait();
task4.Wait();
```

run the program and you should get something like this:

ADDING A R  **ASK**

We now add some random time delays to each action.

```
Action action1 = () =>
{
    Console.WriteLine("Task 1 started ");
    Thread.Sleep(3000);
    Console.WriteLine("Task 1 done ");
};
```

```
Action action2 = () =>
{
    Console.WriteLine("Task 2 started ");
    Thread.Sleep(5000);
    Console.WriteLine("Task 2 done ");
};
```

```
Action action3 = () =>
{
    Console.WriteLine("Task 3 started ");
    Thread.Sleep(2000);
    Console.WriteLine("Task 3 done ");
};
```

```
Action action4 = () =>
{
    Console.WriteLine("Task 4 started ");
```

```
Thread.Sleep(4000);  
Console.WriteLine("Task 4 done ");  
};
```

```
Task task1 = new Task(action1);  
Task task4 = new Task(action4);  
  
task1.Start();  
Task task2 = Task.Factory.StartNew(action2);  
Task task3 = Task.Run(action3);  
task4.RunSynchronously();
```

```
task1.Wait();  
task2.Wait();  
task3.Wait();  
task4.Wait();
```

Run the program you should get something like this:

```
Task 4 started  
Task 1 started  
Task 2 started  
Task 3 started  
Task 3 done  
Task 1 done  
Task 4 done  
Task 2 done  
.
```

USING ASYNCC AND AWAIT

The [async](#) and [await](#) keywords in C# are the heart of asynchronous programming.

The `await` keyword provides a non-blocking way to start a task, then continue execution when that task completes.

There are some rules for writing the Async method:

- The method signature must have the `async` keyword.
- The method name should end with `Async` (this is not enforced, but it is a best practice).
- The method should return `Task`, `Task<T>`, or `void`.

To use this method, you should wait for the result (i.e., use the `await` method). Following these guidelines, when the compiler finds an `await` method, it starts to execute it and will continue the execution of other tasks. When the method is complete, the execution returns to its caller

async

The `async` keyword is added to the method signature to enable the usage of the `await` keyword in the method. It also instructs the compiler to create a state machine to handle asynchronicity.

The return type of an `async` method is always `Task` or `Task<T>`.

await

The `await` keyword is used to asynchronously wait for a `Task` or `Task<T>` to complete. It pauses the execution of the current method until the asynchronous task that's being *awaited* completes. The difference from calling `.Result` or `.Wait()` is that the *await* keyword sends the current thread back to the thread pool, instead of keeping it in a *blocked* state.

Under the hood, it:

- creates a new `Task` or `Task<T>` object for the remainder of the **async** method
- assigns this new task as a continuation to the *awaited* task,
- assigns the context requirement for the continuation task

We first make two task methods that return a Task object

```
// asynchronous task 1
static async Task task1()
{
    Console.WriteLine("task1");
    await Task.Delay(1000);
}

// asynchronous task 2
static async Task task2()
{
    Console.WriteLine("task2");
    await Task.Delay(2000);
}
```

Each task has a delay

Our main method also has a async and returns a Task object and calls the 2 tasks

```
// using async and await
static async Task Main(string[] args)
{
    await task1();
    await task2();
}
```

run the above program you should get something like this:

```
task1
task2
```

async method returning a value

Our task methods can also return values using **Task<T>**

Our task methods now look like this:

```
// asynchronous task 1 returning a value
static async Task<string> task1()
{
```

```

        Console.WriteLine("starting task1");
        await Task.Delay(1000);
        return "task1";
    }

    // asynchronous task 2 returning a value
    static async Task<string> task2()
    {
        Console.WriteLine("starting task2");
        await Task.Delay(2000);
        return "task2";
    }
}

```

We are returning the names of the tasks

Our main method changes to return the task names and print them out.

```

// using async and await
static async Task Main(string[] args)
{
    // retrieve task names
    string name1 = await task1();
    string name2 = await task2();

    // print out task names
    Console.WriteLine(name1);
    Console.WriteLine(name2);
}

```

run the above program you should get something like this:

```

starting task1
starting task2
task1
task2
.

```

using async and await and lambda

We can replace our methods with lambdas. We use the keyword `async` on the lambda to turn the lambda into an `async` function. We use the `func` delegate because we have 1 output but no inputs.

```
// using async and await and lambda
static async Task Main(string[] args)
{
    Func<Task<string>> func1 = async () =>
    {
        Console.WriteLine("starting task1");
        await Task.Delay(1000);
        return "task1";
    };

    Func<Task<string>> func2 = async () =>
    {
        Console.WriteLine("starting task2");
        await Task.Delay(2000);
        return "task2";
    };

    // retrieve task names
    string name1 = await func1();
    string name2 = await func2();

    // print out task names
    Console.WriteLine(name1);
    Console.WriteLine(name2);
}
```

Run the above program you should get something like this:

```
starting task1
starting task2
task1
task2
.
```

244

HOMWORK 2

Create a task that uses an event to get an message from the keyboard, when the event is called call an asynchronous methods to print out the message.

Threads

Multitasking is the simultaneous execution of multiple tasks or processes over a certain time interval.

The operating system uses a term known as a *process* to execute all these applications at the same time. A process is a part of an operating system that is responsible for executing an application. Every program that executes on your system is a process and to run the code inside the application a process uses a term known as a *thread*.

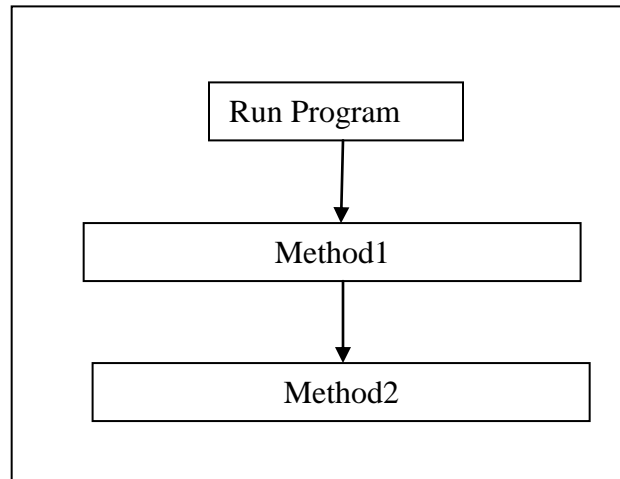
A thread is a lightweight process, or in other words, a thread is a unit which executes the code under the program.

Every program by default carries one thread to executes the logic of the program and the thread is known as the *Main Thread*, so every program or application is by default single-threaded model.

The single thread runs all the process present in the program in synchronizing manner, means one after another. So, the second process waits until the first process completes its execution, it consumes more time in processing.

We have a class that contains two different methods, i.e. *method1*, *method2*. Now the main thread is responsible for executing all these methods, so the main thread executes all these methods one by one.

Without multi-threading



To do:

Put the two following methods in your cs file

```
// method1
public static void method1()
{
    for (int i = 0; i < 5; i++)
    {
        Console.WriteLine("method 1");
    }
}

// method 2
public static void method2()
{
    for (int i = 0; i < 5; i++)
    {
        Console.WriteLine("method 2");
    }
}
```

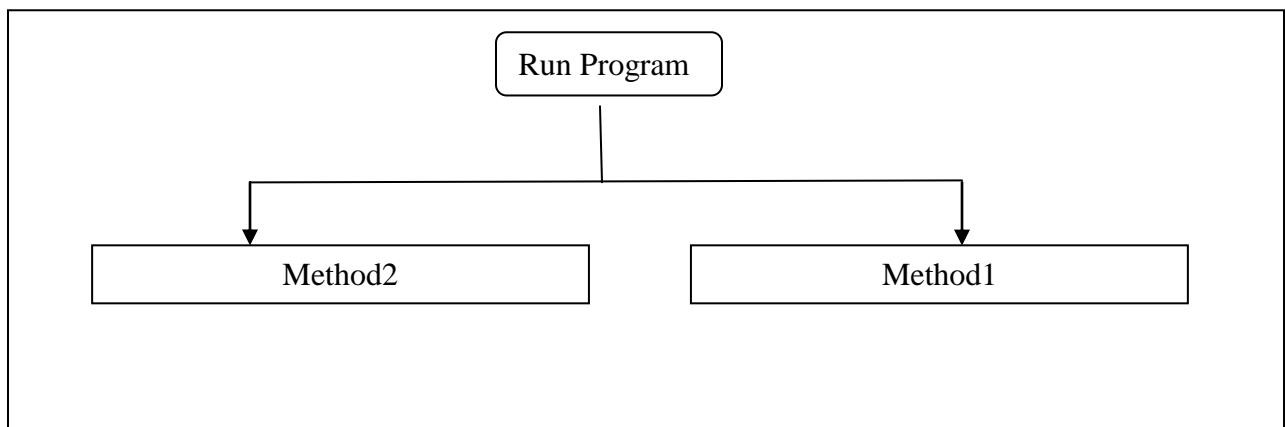
To do

Execute the following statements in your main method
You should get something like this:

```
Threading
Not using Threads
method 1
method 1
method 1
method 1
method 1
method 2
method 2
method 2
method 2
method 2
.
```

Using threads

Threads operate tasks in parallels (actually executes each task in turns)
That appear in operating in parallel.



We can make a Thread by calling the Thread class constructor with a method name

```
Thread thread1 = new Thread(method1);
```

Then we start the thread

```
thread1.Start();
```

we need to wait till the thread has finished executing the program will end before the thread completes executing

```
thread1.Join();
```

you may need to suspend a thread and let other threads execute with the Yield method

```
Thread.Yield();
```

```
// method1
public static void method1()
{
    for (int i = 0; i < 5; i++)
    {
        Console.WriteLine("method 1");
        Thread.Yield();
    }
}
```

Todo:

Type in the following code and run it

```
// Creating and initializing threads
```



```
Console.WriteLine("Using Threads");
Thread thread1 = new Thread(method1);
Thread thread2 = new Thread(method2);
thread1.Start();
thread2.Start();

// Join thread
thread1.Join();
thread2.Join();
```

You should get something like this:

```
Threading
using Threads
method 1
method 1
method 1
method 2
method 2
method 2
method 2
method 2
method 2
method 1
method 1
```

Question16 Homework

use 2 threads that receive an array. One thread will add up all the odds numbers and a second thread will add up all the even umbers. When both threads have finished executing add the two sums together and print out the results.

END

LESSON 17 PROJECT

Project 1 IntArray Class

Make an IntArray class to store an int values in an internal array called items. Make a default constructor that makes an empty array. Make another constructor that takes in the initial size of the array. Make another constructor the receives an ordinary array. You need to copy the elements in the receiving array to your internal array. Make another constructor that receives your IntArray. Again, you need to copy the elements in the receiving IntArray to your internal array. Make methods to access array elements by array index. Make operational methods to add items to the end, insert at a certain index, and remove at a certain index.

When adding and inserting items the internal array should just increase in size by 1. When removing items from the internal array just shift the other values down and set the last value to 0. Make operational methods to sort the array ascending and search for values. Use bubble sort to sort the array and use binary search to search for items in the internal array when it is sorted. You can find the code for bubble sort and binary search on the internet. Lastly make a toString method to print out the array elements enclosed in square brackets like this: [9 4 9 3 6 4 8] Make a TestArray class with a main method to test all the methods of your Array class or alternately for convenience put the main method inside your IntArray class.

Project 2 Int Matrix class

Make a Matrix class that has rows and column variables and a two-dimensional array of the specified rows and columns. Make a default constructor to make an empty Matrix of rows and columns. Have private variables to store rows and columns. Make another constructor that receives an ordinary two-dimensional array. You need to copy the elements in the receiving array. Make another constructor that receives your IntMatrix. Again, you need to copy the elements in the receiving IntMatrix. Make setters and getters to access the matrix elements. Make a toString method that will print out the matrices. Make operational methods to add, subtract, multiply, divide, transpose and rotate matrices by a specified rotation.

Make a TestMatrix class with a main method to test all the methods of your Matrix class.

Project 3 Spelling Corrector

Read in a text file with spelling mistakes, find the incorrectly spelled words and offer corrections. The user should be able to choose the correct choice from a menu. Look for missing or extra letters or adjacent letters on the keyboard. Download a word dictionary from the internet as to check for correctly spelled words. Use a Hash table to store the words. Store the correctly spelled file.

Project 4 MathBee

Make a Math bee for intermixed addition, subtraction, multiplication and division single digit questions. Use random numbers 1 to 9 and use division numbers that will divide even results. Have 10 questions and store results in a file. Keep track of the users score.

Project 5 Quiz App

Make a quiz app with intermixed multiple choice, true and false questions. You should have a abstract Question super class and two derived classes MultipleChoice and TrueAndFalse. Each derived class must use the abstract methods to do the correct operation. Store all questions in one file. Store the results in another file indicating the quiz results.

Project 6 Phone Book App

Make a phone book app that uses a HashMap to store Phone numbers and names. You need a Contact class to store name and phone number. You should be able to view, add, delete, scroll up and down contacts as menu operations. Contacts need to be displayed in alphabetically orders. Offer to lookup by name or by phone number. Contacts should be stored in a file, read when app runs, and saved with app finished running. Bonus, add email and address lookups as well.

Project 7 Appointment App

Make an Appointment book app that uses a HashMap to store Appointments. You need an Appointment class to store name, description and time. You should be able to view, add, delete, scroll up and down appointments as menu operations. Appoints need to be displayed in chronological orders. Appointments should be stored in a file, read when app runs, and saved with app finished running.

Project 8 GenericArray Class

Make the IntArray class to be a Generic Matrix class TArray so that it can store any data type. You need to instantiate generic array with a Comparable object to hold your array T items, since the java compiler cannot make a T object for you. You then need to type cast your created Comparable array to a T type array.

```
items = (T[])new Comparable[size];
```

Make a TestTArray class with a main method to test all the methods of your TArray class.

Project 9 Generic Matrix class

Make the IntMatrix class to be a Generic Matrix class TMatrix so that it can store any data type.

Project 10 Fraction class

Make a **Fraction** class that stores an int numerator and denominator. Make two constructors, a default constructor that initializes the numerator to 0 and initializes the denominator to 1, and an initializing constructor that initializes the numerator and denominator with user values.

Make methods **add**, **subtract**, **multiply** and **divide**, that receives two Fraction classes and return a Fraction class result. Your empty add method may look like this:

```
public Fraction add(Fraction f2)
{
    Fraction f3 new Fraction();
    return f3.
}
```

Make a **toString** method that returns a String representation of a fraction class as a whole number like 5, when the denominator is 1 and a fraction like 2/3 when the

denominator is not 1. Make a main or standalone Test class to test your Fraction class.

Bonus marks:

Make a **reduce** method using GCD that will reduce a fraction to lowest terms. You can make a iterative or recursive GCD private method as follows:

iterative GCD

```
static long gcd (long a, long b) {  
    long r, i;  
    while(b!=0){  
        r = a % b;  
        a = b;  
        b = r;  
    }  
    return a;  
}
```

recursive GCD

```
public int gcd(int a, int b) {  
    if (b==0) return a;  
    return gcd(b,a%b);  
}
```

Also make a **equals** method and a **compareTo** methods to test if two fractions are equal and to compare fractions if they are less or greater to each other. Test these features to your testing.

Project 11 Grocery Store App

Make a Grocery Store App where Customers can purchase items. Preferred customers get a discount. After all items have been entered a receipt is printed.

Step 1: Item class

Make a Item class with private variables product name, quantity ordered, price and discount Price.

If the item is not a discount item then the discount price is 0.

Make a constructor that will receive the item name, quantity, price and discount price.

Make getters and setters for each instance variable

Make a formatted toString method that will return item name price quantity discount price surrounded by round brackets and extension price like this:

Carrots 2 1.29 (.89) 2.58 (1.78)

Step 2: GroceryStore class

Make a GroceryStore class that will store items bought, total items bought, that total's the order and print out a receipt.

The Grocery Store class will store the customer's name, and all items bought in an ArrayList <Item> called items.

The Grocery store constructor will receive the customer name and create the ArrayList <Item> of items.

The grocery store will have a method to add an item object called add.

The grocery store class will also print out a receipt using a method called printReceipt.

The Grocery store class will have a getTotal method to return the total of all items. The getTotal method can also be used to print out the receipt total.

All instance variables are private and you cannot have any getters and setters.

Step 3: DiscountStore class

The DiscountStore class inherits from the GroceryStore class.

The DiscountStore receives the CustomerName and sends the CustomerName to the GroceryStore super class.

If the customer is a preferred customer then the DiscountStore class is used. The DiscountStore class will calculate discount percent, and count of discount items and total of all items using the discount price rather than the regular price. The discount class will override the getTotal class of the grocery store class. The discount store class will also print a receipt showing the number of discount, items and the discount percent obtained.

Step 4 GroceryApp class.

The GroceryApp class is the main class where the cashier enters the customer items, bought.

The cashier will ask if the customer is a preferred customer. if it is a preferred customer then the DiscountStore class is used else the Grocery Store class Is used.

The cashier will enter the items bought. Once all items have been enters then the receipt is printed out.

You will need to store a list of products in a file to simulate the entering of products or make an array of items like this:

```
Item[] items = {new Item("apples",2,1.26,1.08)
new Item("oranges",2,1.26,1.08),
new Item("carrots",2,1.26,1.08)
new Item("apple",2,1.26,1.08)};
```

The file format will be like this

Customer name
Preferred or not preferred
Number of products
Item name, quantity, price, discount price

Example file:

Tom Smith
Preferred
3
Carrots , 2.49, 1.78, 2

Fish,12.67, 11.89,3

Milk 4.89.3.75, 2

In either case the items will be added to the store.

The main method will have a menu as follows:

- (1) add items to Grocery store
- (2) add items to Discount store
- (3) print receipt
- (4) exit program

END