C#_2.1 Class

A ***class*** is a ***template*** that defines the ***form*** of an ***object***. ***Objects*** are ***instances of a class***. A ***class*** is a ***logical abstraction***. ***Methods*** and ***variables*** that constitute a class are called ***members*** of the class. A class is created by using the keyword ***class***. The general form of a class definition that contains only instance variables and methods is:

```
class classname {
        access type var1;     // Declare instance variables.
        access type var2;
        // ...
        access type varN;

        access ret-type method1(parameters) { /* body of method */ }     // methods.
        access ret-type method2(parameters) { /* body of method */ }
        // ...
        access ret-type methodN(parameters) { /* body of method */ }
}
```

- Here, ***access*** is an ***access specifier***, such as ***public***. [Different in JAVA]
- The general form for declaring an instance variable is:
  access type var-name;
  Here, ***access*** specifies the access, ***type*** specifies the type of variable, and ***var-name*** is the variable's name.

Example: **class** Vehicle { **public int** Passengers;
public int FuelCap;
public int Mpg; }

- C# defines several specific ***flavors of members***, which include ***instance*** variables, ***static*** variables, ***constants***, ***methods***, ***constructors***, ***destructors***, ***indexers***, ***events***, ***operators***, and ***properties***.
- To actually create a ***Vehicle*** object, you will use a statement like: `Vehicle minivan = new Vehicle();` [Which is similar to Java]
- Each time you create an ***instance*** of a class, you are creating an ***object*** that contains its ***own copy*** of ***each instance*** variable defined by the class. Each ***object*** has its own ***copies*** of the ***instance variables*** defined by its ***class***. Thus, the ***contents*** of the ***variables*** in one object can ***differ*** from the contents of the variables in another. There is ***no connection*** between the two objects, except for the fact that they are both ***objects*** of the ***same type***.
- Use the dot operator to access instance variables and methods. Eg: `minivan.FuelCap = 16;` General form of the dot operator is: `object.member`
- access to a class can be controlled by using ***access specifiers.***

❒ ***How Objects Are Created (Similar to Java part 2.1):*** ***Class_name object_name = new Class_name ();***

- First, it declares a variable of the ***class type***. This variable is ***not***, itself, an ***object***. Instead, it is simply a ***variable*** that can ***refer*** to an ***object***.
- Second, the ***declaration*** creates an actual, ***physical instance*** of the ***object***. This is done by using the ***new*** operator. Finally, it assigns to object a ***reference*** to that object. The ***new*** operator ***dynamically allocates*** (that is, allocates at runtime) memory for an object and returns a reference to it. This reference is then stored in a variable. Thus, in C#, all class objects must be dynamically allocated.
- `Class_name object_name = new Class_name ();` can be written as
  `Class_name object_name;` *// declare a reference to an object*
  `object_name = new Class_name ();` *// allocate a object*
- The fact that ***class objects*** are accessed through a reference explains why classes are called ***reference types***.

❒ ***Reference Variables and Assignment:*** Like ***Java***, where instances are references of same object. Recall ***Java part 2.2.***

C#_2.2 Methods and Returning from a Method

The basic ideas are ***same*** as ***JAVA***. The general form of a method is shown here: ***access ret-type name(parameter-list) {*** */* body of method */* ***}***

- ***access*** is an access modifier. If any access modifier is ***not present***, the method is ***private*** to the class in which it is declared.
- The ***ret-type*** specifies the type of data returned by the method. If the method ***does not return a value***, its return type must be ***void***.
- The ***parameter-list*** is a sequence of type and identifier pairs separated by ***Commas***. If the method has no parameters, the parameter list will be ***empty***.

❒ ***Return from a Method:*** ***Return from a void*** Method (methods which don't return values) and ***Return a value*** from a Method. Same as JAVA/ ***Recall 2.3*** Java part.

- ***Unreachable code warning:*** The C# compiler will issue a ***warning message*** if you create a method that contains code that no path of execution ***will ever reach*** (Eg: due to *immediate return* from a ***return statement***). Consider this example:

```
public void m() { char a, b;  // ...
        if(a==b) { Console.WriteLine("equal"); return; }
        else { Console.WriteLine("not equal"); return; }
        Console.WriteLine("this is unreachable"); }
```

Here, the method ***m()*** will always return before the final ***WriteLine()*** statement is ***executed***. If you try to ***compile*** this method, you will receive a ***warning***. In general, ***unreachable code*** constitutes a ***mistake*** on your part, so it is a good idea to take unreachable code warnings seriously!

C#_2.3 Methods with parameters (same as Java, Recall 2.4 Java part)

C#_2.4 Constructor and Destructor and NEW operator

A ***constructor*** initializes an object when it is created [Same as ***Java***]. It has the same name as its class and is syntactically similar to a method. However, constructors have no ***explicit return type***. The general form: ***access class-name(param-list) {*** */* constructor code */* ***}***

- Use a constructor to give ***initial values*** to the ***instance variables*** of a class, or to perform any other ***startup procedures*** required to create a fully formed object.
- Often, ***access*** is ***public*** because a constructor is usually called from outside its class. The ***parameter-list*** can be ***empty***, or it can specify one or more parameters.

❒ ***C# has default constructor as JAVA:*** C# automatically provides a ***default constructor*** that causes all member variables to be initialized to their ***default values***. For most ***value types***, the default value is ***zero***. For ***bool***, the default is ***false***. For ***reference types***, the default is ***null***. However, once you define your own constructor, the *default constructor* is no longer used.

❒ ***The new Operator In object declaration (Same as 2.6 Java part):*** ***new*** operator has this general form: ***new class-name(arg-list)***

- Here, ***class-name*** is the name of the class that is being instantiated. The class name followed by parentheses specifies the ***constructor*** for the class. If a class does not define its ***own constructor***, ***new*** will use the ***default constructor*** supplied by ***C#***.
- Since ***memory*** is ***finite***, it is possible that ***new*** will not be able to ***allocate memory*** for an object because ***insufficient memory*** exists. If this happens, a ***runtime exception*** will occur.
- In C#, a variable of a ***value type*** contains its own ***value***. Memory to hold this value is ***automatically provided*** when the program is ***run***. Thus, there is no need to ***explicitly allocate*** this memory using ***new***. Conversely, a ***reference variable*** stores a reference to an object. The memory to hold this object is allocated ***dynamically*** during ***execution***.

❒ ***Garbage Collection and Destructors:*** ***Garbage Collection*** is similar as ***JAVA*** (2.7 Java part). ***Destructors*** in C# is more like ***Garbage Collection***, it not act as C++'s ***destructors***. ***Destructor*** is called just prior to ***garbage collection***. It is ***not called*** when a ***variable*** containing a ***reference*** to an object ***goes out of scope***. *[Destructors in C++ are called when an object goes out of scope.]* You cannot know precisely when a destructor will be executed. It is possible that a program to end *before garbage collection occurs*, so a *destructor might not get called at all.* Destructors have this general form: ***~class-name( ) {*** */* destruction code */* ***}***

- Here, ***class-name*** is the name of the class. Thus, a ***destructor*** is declared like a ***constructor***, except that it is preceded with a **~** (tilde). Notice it has ***no return type***.

C#_2.5 this Keyword (Similar to Java , Recall 2.8 Java part)

When a method is called, it is ***automatically passed a reference*** to the ***invoking object*** (that is, the object on which the method is called). This ***reference*** is called ***this***. Therefore, ***this*** refers to the ***object*** *on which* the ***method*** is ***acting***.

C#_2.6 Arrays (same as Java)

One-Dimensional Arrays: general form: `type[ ] array-name = new type[size];` Which is similar to object declaration. [Recall 2.9 Java part.] Here, ***type*** declares the ***element type*** of the array. Notice the ***square brackets*** that follow ***type***. They indicate that a ***reference*** to a ***one-dimensional array*** is being declared. The *number of elements* that the array will hold is determined by ***size***.

- ***Initialize an Array:*** The general form for initializing a one-dimensional array is: `type[ ] array-name = { val1, val2, val3, ... , valN };`
  - Here, the initial values are specified by ***val1*** through ***valN***. They are assigned in sequence, left to right, in index order.
  - C# ***automatically allocates*** an array large enough to hold the ***initializers*** that you specify. There is no need to explicitly use the ***new*** operator. Although not needed, you can use ***new*** when initializing an array. For example, this is a proper, but redundant, way to initialize nums:
    `int[] nums = new int[] { 99, -10, 100123, 18, -978, 5623, 463, -9, 287, 49 };`
  - While redundant here, the ***new*** form of array initialization is useful when you are assigning a new array to an already existent array reference variable. Eg:

```
int[] nums;
nums = new int[] { 99, -10, 100123, 18, -978, 5623, 463, -9, 287, 49 };
```

    In this case, ***nums*** is declared in the ***first*** statement and ***initialized*** by the ***second***.
- ***Boundaries Are Enforced:*** Array boundaries are strictly enforced in ***C#***; it is a ***runtime error*** to ***overrun*** or ***underrun*** the ***end*** of an array.
- ***Two-Dimensional Arrays:*** To declare a ***two-dimensional*** integer array ***table*** of size ***10***, ***20***, you would write `int[,] table = new int[10, 20];`
  - Notice that the two dimensions are separated by a ***comma***. The syntax `[,]` indicates that a two-dimensional array reference variable is being created. When memory is actually allocated for the array using ***new***, this syntax is used: `int[10, 20]`.
  - To access an element in a two-dimensional array for example, to assign the value ***10*** to location ***3, 5*** of array table, you would use `table[3, 5] = 10;`
- ***Arrays of Three or More Dimensions:*** Here is the general form of a multidimensional array declaration:
  `type[,...,] name = new type[size1,size2,...,sizeN];`
- ***Jagged Arrays/ Irregular arrays:*** A jagged array is an array of arrays in which the length of each array can differ. Thus, a jagged array can be used to create a table in which the row lengths are not the same. ***Recall 2.10 Java part irregular array.*** Jagged arrays are declared by using sets of square brackets to indicate each dimension. For example, to declare a two-dimensional jagged array, you will use this general form: `type[ ] [ ] array-name = new type[size][ ];` Here, ***size*** indicates the number of rows in the array. The *rows themselves have not been allocated*. Instead, the *rows are allocated individually*. This allows for the length of each row to vary.
- ***Assign Array References:*** As with other objects, when you ***assign*** one ***array reference variable*** to another, you are simply making ***both variables refer to the same array***. You are *not causing a copy* of the array to be made, *nor* are you causing the *contents* of one array to be *copied* to the other.
- ***Use the Length Property with Arrays (Same as 2.11 JAVA): Length*** property of an array contains the *number of elements that an array can hold*. Eg:
  `int[] list = new int[10]; Console.WriteLine("length of list is " + list.Length);`

C#_2.6 Implicitly Typed Array

An ***implicitly typed array*** is declared using the keyword ***var***, but you do not follow ***var*** with ***[ ]***. Furthermore, the array ***must be initialized***. It is the type of ***initializer*** that ***determines*** the element ***type of the array***. All of the ***initializers*** must be of the ***same*** or a ***compatible type***. Example of an implicitly typed array:

`var vals = new[] { 1, 2, 3, 4, 5 };`

- This creates an array of ***int*** that is five elements long. A ***reference*** to that array is assigned to ***vals***. Thus, the type of ***vals*** is "***array of int***" and it has five elements.
- Notice that ***var*** is not followed by ***[ ]***. Also, even though the *array is being initialized*, you must include ***new[ ]***. It's not optional in this context.

Here is another example. It creates a two-dimensional array of ***double***. ***var vals = new[,] { {1.1, 2.2}, {3.3, 4.4},{ 5.5, 6.6} };*** In this case, ***vals*** has the dimensions ***2*** by ***3***.

Implicitly typed arrays are most applicable to ***LINQ***-based queries. They are *not meant for general use*. In most cases, you should use explicitly typed arrays.

C#_2. 7 For-each loop (Recall 2.13 JAVA part)

The ***foreach*** loop is used to cycle through the elements of a ***collection***. A ***collection*** is a ***group of objects***. C# defines several types of collections, of which one is an ***array***. The general form of ***foreach*** is: `foreach(type loopvar in collection) statement;`

- Here, type ***loopvar*** specifies the ***type*** and ***name*** of an ***iteration variable***. The ***iteration variable*** receives the value of the ***next element*** in the ***collection*** each time the ***foreach*** loop ***iterates***. The collection being cycled through is specified by ***collection*** (here we used ***array*** as ***collection***). Thus, type must be the same as (or compatible with) the ***element type*** of the ***array***. [type can also be ***var***, in which case the ***compiler*** determines the type based on the element ***type*** of the ***array***.]
- When the ***loop*** begins, the first element in the array is obtained and assigned to ***loopvar***. Each subsequent iteration obtains the next element from the array and stores it in ***loopvar***. The loop ends when there are no more elements to obtain. Two examples of foreach lopp are given below.
- The ***iteration*** variable ***loopvar*** is ***read-only***. This means that you can't change the ***contents*** of an ***array*** by assigning the ***iteration variable*** a ***new value***.

```
foreach(int x in nums) {     Console.WriteLine("Value is: " + x);
                             sum += x; }
```

```
foreach(int val in nums) {   if(val < min) min = val;
                             if(val > max) max = val; }
```

C#_2.8 strings (Recall Java part 2.14, 2.15,)

In C#, ***strings*** are ***objects***. Thus, ***string*** is a ***reference*** type of the ***string class*** (Same as Java). Once you have created a ***string object***, you can use it nearly anywhere that a ***string literal*** is allowed. ***For example:*** here ***str*** is a *string reference variable* that is assigned a ***reference*** to a ***string literal***: `string str = "C# strings are powerful.";` In this case, ***str*** is initialized to the character sequence "*C# strings are powerful.*"

- You can also ***create a string*** from a ***char array***. For example: `char[] chrs = {'t', 'e', 's', 't'}; string str = new string(chrs);`

- ***Operating on Strings:*** The string class contains several methods that operate on strings. Here are a few:

`static string Copy(string str)`	Returns a ***copy*** of ***str***.
`int CompareTo(string str)`	Returns ***less than zero*** if the invoking *string is less than str*, ***greater than zero*** if the invoking *string is greater than str*, and ***zero*** if the *strings are equal*.
`int IndexOf(string str)`	***Searches*** the invoking string for the ***substring*** specified by ***str***. Returns the ***index*** of the first ***match***, or –1, on ***failure***.
`int LastIndexOf(string str)`	***Searches*** the invoking string for the ***substring*** specified by ***str***. Returns the ***index*** of the last ***match***, or –1, on ***failure***.

  - The ***string type*** also includes the ***Length property***, which contains the ***length of the string***.
  - To ***obtain*** the ***value of an individual character*** of a string, you simply ***use*** an ***index***. For example: `string str = "test"; Console.WriteLine(str[0]);`
  - ***CompareTo( ) and ==, !=*** : To test two strings for ***equality***, you can use the = = operator. Normally, when the = = operator is applied to ***object references***, it determines if ***both references*** refer to the ***same object***. This ***differs*** for objects of ***type string***. When the = = is applied to two ***string references***, the ***contents of the strings themselves*** are ***compared*** for ***equality***. The same is true for the != operator: When comparing ***string objects***, the ***contents of the strings*** are compared. For other types of ***string comparisons***, you will need to use the `CompareTo()` method.
- ***Arrays of Strings:*** Like any other data type, strings can be assembled into arrays.
- ***Strings Are Immutable:*** In ***C#*** the ***contents*** of a string object are ***immutable***. i.e, once created, the *character sequence* comprising that string *can't be altered*.
  - ***To create a string that can be changed:*** C# offers a class called ***StringBuilder*** that is in the ***System.Text*** namespace. It creates string objects that can be changed.

- ***ToCharArray():*** `Copies the characters` in this instance to a **Unicode character array**. General form: `public char[] ToCharArray ();`
  - Returns `Char[]`. A *Unicode character array* whose elements are the **individual characters** of this **instance**. If this instance is an **empty string**, the returned array is **empty** and has a **zero length**.
  - This method copies each **character** (that is, each **Char object**) in a **string** to a **character array**. The ***first*** *character copied is at* ***index zero*** of the ***returned character array***; the last character copied is at index **Array.Length - 1**.
  - To ***create a string from*** the characters in a ***character array***, call the `String(Char[])` constructor.
- ***ToCharArray(Int32, Int32):*** `Copies the characters` in a specified substring in this instance to a Unicode character array. General form:

`public char[] ToCharArray (int startIndex, int length);`

Parameters:	`startIndex Int32`	The starting position of a substring in this instance.	`length Int32`	The length of the substring in this instance.

  - Returns `Char[]`. A *Unicode character array* whose elements are the ***length number*** of characters in this instance starting *from character position* ***startIndex***.
  - This method copies the **characters** in a portion of a **string** to a ***character array***.
  - The **startIndex** parameter is **zero-based**. That is, the index of the **first character** in the **string** instance is **zero**. If *length is* ***zero***, the returned array is **empty** and has a zero length. If this instance is **null** or an **empty string** (""), the returned array is **empty** and has a **zero length**.
  - To create a **string** from a *range* of **characters** in a *character array*, call the `String(Char[], Int32, Int32)` constructor.

Exceptions: `ArgumentOutOfRangeException` **startIndex** or **length** is less than **zero**. Or, **startIndex** plus length is *greater than* the **length** of this **instance**.

C#_2.9 The Bitwise Operators (Same as Java part 2.17)

The ***bitwise*** operators act directly upon the bits of their operands. They are defined ***only for integer*** operands. They ***cannot*** be used on ***bool***, ***float***, ***double***, or ***class*** types. Bitwise operations are important to a wide variety of systems-level programming tasks, such as when status information from a device must be interrogated or constructed.

Operator	&	|	^	>>	<<	~
Result	Bitwise AND	Bitwise OR	Bitwise XOR	Shift right	Shift left	One's complement (unary NOT)

C#_2.10 The ? Operator

The ***?*** operator is often used to replace ***if-else*** statements of this general form:

```
if (condition) variable = expression1;
else variable = expression2;
```

The ***?*** takes the general form, `Exp1 ? Exp2 : Exp3;` where ***Exp1*** is a ***bool*** expression, and ***Exp2*** and ***Exp3*** are ***expressions***. The ***type*** of ***Exp2*** and ***Exp3*** must be the ***same*** (or ***compatible***). Notice the use and placement of the ***colon***.

C#_2.10 C#'s Access Modifiers/Specifiers (In Java *public* is default, In C#/C/C++ *private* is default)

Member access control is achieved through the use of ***four access modifiers***: ***public***, ***private***, ***protected***, and ***internal***.

- The ***protected*** modifier applies only when inheritance is involved.
- The ***internal*** modifier applies mostly to the use of an assembly, which for C# loosely means a deployable program or library.
- When a member of a class is modified by the ***public*** specifier, that member can be accessed by any other code in your program. This includes methods defined inside other classes.
- When a member of a class is specified as ***private***, then that member can be accessed only by other members of its class. Thus, methods in other classes are not able to access a private member of another class.
- If ***no access specifier*** is used, a class member is ***private*** to its class by ***default***. Thus, the *private specifier is optional* when creating private class members.
- An access specifier *precedes the rest of a member's type specification*. That is, it must ***begin*** a member's ***declaration statement***.

C#_2.11 Pass an Object Reference to a Method

Object references can be passed to methods. Consider the following Example:

```
using System;
class Block { int a, b, c, volume;
public Block(int i, int j, int k) { a = i; b = j; c = k; volume = a * b * c; }

public bool SameBlock(Block ob) { if((ob.a == a) & (ob.b == b) & (ob.c == c)) return true; else return false; }
public bool SameVolume(Block ob) { if(ob.volume == volume) return true; else return false; }
}
```

```
class PassOb { static void Main() { Block ob1 = new Block(10, 2, 5);
                                    Block ob2 = new Block(10, 2, 5);
                                    Block ob3 = new Block(4, 5, 5);
Console.WriteLine("ob1 dim= ob2: " + ob1.SameBlock(ob2));
Console.WriteLine("ob1 dim= ob3: " + ob1.SameBlock(ob3));
Console.WriteLine("ob1 vol= ob3: " + ob1.SameVolume(ob3)); }}
```

C#_2.12 *CALL-BY-VALUE* and *CALL-BY-REFERENCE*

- ***call-by-value:*** This method copies the value of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument.
- ***call-by-reference:*** In this method, a reference to an argument (not the value of the argument) is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine.
  - By default, C# uses ***call-by-value***, when you pass a ***value type***, such as ***int*** or ***double***.
  - When an ***object reference*** is passed to a method, the ***reference*** itself is passed by use of ***call-by-value***. Thus, a copy of that ***reference*** is made. However, since the value being passed ***refers*** to an ***object***, the copy of that value will still ***refer*** to the ***same object*** as its corresponding argument.
- ***ref and out Parameters:*** By default, ***value types***, such as ***int*** or ***char***, are ***passed by value*** to a method. Through the use of the ***ref*** and ***out*** keywords, it is possible to pass any of the ***value types*** by ***reference***. Doing so allows a method to alter the argument used in the call.
  - ***ref :*** The ***ref*** modifier makes a method to be *able to operate on the actual arguments* that are passed to it.
  - ***out :*** A method can return only one value each time it is called. If you need to return two or more pieces of information use the ***out*** modifier.
  - ***Using ref:*** The ***ref*** *parameter modifier* causes C# to create a ***call-by-reference*** rather than a ***call-by-value***. The ***ref*** modifier is used when the method is ***declared*** and when it is ***called***. The following program creates a method called ***Sqr()*** that returns in place the square of its integer argument. Notice ***use*** and ***placement*** of ***ref***.

```
using System;
class RefTest { public void Sqr(ref int i) { i = i * i; }  /* ref precedes the parameter declaration */    }
class RefDemo { static void Main() {   RefTest ob = new RefTest();
                                       int a = 10;
                                       Console.WriteLine("a before call: " + a);
                                       ob.Sqr(ref a); /* ref precedes the argument */
                                       Console.WriteLine("a after call: " + a);    }}
```

```
OUTPUT:
a before call: 10
a after call: 100
```

  - Notice that ***ref*** precedes the ***entire parameter declaration*** in the method and that it ***precedes*** the ***name of the argument*** when the method is called.
  - Sometimes you will want to use a ***reference parameter*** to receive a value from a method but ***not pass in*** a value. For example, you might have a method that performs some function, such as opening a ***network socket***, that returns a ***success/fail*** code in a reference parameter.

- In this case, *there is no information to pass* into the method, but there is *information to pass back out*.
- The problem with this scenario is that a ***ref*** parameter ***must be initialized*** to a value ***prior to the call***. Thus, to use a ***ref*** parameter would require giving the argument a ***dummy value*** just to ***satisfy*** this constraint. Fortunately, C# provides a better alternative: the ***out*** parameter.

Using out: An ***out*** parameter is similar to a ***ref*** parameter with this one exception: It can *only be used to pass a value out of a method*. It is not necessary (or useful) to ***give the variable*** used as an ***out parameter*** an initial value prior to ***calling the method***. The ***method*** will ***give*** the variable a ***value***.

- Inside the method, an ***out*** parameter is always considered unassigned; that is, it is assumed to have ***no initial value***. Instead, the method ***must assign*** the parameter a ***value prior*** to the method's ***termination***. Thus, after the call to the method, the variable referred to by an ***out*** parameter will contain a value.
- Here is an example that uses an ***out*** parameter. The method ***RectInfo()*** returns the ***area of a rectangle*** given the ***lengths*** of its ***sides***. In the parameter ***isSquare***, it returns ***true*** if the rectangle is a ***square*** and ***false*** otherwise. Thus, ***RectInfo()*** returns ***two pieces of information*** to the caller.

```
using System;
class Rectangle {     int side1, side2;
          public Rectangle(int i, int j) {  side1 = i;  side2 = j; }   /* Following return area and determine if square. */
          public int RectInfo(out bool isSquare) {   /* Pass information out of the method via an out parameter. */
                    if(side1==side2) isSquare = true;
                    else isSquare = false;
                    return side1 * side2; }
}
```

```
class OutDemo { static void Main() {       Rectangle rect = new Rectangle(10, 23);
                                           int area; bool isSqr;
                                           area = rect.RectInfo(out isSqr);
                                           if(isSqr) Console.WriteLine("rect is a square.");
                                           else Console.WriteLine("rect is not a square.");
                                           Console.WriteLine("Its area is " + area + "."); }}
```

Notice ***isSqr*** is not assigned a value ***prior*** to the call to ***RectInfo()***. This would not be allowed if the parameter to ***RectInfo()*** had been a ***ref*** rather than an ***out*** parameter. After the method ***returns***, ***isSqr*** contains either ***true*** or ***false***, depending upon whether the rectangle is square or not. The area is returned via ***return***.

ref and out can be used on reference-type parameters: When ref or out modifies a reference-type parameter, it causes the reference itself to be passed by reference. This allows a method to change what the reference is referring to.

```
class Test { public int a;
          public Test(int i) { a = i; }

     // This will not change the argument.
          public void NoChange(Test o) { Test newob = new Test(0);
          o = newob; /* this has no effect outside of NoChange()*/      }

     // This will change what the argument refers to.
          public void Change(ref Test o) { Test newob = new Test(0);
          o = newob; /* this affects the calling argument. */       }
          }
```

```
class CallObjByRef {
          static void Main() { Test ob = new Test(100);
                    Console.WriteLine("ob.a before call: " + ob.a);

                    ob.NoChange(ob);
                    Console.WriteLine("ob.a after call to NoChange(): " + ob.a);

                    ob.Change(ref ob);
                    Console.WriteLine("ob.a after call to Change(): " + ob.a); }}
```

C#_2.13 Variable Number of Arguments : params modifier

To create a method that can be passed an ***arbitrary number of arguments*** you must use a special type of parameter: a ***params*** parameter. The ***params*** modifier is used to declare an ***array parameter*** that will be able to receive ***zero*** or ***more*** arguments. The ***number of elements*** in the ***array*** will be equal to the ***number of arguments*** passed to the ***method***. Your program then ***accesses*** the ***array*** to obtain the ***arguments***.

Here is an example that uses ***params*** to create a method called ***MinVal()***, which returns the ***minimum value*** from a set of values:

```
using System;
class Min { public int MinVal(params int[] nums) {              int m;
          if(nums.Length == 0) { Console.WriteLine("Error: no args."); return 0; }

          m = nums[0];
          for(int i=1; i < nums.Length; i++) if(nums[i] < m) m = nums[i];   /*finds min*/
          return m;         }
}
```

```
class ParamsDemo { static void Main() {
                                        Min ob = new Min();
                                        int min;
min = ob.MinVal(10, 12, -1);  Console.WriteLine(min);
min = ob.MinVal(18, 23, 3, 14, 25);  Console.WriteLine(min);
int[] args = { 45, 67, 34, 9, 112, 8 };  // call with an int array, too.
min = ob.MinVal(args);  Console.WriteLine(min); }}
```

- In a ***params*** parameter, all ***arguments*** must be of a ***type compatible*** with the ***array type*** specified by the ***parameter***. For example, calling ***MinVal()*** like this: ***min = ob.MinVal(1, 2.2);*** is ***illegal*** because there is no ***implicit conversion*** from ***double*** (2.2) to ***int***, which is the type of ***nums*** in ***MinVal()***.
- When using ***params***, you need to be careful about ***boundary conditions*** because a ***params*** parameter can accept any number of arguments—even ***zero***! This is why ***there is a check*** in ***MinVal()*** to confirm that at least one element is in the ***nums array*** before there is an attempt to access that element. If the check was not there, a ***runtime exception*** would result if ***MinVal()*** were called with ***no arguments***.
- A method can have ***normal parameters*** and a ***variable-length parameter***. In cases where a method has ***regular parameters*** and a ***params parameter***, the ***params*** parameter must be the ***last one*** in the *parameter list*. Furthermore, ***in all situations, there must be only one params parameters***. For example,

```
public void ShowArgs(string msg, params int[] nums) { }
```

C#_2.14 Returning Objects (Similar to Java part 2.23)

C#_2.15 Method and Constructor Overloading (Same Java part 2.24 & 2.25)

Method Overloading: To overload a method, simply declare different versions of it. The compiler takes care of the rest. The ***type*** and/or number of the ***parameters*** of each overloaded method ***must differ***. It is ***not sufficient*** for two methods to differ ***only*** in their ***return*** types. They must ***differ*** in the ***types*** or number of their ***parameters***.

- ***Implicit type conversions*** also apply to *parameters* of *overloaded* methods.
- Both ***ref*** and ***out*** participate in *overload resolution*. For example, the following define two distinct and separate methods:

```
public void MyMeth(int x) { Console.WriteLine("Inside MyMeth(int): " + x); }
public void MyMeth(ref int x) { Console.WriteLine("Inside MyMeth(ref int): " + x); }
```

  - Thus, ***ob.MyMeth(i)*** invokes ***MyMeth(int x)***, but ***ob.MyMeth(ref i)*** invokes ***MyMeth(ref int x)***.

- Although ***ref*** and ***out*** participate in overload resolution, the difference between the two alone is not sufficient. Thus, these two versions of ***MyMeth()*** are invalid:

```
public void MyMeth(out int x) { // ...
public void MyMeth(ref int x) { // ...
```

  - In this case, the compiler cannot differentiate between the versions of ***MyMeth()*** simply because one uses ***ref*** and one uses ***out***.

- ***Signature used in C# is same as JAVA***
- ***Overloading Constructors same as JAVA***
- ***Invoking an Overloaded Constructor Through this:*** When working with ***overloaded constructors***, it is sometimes useful for one constructor to ***invoke*** another. In C#, this is accomplished by using another form of the ***this*** keyword. The general form is shown here:

```
constructor-name(parameter-list1) : this(parameter-list2) { /*... body of constructor, which may be empty */ }
```

☞ When the ***constructor*** is executed, the overloaded constructor that matches the parameter list specified by ***parameter-list2*** is **first executed**. Then, if there are any statements inside the ***original constructor***, they are executed. Here is an example:

```
class XYCoord {     public int x, y;
                    public XYCoord() : this(0, 0) { Console.WriteLine("Inside XYCoord()"); } /* Use this to invoke an overloaded constructor */
                    public XYCoord(XYCoord obj) : this(obj.x, obj.y) { Console.WriteLine("Inside XYCoord(XYCoord obj)"); }
                    public XYCoord(int i, int j) {  Console.WriteLine("Inside XYCoord(XYCoord(int, int)");   x = i; y = j; }        }
```

▶ In the ***XYCoord*** class, the only constructor that actually initializes the ***x*** and ***y*** fields is ***XYCoord(int, int)***. The other two constructors simply invoke ***XYCoord(int, int)*** through ***this.***

☞ ***Invoking overloaded constructors*** through this can be useful is that it can prevent the ***unnecessary duplication*** of code.

☞ You can create constructors with implied "***default arguments***," which are used when these arguments are not explicitly specified. For example, you could create another ***XYCoord*** constructor, as shown here: ***public XYCoord(int x) : this(x, x) { }*** This constructor automatically defaults the ***y*** coordinate to the same value as the ***x*** coordinate. Of course, it is wise to use such "***default arguments***" carefully because their misuse could easily confuse users of your classes.

C#_2.16 Returning Values from Main( )

When a program ends, you can return a value to the calling process (often the operating system) by ***returning a value*** from ***Main( )***. To do so, you can use this form of ***Main( )***: ***static int Main()*** Notice that instead of being declared ***void***, this version of ***Main( )*** has a return type of ***int***. Usually, the ***return value*** from ***Main( )*** indicates whether the program ended ***normally*** or due to some ***abnormal condition***. By convention, a ***return*** value of ***0*** usually indicates ***normal termination***. All other values indicate that some type of ***error occurred***.

C#_2.17 Passing Arguments to Main( )

A command-line argument is the information that ***directly follows the program's name*** on the command line when it is ***executed***. For C# programs, these arguments are then passed to the ***Main( )*** method. To receive the arguments, you must use one of these forms of ***Main( )***:

static void Main(string[ ] args)
static int Main(string[ ] args)

☛ The ***first*** form returns ***void***; the ***second*** can be used to ***return an integer*** value. For both, the ***command-line arguments*** are stored as ***strings*** in the ***string array*** passed to ***Main( )***. The ***length*** of the ***args*** array will be equal to the *number of command-line arguments*.

☯ For example, the following program displays all of the command-line arguments that it is called with:

```
using System;
class CLDemo { static void Main(string[] args) {
        Console.WriteLine("There are " + args.Length + " command-line arguments.");
        Console.Write ("They are: ");
        for(int i=0; i < args.Length; i++) Console.WriteLine(args[i]); }          }
```

If CLDemo is executed like this: ***CLDemo one two three***

OUTPUT: There are 3 command-line arguments. They are:
one
two
three

C#_2.18 Recursion (Same as Java 2.26)

```
class Factorial { public int FactR(int n) { if(n==1) return 1;  else return FactR(n-1) * n; }
```

❒ When a method calls itself, new ***local variables*** and ***parameters*** are allocated storage on the ***system stack***, and the method code is executed with these new variables from the ***start***. As each recursive call returns, the ***old local variables*** and ***parameters*** are ***removed*** from the ***stack***, and execution resumes at the point of the call inside the method. Recursive methods could be said to "telescope" out and back.

❒ Too many recursive calls to a method could cause a ***stack overrun***. Because storage for ***parameters*** and ***local variables*** is on the ***stack*** and each ***new call*** creates a ***new copy*** of these variables, it is possible that the ***stack*** could be ***exhausted***. If this occurs, the ***CLR*** will throw an ***exception***.

❒ For ***recursive methods***, you must have a ***conditional statement*** somewhere, such as an ***if***, to force the method to ***return*** without the ***recursive*** call being ***executed***.

C#_2.19 Static in C# (same as Java part 2.27)

It is important to understand that although a static method cannot directly call instance methods or access instance variables of its class, it can call an instance method or access an instance variable if it does so through an object of its class. It just cannot use an instance variable or method directly, without an object qualification.

❒ ***Static Constructors and Classes:*** It is possible to specify a constructor as static. For example, **class** Sample{ /*... */ **static** Sample(){ ...

▶ Here, ***Sample( )*** is declared ***static*** and is, therefore, a ***static constructor*** for the ***Sample class***.

☛ A ***static constructor*** is called ***automatically*** when the class is ***first loaded***, before any ***objects*** are created and before any ***instance constructors*** are called. Thus, the primary use for a ***static constructor*** is to ***initialize features*** that apply to the class as a ***whole***, rather than to an ***instance*** of the class.

☛ A ***static constructor*** cannot have ***access modifiers*** and cannot be ***called directly*** by your program. Furthermore, it has the same restrictions as ***static methods***, described in Java part 2.27.

☛ You can also specify a ***class*** as ***static***. For example, ***static class Test { // ...***

▶ A ***static class*** has two important ***features***. ***First***, no object of a static class can be created. ***Second***, a ***static class*** must contain ***only static members***. A primary use of a ***static class*** is found when working with ***extension methods***.