WORKING WITH THE TYPE SYSTEM

Value types

Store the values directly.
Are an alias for System types such as int for System.Int32.
Are passed as a copy to methods.
Framework includes standard data types most commonly required.
Legal values are based on number of bits used to store the type.

```
// create a variable to hold a value type using the alias form
// but don't assign a variable
int myInt;
int myNewInt = new int();

// create a variable to hold a .NET value type
// this type is the .NET version of the alias form int
// note the use of the keyword new, we are creating an object from
// the System.Int32 class
System.Int32 myInt32 = new System.Int32();

// you will need to comment out this first Console.WriteLine statement
// as Visual Studio will generate an error about using an unassigned
// variable.  This is to prevent using a value that was stored in the
// memory location prior to the creation of this variable
Console.WriteLine(myInt);

// print out the default value assigned to an int variable
// that had no value assigned previously
Console.WriteLine(myNewInt);

// this statement will work fine and will print out the default value for
// this type, which in this case is 0
Console.WriteLine(myInt32);
```

```
// assign values to these types and then
// print them out to the console window
// also use the sizeOf operator to determine
// the number of bytes taken up be each type

myInt = 5000;
Console.WriteLine("Integer");
Console.WriteLine(myInt);
Console.WriteLine(myInt.GetType());
Console.WriteLine(sizeof(int));
Console.WriteLine();
```

Data structures

Data structures involved structs, enumerations, and classes.
Structs are lightweight data structures.
Structs can contain member variables and methods.
Structs are passed by value unlike reference types, which are passed by reference.

Enumerations

Enumerations contain a list of named constants.
They make code more readable.
They use an underlying value for the named constant.
Underlying values of type int start at 0 and increment by one unless otherwise indicated in the declaration.

```
class Program
{
    enum Months
    {
        Jan = 1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sept,
        Oct, Nov, Dec
    };

    static void Main(string[] args)
    {
        string name = Enum.GetName(typeof(Months), 8);
        Console.WriteLine("The 8th month in the enum is " + name);

        Console.WriteLine("The underlying values of the Months enum:");
        foreach (int values in Enum.GetValues(typeof(Months)))
        {
            Console.WriteLine(values);
        }
    }

}
```

Reference types

Reference types are also commonly referred to as *classes*.
Classes contain member variables to store characteristics.
Classes contain member functions to provide functionality.
Class files encompass data and functionality in one package.

Modifiers

Modifiers are used to determine access for classes and class members.
Modifiers are listed first in declarations.

Fields

Fields contain the data for classes.
Fields are also known as member variables.
They describe characteristics of the class.
They should be marked private to avoid unwanted modification.

Properties Overview

Properties enable a class to expose a public way of getting and setting values, while hiding implementation or verification code.
A get property accessor is used to return the property value, and a set accessor is used to assign a new value.
The value keyword is used to define the value being assigned by the set accessor.
Properties that do not implement a set accessor are read only.
For simple properties that require no custom accessor code, consider the option of using auto-implemented properties.

Accessing Class Fields With Properties

```
using System;
public class Customer
{
   private int m_id = -1;

   public int ID
   {
      get
      {
         return m_id;
      }
      set
      {
         m_id = value;
      }
   }

public class CustomerManagerWithProperties
{
   public static void Main()
   {
      Customer cust = new Customer();

      cust.ID = 1;

            Console.WriteLine(
         "ID: {0}",cust.ID);

      Console.ReadKey();
   }
}
```

Read-Only Properties

```
using System;
public class Customer
{
   private int m_id = -1;
   private string m_name = string.Empty;

   public Customer(int id, string name)
   {
      m_id = id;
      m_name = name;
   }

   public int ID
   {
      get
      {
         return m_id;
      }
   }



   public string Name
```

```
    {
        get
        {
            return m_name;
        }
    }
}

public class ReadOnlyCustomerManager
{
    public static void Main()
    {
        Customer cust = new Customer(1, "Amelio Rosales");

        Console.WriteLine("ID: {0}, Name: {1}",cust.ID,cust.Name);

        Console.ReadKey();
    }
}
```

Write-Only Properties

```
using System;
public class Customer
{
    private int m_id = -1;

    public int ID
    {
        set
        {
            m_id = value;
        }
    }

    public void DisplayCustomerData()
    {
        Console.WriteLine("ID: {0}", m_id);
    }
}

public class WriteOnlyCustomerManager
{
    public static void Main()
    {
        Customer cust = new Customer();
        cust.ID = 1;
        cust.DisplayCustomerData();
        Console.ReadKey();
    }
}
```

Auto-Implemented Properties

```
using System;
public class Customer
{
    public int ID { get; set; }
    public string Name { get; set; }
}
```

```
public class AutoImplementedCustomerManager
{
   static void Main()
   {
      Customer cust = new Customer();
      cust.ID = 1;
      cust.Name = "Amelio Rosales";

      Console.WriteLine("ID: {0}, Name: {1}", cust.ID,cust.Name);

      Console.ReadKey();
   }
}
```

Constructors

Use to initialize classes.
Do not include a return type.
Use the same name as the class.
May contain no parameters (default constructor).
If no constructor is defined, compiler generates a default constructor.

Methods

Provide functionality for a class
Can be used with modifiers
Can return values or not (return type void)
Can accept arguments through parameters in the signature
Can use optional and named parameters

Overloaded methods

Same method name with multiple instances for different functionality
Defined by the signature (name, types, and kinds of parameters)

_____

```
public class studentStudent
{
    private string firstName;
    private char middleInitial;
    private string lastName;
    private int age;
    private string program;
    private double gpa;

    public studentStudent(string first, string last)
    {
        this.firstName = first;
        this.lastName = last;
    }

    public string FirstName
    {
        get { return firstName; }
        set { firstName = value; }
    }

    public string LastName
    {
        get { return lastName; }
```

```
            set { lastName = value; }
        }

        public char MiddleInitial
        {
            get { return middleInitial; }
            set { middleInitial = value; }
        }

        public int Age
        {
            get { return age; }
            set
            {
                if (value > 6)
                {
                    age = value;
                }
                else
                {
                    Console.WriteLine("StudentStudent age must be greater than 6");
                }
            }
        }

        public double GPA
        {
            get { return gpa; }
            set
            {
                if (value <= 4.0)
                {
                    gpa = value;
                }
                else
                {
                    Console.WriteLine("GPA cannot be greater than 4.0");
                }
            }
        }
        public void displayDetails()
        {
            Console.WriteLine(this.FirstName + " " + this.MiddleInitial + " " +
this.LastName);
            Console.WriteLine("Has a GPA of " + this.GPA);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            studentStudent myStudentStudent = new studentStudent("Tom", "Thumb");
            myStudentStudent.MiddleInitial = 'R';
            myStudentStudent.Age = 15;
            myStudentStudent.GPA = 3.5;
            myStudentStudent.displayDetails();
```

```
        }
    }
```

Optional parameters

Enable you to choose which parameters are required in a method.
Defined as optional by including a default value.
The default value is used if none is passed by caller. Must exist after required parameters. If multiple optional parameters exist and a value is specified for one, all preceding optional parameters must also be supplied values.
The default value is used if none is passed by caller. Must exist after required parameters. If multiple optional parameters exist and a value is specified for one, all preceding optional parameters must also be supplied values.

Named parameters

Allow for giving parameters in a method a name
Increase code readability
Enable you to pass arguments to a method in an order other than in the method signature

Indexed properties

Allow array-like access to groups of items .
Must be access using an index in the same manner as arrays.
In the following example, a generic class is defined and provided with simple get and set accessor methods as a means of assigning and retrieving values. The Program class creates an instance of this class for storing strings.

```
class SampleCollection<T>
{
   // Declare an array to store the data elements.
   private T[] arr = new T[100];

   // Define the indexer, which will allow client code
   // to use [] notation on the class instance itself.
   // (See line 2 of code in Main below.)
   public T this[int i]
   {
      get
      {
         // This indexer is very simple, and just returns or sets
         // the corresponding element from the internal array.
         return arr[i];
      }
      set
      {
         arr[i] = value;
      }
   }
}

// This class shows how client code uses the indexer.
class Program
{
   static void Main(string[] args)
   {
      // Declare an instance of the SampleCollection type.
      SampleCollection<string> stringCollection = new SampleCollection<string>();

      // Use [] notation on the type.
      stringCollection[0] = "Hello, World";
      System.Console.WriteLine(stringCollection[0]);
   }
}
```

readonly modifier

The readonly keyword is a modifier that you can use on fields. When a field declaration includes a readonly modifier, assignments to the fields introduced by the declaration can only occur as part of the declaration or in a constructor in the same class.

```
class Age
{
    readonly int _year;
    Age(int year)
    {
        _year = year;
    }
    void ChangeYear()
    {
        //_year = 1967; // Compile error if uncommented.
    }
}
```

When the variable is initialized in the declaration, for example:

```
public readonly int y = 5;
```

static modifier

Use the static modifier to declare a static member, which belongs to the type itself rather than to a specific object.

```
public class Employee4
{
    public string id;
    public string name;

    public Employee4()
    {
    }

    public Employee4(string name, string id)
    {
        this.name = name;
        this.id = id;
    }

    public static int employeeCounter;

    public static int AddEmployee()
    {
        return ++employeeCounter;
    }
}



class MainClass : Employee4
{
    static void Main()
    {
        Console.Write("Enter the employee's name: ");
```

```
            string name = Console.ReadLine();
            Console.Write("Enter the employee's ID: ");
            string id = Console.ReadLine();

            // Create and configure the employee object:
            Employee4 e = new Employee4(name, id);
            Console.Write("Enter the current number of employees: ");
            string n = Console.ReadLine();
            Employee4.employeeCounter = Int32.Parse(n);
            Employee4.AddEmployee();

            // Display the new information:
            Console.WriteLine("Name: {0}", e.name);
            Console.WriteLine("ID:   {0}", e.id);
            Console.WriteLine("New Number of Employees: {0}",
                        Employee4.employeeCounter);
        }
    }
```

const keyword

You use the const keyword to declare a constant field or a constant local.

```
const int x = 0;
public const double gravitationalConstant = 6.673e-11;
private const string productName = "Visual C#";
```

The static modifier is not allowed in a constant declaration.

The readonly keyword differs from the const keyword. **A const field can only be initialized at the declaration of the field. A readonly field can be initialized either at the declaration or in a constructor. Therefore, readonly fields can have different values depending on the constructor used.** Also, although a const field is a compile-time constant, the readonly field can be used for run-time constants, as in this line: public static readonly uint l1 = (uint)DateTime.Now.Ticks;

Is Operator

Checks if an object is compatible with a given type. For example, the following code can determine if an object is an instance of the MyObject type, or a type that derives from MyObject:

```
if (obj is MyObject)
{
}
```

An is expression evaluates to true if the provided expression is non-null, and the provided object can be cast to the provided type without causing an exception to be thrown.
The is keyword causes a compile-time warning if the expression is known to always be true or to always be false, but typically evaluates type compatibility at run time.
The is operator cannot be overloaded.

Note that the is operator only considers reference conversions, boxing conversions, and unboxing conversions.

Example

```
class Class1 {}
class Class2 {}
class Class3 : Class2 { }

class IsTest
{
    static void Test(object o)
    {
        Class1 a;
        Class2 b;

        if (o is Class1)
        {
            Console.WriteLine("o is Class1");
            a = (Class1)o;
            // Do something with "a."
        }
        else if (o is Class2)
        {
            Console.WriteLine("o is Class2");
            b = (Class2)o;
            // Do something with "b."
        }

        else
        {
            Console.WriteLine("o is neither Class1 nor Class2.");
        }
    }

     static void Main()
     {
        Class1 c1 = new Class1();
        Class2 c2 = new Class2();
        Class3 c3 = new Class3();
        Test(c1);
        Test(c2);
        Test(c3);
        Test("a string");
    }
}
/*
Output:
o is Class1
o is Class2
o is Class2
o is neither Class1 nor Class2.
*/
```

As operator

You can use the as operator to perform certain types of conversions between compatible reference types or nullable types. The following code shows an example.

```
class csrefKeywordsOperators
   {
       class Base
       {
           public override string  ToString()
           {
                     return "Base";
           }
       }
       class Derived : Base
       { }

       class Program
       {
           static void Main()
           {

               Derived d = new Derived();

               Base b = d as Base;
               if (b != null)
               {
                   Console.WriteLine(b.ToString());
               }

           }
       }
   }
```

The as operator is like a cast operation. However, if the conversion isn't possible, as returns null instead of raising an exception. Consider the following example:

```
expression as type
```

The code is equivalent to the following expression except that the expression variable is evaluated only one time.

```
expression is type ? (type)expression : (type)null
```

Note that the as operator performs only reference conversions, nullable conversions, and boxing conversions. The as operator can't perform other conversions, such as user-defined conversions, which should instead be performed by using cast expressions.