CSC-200

Module 2

Object-Oriented Programming with C#

1 Implementing a Class

1.1 Class

An object-oriented programming language centers on creation and usage of objects. A class is used to define the structure and behavior of an object. C# is fully object-oriented and thus class is the primary program element of the language. The **class** keyword is used to declare a class and assign a name. You must place the content of the class between the characters **{ }** which we call as the scope of the class.

C# declaration of a class: MyLib1\Time.cs

```
class Time {
      // scope of the class
}
```

Even when you compile a class to a library instead of an application, it still cannot be accessed externally because the default accessibility for a class is **internal** which only allows the class to be accessed from within the same assembly. We can use **public** to ensure that the class can be accessed by the code in other assemblies.

Declaration of a public class

```
public class Time {
      // define class members here
}
```

A class is placed in the global namespace by default. A class name must be unique in one namespace. If you are building a library for other developers to use, you should use a separate namespace for the classes in the library to ensure they do conflict with the classes from other libraries. Use **namespace** keyword to define a namespace and assign the name. To place a class within the namespace, you need to declare it within the scope of the namespace.

Declaring a class within a namespace

```
namespace MyLib1 {
      public class Time {
      }
}
```

1.2 Fields

Not all classes will need to store information. If you are implementing a class to store and manage data, you can use create fields that will cause memory to be allocated during runtime. In the following example, we will define fields to store data to represent time information. Notice the use of a **region** directive. This is not compulsory but creates a collapsible region when the source code is open in an editor that has special support for C#. Using regions can help to organize the content of a complex class.

Defining field members

```
public class Time {
      #region fields
      public short Hour;
      public short Minute;
      public short Second;
      #endregion
}
```

Since the fields are non-static, memory would only be allocated for the fields when an object is instantiated from the class by using the **new** operator. In the example below we demonstrate how to instantiate multiple objects from a class to store and retrieve data. Since the fields are marked **public**, they can be accessed directly.

Instantiating and using objects

```
using System;
using MyLib1;

class Program {
      static void Main() {
            Time a = new Time();
            Time b = new Time();
            a.Hour = 10; a.Minute = 20; a.Second = 30;
            b.Hour = 11; b.Minute = 22; b.Second = 33;
            Console.WriteLine("{0:00}:{1:00}:{2:00}",
                  a.Hour,a.Minute,a.Second);
            Console.WriteLine("{0:00}:{1:00}:{2:00}",
                  b.Hour,b.Minute,b.Second);
      }
}
```

1.3 Properties

Fields only represent memory where data can be stored and accessed. There is no control over the data than can be assigned the fields. The following example shows a program storing invalid time information into an object but it can be compiled and run without any errors or warnings.

Storing invalid time information

```
Time t = new Time();
t.Hour = 9999;
t.Minute = 67;
t.Second = -1;
Console.WriteLine("{0:00}:{1:00}:{2:00}",
      t.Hour,t.Minute,t.Second);
```

If we need to validate the data before storing into the fields, you must not allow any external code to access the fields directly. In the following example, we will mark the field members as **private** which does not allow the members to be accessed outside of the class. We will implement properties to validate the data before storing into fields. We change the name of the fields because we wish to use those names for the properties instead.

Using the private access modifier

```
public class Time {
      #region fields
      private short hour;
      private short minute;
      private short second;
      #endregion
}
```

Properties are used exactly like fields but they represent code and not memory space. A property can consist of a **get** and a **set** method. The **get** method is called when you attempt to retrieve a value from the property while the **set** method is called when you assign a value to the property. It is possible to have a **get** method without a **set**. This would create a read-only property. For a write only property, there will be a **set** method but no **get**. In the following example, we implement a property for each of the fields. In the **get** method, we will simply return the contents of the field. However in the **set** method, we would validate the input **value** first before storing it. When the validation fails we will generate a runtime error by throwing an exception object using the **throw** keyword. We use exceptions to signal problems to the application. We can program the application to catch and handle the exceptions. We will cover exceptions in more detail in later section. Import **System** namespace to use the **Exception** class directly.

Implementing properties

```
public short Hour {
      get { return hour; }
      set {
            if (value < 0 || value > 23)
                  throw new Exception("Invalid hour value.");
            hour = value;
      }
}

public short Minute {
      get { return minute;  }
      set {
            if (value < 0 || value > 59)
                  throw new Exception("Invalid minute value.");
            minute = value;
      }
}

public short Second {
      get { return second; }
      set {
            if (value < 0 || value > 59)
                  throw new Exception("Invalid second value.");
            second = value;
      }
}
```

Properties do not always have to store and retrieve data from fields only. A property may choose to store data into an external file or a database. Some data may actually be processed or calculated before storing or returning. In the following example, we will implement a **Value** property that returns the total number of seconds and parse a total number of seconds to time information for storage.

Calculated property

```
public int Value {
      get { return hour * 3600 + minute * 60 + second; }
      set {
            if (value < 0) throw new Exception("Invalid time value.");
            hour = (short)(value / 3600 % 24);
            minute = (short)(value % 3600 / 60);
            second = (short)(value % 60);
      }
}
```

If you execute the application an exception occur because the program attempt to store invalid data into a **Time** object but is blocked by the properties. Correct the data and execute the application again and this time it will work. You can also retrieve and store time information using the **Value** property.

Storing valid time information

```
Time t = new Time();
t.Hour = 23;
t.Minute = 59;
t.Second = 59;
Console.WriteLine("{0:00}:{1:00}:{2:00}",t.Hour,t.Minute,t.Second);
Console.WriteLine("Total seconds:{0}",t.Value); t.Value = t.Value + 120;
Console.WriteLine("{0:00}:{1:00}:{2:00}",t.Hour,t.Minute,t.Second);
```

1.4 Constructors

By default, all value fields are initialized to **0** and reference fields are initialized to **null**. When you create an object, a special method called as the constructor will be called. The constructor is used to initialize the object upon creation. By default, the compiler will generate a default constructor for you. You can choose to implement the constructor manually so that you can perform your own object initialization as shown below.

Initialize fields to current time

```
public Time() {
      DateTime dt = DateTime.Now;
      hour = (short)dt.Hour;
      minute = (short)dt.Minute;
      second = (short)dt.Second;
}
```

When you use the **new** keyword to instantiate an object from the class, you are also calling the constructor method. The following example will prove that the constructor will be called since the current time would be displayed.

Calling the parameterless constructor

```
Time t = new Time();
Console.WriteLine("{0:00}:{1:00}:{2:00}",
      t.Hour,t.Minute,t.Second);
```

Overloading is a technique where multiple methods may have the same name as long as number and type of parameters are different between them. The C# compiler can detect which method to call depending on the number and type of parameters that you pass to the method. Constructors also support overloading and we can use this to provide another way to initialize the object. Following is an example of a constructor which can initialize the object with custom values passed as parameters. When you instantiate an object, you can then pass the correct number and type of parameters to call the right constructor.

Overloading the constructor

```
public Time(short hour, short minute, short second) {
      Hour = hour; Minute = minute; Second = second;
}
```

Calling constructor with parameters

```
Time t = new Time(10,20,30);
Console.WriteLine("{0:00}:{1:00}:{2:00}",
      t.Hour, t.Minute, t.Second);
```

1.5 Methods

Before we implement the methods in our **Time** class, we should examine how parameters can be passed to methods. A parameter can be for input, output or both. By default, all parameters are passed for input. This would not allow the method to change the contents of variables passed in. Even if the method changes the parameter, it is only changing a copy and not the original. We call this technique as pass by value.

Method accepting an input parameter: Parameter1\Program.cs

```
static void Proc1(int value) {
      Console.WriteLine("Proc1({0})", value);
      value = 99;      // changing copy of variable
}
```

A parameter can also be marked for output by using **out** keyword. This will allow the method to change the contents of any variable passed as a parameter but does not allow you to access the original value since it is not for input.

Method accepting an output parameter

```
static void Proc2(out int value) {
      Console.WriteLine("Proc2(?)");
      value = 98;      // changing original variable
}
```

You can also pass a parameter for both input and output using the **ref** keyword. This would allow the method to access the original content and also modify it as shown in the following example. We can also call this technique as passing by reference. Both **out** and **ref** keyword will also have to be used when calling the methods. This will ensure that developer knows that the methods may change the variable parameters.

Method accepting an output parameter

```
static void Proc3(ref int value) {
     Console.WriteLine("Proc3({0})", value); value = 97;
}
```

Passing input and output parameters

```
static void Main() {
     int var = 100;
     Proc1(var);     Console.WriteLine(var);
     Proc2(out var); Console.WriteLine(var);
     Proc3(ref var); Console.WriteLine(var);
}
```

We will now implement some methods in our **Time** class. We will add a method to set the entire time value instead of having to set three separate properties. The values will be passed in as input parameters and will have to be validated by assigning them to the property.

Method to set entire time

```
public void SetTime(short hour, short minute, short second) {
     Hour = hour; Minute = minute; Second = second;
}
```

Methods can be overloaded. In the following example, we add another method with the same name but this time will only accept two parameters. The **second** value will be initialized to **0** instead. This is useful when a program only wants to set the **hour** and **minute** but not the **second**. We can rewrite the code for new method but we will show you how one overloaded method can call another.

Overloading SetTime method

```
public void SetTime(short hour, short minute) {
//    Hour = hour; Minute = minute; Second = 0;
     SetTime(hour, minute, 0);
}
```

In C# 4.0 you can have default parameters so rather than overloading you may also just assign default values to each parameter so that the same method can be called with different number of parameters.

Method using default parameters

```
public void SetTime(short hour = 0, short minute = 0, short second = 0) {
      Hour = hour; Minute = minute; Second = second;
}
```

In the following example, we will add another method to access the entire time value from the object. We can obtain the hour, minute and second by passing in output parameters. The example program declares three variables to store the output value of the parameters. Note that we use **this** keyword to differentiate the fields from the parameters. The keyword always returns a reference to the current object.

Method accepting output parameters

```
public void GetTime(out short hour, out short minute, out short second) {
      hour = this.hour; minute = this.minute; second = this.second;
}
```

Following are extra methods to update a time object by adding or subtracting a number of seconds. Methods can also perform validation on parameters before using them. In both cases, the input parameter must not be negative.

Additional method to update time

```
public void Add(int seconds) {
      if (seconds < 0) throw new Exception("Invalid seconds parameter.");
      Value = Value + seconds;
}
public void Subtract(int seconds) {
      if (seconds < 0) throw new Exception("Invalid seconds parameter.");
      Value = Value - seconds;
}
```

Note that when you assign the result of a calculation back to the same variable or field that the original value was retrieved from, you can use the assignment operator to perform the calculation. It would simplify the statements.

Using assignment add and subtract operators

```
Value += seconds; // instead of Value = Value + seconds;
Value -= seconds; // instead of Value = Value - seconds;
```

Calling methods and passing parameters: Test6.cs

```
Time t = new Time();
t.SetTime(10,20,30); t.SetTime(20,30);
t.Add(60); t.Subtract(3600); t.GetTime(out h, out m, out s);
Console.WriteLine("{0:00}:{1:00}:{2:00}", h, m, s);
```

2 Implementing Operators

2.1 Math Operators

.NET does not implement operators. Operators are provided by the language itself. By default C# operators only work with simple types. However, you can overload them to work with custom types as well. For example, rather than calling the **Add** and **Subtract** method to update the value in a **Time** object. You may wish to use the **+** and the **–** operators instead. You may also wish to be able to subtract two different **Time** objects to get the duration in between. Following is an example program that attempts to perform these operations. However, you will not be able to compile it at the moment since we have not implemented the operators for our class yet.

Using operators on Time objects

```
Time t1 = new Time(10,20,30);
Time t2 = new Time(20,30,40);
Console.WriteLine(t1.Value);
Console.WriteLine(t2.Value);
t1 += 100; Console.WriteLine(t1.Value);
t2 -= 200; Console.WriteLine(t2.Value);
Console.WriteLine(t1 - t2);
```

To implement an operator method, use the **operator** keyword followed by the actual operator symbol. A binary operator would have two operands; one on the left side of the operator and one on the right side. These two operands are passed in as two separate parameters. Also note that an operator must always have a return value. It is up to you to decide what the return value is. Normally it will be the left operand. Following is the methods for the **+** and **–** operators.

Methods to implement + and – operators

```
public static Time operator +(Time obj, int seconds) {
     obj.Add(seconds); return obj; }

public static Time operator -(Time obj, int seconds) {
     obj.Subtract(seconds); return obj; }
```

The following is the operator method to subtract one **Time** object from another and returns number of seconds in between. In this case, the return type is an **int** value instead. Once you have implemented the operators the program can now be compiled and executed successfully.

Operator to subtract Time objects

```
public static int operator -(Time obj1, Time obj2) {
      return obj1.Value - obj2.Value;
}
```

2.2 Comparison Operators

C# provides comparison operators to compare only simple types and does not work for complex types like structures and classes. Again you can choose to overload the operators to support the features. For example, we may wish to compare one **Time** object against another **Time** object or compare against an **int** value representing the total number of seconds. The following program attempts to perform the operations. However, it cannot be compiled since the operators are not implemented to work with **Time** objects yet.

Operations to comparing Time objects

```
Time t1 = new Time(10,20,30);
Time t2 = new Time(10,20,30);
Console.WriteLine(t1 == t2);
Console.WriteLine(t1 == 37230);
```

The following is the implementation of the equal comparison operators for the **Time** class. All comparison operators should return a **bool** result. We overload the same operator to support comparison of different types. Note that if you are using C# 2.0, you have to implement the not equal operator well if you overload the equal operator.

Implementing equal and not equal comparison operator

```
public static bool operator ==(Time obj1, Time obj2) {
      return obj1.Value == obj2.Value; }
public static bool operator ==(Time obj1, int value) {
      return obj1.Value == value; }

public static bool operator !=(Time obj1, Time obj2) {
      return obj1.Value != obj2.Value; }
public static bool operator !=(Time obj1, int value) {
      return obj1.Value != value; }
```

However, overloading comparison operators can be cumbersome since there are so many of them including equal (**==**), not equal (**!=**), less than (**<**), greater than (**>**), less or equal (**<=**), greater or equal (**>=**). Not to mention that you need to overload them to support different operand types on both the left side and the right side. For example, a method that compares an **int** value to **Time** cannot be used to compare a **Time** to an **int** value. You need to implement two separate methods.

There is an easier way to solve this problem by converting a **Time** object to a simple value type where the operators are already available so we do not have to overload them.

2.3 Conversion Operators

We can implement conversion operators that C# can automatically call when it needs to convert a **Time** object to other types. The following example program attempts to assign **Time** object to an **int** variable and also assign an **int** value to a **Time** variable. This program cannot be compiled since we have not implemented any conversion operators in the class.

Converting between Time and int

```
Time t1 = new Time(10, 20, 30);
int value = t1;       // convert Time to int
Time t2 = value;      // convert int to Time
Console.WriteLine(t1.Value);
Console.WriteLine(t2.Value);
```

You can use **implicit** or **explicit** to implement a type operator. Following shows how to implement an operator to convert a **Time** object to **int** value and another operator method to convert **int** value to **Time** object. Once this is done the program can be compiled.

Type conversion operator methods

```
public static implicit operator int(Time obj) { return obj.Value; }
public static implicit operator Time(int value) {
     Time obj = new Time(); obj.Value = value; return obj; }
```

Since C# operators already work with **int** values. You can use all of them on a **Time** object without having to overload operators. The following shows the use of operators on **Time** objects. C# will automatically convert all the objects to **int** value before applying operators. Comment out all the comparison operators in **Time** class.

Applying int operators on Time objects

```
Time t1 = new Time(10,20,30); Time t2 = new Time(20,30,40);
Console.WriteLine(t1 == t2); Console.WriteLine(t1 != t2);
Console.WriteLine(t1 < t2); Console.WriteLine(t1 > t2);
Console.WriteLine(t1 == 37230); Console.WriteLine(t2 >= 37230);
```

Conversion is automatic when you use **implicit**. If you do not wish this, you can then use **explicit** instead. When explicit is used, the programmer has to make use of type-casting to enforce the conversion. In the following example, we require casting when converting **int** value to **Time** object.

Implementing explicit operators

```
public static explicit operator Time(int value) {
```

Typecasting required to perform conversion

```
Time t = (Time)37230; Console.WriteLine(t.Value);
```

2.4 Indexer Operator

Remember that you can use the indexer operator **[]** to access information in an array or collection. Arrays and collections are actually objects. This operator has already been overloaded for those objects. If you wish to allow the indexer to be used to access information in your own objects, you can overload it in your class. However, it has to be overloaded as a **property**. Use **this** keyword as the name for the property.

We wish to use the indexer to access the individual parts of a **Time** object. The index value in the operator determines which part to access. We can test the index value and access the correct field or property by using a **switch** block. In the **set** method, make sure that there is a **break** for each of the **case** otherwise the operation will flow from one case to another.

Implementing the indexer operator

```
public short this[int index] {
      get {
            switch (index) {
                  case 0: return hour;
                  case 1: return minute;
                  case 2: return second;
                  default:
                        throw new Exception(
                              "Index out of range.");
            }
      }
      set {
            switch (index)
            {
                  case 0: Hour = value; break;
                  case 1: Minute = value; break;
                  case 2: Second = value; break;
                  default:
                        throw new Exception(
                              "Index out of range.");
            }
      }
}
```

Using the indexer operator

```
Time t = new Time();
t[0] = 10;
t[1] = 20;
t[2] = 30;
Console.WriteLine(t[0]);
Console.WriteLine(t[1]);
Console.WriteLine(t[2]);
```

The index parameter does not have to be an **int** value. You can use any type as an index. In the following example, we overloaded the indexer to support using **char** type as index value. This would allow you to pass

Using a string index

```
public short this[char index] {
      get
      {
            if (index == 'h') return hour;
            if (index == 'm') return minute;
            if (index == 's') return second;
            throw new Exception(
                  "Invalid index value.");
      }
      set
      {
            if (index == 'h') Hour = value; else
            if (index == 'm') Minute = value; else
            if (index == 's') Second = value; else
                  throw new Exception(
                        "Invalid index value.");
      }
}
```

Example program using string indexes

```
Time t = new Time();
t['h'] = 10;
t['m'] = 20;
t['s'] = 30;
Console.WriteLine(t['h']);
Console.WriteLine(t['m']);
Console.WriteLine(t['s']);
```

3 Standard Object Features

3.1 Overriding Object Methods

All types directly or indirectly derive from the **object** class. This will ensure that all objects are compatible to the **object** class. This is why you can assign any object to a field, variable or parameter marked as **object**.

Time is compatible with object

```
object o = new Time(); Time t = (Time)o;
```

When one class is extended from another class, all the members of the original class will be inherited by the new class. There are special methods that we have inherited from the **object** class. One inherited method is **ToString**. The method is supposed to convert an object to string. However if you execute the following code, the method would return the name of the type.

Converting Time object to string

```
Time t = new Time(); Console.WriteLine(t.ToString());
```

Even though the method is provided by the **object** class, it does not work since it is suppose to be replaced by your own class. To replace an inherited member you can use **override** keyword. In the following example, we replaced the **ToString** method with our own. When you run the program again, the results will now be correct.

Overriding ToString in Time class

```
public override string ToString() {
//    return string.Format("{0:00}:{1:00}:{2:00}",hour,minute,second);
      return $("{hour:00}:{minute:00}:{second:00}");
}
```

Besides **ToString** method, there are also two additional other methods named as **Equals** and **GetHashCode** that should be overridden by a data-oriented class. **Equals** method is used to check whether two objects are the same type and content.

The Equals method does not work by default

```
Time t1 = new Time(10,20,30), t2 = new Time(10,20,30);
Console.WriteLine(t1.Equals(t2));
```

Implementing our own Equals method

```
public override bool Equals(object obj) {
      Time item = (Time)obj;
      return hour == item.hour && minute == item.minute &&
            second == item.second;
}
```

If you execute the above program the **Equals** method should work. However, there is still a slight problem with the above method. A runtime error will occur if you pass any other object instead of a **Time** object. This is because we are using hard-casting in the above method. In the following example, we will be using safe-casting with the **as** keyword. There will not be a runtime error if the object we are casting is incompatible. Instead a **null** reference will be returned which we can check and avoid any runtime error.

Runtime error for wrong object type

```
Console.WriteLine(t1.Equals("abcdef"));
```

Using safe typecasting to avoid runtime errors

```
public override bool Equals(object obj) {
      Time item = obj as Time; if (item == null) return false;
      return hour == item.hour && minute == item.minute &&
            second == item.second;
}
```

It is possible that an object is compared against itself. We can detect this very easily by comparing the object passed in against **this** keyword. This keyword is used to obtain a reference to the current object. You can then compare against the object reference pass as a parameter. If it is the same object, it will contain the same content so we do not even need to cast or check the content.

Same object should have same value

```
public override bool Equals(object obj) {
      if (obj == this) return true;
                  :
```

The **GetHashCode** method is support to return a value that identity each object. By default, each object will be assigned a unique number when you create it. It can be accessed by calling the method. If you wish to provide your own identity the object, you can override the method. In the following example, we use the value of the time as its identity.

Providing identity for each object

```
public override int GetHashCode() { return Value; }
```

3.2 Implementing ICloneable

The above methods we implemented are inherited from **object** class. This is standard and cannot be avoided. However, we can also implement optional features by using interfaces. An interface is basically a declaration of a set of properties, events and methods but contains no code. The code will have to be implemented in the class if you wish to support the interface. By supporting an interface, you can interface to certain features that are built-in into Microsoft .NET or to integrate better to other applications and libraries. In Microsoft .NET there is an **ICloneable** interface for those objects that can make copies of themselves. Any class implementing this interface must have a **Clone** method.

Implementing the ICloneable interface

```
public class Time : ICloneable {
            :
      public object Clone() {
            return new Time( hour, minute, second);
      }
}
```

Calling the clone method in the class

```
Time t1 = new Time(10,20,30); Time t2 = (Time)t1.Clone();
```

Since the method is marked **public**, it is possible to call the method directly in the class as shown in the above program. Notice that type-casting is required since the **Clone** method is declared to return **object** rather than **Time**. There is no way we can change this as this is defined in the **ICloneable** interface. We can still avoid this issue by providing two methods; one for the interface and one for the class. If you only wish to call the method thru the interface, specify the interface name along with the method name.

Method callable thru interface only

```
object ICloneable.Clone() { return new Time(hour, minute, second); }
```

Method callable thru the class

```
public Time Clone() { return new Time(hour, minute, second); }
```

When you call the method thru the class, it will return a **Time** object. Thus no type-casting is required but will still be required if you call through the interface instead. If you do not wish to duplicate the code, the interface can call the method in the class to perform the same operation.

Interface method calling class method

```
object ICloneable.Clone() { return Clone(); }
```

The following example program shows how to call both methods. If you wish to call methods through an interface, you will need to type-cast the object to the particular interface first before calling the method.

Calling methods in class and interface

```
Time t1 = new Time(10, 20, 30); Time t2 = t1.Clone();
Time t3 = (Time)(ICloneable)t1.Clone();
Console.WriteLine(t1.Equals(t2));
Console.WriteLine(t1.Equals(t3));
```

3.3 Implementing IComparable

You may wish to put multiple **Time** objects into a collection like **ArrayList**. This is shown in the following example. The **ArrayList** object has a **Sort** method we can call to sort the items in the collection. However, when you run the program, a runtime error can occur because a **Time** object cannot be sorted.

Sorting collection of Time objects

```
ArrayList list = new ArrayList();
list.Add(new Time(11,22,33));
list.Add(new Time(22,33,44));
list.Add(new Time(10,20,30));
list.Sort();
```

To support sorting, an object must implement **IComparable** interface. The interface requires that the class must have a **CompareTo** method. The method will return -1 if a source object is less than the target object or +1 if greater. If both objects are the same, 0 should be returned instead. Once this is done, if you run the program again, the **Sort** method will work.

Implementing the IComparable interface

```
public class Time :
      ICloneable, IComparable {
```

Implementing the CompareTo method

```
public int CompareTo(object obj) {
      if (obj == this) return 0;
      Time item = (Time)obj;
      int value1 = this.Value;
      int value2 = item.Value;
      if (value1 < value2) return -1;
      if (value1 > value2) return +1;
      return 0;
}
```

The problem with interfaces is that they have to support a variety of types so a lot of interfaces using **object** as parameter and return value type. However this can cause boxing when values are used. To avoid this you can then implement the generic version of the interface if available. An **IComparable<T>** interface is available if you want to provide exact type comparisons.

Support exact type comparisons

```
public class Time : ICloneable, IComparable,
      IComparable<Time>, IComparable<int> {
                  :
      public int CompareTo(Time item) {
            if (item == this) return 0;
            int value1 = this.Value;
            int value2 = item.Value;
            if (value1 < value2) return -1;
            if (value1 > value2) return +1;
            return 0;
      }
      public int CompareTo(int value) {
            int value1 = this.Value;
            if (value1 < value) return -1;
            if (value1 > value) return +1;
            return 0;
      }
}
```

3.4 Static Members

Data can also be stored together with the class rather than per object. This can be done by using **static** modifier on a field. To initialize static fields, you can implement a static constructor. This constructor is called when the class is loaded and not when you create an object. We will also use **readonly** modifier on the fields so that once initialized by a constructor they cannot be changed. Thus there is no need to make them **private** or create properties to access them.

Static and readonly fields

```
public static readonly Time Midnight;
public static readonly Time Afternoon;
```

constructor to initialize static fields

```
static Time() {
      Midnight  = new Time( 0, 0, 0);
      Afternoon = new Time(12, 0, 0);
}
```

One issue though with **readonly** fields is that even though you cannot replace it with another value but if it is an object they can still change the contents of that object. Another way is to use static properties where you will always return the same time but in a new object.

Using read-only static properties

```
public static Time Midnight { get { return new Time(0, 0, 0); }}
public static Time Afternoon { get { return new Time(12, 0, 0); }}
```

You can also use constants instead of readonly fields but only simple values and strings are supported by constants. All structures and objects still require using readonly fields or static readonly properties. Use the **const** keyword to declare a constant value.

Declaring constants

```
public const int MinHour = 0;
public const int MinMinute = 0;
public const int MinSecond = 0;
public const int MaxHour =  23;
public const int MaxMinute = 59;
public const int MaxSecond = 59;
```

Using constants

```
public short Hour {
      get { return hour; }
      set { if (value < MinHour || value > MaxHour)
                  throw new Exception("Invalid hour value.");
            hour = value;
      }
}
```

3.5 Enumeration

An enumeration contains identifiers associated with a value. Internally the value is stored as a 32-bit integer number. You can choose to assign the value or you can allow the compiler to generate it automatically. By default the first member will be assigned the value 0. Subsequent members will be assigned the previous value plus 1. Following are examples of enumeration types.

Enumeration with implicit values: Enum1\Program.cs

```
public enum Gender {
      Female,               // value defaults to 0
      Male                  // value is Female + 1 = 0
}
```

Enumeration with all explicit values

```
public enum AsiaCallCode {
      Australia = 61, Brunei = 615, China = 86, Indonesia = 62,
      Japan = 81, Malaysia = 60, Myanmar = 95, NewZealand = 64,
      Philippines = 63, Singapore = 65, Taiwan = 886, Thailand = 66,
      Vietnam = 84
}
```

Using enumeration as data type

```
Gender g = Gender.Male;
AsiaCallCode a = AsiaCallCode.Malaysia;
AsiaCallCode b = (AsiaCallCode)84;
Console.WriteLine(g);
Console.WriteLine(a);
Console.WriteLine(b);
Console.WriteLine((int)g);
Console.WriteLine((int)a);
Console.WriteLine((int)b);
```

3.6 Anonymous & Dynamic Objects

If you only need to use an object within a method or lambda expressions you can use an anonymous object. This means that you do not need to declare any class. The .NET compiler will generate the class for you automatically. You will need to use **var** to assign the object to a variable as the type is only available during compilation.

Instancing and initializing an anonymous object: Anonymous1

```
var a1 = new { ID = 100, Name = "ABC Trading", Balance = 10000 };
Console.WriteLine("{0},{1},{2}", a1.ID, a1.Name, a1.Balance);
```

However these objects can only be accessed directly in the same method since parameters have to be typed in order to access them from in other methods. It is possible to pass them using an **object** typed parameter but then you will not be able to typecast it back to access the properties directly. In C# 4.0 Microsoft has added a DLR (Dynamic Language Runtime) system which allows you to use **dynamic** objects. A dynamic object allows .NET to access the type information of the object at runtime rather than at compilation time. Thus the compiler does not know or care what an object is during compilation time. So you can say that an object has a property named **Value** or a method called **Sum** and C# simply generate runtime code to access these members without knowing if the object really have them or not. Runtime error will occur if it doesn't. Bad side of using dynamic objects is they are much slower and no intellisense is available.

Method accepting a dynamic object

```
static void Display(dynamic obj) {
      Console.WriteLine("{0},{1},{2}", obj.ID, obj.Name, obj.Balance);
}
```

Passing an anonymous object as dynamic

```
Display(new { ID = 200,
      Name = "XYZ Limited",
      Balance = 32000m});
```

Unlike using generic types where code is pre-checked and pre-compiled when you build an assembly, code that uses any dynamic types will be compiled into more like scripting code rather than compiled code. This will also allow you to also use any types that are not created by .NET languages. Thus it is possible to integrate C# to other scripting languages such as Ruby, Python and Javascript, if custom DLR binders are available for them. The following shows a method that uses dynamic types and since no type-checking is done during compilation, we can pass anything to the method. However runtime error may occur if the dynamic code fails during runtime. **Microsoft.CSharp** must be referenced to use dynamic objects in C# code.

Using dynamic type: Dynamic1

```
static dynamic Add(dynamic a, dynamic b) { return a + b; }

static void Main() {
      Console.WriteLine(Add(10, 20));
      Console.WriteLine(Add(1.99, 99.1));
      Console.WriteLine(Add("Hello!", "Goodbye!"));
      Console.WriteLine(Add(DateTime.Now, TimeSpan.FromDays(7)));
      Console.WriteLine(Add("Hello!", 999));
      Console.WriteLine(Add(999, 9.99));
      Console.WriteLine(Add(DateTime.Now, 9.99));
}
```

The main purpose of the DLR is to allow non .NET languages including scripting languages like **Python**, **TCL** and **Perl** to expose themselves to .NET through a DLR. These languages are not pre-compiled and type metadata is only available during runtime. The C# compiler will generate dynamic code that does not need any type information during compilation. The code will locate type information only at runtime. Since the entire .NET Framework can be accessed through DLR as well, you can decide whether to also use .NET types dynamically in sacrifice of performance at runtime and not having intellisense during authoring.