Programming in .NET

Microsoft Development Center Serbia programming course

Lesson 3 – Inheritance and generics

Inheritance

`System.Object` class is a root base class of all other classes. All classes will inherit Object's four instance methods: `ToString`, `Equals`, `GetHashCode`, `GetType`. This also means that code written to use `System.Object` will operate on any instance of any class. In general when a class is inheriting some base class it is actually inheriting the following:

- The method signatures – this allows code to think that it is opeating on an instance of the base class, when in fact, it could be operating on an instance of some derived class
- The implementation of these methods – which can be reused or simply not required to be changed or re-implemented when not necessary.

Inheriting a class in C# is done with the following syntax:

```
class Asset
{
    public string Name;
}

class Stock : Asset // Inherits from Asset
{
    public long NumberOfShares;
}

class Car : Asset // Inherits from Asset
{
    public string Make;
}
```

A type can inherit one or more interfaces, but only one class. Therefore diamond shaped multi inheritance is not possible.

Inheritance is used typically when we want to abstract the behavior of an object. Method, property or field is defined in a base class and later implemented or defined in a derived class. This way some other code can work with a base class without knowing the exact implementation.

There are however situations when a base class need to be down casted to a derived class. C# does require the developer to explicitly cast an object to any of its derived types because such a cast could fail

at runtime. `InvalidCastException` will be thrown at runtime for example if base class is not of a derived type.

The `as` operator works just as casting does except that the as operator will never throw an exception. Instead, if the object can't be cast, the result is `null`.

Another way of checking if objects are compatible is to use the `is` operator. It checks whether an object is compatible with a given type and returns true if object is compatible and false if not. The `is` operator will never throw an exception.

If object would need to be casted anyway it's more efficient to use as operator since if operation is successful we will get casted object. With is operator we would need to cast again and that can be slower.

The following code demonstrates casting to the derived types and displaying a specific properties not visible in a base class. Note that `Name` property is a base class property so it can be used without casting.

```
class InheritanceProgram
{
    private static void DisplayAsset(Asset asset)
    {
        Console.WriteLine(asset.Name);

        // Is operator checks whether a variable is a reference to a compatible type.
        if (asset is Car)
        {
            Car car = (Car)asset;
            Console.WriteLine(car.Make);
        }

        Stock stock = asset as Stock; // If asset is not a Stock object, null is returned
        if (stock != null)
        {
            Console.WriteLine(stock.NumberOfShares);
        }
    }
```

The following code demonstrates how classes `Asset`, `Car` and `Stock` can be used. References to `Car` or `Stock` can be used where `Asset` is expected, because a `Car` is an `Asset`. Thus, references are said to be *polymorphic*.

```
    public static void Main(string[] args)
    {
        Car car = new Car();
        car.Name = "Yugo 45";
        car.Make = "Zastava";

        var stock = new Stock { Name = "MSFT", NumberOfShares = 10000 };

        DisplayAsset(car);
        DisplayAsset(stock);

        // Here car is being upcast to its base type Asset. Upcasting can be implicit.
        Asset asset = car;

        // However, downcasting to subclass references must be explicit.
        Car car2 = (Car)asset;

        // Car car2 = asset; // Compile time error.

        // Stock stock2 = (Stock)asset; // Runtime error (since asset is of Car type).
```

```
            // Even though asset is actually a Car, you cannot use a reference to Asset
            // to call Car members. A cast is required.
            // Console.WriteLine(asset.Make); // Compile time error.
        }
    }
```

Given that C# doesn't support multiple inheritance, it offers a "scaled down" version via interfaces. An interface is really just a way to give a name to a set of method or property signatures without providing any implementation at all. Interface can define method signatures, parameterless properties, index properties and events. However, an interface cannot define any constructor methods nor an instance fields.

```
    // An interface defines all things that can return a value.
    interface IValuable
    {
        decimal Value { get; }
    }

    // An interface that defines all things that can have a name.
    interface INameable
    {
        string Name { get; set; }
    }
```

A class inherits an interface the same way it inherits any other base class, however that class must explicitly provide implementations of the interface's methods.

In the following snipped, Asset class is defined as abstract. An abstract class cannot be instanced using the "new" operator, but is intended to be inherited. A class can implement many interfaces, but it can derive only from a single class. All members derived from an interface must be declared as public.

```
    abstract class Asset : IValuable, INameable
    {
        // A constructor which initializes the object by setting the Name value.
        public Asset(string name)
        {
            Name = name;
        }

        // A constructor may call another constructor by using the "this" keyword.
        public Asset()
            : this(string.Empty)
        {
        }

        // A private field to store the name.
        private string _name;

        // A public property to expose the name value.
        public string Name
        {
            get
            {
                return  name;
            }
            set
            {
                _name = value;
            }
        }

        public virtual void Display()
        {
            Console.WriteLine(Name);
```

```
        }

        public abstract decimal Value { get; set; }
    }
```

Display method is marked as *virtual*. Marking a member virtual means that it can be re-implemented, or "overridden" by a derived class in a polymorphic way. What that means is that no matter whether the reference type we have is `Asset`, or say `Car`, the correct implementation will be called, that is the implementation defined by the derived class. Omitting the virtual keyword would cause that `((Asset)o).Display()` and `((Car)o).Display()` may call different methods.

In addition to `virtual` members, a base class can have `abstract` members like property `Value`. An abstract member does not provide an implementation and any subclass must either implement the abstract member of the base class, or it must be abstract itself. An abstract member is implicitly virtual.

If the constructor of the base class is not invoked explicitly, the parameterless constructor of the base class is called implicitly - if it exists. If it doesn't a compile time error is reported. When implementing a virtual member of the base class, keyword "override" must be specified. Failing to do so would cause the original method to be hidden.

```
    class Car : Asset // Inherits from Asset
    {
        // A constructor may call a constructor of the base class by using the "base" keyword.
        public Car(string name, string make)
            : base(name)
        {
            Make = make;
        }

        public Car()
        {
        }

        public string Make { get; set; }

        public override void Display()
        {
            base.Display(); // Here we call the implementation from the base class.
            Console.WriteLine(Make);
        }

        public override decimal Value { get; set; }
    }
```

In the following example since the property `Name` is not marked as "virtual" in the base class we cannot override it. This new property actually “hides” it. What this means is that both properties physically exist in a Stock object instance, but which one is used depends on the type of the reference use. `((Asset)o).Name` cause the property defined in `Asset` to be used, while `((Stock)o).Name` uses this implementation. This is generally not a desired behavior and that's why C# compiler issues a warning when this is detected, unless we specify the "new" keyword which basically tells the compiler that this is in fact our intention. Note that the same would happen if Name were defined as virtual, but we failed to include "override" keyword here.

```
    class Stock : Asset // Inherits from Asset
    {
```

```
        public new string Name { get; set; }

        public override decimal Value { get; set; }

        public long NumberOfShares { get; set; }
    }
```

The following code is an example on how our derived classes `Car` and `Stock` can be used together through a common interface `INameable`.

```
    class InheritanceProgram
    {
        private static void DisplayNames(IEnumerable<INamable> namables)
        {
            foreach (var namable in namables)
            {
                Console.WriteLine(namable.Name);
            }
        }

        public static void _Main(string[] args)
        {
            List<Asset> assets = new List<Asset>
            {
                new Car { Name = "Yugo 55", Make = "Zastava", Value = 250 },
                new Car { Name = "Kombi", Make = "Volkswagen", Value = 300 },
                new Stock {Name = "MSFT", NumberOfShares = 10000, Value = 300000 },
            };

            DisplayNames(assets);
        }
    }
```

Generic types

The .NET Framework allows you to define flexible type-safe data structures, without committing to actual data types. These data structures are called generics. Generics introduce to the .NET Framework the concept of type parameters, which make it possible to design classes and methods that defer the specification of one or more types until the class or method is declared and instantiated by client code.

Generic classes

The standard classes in C# .Net have explicitly defined types for all fields, properties, and methods. In the following example is shown `Point` class with three coordinates defined as integer values:

```
public class Point
{
    public int x, y, z;
}
```

Types of the fields are hardcoded and cannot be used if we need points with `long` or `double` coordinates. In that case we should create new classes that would be copy of the existing one.

In order to create more flexible class that can be easily extended, we can use generic types. To do that, use the class definition **<** and **>** brackets, enclosing a generic type parameter, and instead of actual types use the generic type placed in the brackets. For example, here is how you define and use a generic point:

```
public class Point<T>
{
    public T x, y, z;
}
```

Generic type <T> represents some type that will be defined in the concrete class definitions. This class does not explicitly define types of fields and it can be used as a template for generating points that have various types of coordinates. Example of point classes with `int`, `long`, and `double` types of fields that are defined using this template are shown in the following listing:

```
public class IntPoint : Point<int> { }

public class LongPoint : Point<long> { }

public class DoublePoint : Point<double> { }
```

The actual instances of these classes will have properties with type that is defined in the **<** and **>** brackets, and they can be used as a regular classes. While creating objects of these types, you can either use these derived classes, or directly create instance of generic class by specifying type as shown in the following example:

```
Point<int> p1 = new Point<int>() { x = 1, y = 3 };
p1.z = 4;
```

```
Point<double> p2 = new Point<double>() { x = 1.3, y = 3.7, z = 4.2 };

LongPoint p3 = new LongPoint() { x = 1L, y = 3L, z = 4L };
p3.z = 7L;
```

We can create instances either as specialization of generic class (e.g. `Point<int>`) or as instance of classes that has already defines a type for this this generic class (e.g. `LongPoint` class).

The most common usage of generics are collection processing. Standard collections in C# .Net are structures that contain objects (type `System.object`) where can be placed any types. As an example, we can create stack that should contain strings; however, there is a possibility that other types can be placed there too. Example of such kind of stack is shown in the following example.

```
var stringStack = new System.Collections.Stack();
stringStack.Push("First");
stringStack.Push("Second");
stringStack.Push("Third");
stringStack.Push("Fourth");
stringStack.Push("Fifth");

//Valid call: no semantic type checking is done here.
stringStack.Push(new DateTime()); // Objects other than strings can be pushed to the stack

int length = 0;
foreach (object elem in stringStack)
{
    if (elem is string) // We need to check is the element in the stack actually a string.
    {
        string str = (string)elem;
        length += str.Length;
    }
}
```

Although we want to use stack of string, a standard stack class enables us to put any value (e.g. `DateTime`). Therefore, while we reading data from the stack, we need to check is the element actually string, or someone has placed data type we are not expecting. This code requires a lot of type checking, casting from object to desired type, and also it can cause a lot of errors.

Instead of stack of objects, we can use stack specialized for strings using generic `Stack<T>` class as it is shown in the following listing:

```
var stringStack = new System.Collections.Generic.Stack<string>();
stringStack.Push("First");
stringStack.Push("Second");
stringStack.Push("Third");
stringStack.Push("Fourth");
stringStack.Push("Fifth");

int length = 0;
foreach (var elem in stringStack)
{
    length += elem.Length;
}
```

Beside the fact that code is cleaner and that no casting/type checking is required, we have type safe structure where we can work with string only. If someone tries to put some other type of object into the stack, compile time error will be reported.

Generic methods

When the generic types are defined at the class level, they can be used in any fields, properties or methods in that class. However, in some cases we might want to create generic types per individual methods. In that case we will put generic type in the method name in the **<** and **>** brackets, and that types will be applied in the method. Example of the generic method defined in the class is shown in the following example:

```
public class Klass<T1>
{
    public T1 x, y, z;

    public static void Replace<T2>(ref T2 x, ref T2 y)
    {
        T2 temp;
        temp = x;
        x = y;
        y = temp;
    }
}
```

Generic type T1 that is defined at the class level can be used in any field or method in the class (e.g. as return value, parameter, or local variable in the method). However, generic type T2 is defined in the method, and can be used only within that method.

When generic method is called, type should be defined in the call as shown in the following example:

```
int x = 2, y = 3;
Replace<int>(ref x, ref y);

long a = 2, b = 3;
Replace<long>(ref a, ref b);
```

Another example of generic method that returns a default value for some type is shown in the following example:

```
static T CreateDefault<T>()
{
    return default(T);
}
```

This method will use a type, determine default value for that type and return this object as a return value using the `default(T)` method. Some examples of calls are shown in the following code:

```
int number = CreateDefault<int>();// number = 0

string str = CreateDefault<string>(); // str = null

int? nullable = CreateDefault<int?>(); // nullable = null

DateTime date = CreateDefault<DateTime>();// date = {1/1/0001 12:00:00 AM}
```

Generics namespace in .Net library

In the .Net library can be found a lot of generic glasses that enables you to create your own flexible data structures.

The most commonly used generic classes are generic collections. Generic collections are placed in the `System.Collection.Generics` namespace where you can find a generic versions of the most commonly used structures such a lists, sets, stacks, queues, etc. In the following example is shown how you can use various collections in this namespace:

```
var stringStack = new System.Collections.Generic.Stack<string>();
stringStack.Push("First");
stringStack.Push("Second");
stringStack.Push("Third");
stringStack.Push("Fourth");
stringStack.Push("Fifth");

var stringQueue = new System.Collections.Generic.Queue<string>();

while (stringStack.Peek() != null)
{
    stringQueue.Enqueue(stringStack.Pop());
}

var map = new System.Collections.Generic.Dictionary<string, int>();

while(stringQueue.Count>0){
    string element = stringQueue.Dequeue();
    if(map.ContainsKey(element))
        map[element]++;
    else
        map[element] = 1;

}

System.Collections.Generic.List<string> stringList = stringQueue.ToList();
foreach(var str in stringList){
    Console.Out.WriteLine(str);

}
```

First, we have created one stack of strings where we have placed few string objects. Any attempt to put object with types other than `string`, will cause compile-time errors. Then strings are taken from the stack and placed in the queue of strings. As you can see no type checking/casting is required because `Pop` method will return string, and `Enqueue` method accepts string as an argument.

If we need to count the number of occurrences in the queue, we can create a map where the key will be `string` (the unique value in the queue) and value will be `int` (the number of occurrences of the key). In the while loop are taken elements from the queue and if this is a first occurrence of the word number 1 is added in the map for that key, or the number of occurrences is incremented.

Finally, a queue of strings is converted to list so it can be accessed as a standard string list, and printed to console.

If you need a flexible structure that holds an arbitrary number of strongly typed values, you can use Tuple class. Tuple is a fixed size set of ordered items (e.g. vector) where you can define types for each dimension.

```
Tuple<int, string, bool> t1 = Tuple.Create(5, "tuple", true);

//Dimensions can be accessed via types properties Item1, Item2, Item3
if (t1.Item1 == t1.Item2.Length && t1.Item3)
{
    Tuple<int, string, bool> t2 = Tuple.Create(5, "tuple", false);
    t1 = t2;
}

Tuple<long, string, bool> t6 = Tuple.Create(5L, "tuple", false);
Tuple<long, string, bool> t7 = Tuple.Create<long, string, bool> (5, "tuple", false);
```

In the example is created a tuple with items of type `int`, `string` and `bool`. You can access individual items in the tuple using the `Item1`, `Item2`, and `Item3` properties – number of items is equal to the dimension of tuple. However, these items are read-only – you cannot modify items in the tuple once they are created. Items in the tuple object are strongly typed, so they can be used without any casting.

`Tuple.Create()` method determines types of items in the tuple using the types of arguments passed to the method (e.g. value 5 generate `int` type, while 5L generates `long` type as you can see in the variables t1 and t6). However, explicit types are passer to the function call values passed as arguments are converted (see variable t7 in the example above).

Constraints

In the previous examples, once we have defined generic class, we were able to specialize it with any type. However, if we need control over the type of classes that can be used to specialize classes; C# enables us to define various constraints that can be applied.

In order to specify characteristics of generic type, we can add a where clause with description of the type.

As an example, we might want to create generic lists that can contain variables of some class (reference) types, or just some value types. In that case, we can define so called constraints that define what concrete classes can be used instead of the generic types.

```
public class GenericList<T> : List<T> where { }
public class ValueList<T> : List<T> where T : struct { }
public class ObjectList<T> : List<T> where T : class { }
```

`GenericList` is a class inherited from standard `List<T>` class; therefore, any type can specialize this template. If we want to have a list that can be specialized with value types only (e.g. `DateTime`, `int`), or reference types only (e.g. `string`, `Exception`), we can add a `where` clause with a constraint specifying that generic type must be structure or class. Specializing `ValueList<T>` with any class type, or `ObjectList<T>` with any structure type, will cause compile-time error.

We can add even more precise description, such as that type must implement some interface, be derived from the particular class or even another generic type used in template, as shown in the following example:

```
public class CloneableList<T> : List<T> where T : ICloneable { }
```

```
public class ExceptionList<T> : List<T> where T : Exception {

        public static TList Add<TList, T>(TList list, T data) where TList : IList<T>
        {
            list.Add(data);
            return list;
        }
}
```

We have defined a class `CloneableList` that can be specialized with any class that inherits `ICloneable` interface. `ExceptionList` is similar and it can be specialized with any class that derives Exception class. In the Exception list is defined a static method that accepts a list defined as generic `TList` type, with a constraint that `TList` must be a list of items with type T.

If we do not specify that `TList` is type of `IList<T>`, we cannot use `Add()` method in the body of method. Without this constraint, compiler would not be aware that object of type `TList` has any method from `IList` interface that could be called, and this statement will cause a compile-time error.

Another interesting constraint is `new()`. With this constraint, we can define that generic type must have a constructor without parameters.

```
public static T Create<T>() where T : new()
{
    return new T();
}
```

Calling this method with any type that do not have parameterless constructor (e.g. `string` class) would cause a compile-time error. This constraint enables us to create instances of objects within the methods by calling this constructor.

We can even specify multiple constraints on the same type separating conditions with comma as it is shown in the following example:

```
public static TResultList AddToList<TResultList, TList, TData>(TList list, TData data)
    where TData : class, new()
    where TResultList : IList<TData>
    where TList : TResultList , ICloneable
{
    list.Add(data);
    return list;
}
```

In the example, `TData` is a class with parametarless constructor, `TResultList` is a list of `TData`, and the `TList` is cloneable class derived from the `TResult`.

Recap

The purpose of this lesson was to provide you the overview of .NET generics and how they are used in C#. Generics are important language construction that enables you to create clean and elegant code, reduce repetitive task and simplify your code.