Value Type and Referance Type

C# dilinde iki tür veri tipi mevcuttur. Bunlar değer ve referans tipleridir. Değer tipleri; veriyi taşıyan ve taşıdığı veriye göre bellek üzerinde yer dolduran değişken türleridir.Referans türleri ise, bellek bölgesinde veri yerine adresi tutarlar ve o adresin gösterdiği yerde de veri tutulur.

Değer Tipleri: “int”, “long”, “float”, “double”, “decimal”, “char”, “bool”, “byte”, “short”, “struct”, “enum”

Referans Tipleri: “string”, “object”, “class”, “interface”, “array”, “delegate”, “pointer”

Bir metoda parametre olarak Değer Tipli bir değişken verildiğinde; bellekte yeni bir alan oluşturulur ve değişkenin taşıdığı verinin kopyası bu alana yerleştirilir. Metot içerisinde yapılan işlemlerde de yeni oluşturulan bellek bölgesinde bulunan veri kullanılır. Bu yüzden metot içerisinde yapılan değişikliklerden ana değişken herhangi bir şekilde etkilenmez.

Bir metoda parametre olarak Referans Tipli bir değişken verildiğinde; bellekte yeni bir alan oluşturulur ve değişkenin bellekteki adresi bu alana yerleştirilir. Metot içerisinde yapılan işlemlerde yeni oluşturulan bellek bölgesinde bulunan adres kullanılır, bu adreste ana değişkeni gösterdiği için metot içerisinde yapılan değişikler doğrudan ana değişkeni etkileyecektir.

Nullable Type

Bilindiği gibi değişken tipleri, depolanma durumları ve davranışları bakımından değer ve referans olmak üzere ikiye ayrılır. Bu iki veri tipi arasındaki önemli bir farktan bahsedeceğiz; referans türlerinin null(nothing) değeri alabiliyorken değer türlerinin bu özelliği taşımıyor olmasıdır. Yani değer türleri, her zaman bir değer taşımak zorundadır. Bir değişkenin, null olması, herhangi bir nesneye işaret etmiyor olduğu anlamına gelir. Özellikle veritabanı işlemlerinde null tabanlı kolonlarla çalışıldığında veya uygulama içerisinde değişkenlerin herhangi bir değer taşımaması gereken durumlarda bu özellik önem kazanmaktadır.

? takısı ile kullanılabililr. Compiler (derleyici) ? ifadesini gördüğü yerde Nullable yapısına dönüştürmektedir.Örnek olarak ;

```
1 int myAge = 29;
2 int? nullableAge = myAge;
```

şeklinde tanımlanabilmektedir.

Abstract Class

Abstract sınıflar,sınıf hiyerarşisinde genellikle base class (temel sınıf) tanımlamak için kullanılan ve soyutlama yeteneği kazandıran sınıflardır. Bir sınıfı abstract yapmak için abstract keywordünü kullanırız.Static tanımlanamazlar.Bu sınıftan new anahtar kelimesi kullanılarak bir nesne oluşturulamaz.

Örnek olarak, Ucak adında bir tane Abstract(taban) sınıfımız olacak. Ve daha sonra bu Abstract Class'dan BuyukUcak ve KucukUcak sınıflarını kalıtacağız. Ve Ucak Abstract Class'ımızın içerisine basit bir "UcakFiyatı" isimli method yazıp, bu method'u kalıtım yaptığımız Class'larda override ederek, UcakFiyatı isimli methodumuza gövdeyi implement edeceğiz.

Öncelikle "Ucak" Abstract Class'ımızı yazalım.

```
abstract class Ucak
{
    0 references
    public int YolcuKapasitesi { get; set; }
    0 references
    public string UcakSirketi { get; set; }

    2 references
    public abstract void UcakFiyati();
}
```

Abstract Method

Sadece soyut sınıflar içerisinde kullanılabilirler. Mirasçı sınıflarda override edilmek zorundadırlar.Abstarct metotlar sadece tanımlanır. Herhangi bir işlemi yerine getirmezler. Yapacakları işlemler override edildikleri sınıfta kodlanmalıdır.Örnek olarak yukarıda tanımladığımız Abstract Class'ımızda, sonradan oluşturulabilecek Ucak Class'larının ortak özellikleri ve bir tane override ederek gövdesini oluşturacağımız bir Abstract Method vardır.Ucak Abstract Class'ından kalıttım yaptığımız BuyukUcak ve KucukUcak sınıflarında, UcakFiyati Abstract Method'unu override ederek gövdesini dolduruyoruz.

```
 class BuyukUcak : Ucak
{
    2 references
    public override void UcakFiyati()
    {
        Console.WriteLine("Büyük Uçağın Fiyatı 5m");
        Console.ReadLine();
    }
}

0 references
class KucukUcak : Ucak
{
    0 references
    public int UcusSuresi { get; set; }
    2 references
    public override void UcakFiyati()
    {
        Console.WriteLine("Küçük Uçağın Fiyatı 1m");
        Console.ReadLine();
    }
}
```

Patial Class

Partial class ise bir class' ı birden fazla class olarak bölmemize olanak sağlar. Fiziksel olarak birden fazla parça ile oluşan partial class' lar, Çalıştığı zaman tek bir class olarak görev yapar.Partial class ile fiziksel olarak parça class'ların birleşmesi için class isimlerinin aynı olması gerekmektedir.

Syntax:

public partial Clas_name

{

// code

}

Mutable and Immutable

Immutable (değişmez), nesneler bir kez oluşturulduktan sonra içeriği değiştirilemeyen sınıflardır. Tam tersi olarak, değiştirilebilen sınıflar da Mutable (değişebilir) sınıflardır.

Mutable veri tiplerine örnek olarak ise, Date, StringBuilder gibi tipleri Immutable veri tiplerine örnek olarak, string, integer, double, byte gibi tipleri örnek verebiliriz.

Indexer

Indexer özel tanımlı bir property'dir ve sadece class içerisinde tanımlanabilir. Tanımlandığı class'a indexlenebilir özelliği kazandırır. Array işlemlerinde kullandığımız [ ] operatörünü tanımlamış olduğumuz bir class'ı diziymiş gibi işlemler yapabilmek içinde kullanabiliriz.

```
public static void Main()
{
    Department dprt = new Department();
    dprt[0] = "Bilgi İşlem";
    dprt[1] = "Proje Yönetimi";
    dprt[2] = "Analiz";
    dprt[3] = "İş Geliştirme";
    dprt[4] = "Destek Sistemler";
    Console.WriteLine(dprt[4]); //Destek Sistemler
}
1 reference
public class Department
{
    0 references
    public string Name { get; set; }
    0 references
    public int ID { get; set; }
    //indexer tanımlaması
    private string[] names = new string[5];
    6 references
    public string this[int index]
    {
        get { return names[index]; }
        set { names[index] = value; }
    }
}
```

ReadOnly

Birçok yerde readonly ifadesini görmüşsünüzdür. readonly olarak tanımlanan bir değişken yalnızca okunmak üzere tanımlanır ve yalnızca 2 şekilde değer ataması yapılabilir.

1) İlk tanımlandığında değer ataması yapılarak

2)Bir constructor içinde değer ataması yapılarak

Yukardaki 2 durum hariç readonly değişkenine değer ataması yapılamaz.Bu durum, aslında sadece bir yapılandırıcı tarafından değer ataması yapılması gerektiği durumlarda kullanışlıdır. Böylece daha sonra değer değiştirmesi önlenmiş olur. Kullanım alanı oldukça fazladır.Aşağıdaki örneği incelersek ;

```
class DenemeClass
{
    readonly double x = 30;
    1 reference
    public DenemeClass(int _x) {
        x = _x; //İlk değer ataması yapıldı
        x = x * 5; //Burda değeri değiştiriliyor. Doğru ifade çünkü constructor içindeyiz.
    }
    1 reference
    public DenemeClass(double _x){
        x = _x;
    }
    2 references
    public void Show() {
        Console.WriteLine(x);
    }
    0 references
    static void Main(){
        DenemeClass dc = new DenemeClass(10);
        dc.Show();

        DenemeClass dc2 = new DenemeClass(10.2);
        dc2.Show();

        dc2.x = 10.3;//HATA!!
        Console.Read();
    }
```

dc2.x = 10.3; ataması yapıldığında program kızacaktır. Çünkü bir readonly değişken, hata açıklamasında yazdığı üzere sadece ilk değer atamasında ya da bir constructor'da set edilebilir.

Const

Const da aslında readonly'e benzer özellikler gösterir. Aralarındaki fark,Bir değişkenin değerinin program boyunca sabit olarak tutulması istendiğinde const (sabit) ifadesinden yararlanılır. Tanımlandığı satırda değeri atanmalıdır.

const int a; --> HATA!! Değer ataması yapılmalıdır

const int a = 10; --> Doğru ifade.

Constructors

Yapılandırıcıların (constructor) görevi oluşturulan nesneyi ilk kullanıma hazırlamasıdır. C# da tüm sınıflar (class) tanımlansın ya da tanımlanmasın değer tiplerine sıfır, referans tiplerine "null" değerini atayan varsayılan bir yapılandırıcı vardır. Yapılandırıcısı tanımlandıktan sonra varsayılan yapılandırıcı bir daha kullanılmaz.

Func and Action

Action dönüş tipi olmayan (void), Func dönüş tipi generic olan özel delege yapılarıdır.

Delegate

Delegeler referans türlü bir tiptir. Dolayısı ile nesneleri heap'de durur. Girişte bahsettiğimiz gibi, görevleri metot adresi tutmaktır. Burada dikkat edilmesi gereken nokta; delegenin imzası, tuttuğu metodun imzası ile aynı olmalıdır. İmzadan kastımız, geriye dönüş tipi ve aldığı parametrelerdir. Bir delege, birden fazla metot adresi tutabilir. Bu durumda FIFO (ilk giren ilk çıkar) prensibi geçerlidir. Yani metotlar, delegeye bağlanma sırasına göre çalışırlar. Sonuç almak istediğimiz zaman, en son eklenen metodun yaptığı işi görürüz.

```
namespace deneme
{
    delegate void Temsilci();
    //Tanımladığımız bu delege, geriye dönüş tipi void olan ve parametre almayan metotların adreslerini saklayabilir.
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Temsilci t = new Temsilci(Test);
             t.Invoke();
        }

        1 reference
        static void Test()
        {
            Console.WriteLine("Metot çalıştı");
        }

        }

    }
```