Generice

Introducere (scop si beneficii)

Parametrii generici

Constrangeri asupra parametrilor generici

Clase generice

Interfete generice

Metode Generice

Delegari generice

Clasa Nullable

Introducere (Scop si beneficii)

- Genericele (Generics) au fost adaugate odata cu versiunea 2.0 a limbajului C# si motorului comun de programare (CLR).
- Prin conceptul de parametru generic (numit si parametru de tip) este posibila realizarea unei clase (sau metode) care nu specifica unul sau mai multe tipuri utilizate pana la momentul cand clasa (sau metoda) este declarata si instantiata.
- Spre exemplu, prin utilizarea unui parametru generic de tip T, se poate crea o clasa pe care o alta portiune de cod sa o utilizeze fara a suporta costurile operatiilor de impachetare sau despachetare:

```
public class ListaGenerica<T> // Declaram clasa generica.
{    void Add(T input)  { //codul metodei Add
     }
}
   class Test {
      private class A { }
      static void Main(){
         ListaGenerica<int> lista1 = new ListaGenerica<int>(); // Declaram o lista de tip int.
         ListaGenerica<string> lista2 = new ListaGenerica<string>(); // Declaram o lista de tip string.
         ListaGenerica<A> lista3 = new ListaGenerica<A>(); // Declaram o lista de tip A.
      }
   }
```

-Clasele si metodele generice combină conceptele: reutilizare, tipuri sigure (type safety) și eficiență într-un mod pe care clasele non-generice nu-l pot realiza.

-Generice sunt cel mai frecvent utilizate cu colecțiile și metodele care operează asupra lor. Versiunea . NET 2.0 oferă un nou spațiu de nume, System.Collections.Generic, care conține mai multe clase colectii care utilizeaza genericele.

-Pentru crearea unor tipuri sigure, se recomanda utilizarea colectiilor generice in locul colectiilor non-generice.

-Următorul program prezinta un exemplu simplu in care se poate crea o lista generica. Insa, in cele mai multe cazuri, ar trebui utilizata clasa Lista <T> furnizata de Biblioteca .NET.

```
using System.Collections.Generic;
// pentru parametrul generic T se folosesc
    paranteze unghiulare.
public class ListaGenerica<T>{
    // O clasa imbricata este si ea
    generica in T.
    private class Imbricat{

        private Imbricat next;
        // T ca membru privat.
        private T data;

        public Imbricat(T t){
            next = null;
            data = t;
        }
        public Imbricat Next {
            get { return next; }
            set { next = value; }
        }
        // T ca tip rezultat al unei
   proprietati.
        public T Data{
            get { return data; }
            set { data = value; }
        }
    }
    private Imbricat head;
     // constructor
    public ListaGenerica(){
        head = null;
    }
    // T ca tip al unui parametru formal
   pentru o metoda.
    public void AddHead(T t){
        Imbricat n = new Imbricat(t);
        n.Next = head;
        head = n;
    }
    public IEnumerator<T> GetEnumerator() {
        Imbricat current = head;
        while (current != null)
        {     /*Cuvantul cheie yield se
   foloseste intr-o declaratie
              pentru a indica faptul  că
   metoda, operatorul,
               sau accesorul get
               în care apare este un
   iterator.*/
              yield return current.Data;
             current = current.Next;
        }
    }
}
```

```
public class A {
    int i;
    public A(int i){
    this.i = i;
    }
    public override string ToString(){
    return i.ToString();
    }
}
class TestListaGenerica{
    static void Main(){
        // A este tipul argumentului
        ListaGenerica<A> lista =new ListaGenerica<A>();

        for (int x = 0; x < 10; x++){
            lista.AddHead(new A(x));
        }

        foreach (A a in lista)        {
            System.Console.Write(a + " ");
        }
        System.Console.WriteLine("\nDone");
    }
}
```

-Genericele sunt importante la crearea si utilizarea colectiilor generice. Ele permit crearea unei colectii “type-safe” la momentul compilarii.
-Limitele colectiilor non-generice pot fi demonstrate prin urmatorul exemplu, care utilizeaza o colectie ArrayList pentru a pastra obiecte de orice tip:

```
using System.Collections;
class Demo{
   public static void Main() {
      ArrayList lista1 = new ArrayList();
      lista1.Add(3);
      lista1.Add(105);
      ArrayList lista2 = new System.Collections.ArrayList();
      lista2.Add("Primul string in colectie");
      lista2.Add("Al doilea string in colectie");
   } }
```

- Dar acest confort are un cost. Orice tip referință sau tip valoric, care se adaugă la un ArrayList este implicit transformat intr-un obiect de tipul Object. În cazul în care elementele sunt tipuri valorice, acestea sunt impachetate atunci când sunt adăugate in lista, și despachetate când sunt preluate. Operatiile de impachetare si despachetare scad performanta, iar efectul poate fi foarte important în scenarii care utilizeaza colecții mari.

-Altă limitare conduce la lipsa unei verificarei a codului la momentul compilarii. Nu exista nici un mod de a preintampina codul clientului sa faca ceva de genul acesta:

```
ArrayList listaNoua = new ArrayList();
listaNoua.Add(3);  // Adauga un intreg in lista.
listaNoua.Add("Primul string in colectie."); // Adauga un string in lista.

int t = 0;
// Codul de mai jos lanseaza o exceptie de tipul InvalidCastException.
foreach (int x in listaNoua)
{
    t += x;
}
```

-Deși este perfect acceptabil, și poate ca uneori în mod intenționat doriți să creați o colecție eterogenă, combinarea stringurilor si intregilor intr-un singur ArrayList este mult mai probabil să fie o eroare de programare, iar această eroare nu va fi detectata până la execuție (se lanseaza o exceptie).

-ArrayList și alte clase similare au nevoie de o modalitate prin care codul utilizatorului sa specifice tipul de date special pe care aceste clase intentioneaza sa-l foloseasca. Aceasta ar elimiana nevoia de utilizare a castului si ar facilita calea compilatorului să facă verificare a tipului.
-Cu alte cuvinte, ArrayList are nevoie de un parametru care sa specifice tipul elementelor din colectie. Asta este exact ceea ce oferă genericele. În colecția generica Lista <T>, din spatiul de nume System.Collections.Generic, aceeași operațiune de adăugarea de elemente unei colectii seamănă cu aceasta:

```
List<int> lista1 = new List<int>();
 // Fara impachetare sau cast:
 lista1.Add(3);
 // Eroare la compilare:
 // lista1.Add("Un string de adaugat in colectie.");
```

- Pentru codul client, sintaxa de adăugat atunci cand se utilizeaza List <T> in detrimentul unui ArrayList este argumentul <T> atat in declaratie cat si in instantiere. În schimb pentru această mica extindere, puteți crea o listă, care nu este doar mai sigura decât ArrayList, dar, de asemenea, semnificativ mai rapida, mai ales atunci când elementele din listă sunt tipuri valorice.

Parametrii generici

-Un parametru generic (sau parametru de tip) reprezinta un substituent pentru un anumit tip care urmeaza a fi specificat de utilizator atunci când instantiaza o variabilă de tip generic.
-O clasă generica, cum ar fi ListaGenerica <T> utilizata intr-un slide anterior, nu poate fi utilizata ca atare, pentru că nu este într-adevăr un tip, este mai mult un plan pentru un tip.
-Pentru a utiliza ListaGenerica <T>, codul client trebuie să declare și instantieze un tip prin specificarea unui argument în interiorul parantezelor unghiulare. Argumentul pentru această clasă particulară poate fi orice tip recunoscut de compilator. Exemplu:

```
ListaGenerica<int> lista1 = new ListaGenerica<int>();
ListaGenerica<string> lista2 = new ListaGenerica<string>();
ListaGenerica<A> lista3 = new ListaGenerica<A>();
```

-În fiecare din aceste instante ale clasei GenericList <T>, fiecare apariție a parametrului T în clasa va fi înlocuit în timpul rulării cu argumentul specificat. Prin această înlocuire, am creat trei obiecte de tipuri diferite utilizând o singură definiție de clasă.

Numele parametrilor generici:

-Denumiti parametrii generici cu nume descriptiv, cu exceptia cazului cand o singura litera este auto-explicativa si un nume ar adăuga nimic in plus. Exemple:

```
public interface IDictionary<TKey,TValue>
public delegate TOutput Converter<TInput, TOutput>( TInput input)
```

-Considerati utilizarea literei T ca numele parametrului pentru tipurile cu un singur parametru. Exemple:

```
public int IComparer<T>() { return 0; }
public delegate bool Predicate<T>(T item);
public struct Nullable<T> where T : struct { /*...*/ }
```

Constrangeri asupra parametrilor generici

- Când definiți o clasă generica, se pot aplica restricții asupra tipurilor pe care codul client le poate utiliza drept argumente de tip atunci cand se instanțiază clasa generica.
- În cazul în care codul client încearcă să instantieze clasa generica, prin utilizarea unui tip care nu este permis, rezultatul este o eroare de compilare. Aceste restricții sunt numite constrângeri.
- Constrângerile sunt specificate prin folosirea cuvantului cheie where.

Următorul tabel listează cele șase tipuri de constrângeri:

Constrangere	Descriere
where T: struct	Argumentul de tip trebuie sa fie un tip valoric. Asadar T poate fi orice tip valoric cu exceptia tipului Nullable.
where T : class	Argumentul T trebuie sa fie un tip referinta; aceasta se aplica pentru orice clasa, interfata, delegare sau tablou.
where T : new()	Argumentul de tip trebuie sa aiba un constructor public fara nici un parametru. Atunci cand se utilizeaza cu mai multe constrangeri, new() trebuie specificat ultimul.
where T : <base class name>	Argumentul T trebuie sa fie de tipul clasei sau sa derive din clasa specificata.
where T : <interface name>	Argumentul T trebuie sa fie sau sa implementeze interfata. Se pot include constrangeri interfata multiple. Contrangerea interfata poate fi de asemenea generica.
where T : U	Argumentul T trebuie sa fie de tipul sau sa derive din argumentul specificat pentru U.

De ce se utilizeaza constrangerile?

- Dacă doriți a examina un element dintr-o listă generică pentru a determina dacă acesta este valid sau nu sau pentru a-l compara cu un alt element, compilatorul trebuie să aibă o oarecare garanție că metoda pe care o utilizeaza este suportata de către orice tip de argument care ar putea fi specificat de codul client. Această garanție este obținută prin aplicarea uneia sau mai multor constrângeri în definiția clasei generice.

-De exemplu, constrângerea "clasa de baza" spune compilatorului că numai obiecte de acest tip sau derivate din acest tip vor fi utilizate ca argumente de tip. Odata ce compilatorul are această garanție, se poate permite metodei de acest tip să apeleze clasa generica.

- Clasa ListaGenerica <T> o putem modifica astfel:

```
public class ListaGenerica<T> where T:A
{
}
```

Clase generice

-Clasele generice includ operațiuni care nu sunt specifice pentru un tip particular de date.

- Cea mai comună utilizare a claselor generice este in colecții, cum ar fi: liste, stive, cozi etc. Operații cum ar fi adăugarea sau eliminarea de elemente dintr-o colectie sunt efectuate în esență in același mod, indiferent de tipul de date stocate.

- Pentru cele mai multe scenarii care necesită clase colectii, se recomandă utilizarea claselor prevăzute în Biblioteca .NET.

-De obicei, creați clase generice plecand de la o clasa concreta (non-generica) si schimband pe rand tipurile dorite in tipuri generice pana se atinge un echilibru optim intre generalizare si usurinta in utilizare.

Când creati propriile clase generice, considerati importante următoarele considerente:

-*Ce tipuri ar trebui generalizate in parametrii generici?*

Ca o regulă, cu cat mai multe tipuri puteți parametrizarea cu atat codul devine mai flexibil și reutilizabil. Cu toate acestea, o generalizare prea extinsa poate crea cod care este dificil pentru alți dezvoltatori sa-l citeasca sau sa-l înțeleaga.

-*Ce constrângeri, dacă este cazul, să se aplice parametrilor generici?*

O regulă bună este să se aplice constrângeri maxime posibile, care vor permite în continuare manipularea tipurilor pe care aplicatia (programul) le are in vedere. De exemplu , dacă știți că acea clasa generica este destinata utilizarii numai cu tipurile referință, se aplică restricția de clasă. Aceasta va preintampina utilizarea clasei pentru tipuri valorice si facilita folosirea operatorului as si testarea valorilor null.

-*Dacă se impune utilizarea comportamentului generic in clase si subclase.*

Deoarece clasele generice poate servi drept clase de bază, aceleași considerente de proiectare se aplică aici ca si in cazul claselor non- generice.

-*Daca se impune implementarea unei (sau mai multor) interfețe generice.*

De exemplu, dacă proiectați o clasă care va fi folosita pentru a crea elemente ale unei colectii generice, va trebui să implementati o interfață, cum ar fi IComparable <T> unde T este tipul clasei dumneavoastra.

Regulile pentru parametrii genericii și pentru constrângeri au mai multe implicații asupra comportamentului clasei generice, în special ceea ce privește moștenirea și accesibilitate membrilor. Astfel:

- Pentru o clasa generica ClasaMeaGenerica <T>, codul client poate face referire la clasa, fie prin specificarea unui argument de tip, pentru a crea un **tip închis** (ClasaMeaGenerica <int>). Alternativ, se poate lăsa parametrul de tip nespecificat, de exemplu, atunci când specificați o clasa de baza generica, pentru a crea un **tip deschis** (ClasaMeaGenerica <T>) . Clase generice pot moșteni o clasa concreta, un tip inchis sau un tip deschis:

```
class ClasaMea { }
class ClasaMeaGenerica<T> { }

// clasa generica mosteneste clasa nongenerica (concreta)
class MostenesteClasaConcreta<T> : ClasaMea { }
// clasa generica mosteneste clasa generica inchisa
class MostenesteClasaInchisa<T> : ClasaMeaGenerica<int> { }
//clasa generica mosteneste clasa generica deschisa
class MostenesteClasaDeschisa<T> : ClasaMeaGenerica<T> { }
```

-Clasele non-generice (concrete) pot moșteni tipuri închise, dar nu tipuri deschise sau parametri de tip deoarece în timpul rulării nu există nici o modalitate pentru codul client de a furniza argumentul de tip necesar pentru a instantia clasa de baza:

```
class ClasaMea : ClasaMeaGenerica<int> { } //fara eroare
//class ClasaMea : ClasaMeaGenerica<T> { } //Genereaza o eroare
//class ClasaMea : T {} //Genereaza o eroare
```

- Clasele generice care moștenesc tipuri deschise trebuie să furnizeze argumente de tip pentru fiecare parametru de tip al clasei de baza care nu intervine explicit în clasa derivata, așa cum se arata în următorul cod:

```
class ClasaMeaGenerica<T, U> { }
class ClasaA<T> : ClasaMeaGenerica<T, int> { } //fara eroare
class ClasaB<T, U> : ClasaMeaGenerica<T, U> { } //fara eroare
//class ClasaC<T> : ClasaMeaGenerica<T, U> {}  //genereaza eroare
```

-Clasele generice care moștenesc tipuri deschise trebuie să implice constrângerile tipului:

```
class ClasaMeaGenerica<T> where T : System.IComparable<T>, new() { }

class ClasaMeaGenericaSpeciala<T> : ClasaMeaGenerica<T> where T :
System.IComparable<T>, new() { }
```

-Tipurile generice pot utiliza parametrii de tip multiplii și impune constrângeri multiple, după cum urmează:

```
class ClasaMeaGenerica<K, V, U>
      where U : System.IComparable<U>
      where V : new()
  { }
```

-Tipurile deschise sau inchise pot fi utilizate ca parametrii pentru metode:

```
 void MetodaMea<T>(List<T> lista1, List<T> lista2)
  {      //codul metodei    }

  void MetodaMea(List<int> lista1, List<int> lista2)
  {      //codul metodei    }
```

Interfete generice

Adesea este util să se definească interfețe, fie pentru clase colectii generice, sau pentru clase generice care reprezintă elemente intr-o colecție.

De preferat, atunci cand se utilizeaza clase generice, este de a utiliza interfețe generice, cum ar fi IComparable <T> in locul lui IComparable, aceasta pentru a evita operatiile de impachetare si despachetare a tipurilor valorice.

Biblioteca .NET defineste mai multe interfețe generice pentru a fi utilizate de clasele colectii din spațiul de nume System.Collections.Generic.

Atunci când o interfață este specificata ca o constrângere asupra unui parametru de tip, pot fi utilizate numai tipuri care implementeaza interfata (a se vedea exemplul urmator unde intr-o lista de tipul ListaSortata<T> pot fi adaugate doar instante ale unor clase care implementeaza interfata System.IComparable<T>.

```
using System.Collections.Generic;
using System.Collections:

public class ListaGenerica<T>: IEnumerable<T>{
    protected Imbricat head;
    // O clasa imbricata este si ea generica in T.
   protected class Imbricat{
        private Imbricat next;
       // T ca membru privat.
       private T data;

       public Imbricat(T t){
           next = null;
           data = t;
       }
       public Imbricat Next {
           get { return next; }
           set { next = value; }
       }
       // T ca tip rezultat al unei proprietati.
       public T Data{
           get { return data; }
           set { data = value; }
       }
   }
    public ListaGenerica(){
       head = null;
    }
    public void AddHead(T t){
       Imbricat n = new Imbricat(t);
       n.Next = head;
       head = n;
    }
    public IEnumerator<T> GetEnumerator()  {
       Imbricat current = head;
       while (current != null)   {
           yield return current.Data;
           current = current.Next;
       }    }
    // IEnumerable<T> mosteneste interfata IEnumerable din
    //System.Collections,
    // asadar aceasta clasa trebuie sa implementeze ambele
    // versiuni generica si negenerica
    // ale metodei  GetEnumerator. In cele mai multe cazuri
    // metoda negenerica
    // face apel la metoda generica.
     IEnumerator IEnumerable.GetEnumerator()   {
       return GetEnumerator();
    }
}
```

```
public class ListaSortata<T> : ListaGenerica<T> where
      T : System.IComparable<T>
{
   //Un algoritm de sortare a elementelor
   //de la cel mai mic la cel mai mare

   public void MetodaSortare()
   {
      if (null == head || null == head.Next)
      {
         return;
      }
      bool schimb;

      do
      {
         Imbricat previous = null;
         Imbricat current = head;
         schimb = false;

         while (current.Next != null)
         {
if (current.Data.CompareTo(current.Next.Data) > 0) {
                  Imbricat tmp = current.Next;
                  current.Next = current.Next.Next;
                  tmp.Next = current;

                  if (previous == null)  {
                     head = tmp;
                  }
                  else  {
                     previous.Next = tmp;
                  }
                  previous = tmp;
                  schimb = true;
               }
               else {
                  previous = current;
                  current = current.Next;
               }
            }
         } while (schimb);
      }
}
```

```
public class A: System.IComparable<A>
{
   int i;
   public A(int i)
   {
      this.i = i;
   }

   public int CompareTo(A a)
   {
      return i - a.i;
   }

   public override string ToString()
   {
      return i.ToString();
   }
}
```

```
class TestListaGenerica{
   static void Main()  {
      // A este tipul argumentului
      ListaSortata<A> lista = new ListaSortata<A>();

      for (int x = 0; x < 10; x++)  {
         lista.AddHead(new A(x));
      }

      foreach (A a in lista)    {
         System.Console.Write(a + " ");
      }
      System.Console.WriteLine();

      lista.MetodaSortare();

      foreach (A a in lista)
      {
         System.Console.Write(a + " ");
      }
      System.Console.WriteLine("\nDone");
   }
}
```

-Interfețe multiple pot fi specificate drept constrângeri asupra unui singur tip, după cum urmează:

```
class Stack<T> where T : System.IComparable<T>, IEnumerable<T> { }
```

-O interfață poate defini mai mult de un parametru de tip, după cum urmează:

```
interface IDictionary<K, V> { }
```

-Regulile de moștenire care se aplică la clase, se aplică de asemenea la interfețe:

```
interface ILuna<T> { }
interface IJanuarie: ILuna<int> { } //fara eroare
interface IFebruarie<T> : ILuna<int> { } // fara eroare
interface IMartie<T> : ILuna<T> { } // fara eroare
//interface IAprilie : ILuna<T> {} //eroare
```

Obs: Interfețele generice pot moșteni interfețe non-generice, doar dacă interfața generica este contra-variantă, ceea ce înseamnă că isi folosește parametrul de tip ca o valoare return. În biblioteca .NET, IEnumerable <T> moștenește IEnumerable deoarece IEnumerable <T> folosește T ca valoare return a metodei GetEnumerator.

-Clasele concrete pot implementa interfețe închise, după cum urmează:

```
interface IInterfataDeBaza<T> { }
class ClasaDemo : IInterfataDeBaza<string> { }
```

-Clasele generice pot implementa interfețe generice sau interfețe închise atâta timp cât lista de parametri tip ai clasei cuprinde toate argumentele necesare interfeței, după cum urmează:

```
interface IInterfataDeBaza1<T> { }
interface IInterfataDeBaza2<T, U> { }

class ClasaDemo1<T> IInterfataDeBaza1<T> { } //fara eroare
class ClasaDemo2<T> : IInterfataDeBaza2<T, string> { } //fara eroare
```

Metode Generice

-O metodă generică este o metodă care este declarată cu parametrii de tip, după cum urmează:

```
static void Schimb<T>(ref T stanga, ref T dreapta)
{ T temp;
temp = stanga;
stanga = dreapta;
dreapta = temp; }
```

-Următorul exemplu arată o modalitate de a apela metoda folosind int pe post de argument de tip:

```
public static void TestSchimb() {
    int a = 1; int b = 2;
    Schimb<int>(ref a, ref b);
    System.Console.WriteLine(a + " " + b);
 }
```

-Puteți omite argumentul de tip intrucat compilatorul il va deduce. Următorul apel pentru Schimb este echivalent cu apelul anterior:

```
Schimb(ref a, ref b);
```

Obs. Aceleași reguli de deducere a tipului la apelul metodei, daca tipul nu este precizat (vezi slide-ul anterior) se aplică atat metodelor statice cat și metodelor instantelor. Compilatorul poate deduce parametrii de tip (T in slide-ul anterior), pe baza argumentelor primite de metoda. Compilatorul nu poate deduce parametrii de tip daca metoda nu are parametrii formali. Prin urmare deducerea tipului funcționează cu metode care au parametri formali de tipul T.

-Într-o clasă generică, metodele non-generice pot accesa parametrii de tip (ai clasei), după cum urmează:

```
class ClasaDemo<T> {
        void OMetoda(ref T lhs, ref T rhs) { }
}
```

-Dacă definiți o metodă generică care are aceiași parametri de tip ca si clasa care conține metoda, compilatorul generează avertismentul CS0693, deoarece în blocul metodei, argumentul furnizat de T interior ascunde argumentul furnizat de T exterior.

Dacă aveți nevoie de flexibilitate in apelul unei metode generice cu argumente de tip, altele decât cele furnizate clasei atunci cand aceasta a fost instantiata, atunci se ia în considerare furnizarea unui alt identificator pentru parametrul tip al metodei, așa cum se arată în ListaGenerica2 <T>:

```
class ListaGenerica1<T>
{
        // generaza avertismentul CS0693
         void MetodaDemo<T>() { }
}
 class ListaGenerica2<T>
{
         //nici un avertisment
         void MetodaDemo<U>() { }
}
```

-Utilizați constrângeri, pentru a permite operațiuni specializate asupra parametrilor de tip ai metodelor. Această metoda numita Schimb2 <T>, poate fi utilizata numai cu argumente de tip care implementeaza IComparable <T>.

```
void Schimb2<T>(ref T stanga, ref T dreapta) where T : System.IComparable<T>
        {
                T temp;
                 if (stanga.CompareTo(dreapta) > 0)
                 {
                        temp = stanga;
                         stanga = dreapta;
                         dreapta= temp;
                }
        }
```

-Metodele generice poate fi supraîncărcate pe mai mulți parametri de tip. De exemplu, următoarele metode pot fi situate în aceeași clasă:

```
void  Metoda() { }
void  Metoda<T>() { }
void Metoda<T, U>() { }
```

Delegari generice

-O delegare poate defini proprii sai parametri tip. Codul care refera delegarea generica poate specifica argumentul de tip pentru a crea un tip închis, la fel ca atunci când se instantiaza o clasă generica sau se apeleaza o metodă generica, așa cum se arată în următorul exemplu:

```
public delegate void Del<T>(T item);
public static void Metoda(int i) { }

Del<int> m1 = new Del<int>(Metoda);
```

-Versiunea C# 2.0 are o nouă caracteristică, care functioneaza atat cu delegarile concrete, precum și cu delegarile generice, și vă permite să scrieti ultima linie de mai sus cu această sintaxă simplificată:

```
Del<int> m2 = Metoda;
```

-Delegarile definite într-o clasă generica pot utiliza parametrii de tip clasa generica în același mod în care o fac metodele clasei:

```
class Demo<T> {
        T[ ] items;
        int index;
        public delegate void DelegareDemo(T[ ] items);
}
```

-Codul care face referire la delegare trebuie să specifice argumentul tip al clasei continute, după cum urmează:

```
private static void Metoda(float[] items) { }

public static void TestMetoda() {
        Demo<float> s = new Demo<float>();
Demo<float>.DelegareDemo d = Metoda;
}
```

Clasa Nullable

-Tipurile valorice difera de tipurile referinta prin faptul ca primele contin o valoare. Tipurile valorice pot exista in stare sa zicem “neatribuita” imediat dupa ce sunt declarate si inainte de a li se atribui o valoare. Insa nu pot fi utilizate intr-o expresie daca nu li se atribuie o valoare.

-Din contra, un tip referinta poate fi null.

-Exista cazuri cand este necesar sa avem o valoare pentru orice tip folosit, chiar daca aceasta este null (in particular cand se lucreaza cu baze de date).

-Genericele ofera o modalitate de a face acest lucru prin utilizarea clasei generice System.Nullable<T>, spre exemplu:

```
System.Nullable<int>    nullableInt;
```

-Acest cod declara o variablila care poate avea orice valoare de tip int insa si valoarea null.

-Se poate scrie

```
nullableInt=null;
```

cod echivalent cu

```
nullableInt=new System.Nullable<int>();
```

- Clasa Nullable<T> pune la dispozitie proprietatile HasValue si Value. Daca HasValue este true atunci este garantata o valoare pentru Value. In caz contrar (HasValue este false), daca se apeleaza Value atunci este lansata o exceptie de tipul System.InvalidOperationException.

- Intrucat tipurile Nullable<T> sunt des utilizate, in locul sintaxei:

```
System.Nullable<int> nullableInt;
```

se utilizeaza forma prescurtata:

```
int? nullableInt;
```

-In cazul tipurilor simple precum int, se pot utiliza operatorii +, -, * etc. pentru a lucra cu valori. In cea ce priveste tipurile Nullable nu exista nici o diferenta. Spre exemplu, putem avea:

```
int? op1=5;
int? result=op1*2;
```

Rezultatul este de tip int?

-Insa urmatorul cod nu poate fi compilat:

```
int? op1=5;
int  result=op1*2;
```

-Pentru ca lucrurile sa fie in ordine, trebuie utilizat un cast:

```
int? op1=5;
int  result=(int)op1*2;
```

sau utilizata proprietatea Value

```
int? op1=5;
int  result=op1.Value*2;
```

-Ce se intampla cand unul din operanzi este null?

Raspunsul este urmatorul: pentru toate tipurile simple Nullable, in afara de bool?, rezultatul este null. Pentru bool? rezultatul este dat in tabelul

op1	op2	op1 &op2	op1 |op2
true	null	null	true
false	null	false	null
null	true	null	true
null	false	false	null
null	null	null	null

Exemplu (clasa Vector):

```
using System;
public class Vector
{
    public double? r = null;
    public double? theta = null;

    public double? ThetaRadiani
    {
        get
        {
            //theta in radiani
            return theta * 4 * Math.Atan(1) / 180;
        }
    }
    public Vector(double? r, double? theta)
    {
        theta = theta % 360;

        this.theta = theta;
        this.r = r;
    }
```

```
 public static Vector operator +(Vector op1, Vector op2)    {
       try{
          double X_suma = op1.r.Value * Math.Cos(op1.ThetaRadiani.Value) + op2.r.Value *
Math.Cos(op2.ThetaRadiani.Value);
          double Y_suma = op1.r.Value * Math.Sin(op1.ThetaRadiani.Value) + op2.r.Value *
Math.Sin(op2.ThetaRadiani.Value);
          double r_suma = Math.Sqrt(X_suma*X_suma+Y_suma*Y_suma);
          double theta_suma = Math.Atan2(Y_suma, X_suma)*180/4/Math.Atan(1);
          return new Vector(r_suma, theta_suma);
       }
       catch{
          return new Vector(null, null);
       }
    }
    public static Vector operator -(Vector op)    {
       Vector v = new Vector(null, null);
       v.r = op.r;
       v.theta = op.theta + 180;
       return v;
    }

    public static Vector operator -(Vector op1, Vector op2)    {
       return op1 + (-op2);
    }
    public override string ToString()    {
       string r_string = r.HasValue ? r.ToString() : null;
       string theta_string = theta.HasValue ? theta.ToString() : null;
       return string.Format("{0}[cos({1})+i sin({1})]", r_string, theta_string);
    }
}
```

```
class Program{
    public static void Main()   {
        Vector v1 = ObtineVector("vectorul1");
        Vector v2 = ObtineVector("vectorul2");
        Console.WriteLine("{0}+{1}={2}", v1, v2, v1+v2);
        Console.WriteLine("{0}-{1}={2}", v1, v2, v1 - v2);
        Console.ReadKey();
    }
    public static Vector ObtineVector(string numeVector)   {
        Console.Write("Introduceti magnitudinea pentru {0}: ", numeVector);
        double? r = ObtineNullableDouble();
        Console.Write("\n Introduceti unghiul (in grade) pentru {0}: ", numeVector);
        double? theta = ObtineNullableDouble();
        Console.Write("\n");
        return new Vector(r,theta);
    }
    public static double? ObtineNullableDouble()   {
        double? rezultat;
        string string_introdus=Console.ReadLine();
        try  {
            rezultat = double.Parse(string_introdus);
        }
        catch
        {
            rezultat = null;
        }
        return rezultat;
    }
}
```