Microsoft .net

C# 3.0

Notions avançées

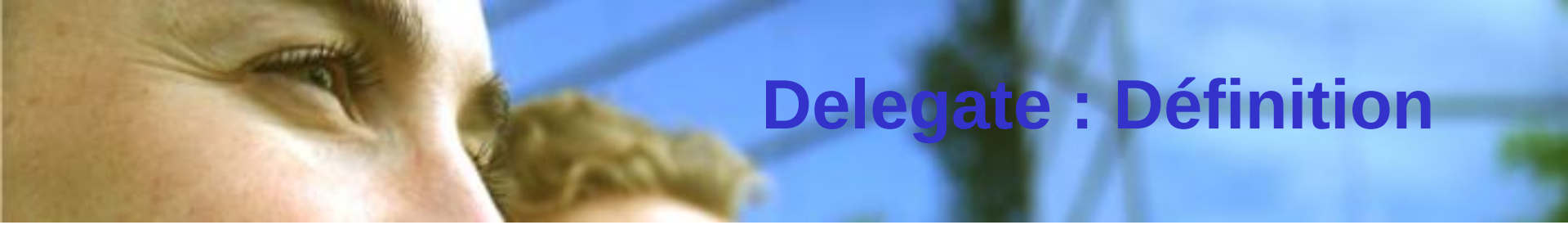
Key Consulting

Agenda

- Delegate
- Methode anonymne
- Threading
- Exception
- Générique
- Itérateur
- Attributs
- Nullable

Agenda

- **Delegate**
- Methode anonymne
- Threading
- Exception
- Générique
- Itérateur
- Attributs
- Nullable



Delegate : Définition

- **Permet de référencer des méthodes**
  - S'apparente à un « *pointeur sur fonction* »
  - L'appel s'effectue dans le *thread* appelant
- **C'est un type qui peut être déclaré dans :**
  - Le scope globale
  - Une classe
  - Une structure

```
delegate void Deleg ();

static void fct();

Deleg d1 = new Deleg(fct);
```

- **Il faut distinguer :**
  - Le « type delegate » et « l'instance delegate »

Delegate : instanciation

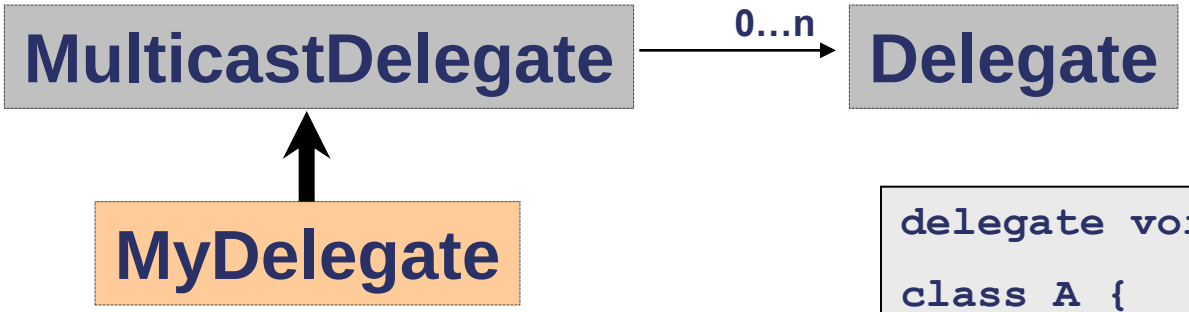
- **L'instanciation peut s'effectuer avec ou sans l'utilisation du mot clé « new »**
- **Un « delegate » peut référencer une méthode avec exactement la même signature :**
  - De classe (statique)
  - Ou d'instance

```
delegate void Deleg ();
class A {
  static void Fct1(){};
  void Fct2(){};
}
```

```
Deleg deleg1 = A.Fct1;
A a = new A();
Deleg deleg2 = a.Fct2()
```

Delegate : Multi référencement

- Un delegate peut référencer plusieurs méthodes




```
delegate void Deleg ();
class A {
 public int _i = 0;
 public void Fct1() { _i++; }
 public void Fct2() { _i+=10; }
}
```

```
A a = new A();
Deleg deleg1 = a.Fct1;
deleg1 += a.Fct2;
deleg1();
```

a._i vaut 11

Delegate : Multi référencement

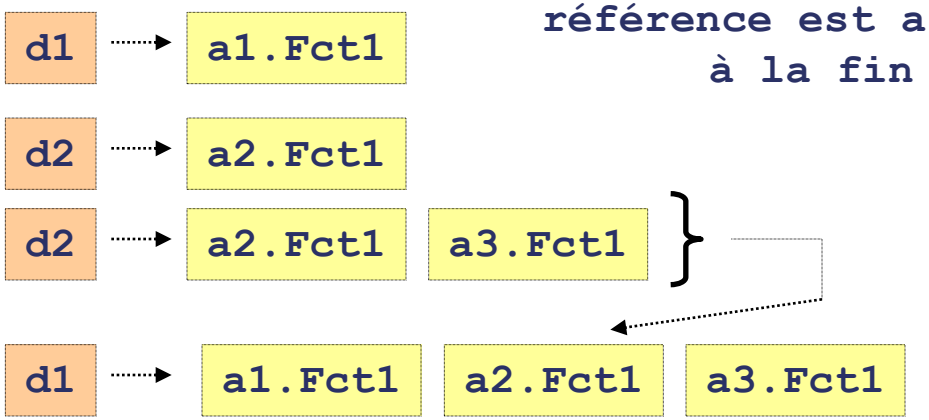
- **Une instance delegate peut être ajoutée a une autre instance de delegate**

```
delegate void Deleg ();
class A {
 int _i = 0;
 public void Fct1() { _i++; }
}
```

```
A a1 = new A();
A a2 = new A();
A a3 = new A();
```

```
Deleg d1  = a1.Fct1;
Deleg d2  = a2.Fct1;
d2 += a3.Fct1;

d1 += d2;
```



La liste des référence est ajoutée à la fin

Delegate : Valeur de retour

- **Un « delegate » peut renvoyer une valeur de retour**

```
delegate int Deleg ();
```

- **Si le « delegate » référence plusieurs méthodes**
  - C’est le retour de la dernière référence qui est renvoyée

- **`MulticastDelegate.GetInvocationList`**
  - Permet de récupérer la liste des méthode et ainsi récupérer chaque retour de méthode

Delegate : Suppression de référence

- **«-=» permet de supprimer de la liste des références :**
  - Une référence vers une méthode
  - Une instance de « delegate »
    - La liste des références de l'instance est supprimée
- **Une instance de « delegate » avec une liste de méthode vide vaut « null »**

```
Deleg d1  = a1.Fct1;
Deleg d2  = a2.Fct1;
d2 += a3.Fct1;
d1 += d2
d1 -= a1.Fct1;
d1 -= d2;
```

3 éléments dans la liste

2 éléments dans la liste

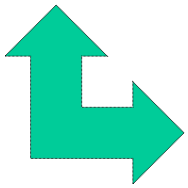
d1 vaut null

Event

- **Un événement est une instance de « delegate »**
  - Possibilité de redéfinir les accesseurs à l' « event »
    - « add » et « remove »
  - Ne peut être activé de l'extérieur de la classe

```
delegate void Deleg();
class A {
  public event Deleg MyDelegate;
}
```

équivalent



```
delegate void Deleg();
class A {
 private event Deleg _MyDelegate;
 public event Deleg MyDelegate {
  add { _MyDelegate += value; }
  remove { _MyDelegate -= value; }
 }
}
```

Event : Pattern

.Net recommande le pattern suivant :

```
delegate void NameEventHandler(object s, NameEventArgs args);
class NameEventArgs : EventArgs { }
class A {
  public event NameEventHandler MyDelegate;
  public OnMyDelegate() {
   if ( MyDelegate!=null ) {
     MyDelegate(this,new NameEventArgs());
   }
 }
}
```

Agenda

- Delegate
- **Methode anonymne**
- Threading
- Exception
- Générique
- Itérateur
- Attributs
- Nullable

Méthode anonyme : Définition (1/2)

- **Une méthode anonyme est :**
  - Une méthode sans nom
  - Adaptée aux petits traitements
  - Associé à un « delegate »
  - Précédé du mot clé « delegate »

```
delegate void Deleg();

Deleg d = delegate()
{ System.Console.WriteLine("Methode anonymne"); };
d();
```

Méthode anonyme : Définition (2/2)

- **Une méthode anonyme peut**
  - Avoir des arguments
    - Ne supporte pas le mot clé « param »
  - Avoir une valeur de retour
    - Le type de la valeur de retour est définit par le « delegate »

```
delegate int Deleg(int p1, ref int p2, out int p3);


Deleg d = delegate(int p1, ref int p2, out int p3)
{ p2+=p1; p3 = p2; return p1 + p2 + p3; };


int i = 10;
int j;
int k = d(1,ref i,out j);
```

```
i = 11
j = 11
k = 23
```

Méthode anonyme : Cas particulier

- **Une méthode anonyme sans paramètre est assignable à tout « delegate » si :**
  - Il n'a pas de valeur de retour
  - Il n'a pas de paramètre « out »

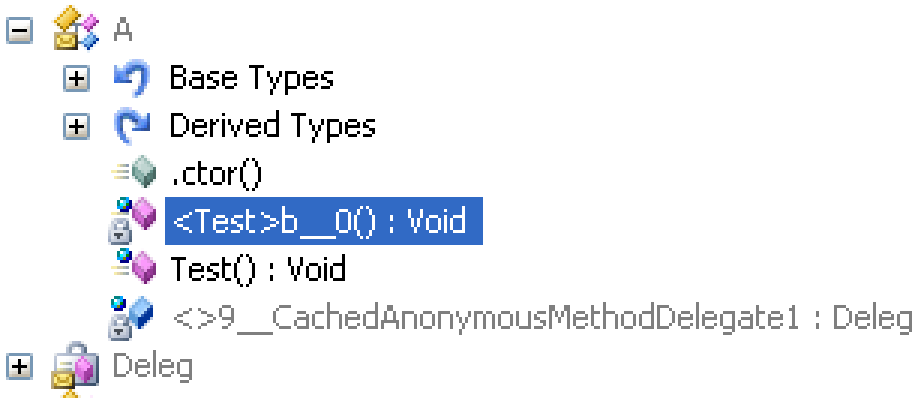
```
delegate void Deleg(int p1, int p2, int p3);
Deleg d = delegate(int p1, int p2, int p3)
  { System.Console.WriteLine( p1 + p2 + p3); };
d += delegate ()
  { System.Console.WriteLine(« No Args"); };
d(1,2,3);
```

```
6
No Args
```

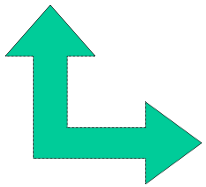
Méthode anonyme : Génération IL

- **L'IL ne connaît pas les méthodes anonymes**
  - C'est le compilateur qui génère les instructions IL

```
delegate void Deleg();
class A {
 static void Test() {
 Deleg d = delegate() {};
}
```



Code généré



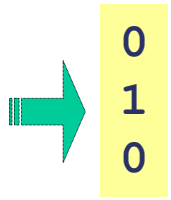
```
[CompilerGenerated]
private static void <Test>b__0()
{
}
```

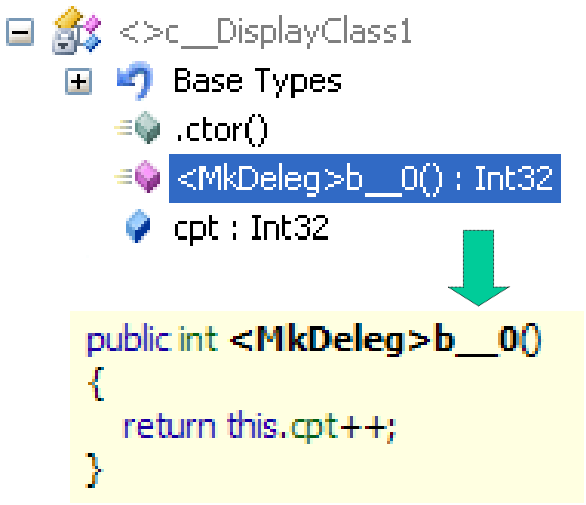
Méthode anonyme : Génération IL

- **Accés à une variable locale de la méthode encapsulante :**
  - Création d'une classe local
  - Création d'un champ du type de la variable

```
public static Deleg MkDeleg() {
  int cpt = 0;
  return delegate() {return cpt++;};
 }
```

```
Deleg d1 = MkDeleg();
Deleg d2 = MkDeleg();
System.Console.WriteLine(d1());
System.Console.WriteLine(d1());
System.Console.WriteLine(d2());
```

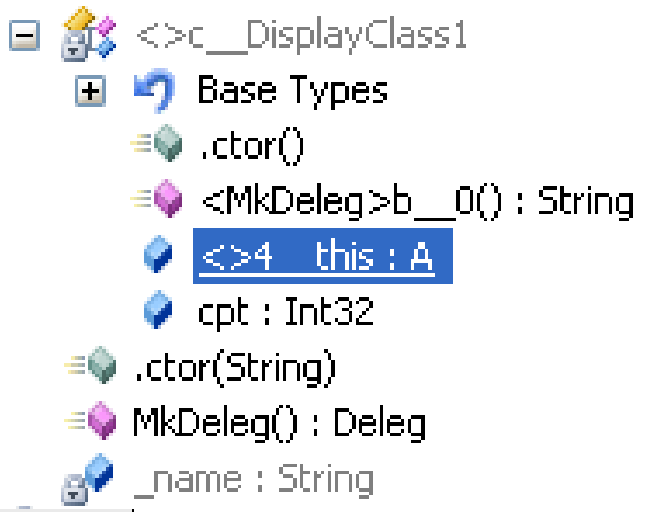




Méthode anonyme : Génération IL

Accés à un membre de la classe encapsulante :

- Création d'une classe local
- Capture de « this »




```
delegate string Deleg();
class A {
 string _name;
 public A(string name) { _name = name;}
 public Deleg MkDeleg() {
  int cpt =0;
  return delegate() { return _name + cpt++; };
}}
```

```
 Deleg d = new A("Class A").MkDeleg();
 System.Console.WriteLine(d());
 System.Console.WriteLine(d());
```

```
Class A0
Class A1
```

Expression Lambda

En lieu et place des méthodes anonymes:

- Une liste de paramètres
- Le signe =>
- Une expression

```
List<int> list = new List<int>(new int [] { -1, 2, -5, 45, 5 });
List<int>positiveNb = list.FindAll(delegate(int i) {return i>0;});
```

```
List<int> list = new List<int>(new int [] { -1, 2, -5, 45, 5 });
var positiveNb = list.FindAll((int i) => i > 0);
```

Agenda

- Delegate
- Methode anonymne
- **Threading**
- Exception
- Générique
- Itérateur
- Attributs
- Nullable

Threading : Définition

- **Instance de `System.Threading.Thread`**
  - Peut avoir une priorité
  - Peut être nommée
  - Peut référencer une méthode statique ou une méthode de classe

```
class Program  {
  static void f1(){};
  void f2(){};
  static void Main() {
    Thread t1 = new Thread(f1);
    t1.Start();
    Program p = new Program();
    Thread t2 = new Thread(p.f2);
    t2.Start();
    t1.Join(); t2.Join();
  }
}
```

Threading : Etats

- **Joindre un Thread**
  - Join(), bool Join(int)
- **Suspendre un Thread**
  - Sleep()
  - Suspend()
- **Reprendre un Thread**
  - Resume()
- **Terminer un Thread**
  - Abort(), Interrupt()

Threading : Pool de thread

- **Un pool de thread par processus : `ThreadPool`**
- **nombre limité de threads (25 par défaut)**
- **Permet de ne pas se soucier de la gestion des threads**
  - « On poste une tâche dans le pool de thread »
- **Poster une tâche :**
  - `ThreadPool.QueueUserWorkItem`

Threading : Interlocked

- **System.Threading.Interlocked**
  - Permet de sécuriser des opérations atomiques
    - Increment
    - Decrement
    - Add

```
static long cpt = 0;
static void fct1() {
  Interlocked.Increment(ref cpt);
}
static void fct2() {
  Interlocked.Decrement(ref cpt);
}
```

```
Thread t1 = new Thread(fct1);
Thread t2 = new Thread(fct1);
t1.Start(); t2.Start();
t1.Join(); t2.Join();
```

Threading : Section critique

Lock : Permet de protéger une partie du code

```
static long cpt = 0;
static void fct1() {
   lock (typeof(Program))
   { cpt++; };
}
static void fct2() {
   lock (typeof(Program))
   { cpt++; };
}
```

- **Lock prend en paramètre un objet de type référence**
  - C'est l'objet de synchronisation
- **Monitor.TryEnter**
  - Attente non bloquante d'accès à une ressource
  - Possibilité d'y mettre une durée

Threading : Wait / Pulse / PulseAll

- **Wait**
  - Le thread se met en attente d'un changement d'état de la ressource
  - Le thread se met dans la liste passive de la ressource
- **Pulse**
  - Passe le premier thread de la liste passive à l'état actif
- **PulseAll**
  - Passe tout les threads de la liste passive de la ressource à l'état actif

Threading : Autres objets de synchronisation

- **Mutex (nommé)**
  - new / WaitOne / ReleaseMutex / Close
  - Permet de faire de la synchronisation entre processus
- **Les événements : Threading.EventWaitHandle**
  - WaitOne / Set
  - Ne définisse pas d'appartenance d'une ressource à un Thread
  - Passe une notification d'un Thread à un autre
- **Sémaphore**

Threading : Lecture multiple

- **ReaderWriterLock**
  - AcquireReaderLock
  - AcquireWriterLock
  - ReleaseWriterLock
- **Accès lecture multiple / écriture unique**
- **Pas de synchronisation sur une ressource mais sur l'objet de synchronisation**

Threading : Méthode Asynchrone

- **Permet de lancer un traitement sans se soucier du Thread qui l'éxecute**
  - Utilise un delegate et les deux méthodes du delegate :
    - `BeginInvoke` → **Lance le traitement**
    - `EndInvoke` → **Se met en attente de la fin du traitement**

```
delegate int Test(string str);
IAsyncResult BeginInvoke([…]);
int EndInvoke(IASyncResult result);
```

Identifie la méthode asynchrone

Threading : Méthode Asynchrone

delegate de finalisation :

void AsyncCallback (IAsyncResult async)

```
delegate int Test(string str);

static void Finalisation(IAsyncResult async) {
  string s = ((string) async.AsyncState);
}

static void TestMethod(string str);
Test test = TestMethod;

Test.BeginInvoke(« test », new AsyncCallback(Finalisation), « fin »);
```

Agenda

- Delegate
- Methode anonymne
- Threading
- **Exception**
- Générique
- Itérateur
- Attributs
- Nullable

Exceptions : Syntaxe

Gestionnaire d'exceptions

```
try {
    …
    throw new Exception(«   »);
    …
} catch ( exception ) {
  …
} catch (System.Exception e) {
  …
} finally {
  …
}
```

Lancement d'un exception

Attrape une exception spécifique

Attrape toutes les exceptions

Code éxecuté dans tout les cas

Exceptions : Définir son exception

`System.Exception` ⟵ **Doit hériter de**

- **`string Message`**
  - Description de l'erreur
- **`string Source`**
  - Nom de l'objet qui a généré l'exception
- **`string HelpLink`**
  - Référence vers une page d'explication
- **`Exception InnerException`**
  - Référence vers l'exception d'origine dans le cas d'une exception rattrapée et relancée

Exceptions : UnhandledException

- **Event UnhandledException sur AppDomain**
  - Déclenché lorsqu'une exception n'est pas attrapée dans le code du domaine d'application

```
AppDomain.CurrentDomain.UnhandledException
  += new UnhandledExceptionEventHandler(MyHandle);
```

```
static void MyHandle(object sender, UnhandledExceptionEventArgs e)
 {}
```

Agenda

- Delegate
- Methode anonymne
- Threading
- Exception
- **Générique**
- Itérateur
- Attributs
- Nullable

Générique : définition (1/2)

- **Permet de paramétrer une classe**

```
class A<T> {
  T _field;
}
```

```
A<int> a1= new A<int>();
A<B> a2 = new A<B>();
```

```
class A<int> {
  int _field;
}
```

```
class A<B> {
  B _field;
}
```

- **Avantage :**
  - Pas de transtypage à l'utilisation
  - Pas de Boxing/UnBoxing avec les types valeurs
  - Type Safe dans le cas de l'utilisation d'ensembles
- **Les interfaces et les structures peuvent être génériques**

Génériques : définiton (2/2)

Une classe peut être paramétré par plusieurs types

```
class A<U,V> {
  U _field1;
  V _field2;
}
```

```
A<int,double> a = new A<int,double>()
```

- **Type générique ouvert**
  - Au moins un des paramètres n'est pas définit
- **Type générique fermé**
  - Tout les paramètres sont définit
  - Peut avoir un alias (using)

```
class A<U,V> {}
class B<U> : A<U,int> {}
class C : B<double> {}
```

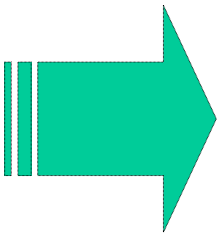
Générique : Génération IL

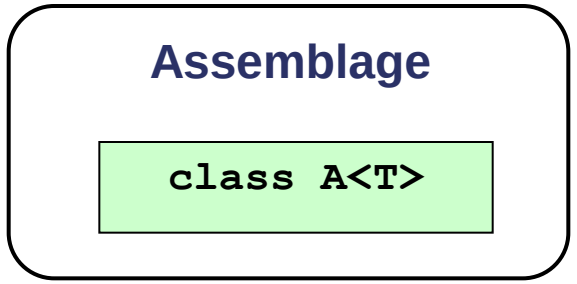
```
class A<U> {}
class B {}
```

```
A<int> a = new A<int>();
A<double> a = new A<double>();
A<bool> a = new A<bool>();

A<B> a = new A<B>();
```

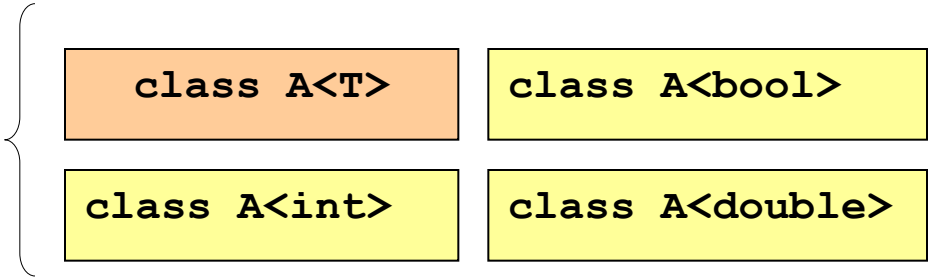
Compilateur C #





Compilateur JIT du CLR

Une classe par type valeur
Une classe pour tout les types références



Génériques : Contraindre un paramètre

- **Trois sortes de contraintes « where » :**
  - Avoir un constructeur par défaut
    - `new()`
  - Dérive ou implémente un type simple, fermé ou ouvert
    - `where X : Y`
  - De type référence ou valeur
    - `where X : class`
    - `where X : struct`

```
public class BaseClass {}
public interface I1 { }
public class A<U, V>
  where U : class, new()
  where V : BaseClass, I1 {}
```

Génériques : Paramètres des méthodes

- **Les méthodes peuvent avoir des types paramètres**
  - Possibilité d'ambiguïté lors de la surcharge d'opérateurs

```
class A<U,V> {
  public void Fct1(U i) {}
  public void Fct1(V i) {}
  public void Fct1(int i) {}
  public void Fct2(U i, V j) {}
  public void Fct2(V i, U j) {}
}
```

```
A<int,int> a = new A<int,int>();
a.Fct1(1);
a.Fct2(1, 1);
a.Fct2(1, 1);
```

Erreur de compilation

- **Si ambiguïté alors la préférence ira vers la méthode qui a le moins de type paramètre**

Générique : Statique

- **Un champ statique existe autant de fois qu'un type fermé existe**
- **Une classe générique peut avoir un constructeur statique**
  - Moyen de placer une contrainte sur le type paramètre

```
class A<U> {
  public static int _nbr = 0;
  public static void Incr (){_nbr++;}
}
class B { }
class C { }
```

```
A<int>.Increment();
A<double>.Increment();
A<B>.Increment();
A<C>.Increment();
```

_nbr vaut 1 pour
tout les types fermés

Génériques : Transtypage

- Le transtypage est permit sur les « Types Paramêtres »
- Transtypage implicite possible si il y a contrainte de dérivation

```
class B  { }
class C : B{ }
```

Conversion implicite

Covariance sur les éléments

Contravariance sur les éléments

```
class A<U> where U : B {
  private U _param;
  U[] _arrayU = new U[10];
  void FCt() {
    B b = _param;
    B[] arrayB = _arrayU;
    U[] arrayU = (U[])arrayB;
  }
}
```

Génériques : Méthodes génériques

- Une méthode d'une classe peut être générique
- Les types paramètres peuvent être contraints

```
class A {
  static public void Fct1<U>(U u) where U : struct {
  }
}
```

```
A<int>.Fct1(1);
A.Fct1(1);
A.Fct1<double>("");
A.Fct1<string>("");
```

Inférence U est int

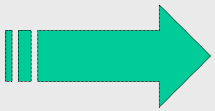
Erreur d'inférence

Erreur de contrainte

Génériques : Méthode anonyme

- **Les méthodes anonymes peuvent avoir des arguments de type générique :**

```
delegate void Concat<U,V>(U arg1,V arg2 );
public static void Main()  {
  Concat<string, string> fct1 = delegate(string a1, string a2) {
   System.Console.WriteLine(a1 + a2);
  };
  Concat<double, double> fct2 = delegate(double a1, double a2) {
    System.Console.WriteLine("{0}+{1}={2}", a1, a2, a1 + a2);
  };
  fct1("A", "B");
  fct2(1, 2);
}
```



AB
1+2=3

Agenda

- Delegate
- Methode anonymne
- Threading
- Exception
- Générique
- **Itérateur**
- Attributs
- Nullable

Itérateur en C#

- **Exemple d'itérateur en C#**

```
public class Test : IEnumerable {
  string[] m_list;
  public Test(params string[] list) {
    m_list = new string[list.Length];
    list.CopyTo(m_list, 0);
  }
  public IEnumerator GetEnumerator() {
    foreach (string s in m_list)
      yield return s;
  }
}
```


```
Test test = new Test(«A», «B», «C»);
foreach (string s in test)
   System.Console.WriteLine(s);
```

Agenda

- Delegate
- Methode anonymne
- Threading
- Exception
- Générique
- Itérateur
- **Attributs**
- Nullable

Attributs : Définition

- **Information positionnée sur un élément du code et inséré dans l'assemblage**
  - Class, Method, Assembly, Parameter …
- **Peut être paramétré**

```
[assembly : CLSCompliant(true)]
namespace Application {
    [CLSCompliant(true)]
    public class Test {
        [CLSCompliant(false)]
        public void Method(UInt32 i) { }
    }
}
```

Attributs : Extension

- **Possibilité de définir son propre attribut**
  - Classe qui dérive de System.Attribute
  - XXXAttribute

```
[AttributeUsage(AttributeTargets.Class,AllowMultiple=false)]
public class TestAttribute : System.Attribute {
  public TestAttribute() { }
  public TestAttribute(int p) { }
  public int Property {
     get { return 0; } set { }
  }
}
```

```
[Test]
[Test(2)]
[Test(Property=2)]
```

Agenda

- Delegate
- Methode anonymne
- Threading
- Exception
- Générique
- Itérateur
- Attributs
- **Nullable**

Nullable : Définition

- **Une référence de type est nulle quand il ne référence rien**
  - Quand est il d'un type valeur ?

- **La solution de C# 2 :**
  - ***System.Nullable<T>***

```
Nullable<int> i1 = null;
Nullable<int> i2 = 1;
Nullable<int> i3 = 2;
Nullable<int> result = i2 + i3;
System.Console.WriteLine(result);
System.Console.WriteLine(i1==null?"(null)":i1.ToString());
i1 = 12;
System.Console.WriteLine(i1);
```

3
(null)
12

Nullable : Syntaxe simplifiée

T? ⟺ System.Nullable<T>

- Le cast implicite n'est pas possible

```
int? i1 = 2;
int i2 = i1; // Erreur de compilation
```

- Le boxing converti un `Nullable` en référence `null`

```
int? i1 = null;
object o = i1;      // o vaut null
int? i2 = (int?)o; // i2 vaut null
```