
C#

Lionel Seinturier

Université des Sciences et Technologies de Lille

Lionel.Seinturier@univ-lille1.fr

10/01/09

C#

1

Lionel Seinturier

Introduction

C#

- langage OO compilé
- inspiré de C/C++

- **très proche** de Java
- réintroduit éléments C/C++ manquants dans Java
 - struct, enum, pointeurs de fonction (*delegate*)

- standardisation ECMA/ISO
- spécification : msdn.microsoft.com/vcsharp/team/language/default.aspx

C#

2

Lionel Seinturier

Plan

1. Types
2. Classes
3. Instructions
4. Documentation de code
5. *Delegates*/événements

6. I/O
7. *Threads*
8. Réseau
9. Reflexivité

10. C# v2.0
11. C# v3.0
12. C# v4.0

C#

3

Lionel Seinturier

1. Types

Types de base

byte, sbyte	8 bits (non signé/signé)	(System.Byte SByte)
short, ushort	16 bits (signé/non signé)	(System.Int16 UInt16)
int, uint	32 bits	(System.Int32 UInt32)
long, ulong	64 bits	(System.Int64 UInt64)
float, double	32/64 bits	(System.Single Double)
char	16 bits Unicode	(System.Char)
bool	true ou false	(System.Boolean)
decimal	grand nombre	(System.Decimal)
string	chaîne de char	(System.String) + +=

boxing/unboxing automatique

```
int x = 345;  
object o = x;  
int x2 = (int) o;
```

C#

4

Lionel Seinturier

1. Types

Types construits

Enumération

```
enum Color { Red, Blue, Green };
Color c = Color.Green;
Console.WriteLine(c.ToString());
```

- un groupe de valeurs
- définit un nouveau type
- arithmétique : + - ++ -- & | ~ ^
- peut être explicitement converti vers int `int i = (int) c`
- tous les types énumérés héritent de `System.Enum`

1. Types

Types construits

Tableau

```
int[] tab; // déclaration
int[] tab = new int[3]; // allocation
int[] tab = {43,6,21}; // initialisation

int taille = tab.Length;
foreach(int elt in tab) { Console.WriteLine(elt); }

int[,] mat = new int[4,2]; // régulier
int[][] foo = new int[2][]; // irrégulier
```

- indice à partir de 0
- tous les types tableau héritent de `System.Array`
- tests de dépassement automatiques à l'exécution (`IndexOutOfRangeException`)

1. Types

Types construits

Structure

```
struct Personne {
    public string Nom;
    public int Age;
    public Personne(int nom, int age) { Nom=nom; Age=age; }
}

Personne bob = new Personne("Robert",15);
Console.WriteLine(bob.Age);
```

- ensemble d'attributs
- mêmes principes qu'une classe
 - attributs typés (par défaut `private`), constructeurs, méthodes
 - instanciés avec `new`
 - peuvent implémenter des interfaces
- transmis par valeur

1. Types

Types construits

Structure – transmission par valeur

```
void foo( Personne p ) {
    p.Age++;
    Console.WriteLine(bob.Age); // affichage: 16
}

Personne bob = new Personne("Robert",15);
foo(bob);
Console.WriteLine(bob.Age); // affichage: 15
// si class Personne, affichage: 16
```

- classe = instances transmises par référence

1. Types

Types construits

Les collections

- gestion d'ensembles de valeurs
- namespace `System.Collections`

ICollection

Queue	Enqueue, Dequeue
Stack	Push, Pop, Peek
ICollection	Add, [idx], Remove, RemoveAt, Insert, IndexOf, ...
ArrayList	
IDictionary	Add(key,value), [key], Remove(key), GetEnumerator, ...
Hashtable	
SortedList	

```
Hashtable h1 = new Hashtable();
h1.Add("One", 1);
Console.WriteLine(h1["One"]);
```

2. Classes

Classes

- attributs + méthodes + propriétés
- par défaut `private`
- constructeurs, destructeur (facultatif)
- convention nommage : nom classe commence par une majuscule

```
class Personne {
    string Nom;
    int Age;

    public Personne(int nom, int age) {
        Nom = nom;
        Age = age;
    }

    public void Anniversaire() {
        Age++;
    }
}
```

2. Classes

Classes

- méthode `Main` : point d'entrée dans un programme

```
static void Main()
static int Main()
static void Main(string[] args)
static int Main(string[] args)
```

2. Classes

Implémentation & héritage

- interface : que des signatures de méthodes, pas de code
- classe implémente 0, 1 ou +sieurs interfaces
- toutes les méthodes définies dans les interfaces doivent être implémentées
- convention nommage : nom interface commence par `I`

```
interface IFoo1, IFoo2 { ... }
class Foo : IFoo1, IFoo2 { ... }
```

- héritage simple

```
class Personne { ... }
class Etudiant : Personne { ... }
```

- classe non extensible `sealed`
- classe abstraite : 0, 1 ou +sieurs méthodes non implémentées (`abstract`)
⇒ ne peut pas être instanciée, doit être sous-classée
- classe peut hériter **et** implémenter

2. Classes

Constructeurs

- même nom que la classe
- peuvent être polymorphes (paramètres ≠)

- appel des constructeurs

```
using System;

class MyException : Exception {
    int Id;
    public MyException(string mess) { this(mess,10); }
    public MyException(string mess, int id) {
        base(mess); // constructeur hérité
        this.Id = id;
    } }
}
```

C#

13

Lionel Seinturier

2. Classes

Membres : méthodes, attributs & propriétés

- accessibilité
 - public, private (par défaut)
 - protected : classe ou sous-classes
 - internal : même assembly
 - protected internal : protected ou internal
- peuvent être `static` = partagés par toutes les instances
- convention de nommage : nom commence par majuscule

```
class Foo {
    public static int Cpt;
    public Foo() { Cpt++; }
}

new Foo();
Console.WriteLine(Foo.Cpt); // affichage: 1
new Foo();
Console.WriteLine(Foo.Cpt); // affichage: 2
```

C#

14

Lionel Seinturier

2. Classes

Propriétés

- peuvent être constantes (`const`)
- peuvent être en lecture seule pour les clients (`readonly`)
- sont associées à des getters/setters

```
class Carre {
    public double Cote; // attribut
    public double Aire { // propriété
        get { return Cote*Cote; }
        set { Cote = Math.Sqrt(value); }
    }
}

Carre c1 = new Carre();
c1.Aire = 9; // appel de set
Console.WriteLine(c2.Cote); // affichage: 3

Carre c2 = new Carre();
c2.Cote = 4;
Console.WriteLine(c2.Aire); // appel de get, affichage: 16
```

C#

15

Lionel Seinturier

2. Classes

Méthodes

- peuvent être polymorphes (même nom, paramètres ≠)
- **non virtuelles** par défaut
- redéfinition doit être signalée explicitement

```
class Foo1 { public void Bar() {...} }
class Foo2: Foo1 { public new void Bar() { ... } }

Foo2 f2 = new Foo2(); f2.Bar(); // dans Foo2.Bar()
Foo1 f1 = new Foo2(); f1.Bar(); // dans Foo1.Bar() (Java Foo2.Bar)
```

- méthodes virtuelle et/ou redéfinies déclarées explicitement

```
class Foo3 { public virtual void Bar() {...} }
class Foo4: Foo3 { public override void Bar() {...} }

Foo4 f4 = new Foo4(); f4.Bar(); // dans Foo4.Bar()
Foo3 f3 = new Foo4(); f3.Bar(); // dans Foo4.Bar()
```

C#

16

Lionel Seinturier

2. Classes

Paramètres

- passage de paramètres

- par valeur types de base
- par référence objets, mot-clé `ref`

```
void Swap( ref int x, ref int y ) {  
    int frigo = x;  
    x = y;  
    y = frigo; }  
  
Swap( ref x, ref y );
```

- paramètres de sortie

```
void MethWithout( out int x ) { x=22; }  
  
int p;  
MethWithout(out p);      // après retour p==22
```

moyen d'avoir +sieurs paramètres "retournés" par une méthode

2. Classes

Opérateurs

- définition d'opérateurs pour les classes

- +-++--== : méthodes `public static`
- [] : propriété
- Java non, C++ oui (avec en plus `new () = || &&`)

```
class Complex {  
    public double X; public double Y;  
    public Complex(double x,double y) { X=x; Y=y; }  
  
    public static Complex operator +( Complex c1, Complex c2 ) {  
        return new Complex(c1.X+c2.X,c1.Y+c2.Y); }  
  
    public double this[int i] {    // défini comme une propriété  
        get { if (i==0) return X; else return Y; }  
        set { if (i==0) X=value; else Y=value; }  
    } }  
  
Complex res = new Complex(1,1) + new Complex(1,-4);  
double X = new Complex(1,2.5)[0];      // -> 1
```

3. Instructions

Instructions de base

- identiques C/C++/Java

- tests

```
if (cond) { ... } else { ... }  
switch(expr) {  
    case valeur1 : ... ; break;  
    case valeur2 : ... ; break;  
    ...  
    default : ...  
}
```

rq : expr peut être de type string

- boucles

```
while (cond) { ... }  
do { ... } while(cond);  
for( init ; tant_que_cond ; incrément ) { ... }  
  
break;  
continue;
```

3. Instructions

foreach

- itération sur tableau, collection, classe implémentant `IEnumerable`

```
int[] tab = {8,76,76};  
foreach( int x in tab ) { Console.WriteLine(x); }
```

rq : pas d'accès à l'indice

```
IList l = new ArrayList();  
l.Add("One");  
l.Add("Two");  
foreach( string s in l ) { Console.WriteLine(s); }
```

rq : collections hétérogènes → `foreach(object`

```
interface IEnumerable { IEnumerator GetEnumerator(); }  
interface IEnumerator {  
    object Current { get; }      // propriété  
    bool MoveNext();  
    void Reset(); }  
  
IEnumerator i = GetEnumerator();  
while (i.MoveNext()) {  
    Console.WriteLine(i.Current);  
}
```

3. Instructions

Exceptions

- levée : throw instance de Exception (ou d'une sous-classe)
- récupération : try/catch/finally
- plusieurs catch possibles
- finally systématiquement exécuté

```
try {
    throw new Exception();
}
catch( Exception e ) {
    // e.StackTrace: la pile d'appel de méthodes
    // e.Message: le message associé à l'exception
    // e.InnerException: en cas d'exception imbriquée
}
finally {
    // tjrs exécuté qu'il y ait une exception ou non
}
```

3. Instructions

using

- *garbage collection* (GC) non maîtrisé par le programmeur
- using permet de forcer le ramassage d'un objet
- la classe de l'objet peut implémenter IDisposable

```
using( Personne p = new Personne("Bob",15) ) {
    // ...
}
// p est ramassé à la sortie de using et Dispose() est appelée

class Personne: IDisposable {
    // ...
    public void Dispose() {
        // on libère les ressources
    }
}
```

3. Instructions

Gestion de types

- conversion de type (cast) avec (*typecible*)

```
Object x = ...
MyClass mc = (MyClass) x;
```

InvalidCastException si la référence n'est pas conforme au type cible

- instruction as
- conversion de type
- affecte null si la conversion n'est pas possible

```
Object x = ...
MyClass mc = x as MyClass;
```

- is permet de tester l'appartenance à un type

```
if ( x is MyClass )
    MyClass y = (MyClass) x;
```

3. Instructions

Quelques méthodes utiles

- Console.WriteLine(...)
- string s = Console.ReadLine()

- Thread.Sleep(ms) *(namespace System.Threading)*
- DateTime.Now.ToString() *date et heure courante*

3. Instructions

Annotations

Informations (*tag*, méta-données, ...) ajoutées en tête d'un élément de code (classe, interface, méthode, propriété, ...)

Exemple d'utilisation

- les méthodes qui doivent être synchronisées
- les propriétés qui doivent être persistantes
- les méthodes dont l'accès doit être protégé par un *login* et un mot de passe
- l'auteur d'une classe

```
[ name( name=value, value, ... ), ... ]
[ ... ]
void foo() {
    // ...
}
```

3. Instructions

Définition d'annotations personnalisées

Définir une classe

- qui étend `System.Attribute`
- dont le nom a le suffixe `Attribute`
- dont les propriétés correspondent aux propriétés de l'annotation
- avec un constructeur pour les propriétés dont le nom peut être omis
- avec un attribut `AttributeUsage` qui précise la portée de l'annotation

```
[AttributeUsage(AttributeTargets.Class)]
public class PersoAttribute : Attribute {
    public string Auteur;
    public int Version;
    public PersoAttribute(string a) { Auteur=a; }
}
```

3. Instructions

Utilisation d'annotations personnalisées

```
[Perso("moi")]
public class MaClasse1 { ... }

[Perso("lui",Version=12)]
public class MaClasse2 { ... }
```

Éléments annotables

All, Assembly, Class, Constructor, Delegate, Enum, Event, Field, Interface, Method, Module, Parameter, Property, ReturnValue, Struct

4. Documentation de code

Documentation de code

Générer automatiquement de la documentation à partir du code

- but : clarté, lisibilité, vue de + haut niveau que le source
- essentiellement tournée vers une documentation d'API
- identique javadoc

- commentaire `///`
- balises XML

```
class Compte {
    /// <summary>Débite le compte bancaire</summary>
    /// <param name="montant">Le montant à débiter</param>
    /// <returns>
    /// true si le compte est suffisamment crédité, false sinon
    /// <returns>
    public boolean Debiter( double montant ) {
        ...
    } }
}
```

4. Documentation de code

Balises principales

<code><summary></code>	description générale de l'élément (classe, méthode, ...)
<code><param name="..."></code>	description d'un paramètre
<code><value></code>	description d'une propriété (ou attribut)
<code><returns></code>	description du paramètre de retour
<code><seealso cref="..."></code>	élément en relation
<code><exception></code>	description d'une exception levée
<code><permission></code>	accessibilité de la méthode

4. Documentation de code

Balises autorisées en fonction des éléments

class	<code><summary></code> <code><remarks></code> <code><seealso></code>
struct	<code><summary></code> <code><remarks></code> <code><seealso></code>
interface	<code><summary></code> <code><remarks></code> <code><seealso></code>
delegate	<code><summary></code> <code><remarks></code> <code><seealso></code> <code><param></code> <code><returns></code>
enum	<code><summary></code> <code><remarks></code> <code><seealso></code>
constructor	<code><summary></code> <code><remarks></code> <code><seealso></code> <code><param></code> <code><permission></code> <code><exception></code>
property	<code><summary></code> <code><remarks></code> <code><seealso></code> <code><value></code> <code><permission></code> <code><exception></code>
method	<code><summary></code> <code><remarks></code> <code><seealso></code> <code><param></code> <code><returns></code> <code><permission></code> <code><exception></code>
event	<code><summary></code> <code><remarks></code> <code><seealso></code>

4. Documentation de code

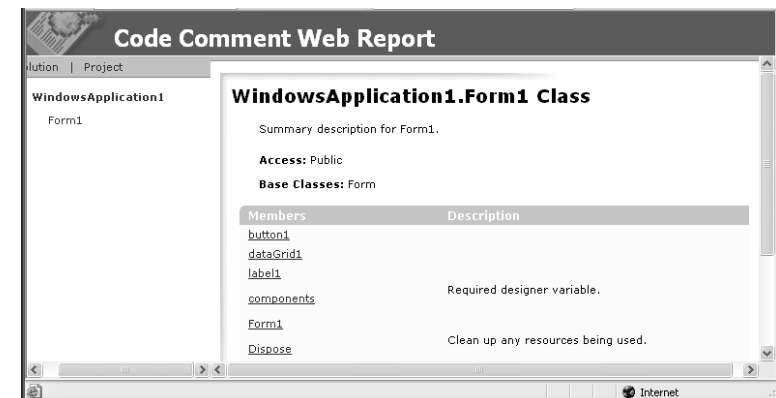
Autres Balises

<code><code></code>	un extrait de code dans la documentation
<code><example></code>	un exemple de mise en œuvre de l'élément
<code><list><item><description></code>	liste
<code><para></code>	créé un paragraphe
<code><paramref name="..."></code>	créé une référence vers un paramètre

4. Documentation de code

Génération du code HTML

Avec Visual Studio .NET



4. Documentation de code

Autres générateurs

```
csc /doc:HelloWorld.xml HelloWorld.cs
```

- extraction des commentaires de documentation dans un fichier XML
- tout traitement (XSLT, ...) possible

```
ndoc.sourceforge.net
```

- javadoc
- LaTeX
- HTML
- MSDN "look"
- VS .NET "look"
- XML

5. Delegates/événements

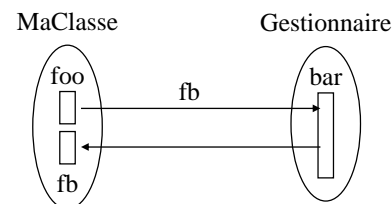
Délégué

- pointeur de méthode
- mécanismes de rappel de l'appelant
ex : on notifie périodiquement l'appelant de l'état d'avancement de la tâche
- injection de code personnalisé
ex : pas de tri dans la classe Listbox mais un délégué pour fournir un tri personnalisé
- signature quelconque pour les délégués
- delegate déclaration d'un type de délégué
- new association entre un type de délégué et un délégué effectif
- les signatures du type de délégué et du délégué effectif doivent être identiques

5. Delegates/événements

Exemple

```
public delegate void FBType(int i);  
class MaClasse {  
    public void foo() {  
        FBType aFb = new FBType(fb);  
        aGestionnaire.bar(aFb);  
    }  
    public void fb(int i) {  
        ...  
    }  
}  
class Gestionnaire {  
    public void bar(FBType aFb) {  
        aFb(25);  
    }  
}
```



5. Delegates/événements

Composition de délégués

- 2 (ou +sieurs) délégués de même type peuvent être combinés avec +=
- un même délégué peut apparaître +sieurs fois dans le délégué composite
- - pour retirer un délégué d'un délégué composite

```
FBType aFb1 = new FBType(fb1);  
FBType aFb2 = new FBType(fb2);  
FBType aFb3 = new FBType(fb3);  
FBType compositeFb1 = aFb1+aFb3;  
compositeFb1 += aFb2;  
compositeFb1(123);  
(compositeFb1-aFb3)(789);
```

5. Delegates/événements

Événements

- mécanisme publish/subscribe
- abondamment utilisé en IHM

- producteur d'événements
 - déclare certain type(s) d'événements qu'il produit
 - produit un ou +sieurs événements
- consommateur d'événements
 - s'abonne à certain(s) type(s) d'événements
- production d'un événement par un producteur
 - tous les consommateurs abonnés sont notifiés

- mécanisme similaire en Java
EventObject, listener, addActionListener, removeActionListener, ...

5. Delegates/événements

Définition des arguments d'un événement

- classes qui héritent de System.EventArgs

```
using System;

class StockChangeEventArgs : EventArgs {

    public StockChangeEventArgs( string stock, int diff ) {
        this.stock = stock;
        this.diff = diff;
    }

    string stock;
    public string Stock { get{ return stock; } }

    int diff;
    public int Diff { get{ return diff; } }
}
```

5. Delegates/événements

Le producteur d'événements

- définit un délégué qui correspond au profil des méthodes abonnées
- définit une liste d'abonnés (mot-clé event) (composition de délégué)
 - l'appel sur event provoque l'appel de toutes les méthodes abonnées

```
class StockManager {

    public delegate void StockChangeEventHandler
        (object src, StockChangeEventArgs e);

    event StockChangeEventHandler OnStockChangeHandler;

    public void Register(StockChangeEventHandler observer) {
        OnStockChangeHandler += new StockChangeEventHandler(observer);
    }

    public UpdateStock(string stock, int diff) {
        StockChangeEventArgs evt = new StockChangeEventArgs(stock,diff);
        if (OnStockChangeHandler != null)
            OnStockChangeHandler(this,evt);
    }
}
```

5. Delegates/événements

Le consommateur d'événements

- fournit une méthode dont le profil respecte celui du délégué

```
class StockWatcher {

    public void OnStockChange(object src, StockChangeEventArgs evt) {
        Console.WriteLine("Le stock "+evt.Stock+" a varié de : "+evt.Diff);
    }
}
```

5. Delegates/événements

Main

- création d'un producteur
- création d'un ou plusieurs consommateurs
- on enregistre les consommateurs auprès du producteur
- on met à jour le stock sur le producteur
 - génère un événement
 - propagé à tous les consommateurs abonnés

```
StockManager mng = new StockManager();
StockWatcher sw = new StockWatcher();
mng.Register(sw.OnStockChange);
```

```
mng.UpdateStock("cafe",2);
mng.UpdateStock("croissant",-3);
```

6. I/O

6.1 Entrées/sorties en mode binaire

6.2 Entrées/sorties en mode caractère

6. Entrées/sorties

Entrées/sorties

Gérées à l'aide des classes du package `System.IO`

But : lire/écrire des données à partir de fichiers, mémoire, réseau, ...

2 formes d'I/O

- mode binaire (`BinaryWriter` et `BinaryReader`)
- mode caractère (`TextWriter` et `TextReader`)

6.1 Mode binaire

BinaryReader

Fournit des méthodes pour **lire** des données

```
int Read()
```

- lecture d'un octet
- bloquant
- retourne -1 si fin du flux

```
BinaryReader br = ...
int i = br.Read();
while ( i != -1 ) {
    byte b = (byte) i;
    ...
    i = br.Read();
}
is.Close();
```

BinaryWriter

Fournit des méthodes pour **écrire** des données

```
void Write(int b)
```

- écriture d'un octet
- `Close()` pour fermer le flux

```
BinaryWriter bw = ...
bw.Write(12);
bw.Write(45);
...
bw.Close();
```

6.1 Mode binaire

BinaryReader

Autres méthodes pour lire des données

```
int Read( byte[] tab, int offset, int length )
```

- le tableau doit être alloué (ex `byte[256]`) **avant l'appel à Read**
- `offset,length` : lecture de **au max** `length` octets, écrits à partir de `offset` dans `tab`
⇒ `offset + length <= tab.length`
- lecture bloquante
- retourne le nombre d'octets lus

Autres méthodes

```
byte ReadByte();           byte[] ReadBytes(int count);  
char ReadChar();          char[] ReadChar(int count);  
ReadDouble, ReadInt16, ReadInt32, ReadInt64, ReadString, ...
```

lèvent `EndOfStreamException` lorsque la fin du flux est atteinte

6.1 Mode binaire

BinaryWriter

Autres méthodes pour écrire des données

```
void Write( byte[] tab )  
void Write( byte[] tab, int offset, int length )
```

- `offset,length` : écriture de `length` octets à partir de `offset`
⇒ `offset + length <= tab.length`

Autres méthodes

```
Write(char), Write(char[]), Write(char[],int,int),  
Write(double), Write(int), Write(short), Write(boolean),  
Write(string), ...
```

6.1 Mode binaire

BinaryWriter / BinaryReader

Création nécessite un flux (*stream*)

Classe Stream

```
FileStream( string name, FileMode mode )  
BufferedStream(Stream)  
MemoryStream  
NetworkStream
```

FileMode

```
FileStream fs = new FileStream("foo.bin",FileMode.Create);  
BinaryWriter bw = new BinaryWriter(fs);
```

- Append ajout dans le fichier
- Create création, écrasement si existence
- CreateNew création, exception si existence
- Truncate ouverture en écriture, exception si non existence
- Open ouverture, exception si non existence
- OpenOrCreate ouverture ou création

6.2 Mode caractère

TextReader

Fournit des méthodes pour **lire** des caractères

```
int Read()
```

- lecture d'un caractère
- bloquant
- retourne -1 si fin du flux

```
TextReader tr = ...  
int i = tr.Read();  
while ( i != -1 ) {  
    char c = (char) i;  
    ...  
    i = tr.Read();  
}  
tr.Close();
```

TextWriter

Fournit des méthodes pour **écrire** des caractères

```
void Write(char b)
```

- écriture d'un caractère
- `Close()` pour fermer le flux

```
TextWriter tw = ...  
tw.Write('c');  
tw.Write('Z');  
...  
tw.Close();
```

6.2 Mode caractère

TextReader

Autres méthodes pour lire des données

```
int Read( char[] tab, int offset, int length )
```

- le tableau doit être alloué (ex char[256]) **avant l'appel à read**
- offset,length : lecture de **au max** length caractères, écrits à partir de offset
⇒ offset + length <= tab.length
- lecture bloquante
- retourne le nombre de caractères lus

Autres méthodes

```
string ReadLine()  
string ReadToEnd()
```

Lecture au clavier

```
string s = Console.ReadLine()
```

6.2 Mode caractère

TextWriter

Autres méthodes pour écrire des données

```
void Write( char[] tab )  
void Write( char[] tab, int offset, int length )
```

- offset,length : écriture de length caractères à partir de offset
⇒ offset + length <= tab.length

Autres méthodes

```
Write(String), Write(Boolean), Write(Double), Write(int), ...  
idem WriteLine(String), ...
```

6.2 Mode caractère

Classe System.IO.File

Permet de gérer des fichiers

Méthodes

```
FileStream static File.Create( string nom )  
FileStream static File.Open( string nom )  
StreamWriter static File.CreateText( string nom )  
idem OpenText, OpenRead, OpenWrite  
  
boolean static Exists(string)  
boolean static Delete(string)  
boolean static Copy(string,string)  
boolean static Move(string,string)  
... + manipulation des attributs (date création, accès, ...) des fichiers
```

Remarque : classe Directory pour la gestion des répertoires

7. Threads

7.1 Introduction

7.2 Modèle de programmation

7.3 Synchronisation

7.4 Exclusion mutuelle

7.5 Autres politiques

7.6 Fonctionnalités complémentaires

7.1 Introduction

Threads

Possibilité de programmer des traitements concurrents

⇒ simplifie la programmation dans de nombreux cas

- programmation événementielle (ex. IHM)
- I/O non bloquantes
- timers, déclenchements périodiques
- servir plusieurs clients simultanément (serveur Web, BD, ...)

⇒ meilleure utilisation des capacités (CPU) de la machine

- utilisation des temps morts

7.1 Introduction

Threads

Processus vs *threads* (= processus légers)

- processus : espaces mémoire séparés
- *threads* : espace mémoire partagé
(seules les piles d'exécution des *threads* sont ≠)

⇒ + efficace

⇒ - robuste

- le plantage d'un *thread* peut perturber les autres
- le plantage d'un processus n'a pas (normalement) d'incidence sur les autres

Approches mixtes : plusieurs processus ayant plusieurs *threads* chacun

Rq : processus manipulables en C# via la classe `System.Diagnostics.Process`

7.2 Modèle de programmation

Modèle de programmation

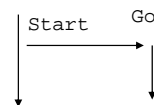
Namespace `System.Threading`

Écriture d'une méthode déléguée (*delegate*)

- de type `delegate void ThreadStart();`
- instantiation de la classe `Thread` avec une instance de `ThreadStart`
- lancement du *thread* en appelant méthode `Start`

```
ThreadStart ts = new ThreadStart(Go);
Thread t = new Thread(ts);
t.Start();
// ... la suite du programme ...

void Go() {
    // les instructions à exécuter dans le thread
}
```



⇒ exécution concurrente du code lanceur et du *thread*

7.2 Modèle de programmation

Modèle de programmation

Remarques

- création d'autant de *threads* que nécessaire (même méthode ou méthodes ≠)
- appel de `Start()` une fois et une seule pour chaque *thread*
- un *thread* meurt lorsque sa méthode se termine
- !! on appelle jamais directement la méthode (`Start()` le fait) !!

7.2 Modèle de programmation

Modèle de programmation

Remarques

- pas de passage de paramètre au *thread* via la méthode `start()`
 - ⇒ définir des variables d'instance
 - ⇒ les initialiser lors de la construction

```
class Foo {  
    int P1;  
    Object P2;  
    public Foo(int p1, Object p2) {this.P1=p1; this.P2=p2; }  
    public void Run() { ... P1 ... P2 ... }  
}
```

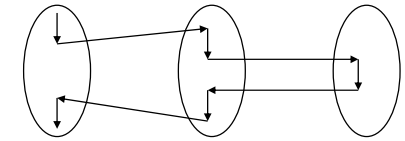
```
Foo aFoo = new Foo(12, aRef);  
new Thread( new ThreadStart(aFoo.Run) ).start();
```

7.3 Synchronisation

Modèle d'objets multi-threadé passifs

En C# : *threads* \perp objets

- *threads* non liés à des objets particuliers
- exécutent des traitements sur éventuellement +sieurs objets
- sont eux-même des objets



"autonomie" possible pour un objet (\approx notion d'agent)

⇒ "auto"-thread

```
public class Foo {  
    public Foo() { new Thread(new ThreadStart(Run)).Start(); }  
    public void Run() { ... }  
}
```

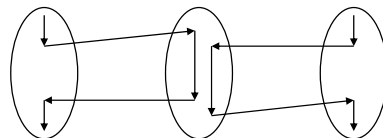
⇒ la construction d'un objet lui assigne des instructions à exécuter

7.3 Synchronisation

Modèle d'objets multi-threadé passifs

2 ou +sieurs *threads* peuvent exécuter la **même méthode** simultanément

- ⇒ 2 flux d'exécutions distincts (2 piles)
- ⇒ 1 même espace mémoire partagé (les champs de l'objet)



7.4 Exclusion mutuelle

Exclusion mutuelle

Besoin : code d'une méthode section critique

- ⇒ au plus 1 *thread* simultanément exécute le code de cette méthode pour cet objet
- ⇒ utilisation d'un attribut pour la méthode

```
[MethodImpl(MethodImplOptions.Synchronized)]  
public void Ecrire(...) { ... }
```

- ⇒ si 1 *thread* exécute la méthode, les autres restent bloqués à l'entrée
- ⇒ dès qu'il termine, le 1er *thread* resté bloqué est libéré
- ⇒ les autres restent bloqués

7.4 Exclusion mutuelle

Exclusion mutuelle

Autre besoin : bloc de code (∈ à une méthode) section critique

- ⇒ au plus 1 *thread* simultanément exécute le code de cette méthode pour cet objet
- ⇒ utilisation du mot clé `lock`

```
public void Ecrire2(...) {  
    ...  
    lock(objet) { ... } // section critique  
    ...  
}
```

objet : objet de référence pour assurer l'exclusion mutuelle (en général `this`)

Chaque objet est associé à un verrou
`lock` = demande de prise du verrou

7.4 Exclusion mutuelle

Exclusion mutuelle

Le contrôle de concurrence s'effectue au niveau de l'objet

- ⇒ +sieurs exécutions d'une même méth. `Synchronized` dans des objets ≠ possibles
- ⇒ si +sieurs méthodes `Synchronized` ≠ dans 1 même objet
au plus 1 *thread* dans **toutes les méthodes** `Synchronized` de l'objet
- ⇒ les autres méthodes (non `Synchronized`) sont tjrs exécutables concurrentement

Remarques

- coût
appel méthode `synchronized` ≈ 3 fois + long qu'appel méthode "normal"
⇒ à utiliser à bon escient

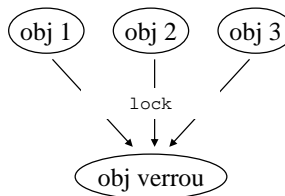
7.4 Exclusion mutuelle

Exclusion mutuelle

Autre besoin : exclusion mutuelle "à +sieurs"

i.e. + méthodes et/ou blocs de codes dans des obj. ≠ en exclusion entre eux

- ⇒ choix d'un objet de référence pour l'exclusion
- ⇒ tous les autres se "`synchronized`" sur lui



7.5 Autres politiques

Autres politiques de synchronisation

Ex : lecteurs/écrivain, producteur(s)/consommateur(s)

⇒ utilisation des méthodes

`Monitor.Wait()` Et `Monitor.Pulse()` (*namespace* `System.Threading`)

`Wait()` : met en attente le *thread* en cours d'exécution

`Pulse()` : réactive un *thread* mis en attente par `Wait()`
si pas de *thread* en attente, RAS

!! ces méthodes nécessitent un accès exclusif à l'**objet exécutant** !!

⇒ à utiliser avec méthode `Synchronized` ou bloc `lock(object)`

```
lock(this) { Monitor.Wait(this); }  
lock(this) { Monitor.Pulse(this); }
```

7.5 Autres politiques

Méthode `wait()`

Fonctionnement

Entrée dans `synchronized` ou `lock`
- acquisition de l'accès exclusif à l'objet

`wait()`
- mise en attente du *thread*
- relachement de l'accès exclusif
- attente d'un appel à `Pulse()` par un autre *thread*
- attente de la réacquisition de l'accès exclusif
- reprise de l'accès exclusif

Sortie du `synchronized` ou `lock`
- relachement de l'accès exclusif à l'objet

7.5 Autres politiques

Méthode `Pulse()`

Réactivation d'un *thread* en attente sur un `wait()`

Si +sieurs *threads*
réactivation du 1er endormi

`Pulse()` pas suffisant pour certaines politiques de synchronisation
notamment lorsque compétition pour l'accès à une ressource

- 2 *threads* testent une condition (faux pour les 2) → `wait()`
- 1 3ème *thread* fait `Pulse()`
- le *thread* réactivé teste la condition (tjrs faux) → `wait()`
→ les 2 *threads* sont bloqués
→ `PulseAll()` réactive **tous les *threads*** bloqués sur `wait()`

7.5 Autres politiques

Politique lecteurs/écrivain

soit 1 seul écrivain, soit plusieurs lecteurs

- demande de lecture : bloquer si écriture en cours ⇒ booléen écrivain
- demande d'écriture : bloquer si écriture ou lecture en cours ⇒ compteur lecteurs

réveil des bloqués en fin d'écriture et en fin de lecture

```
boolean ecrivain = false;  
int lecteurs = 0;
```

7.5 Autres politiques

Politique lecteurs/écrivain

- demande de lecture : bloquer si écriture en cours
- réveil des bloqués en fin de lecture

```
void DemandeLecture(...) {  
    lock(this) {  
        while (ecrivain) { Monitor.Wait(this); }  
        lecteurs++;  
    } }  
  
// On lit  
  
void FinLecture(...) {  
    lock(this) {  
        lecteurs--;  
        Monitor.PulseAll();  
    } }  
}
```

7.5 Autres politiques

Politique lecteurs/écrivain

- demande d'écriture : bloquer si écriture ou lecture en cours
- réveil des bloqués en fin d'écriture

```
void DemandeEcriture(...) {
    lock(this) {
        while (ecrivain || lecteurs>0) { Monitor.Wait(); }
        ecrivain = true;
    } }

// On écrit

void FinEcriture(...) {
    lock(this) {
        ecrivain = false;
        Monitor.PulseAll();
    } }
}
```

C#

69

Lionel Seinturier

7.5 Autres politiques

Politique producteurs/consommateurs

1 ou +sieurs producteurs, 1 ou +sieurs consommateurs, zone tampon de taille fixe

- demande de production : bloquer si tampon plein
- demande de consommation : bloquer si tampon vide

réveil des bloqués en fin de production et en fin de consommation

```
int max = ... // taille du tampon
tampon = ... // tableau de taille max
int taille = 0; // # d'éléments en cours dans le tampon
```

C#

70

Lionel Seinturier

7.5 Autres politiques

Politique producteurs/consommateurs

- demande de production : bloquer si tampon plein
- réveil des bloqués en fin de production

```
void DemandeProd(...) {
    lock(this) {
        while (taille == max) { Monitor.Wait(this); }
    } }

// En exclusion mutuelle
// on produit (maj du tampon)
// taille++

void FinProd(...) {
    lock(this) {
        Monitor.PulseAll();
    } }
}
```

C#

71

Lionel Seinturier

7.5 Autres politiques

Politique producteurs/consommateurs

- demande de consommation : bloquer si tampon vide
- réveil des bloqués en fin de consommation

```
void DemandeCons(...) {
    lock(this) {
        while (taille == 0) { Monitor.Wait(this); }
    } }

// En exclusion mutuelle
// on consomme (maj du tampon)
// taille--

void FinCons(...) {
    lock(this) {
        Monitor.PulseAll();
    } }
}
```

C#

72

Lionel Seinturier

7.5 Autres politiques

Schéma général de synchronisation

- bloquer (éventuellement) lors de l'entrée
- réveil des bloqués en fin

```
lock(this) {
    while (!condition) {
        Monitor.Wait(this);
    }
}

// ...

lock(this) {
    Monitor.PulseAll();
}
```

C#

73

Lionel Seinturier

7.5 Autres politiques

Exemple : sémaphore

```
class Semaphore {
    int NbThreadsAutorises;

    public Semaphore( int init ) { NbThreadsAutorises = init; }

    [MethodImpl(MethodImplOptions.Synchronized)]
    public void P() {
        while ( nbThreadsAutorises <= 0 ) {
            Monitor.Wait(this);
        }
        NbThreadsAutorises --;
    }

    [MethodImpl(MethodImplOptions.Synchronized)]
    public void V() {
        NbThreadsAutorises ++;
        Monitor.Pulse();
    }
}
```

C#

74

Lionel Seinturier

7.6 Compléments

Pool de thread

- Serveurs concurrents avec autant de *threads* que de requêtes
- ⇒ concurrence "débridé"
 - ⇒ risque d'écroulement du serveur

Pool de thread : limite le nb de *threads* à disposition du serveur

Pool fixe : nb cst de *threads*

Pb : dimensionnement

Pool dynamique

- le nb de *threads* s'adapte à l'activité
- il reste encadré : [borne sup , borne inf]

Optimisation : disposer de *threads* en attente (mais pas trop)

- encadrer le nb de *threads* en attente

C#

75

Lionel Seinturier

7.6 Compléments

I/O asynchrone

But : pouvoir lire des données sur un flux sans rester bloquer en cas d'absence

Solution

- un *thread* qui lit en permanence et stocke les données dans un buffer
- une méthode read qui lit dans le buffer

```
class AsyncInputStream {
    Stream S; // flux dans lequel on lit
    int[] Buffer = ... // zone tampon pour les données lues

    AsyncInputStream( Stream s ) { S=s;
        new Thread(new ThreadStart(Run)).Start(); }
}
```

```
public void Run() {
    int b = S.read();
    while( b != -1 ) {
        // stocker b dans buffer
        b = S.Read(); } }
```

```
public int Read() {
    return ...
    // lère donnée dispo dans buffer
}
```

C#

76

Lionel Seinturier

8. Réseau

8.1 Notions générales

8.2 TCP

8.1 Notions générales

Protocoles de transport réseaux

Protocoles permettant de transférer des données de bout en bout

- s'appuient sur les protocoles rx inf. (IP) pour routage, transfert noeud à noeud, ...
- servent de socles pour les protocoles applicatifs (RPC, HTTP, FTP, DNS, ...)
- API associées pour pouvoir envoyer/recevoir des données

UDP mécanisme d'envoi de messages
TCP flux **bi-directionnel** de communication
Multicast-IP envoi de message à un groupe de destinataire

8.1 Notions générales

Caractéristiques des protocoles de transport réseaux

2 primitives de communications

- send envoi d'un message dans un buffer distant
- receive lecture d'un message à partir d'un buffer local

Propriétés associées

fiabilité : est-ce que les messages sont garantis sans erreur ?

ordre : est-ce que les messages arrivent dans le même ordre
que celui de leur émission ?

contrôle de flux : est-ce que la vitesse d'émission est contrôlée ?

connexion : les échanges de données sont-ils organisés en cx ?

8.1 Notions générales

Caractéristiques des protocoles de transport réseaux

2 modes

- synchrone les primitives sont bloquantes
- asynchrone les primitives sont non bloquantes

Exemple

- send sync et receive sync
send reste bloqué jusqu'à l'envoi complet du message
receive reste bloquée jusqu'à ce qu'il y ait un message à lire

- send async et receive sync
send retourne immédiatement

Asynchrone + souple
Synchrone programme + simple à écrire

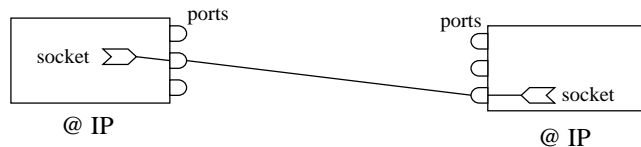
→ receive synchrone + multi-threading ≈ receive asynchrone

	sync	async
send		
receive		

8.1 Notions générales

Adressage

socket = abstraction des extrémités servant à communiquer
chaque socket *associée* à @ IP + n° port local



1 récepteur pour chaque port/machine
éventuellement +sieurs émetteurs vers le même port/machine

8.1 Notions générales

Adressage

Chaque carte rsx (connecté de façon fixe et directe) = 1 point d'accès au rsx
= 1 @ IP permanente

- +sieurs @ IP par machine
- @ IP = 32 bits (128 pour IPv6)

4 classes d'adresses IP

- A : 128 rsx, 16 M @ par réseau
- B : 16 K rsx, 64 K @ par réseau
- C : 2 M rsx, 256 @ par réseau
- D : @ réservées pour la diffusion (Multicast IP)

Correspondance @ symbolique ↔ @ IP assurée par DNS

ex : www.lip6.fr ↔ 132.227.60.13

8.1 Notions générales

Adressage

Classe `System.NET.Dns`

```
static IPHostEntry Resolve(string)    @ IP de la machine <String>  
static string GetHostName()         nom machine locale
```

`IPAddress[] IPHostEntry.AddressList` la liste des @ de la machine

```
IPEndPoint ipe = new IPEndPoint( IPHostEntry.AddressList[0], (int) #port )
```

`ipe.AddressFamily` utilisé pour la connexion sur une socket

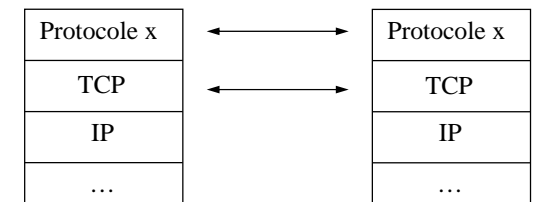
8.1 Notions générales

Problématique de la construction de protocoles applicatifs

TCP, UDP : socles pour la construction de protocoles de + haut niveau

Définition de protocole

- message + paramètres
- format des messages
- enchaînement des messages
- cas d'erreur : message, format, paramètres, enchaînement



!! distinction entre niveaux !!

⇒ utilisation des services du protocole sous-jacent

ex. : ouverture de connexion TCP

8.1 Notions générales

Connexion

Problématique

Les communications entre un client et un serveur sont-elles précédées d'une ouverture de cx ? (et suivies d'une fermeture)

Mode non connecté (le + simple)

- les messages sont envoyés "librement"

Mode connecté

Avantages

- facilite la gestion d'état
- meilleur contrôle des clients (arrivées & départs)

8.1 Notions générales

Connexion

niveau transport et/ou applicatif

	Transport (niveau 4)	Applicatif (niveau 7)
Connecté	TCP	FTP, Telnet, SMTP, POP, JDBC
Non connecté	UDP	HTTP, NFS, DNS, TFTP

Rq : HTTP

- purement non connecté : NFS+UDP
- purement connecté : FTP+TCP
- mixte : HTTP+TCP

- cx lié au transport (TCP)
- pas à HTTP lui-même
- mécanisme session
- ⇒ arrivées & départs

8.1 Notions générales

Pool de connexion

Connexions réseau coûteuse

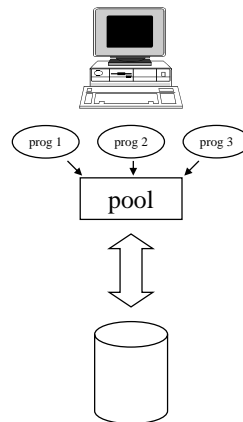
- en ressources occupées
- en temps mis pour ouvrir/fermer les connexions

But du *pool* : mutualiser les cx entre 2 machines
ex. : *pool* de cx vers un SGBD

- tous les progs n'ont pas forcément besoin de toutes les cx en même temps
- le *pool* peut restreindre ou adapter le # de cx simultanées

Si les progs ont besoin de +sieurs cx

- ⇒ politique de gestion de ressources partagées
- ⇒ pb "classique" : réservation, interblocage, ...



8.1 Notions générales

Multiplexage de communications

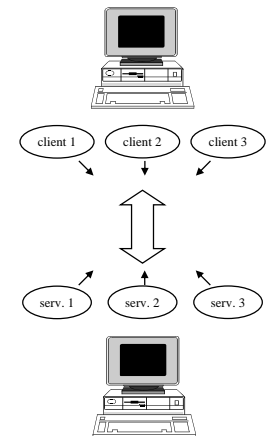
Même objectif que le pool

1 seule cx partagée entre +sieurs prog. c/s

Pb : distinguer les ≠ flux de données

⇒ les encadrer par un protocole de multiplexage

Mécanisme pouvant être couplé avec le pool



8.1 Notions générales

Représentation des données

Problématique

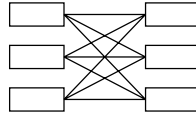
Comm. entre machines avec des formats de représentation de données \neq

→ pas le même codage (*big endian* vs *little endian*)

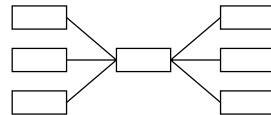
→ pas la même façon de stocker les types (entiers 32 bits vs 64 bits, ...)

2 solutions

On prévoit tous les cas de conversions possibles
(n^2 convertisseurs)



On prévoit un format pivot et on effectue 2 conversions (2n convertisseurs)



Nbreux formats pivots : ASN.1, Sun XDR, sérialisation Java, CORBA CDR, ...

8.1 Notions générales

Traitement des pannes

3 types

- client
- serveur
- réseau - panne de rsx local en général détectable (ex. brin Ethernet)
 - panne rsx large échelle (Internet) non signalée généralement

Dans la majorité des cas

- symptôme : absence de réponse
- cause (rsx, serveur plantée, serveur très lent) **inconnue**
- solution
 - choisir un autre serveur : pas toujours possible
 - abandonner : pas forcément satisfaisant
 - réessayer plus tard (en espérant un retour en service) \Rightarrow mieux

8.1 Notions générales

Traitement des pannes

Problématique

Comportement de l'interaction c/s satisfaisant en présence de ré émissions

(\approx appel de méthode local)

sans que cela soit trop lourd à gérer

\Rightarrow Notion de **sémantique d'invocation**

- au moins 1 fois garantie traitement demandé exécuté au moins 1 fois [1..n]
- au plus 1 fois [0..1]
- exactement 1 fois
 - le plus satisfaisant
 - le plus lourd

8.1 Notions générales

Détection des pannes

Mécanisme complémentaire pour détecter des pannes

en cours d'exécution d'un traitement

- *heart beat* périodiquement le serveur signale son activité au client
- *pinging* périodiquement le client sonde le serveur qui répond

8.2 TCP

Propriétés du protocole TCP

Taille des messages

- quelconque
- envoi en général bufférisé
- vidage autoritaire des buffers possible (dépend OS)

Perte de messages

- acquittements (masqués au programmeur) de messages envoyés
- timeout de ré-émission des messages en cas de non récept. de l'acq. (mécanisme de fenêtre)

Contrôle de flux

- éviter qu'un émetteur trop rapide fasse "déborder" le buffer du récepteur
- blocage de l'émetteur si nécessaire

Ordre des messages

- garantie que l'ordre de réception = ordre de réception
- garantie de non duplication des messages

8.2 TCP

Propriétés du protocole TCP

Connexions TCP

- demande d'ouverture par un client
- acceptation explicite de la demande par le serveur
 - ⇒ au delà échange en mode bi-directionnel
 - ⇒ au delà distinction rôle client/serveur "artificielle"

- fermeture de connexion à l'initiative du client ou du serveur
 - ⇒ vis-à-vis notifié de la fermeture

Pas de mécanisme de gestion de panne

- trop de pertes de messages ou réseau trop encombré
 - connexion perdue

Utilisation

- nombreux protocoles applicatifs : HTTP, FTP, Telnet, SMTP, POP, ...

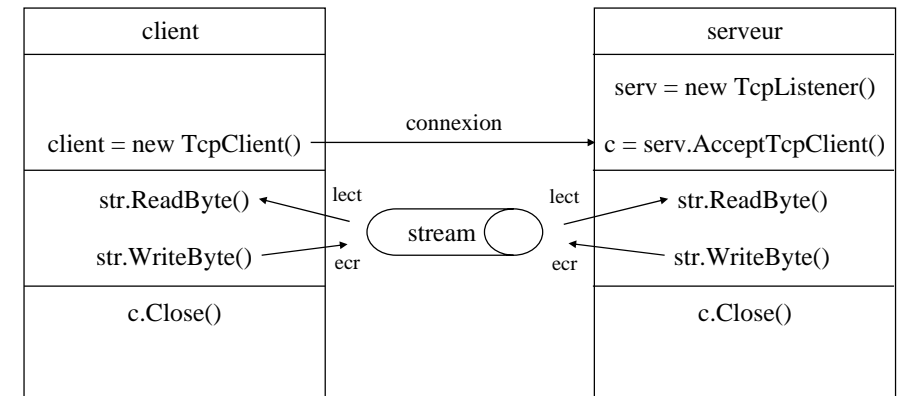
8.2 TCP

Fonctionnement

- serveur crée une *socket* et attend une demande de connexion
- client envoie une demande de connexion
- serveur accepte connexion
- dialogue client/serveur en mode flux
- fermeture de connexion à l'initiative du client ou du serveur

8.2 TCP

Fonctionnement



8.2 TCP

Serveur

```
IPHostEntry iphe = Dns.Resolve("localhost");
IPEndPoint ipep = new IPEndPoint(iphe.AddressList[0],port);

TcpListener s = new TcpListener(ipep);
s.Start(); // début écoute

TcpClient c = s.AcceptTcpClient(); // attente d'un client
Stream str = c.GetStream(); // flux de com avec le client

int i;
while( (i=str.ReadByte()) != -1 ) { ... } // -1 = fin du flux

str.Close();
c.Close();
```

- souvent while (true) { AcceptTcpClient(); ... } + thread
- Stream : ReadByte et/ou Write
- tableau d'octets ou octet par octet : (byte[] data, int offset, int length)

8.2 TCP

Client

```
IPHostEntry iphe = Dns.Resolve("serveurdistant");
IPEndPoint ipep = new IPEndPoint(iphe.AddressList[0],portdistant);

TcpClient c = new TcpClient(ipep);
Stream str = c.GetStream(); // flux de com avec le serveur

str.WriteByte(12); // envoi des données
str.WriteByte(178);
...

str.Close();
c.Close();
```

- Stream : ReadByte et/ou WriteByte
- tableau d'octets ou octet par octet : (byte[] data, int offset, int length)

8.2 TCP

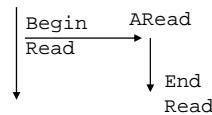
Asynchronisme

- Read et Write **synchrone**
- asynchronisme avec BeginRead/EndRead et BeginWrite/EndWrite

```
BeginRead(
    byte[] buffer, int offset, int length,
    delegate void AsyncCallback(IAsyncResult iar),
    object id ) // objet pour identifier les requêtes read

str.BeginRead( buf, 0, buf.Length, new AsyncCallback(ARead), new Object() )

void ARead( IAsyncResult iar ) {
    int len = str.EndRead(iar); // len = # octets lus
    // traitement octets lus
    // si d'autres octets à lire (rq: il faut savoir)
    str.BeginRead( buf,0,buf.Length,new AsyncCallback(ARead),new Object() )
}
```



8.2 TCP

Autres protocoles de transport

- classe Socket
- fonctionnement identique API C socket : bind/listen/connect
- socket TCP/UDP/multicast IP, read/write sync/async

9. Réflexivité

Réflexivité

Interroger le programme pour qu'il fournisse des informations sur lui-même

2 catégories de réflexivité

- structurelle (la structure d'un programme) oui
- comportementale (son comportement) non

- *namespace* System.Reflection

C#

101

Lionel Seinturier

9. Réflexivité

MemberInfo

MemberInfo

- EventInfo : événement
- FieldInfo : attribut
- MethodBase
 - ConstructorInfo : constructeur
 - MethodInfo : méthode
- PropertyInfo : propriété

Exemples

- MethodBase.GetParameters() : retourne la liste des paramètres (ParameterInfo[])
- ParameterInfo.Name : nom du paramètre
- ParameterInfo.ParameterType : type du paramètre
- MethodInfo.ReturnType : type de retour de la méthode

C#

102

Lionel Seinturier

9. Réflexivité

Chargement dynamique de code

Charger à l'exécution des classes/types

- inconnues à la compilation
- dont le nom est inconnu à la compilation
- générées dynamiquement

Assembly : .exe ou .dll contenant du code (classe, interface) MSIL

Chargement dynamique à partir nom fichier, *path*, URL

Invocation dynamique de méthode

```
Assembly a = Assembly.LoadFrom("serveur.exe"); // chargement
Object o = a.CreateInstance("namespace.MaClasse"); // instantiation
MethodInfo m = o.GetType().GetMethod("Main"); // introspection
m.Invoke(o, new Object[] { null }); // invocation dyn.
```

```
Assembly current = Assembly.GetExecutingAssembly(); // assembly courante
```

C#

103

Lionel Seinturier

9. Réflexivité

API System.Reflection.Emit et System.CodeDom

Générer dynamiquement du code

- Emit: génération de MSIL
- CodeDom: génération de code source à partir du DOM d'un prog. C#

```
CodeMemberMethod cm = new CodeMemberMethod ();
cm.Name = "Add";
cm.ReturnType = new CodeTypeReference(typeof(int));
cm.Parameters.Add(new CodeParameterDeclarationExpression(typeName, "val"));
cm.Attributes = MemberAttributes.Public | MemberAttributes.Final;
cm.Statements.Add (
    new CodeMethodReturnStatement (
        new CodeMethodInvokeExpression (
            new CodeFieldReferenceExpression(
                new CodeThisReferenceExpression(), "List",
                "Add", new CodeArgumentReferenceExpression ("val"))));
```

C#

104

Lionel Seinturier

10. C# v2.0

C# v2.0

Nouvelle version du langage avec .NET Framework 2.0

Nouveautés

- générique
- méthode anonyme
- itérateur
- type partiel
- type *nullable*

Rq : Java 5

10. C# v2.0

Générique

Paramétrer les classes, interfaces avec un type

- principale utilisation : les collections
- collections d'un type particulier
- vérification à la compilation si tentative ajout objet d'un mauvais type

C# v1.1

```
ICollection l = new ArrayList();
l.Add("One");
l.Add(new Object());
foreach( string s in l )
    Console.WriteLine(s)
```

⇒ Exception à l'exécution

```
Unhandled Exception: System.InvalidCastException:
Specified cast is not valid.
```

10. C# v2.0

Générique

C# v2.0

```
using System.Collections.Generic;

ICollection<string> l = new List<string>();
l.Add("One");
l.Add(new Object()); // Erreur de compilation
foreach( string s in l )
    Console.WriteLine(s)
```

Définition d'un type générique

```
class MyClass<T> {
    T element;
    // ... T s'utilise comme n'importe quel type
}

MyClass<MyClass2> obj = new MyClass<MyClass2>();
```

10. C# v2.0

Méthode anonyme

- alléger l'écriture des programmes contenant des méthodes utilisées 1 seule fois par un délégué

C# v1.1

```
delegate void MyDelegate();
static void MyMethod() { /* do something */ }
static MyDelegate Factory() { return new MyDelegate(MyMethod); }

static void Main() {
    MyDelegate md = Factory();
    md();
}
```

10. C# v2.0

Méthode anonyme

C# v2.0

```
delegate void MyDelegate();
static MyDelegate Factory() {
    return delegate() { /* do something */ }
}

static void Main() {
    MyDelegate md = Factory();
    md();
}
```

- MyMethod n'est plus définie
- le corps est directement défini après delegate()

Voir classe anonyme de Java

10. C# v2.0

Itérateur

Retourner 1 à 1 les éléments d'un type énumérable (collection, tableau, ...)

C# v1.1

- classe énumérable doit implémenter IEnumerable
- créer une classe implémentant IEnumerator
- retourner une instance de cette classe dans GetEnumerator

```
interface IEnumerable { IEnumerator GetEnumerator(); }
interface IEnumerator {
    object Current { get; } // propriété
    bool MoveNext();
    void Reset();
}
```

10. C# v2.0

Itérateur

C# v2.0

- implémenter IEnumerable
- nouvelle instruction `yield return`
 - retourne 1 à 1 les éléments à énumérer

```
class Personnes : IEnumerable {
    public IEnumerator GetEnumerator() {
        yield return "Bob";
        yield return "Anne";
    }
}
```

```
Personnes ps = new Personnes();
foreach( string s in ps )
    Console.WriteLine(s);
```

- le 1er appel retourne la valeur du 1er yield return
- le 2ème appel retourne la valeur du 2ème yield return
- ...
- quand il n'y a plus de yield return dans la méthode GetEnumerator, renvoi null
⇒ tout se passe comme si il y avait une "mémoire" du dernier yield return

10. C# v2.0

Type partiel

Permet de séparer la définition d'un type en +sieurs fichiers

- principale utilisation : fragments générés automatiquement

```
public partial class Customer {
    private int id;
    private string name;
    private string address;
    public Customer() { ... }
}

public partial class Customer {
    private List<Order> orders;
    public void SubmitOrder(Order order) { orders.Add(order); }
    public bool HasOutstandingOrders() { return orders.Count > 0; }
}
```

- à la compilation les fragments sont rassemblés dans 1 seule classe

10. C# v2.0

Type *nullable*

Type comportant une valeur spéciale `null`

- évite l'utilisation d'une valeur prédéfinie (ex -1 pour les int) pour les valeurs non def.
- meilleure intégration avec SQL qui supporte en natif les types *nullables*
- idem `null` pour les références d'objets
- déclaration : ? en suffixe du type

```
int? x = null;
int? y = 10;
int? z = (x>=0) ? x : null;
```

Opérateur ??

```
int? z = x ?? y; // x si x non null, sinon y
int a = x ?? -1;
```

11. C# v3.0

C# v3.0

Nouvelle version du langage avec .NET Framework 3.0

Nouveautés

- variable locale typée implicitement
- méthode d'extension
- lambda expression
- initialiseur d'objet
- type anonyme
- LINQ

11. C# v3.0

Variable locale typée implicitement

Mot clé `var`

```
var i = 5;
var s = "Hello World!";
var t = new int[]{ 12, 6 };
var m = new Dictionary<string,int>();

var l;
l = new List<int>();
```

- le type est inféré à partir de la valeur affectée
- erreur de compilation si tentative de changement de type

```
    i = "Essai";
```

- influence des langages de script

11. C# v3.0

Méthode d'extension

- étendre une classe avec de nouvelles méthodes
- les méthodes doivent être `static` et définies dans une classe `static`
- au moins un 1er paramètre : mot-clé `this` + type étendu
- éventuellement paramètres supplémentaires

```
public class A {
    public void Meth() { ... }
}
```

```
public static B {
    public static string NouvelleMethode( this A a, string ext ) {
        return a.ToString() + ext;
    }
}
```

```
A a = new A();
a.Meth();
string s = a.NouvelleMethode("Bar");
```

11. C# v3.0

Méthode d'extension

- pour les types de base aussi (string, int, ...)
- ainsi que pour les classes de la librairie de base (List, Dictionary, ...)

```
public static B {  
    public static string ToUpperCase( this string s ) {  
        return ...  
    }  
}
```

```
string s = "ab";  
string s = s.ToUpperCase();
```

- pas vraiment une extension (au sens de l'héritage)
- les appels aux méthodes étendues sont traduites en appels de méthodes static

```
s.ToUpperCase()    =>    B.ToUpperCase(s)
```

11. C# v3.0

Lambda expression

- construction permettant de définir des fonctions dans un style "mathématique"
- syntaxe
 - liste de paramètres entre parenthèses
 - opérateur =>
 - expression

Exemples

```
( int i ) => i > 0;  
( int i, string s ) => s.Length >= 2 && i==12;
```

- les lambda expressions ont un type et peuvent être passées en paramètre
- exemple : méthode FindAll de la classe List

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

11. C# v3.0

Lambda expression

- chaque lambda expression est traduite en un delegate anonyme
- type de retour : bool
- paramètres : ceux de la lambda expression

```
(int i) => i > 0;    /    delegate( int i ) { return i>0; }
```

- le type du delegate peut alors être utilisé en paramètre

```
delegate bool filter( int i );  
List<int> FindAll( List<int> l, filter myfilter ) {  
    IList<int> res = new List<int>();  
    foreach( int i in l ) { if(myfilter(i)) { res.Add(i); } }  
    return res;  
}  
filter myFilter = (int i) => i > 0;  
FindAll( myList, myFilter );
```

11. C# v3.0

Lambda expression

- un exemple "un peu plus" compliqué (lambda, méthode d'extension, généricité)

```
public static class FilterClass {  
    public delegate bool KeyValueFilter<K,V>(K key, V value);  
  
    public static Dictionary<K,V> FilterBy<K,V>(  
        this Dictionary<K,V> items, KeyValueFilter<K,V> filter ) {  
  
        var result = new Dictionary<K,V>();  
        foreach( KeyValuePair<K,V> element in items ) {  
            if ( filter(element.Key,element.Value) )  
                result.Add( element.Key, element.Value );  
        }  
        return result;  
    }  
}
```

11. C# v3.0

Lambda expression

- un exemple "un peu plus" compliqué (suite et fin)

```
var listLanguages = new Dictionary<string,int>();
listLanguages.Add("C#",5);
listLanguages.Add("VB",3);
listLanguages.Add("Java",-5);

var bestLanguages =
    listLanguages.FilterBy(
        (string name, int score) => name.Length == 2 && score > 0 );

foreach( KeyValuePair<string,int> language in bestLanguages ) {
    Console.WriteLine(language.Key);
}
```

11. C# v3.0

Initialiseur d'objet

- initialisation automatique via les propriétés publiques de la classe

```
public class User {
    public string Name { get {return name;} set {name=value;} }
    public int Age { get {return age;} set {age=value;} }
    private string name;
    private int age;
}

User bob = new User { Name="Bob"; Age=23; };
```

- évite l'écriture des constructeurs
- fonctionne également sur les collections

```
List<int> l = new List<int> { 1, 2, 3, 4 };
```

11. C# v3.0

Type anonyme

- variable **implicitement** typée par un type **sans nom**
- struct avec attribut en lecture seule

```
var MonObjet =
    new { PremierChamp = 1,
          SecondChamp = "Ma Chaîne",
          TroisiemeChamp = new List<int>{1,2,3,4,5} };

Console.WriteLine( MonObjet.PremierChamp );
Console.WriteLine( MonObjet.SecondChamp );
```

- les noms des attributs peuvent être omis et inférés automatiquement

```
int i = 12; string s = "Hello World!";
var MonObjet2 = new { i, s };

Console.WriteLine( MonObjet2.i );
Console.WriteLine( MonObjet2.s );
```

12. C# v4.0

C# v4.0

Nouvelle version du langage avec .NET Framework 4.0

Rq : prévisions

Nouveautés

- support des données typées dynamiquement
- paramètres optionnels et nommés
- variance dans les types génériques

12. C# v4.0

Support des données typées dynamiquement

But : pouvoir intégrer le résultat de l'exécution d'un langage de scripts dans des programmes .NET (C#, VB, C++, ...)

- nouveau mot clé `dynamic`
- le type de la variable dynamique est résolu à l'exécution
- permet l'intégration des langages IronPython, IronRuby, ...
- fonctionnalité similaire dans Java 7

Exemple

```
Object o = ... // un objet qui fournit une méthode Add(int,int)
dynamic d = o;
d.Add(10,20);
```

- sans `dynamic` : `o.Add(10,20)` provoque une erreur de compilation
- avec `dynamic` : compilation ok, résolution de type à l'exécution

12. C# v4.0

Support des données typées dynamiquement

Autre exemple : invocation dynamique de méthode

Avant C# 4.0

```
public static void InvokeMyMethod( Object o ) {
    var myMethod = o.GetType().GetMethod("MyMethod");
    if( myMethod == null )
        throw new InvalidOperationException();
    myMethod.Invoke( o, new object[0] );
}
```

Avec C# 4.0

```
public static void InvokeMyMethod( Object o ) {
    dynamic di = o;
    di.MyMethod();
}
```

12. C# v4.0

Support des données typées dynamiquement

Avantage

- programmation plus souple pour tous les scénarii de dynamicité
 - exécution de scripts
 - chargement dynamique de code
 - introspection

Inconvénient

- programmes moins sûrs
- plus difficile à déboguer

12. C# v4.0

Paramètres optionnels et nommés

But : faciliter l'invocation de méthodes comportant de nombreux paramètres

- définition de valeurs par défaut pour les paramètres "de fin" de signature
- la valeur est prise en compte en cas d'omission du paramètre lors de l'appel

Exemple

```
public void m( int i, string s ="HelloWorld", object o =null ) { ... }

m(12,"FooBar",new MaClasse());

m(12); // compilation ok en C# 4.0
m(12,s:"FooBar"); // nommage obligatoire : s ou o ?
```

- Rq : complique les règles de surcharge en cas de hiérarchie de types

12. C# v4.0

Variance dans les types génériques

But : autoriser les conversions de types pour les types génériques (lorsque c'est possible)

Exemple

```
IList<string> strings = new List<string>();  
IList<object> objects = strings; // Interdit
```

- alors que une string est un objet
- une liste de string n'est pas une liste d'objets
- raison : après coup on pourrait faire

```
objects[0] = 5;
```

- ce qui deviendrait problématique : `strings[0]` ne serait plus une string
- pb lié au fait qu'il y a une méthode (opérateur []) de **modification**

12. C# v4.0

Variance dans les types génériques

But : autoriser la conversion dans les cas où il n'y a pas de modification

Exemple

```
IEnumerable<string> strings = ...;  
IEnumerable<object> objects = strings;
```

Notion de covariance

```
public interface IEnumerable< out T > { ... }
```

out : `IEnumerable<A>` est covariant à `IEnumerable` ssi A peut être converti en B

- string peut être converti en object
- limitation (*boxing*) : `IEnumerable<int>` n'est pas convertible en `IEnumerable<object>`
`IEnumerable<Integer>` ok

12. C# v4.0

Variance dans les types génériques

Notion de contravariance

```
public interface IComparer< in T > { ... }
```

in : `IComparer<A>` est contravariant à `IComparer` ssi B peut être converti en A

Couplage covariance/contravariance

```
public delegate TResult Func< in TArg, out TResult >( TArg arg );
```